cause an invalidation, but the rest can simply accumulate in the block and be transferred in one bus transaction on a flush or a write-back. Sophisticated update schemes might attempt to delay the update to achieve a similar effect (by merging writes in the write buffer), or use other techniques to reduce traffic and improve performance [DaS95]. However, the increased bandwidth demand, the complexity of supporting updates, the trend toward larger cache blocks and the pack-rat phenomenon with sequential workloads underly the trend away from update-based protocols in the industry. We will see in Chapter 8 that update protocols also have some other prob-

[ARTIST: Please remove FSMR, TSMR, CAPMR and COLDMR from legend, and please change the only remaining one, (UPGMR) to Upgrade/Update].

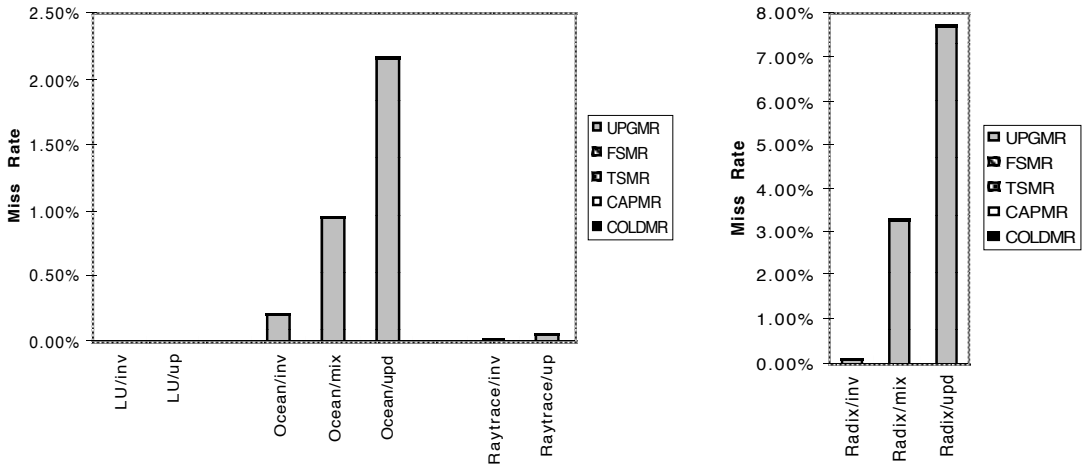


Figure 5-28 Upgrade and update rates for invalidate, update, and mixed protocols

1Mbyte, 64byte, 4-way, and k=4

lems for scalable cache-coherent architectures, making it less attractive for microprocessors to support them.

5.6 Synchronization

A critical interplay of hardware and software in multiprocessors arises in supporting synchronization operations, mutual exclusion, point-to-point events and global events, and there has been considerable debate over the years on what hardware primitives should be provided in multiprocessor machines to support these synchronization operations. The conclusions have changed from time to time, with changes in technology and machine design style. Hardware support has the advantage of speed, but moving functionality to software has the advantage of flexibility and adaptability to different situations. The classic work of Dijkstra [Dij65] and Knuth [Knu66] shows that it is possible to provide mutual exclusion with only atomic read and write operations (assuming a sequentially consistent memory). However, all practical synchronization operations rely on some *atomic read-modify-write* machine primitive, in which the value of a memory loca-

tion is read, modified and written back atomically without intervening accesses to the location by other processors. Simple or sophisticated synchronization algorithms can be built using these primitives.

The history of instruction set design offers a glimpse into the evolving hardware support for synchronization. One of the key instruction set enhancements in the IBM 370 was the inclusion of a sophisticated atomic instruction, the *compare-and-swap* instruction, to support synchronization in multiprogramming on uniprocessor or multiprocessor systems. The compare&swap compares the value in a memory location with the value in a specified register, and if they are equal swaps the value of the location with the value in a second register. The Intel x86 allows any instruction to be prefixed with a lock modifier to make it atomic, so with the source and destination being memory operands much of the instruction set can be used to implement various atomic operations involving even more than one memory location. Advocates of high level language architecture have proposed that the user level synchronization operations, such a locks and barriers, should be supported at the machine level, not just the atomic read-modify-write primitives; i.e. the synchronization "algorithm" itself should be implemented in hardware. The issue became very active with the reduced instruction set debates, since the operations that access memory were scaled back to simple loads and stores with only one memory operand. The SPARC approach was to provide atomic operations involving a register and a memory location, e.g., a simple swap (atomically swap the contents of the specified register and memory location) and a compare-and-swap, while MIPS left off atomic primitives in the early instruction sets, as did the IBM Power architecture used in the RS6000. The primitive that was eventually incorporated in MIPS was a novel combination of a special load and a conditional store, described below, which allow higher level synchronization operations to be constructed without placing the burden of full read-modify-write instructions on the machine implementor. In essence, the pair of instructions can be used to implement atomic exchange (or higher-level operations) instead of a single instruction. This approach was later incorporated in the PowerPC and DEC Alpha architectures, and is now quite popular. Synchronization brings to light an unusually rich family of tradeoffs across the layers of communication architecture.

The focus of this section is how synchronization operations can be implemented on a bus-based cache-coherent multiprocessor. In particular, it describes the implementation of mutual exclusion through lock-unlock pairs, point-to-point event synchronization through flags, and global event synchronization through barriers. Not only is there a spectrum of high level operations and of low level primitives that can be supported by hardware, but the synchronization requirements of applications vary substantially. Let us begin by considering the components of a synchronization event. This will make it clear why supporting the high level mutual exclusion and event operations directly in hardware is difficult and is likely to make the implementation too rigid. Given that the hardware supports only the basic atomic state transitions, we can examine role of the user software and system software in synchronization operations, and then examine the hardware and software design tradeoffs in greater detail.

5.6.1 Components of a Synchronization Event

There are three major components of a given type of synchronization event:

an *acquire method*: a method by which a process tries to acquire the right to the synchronization (to enter the critical section or proceed past the event synchronization)

a *waiting algorithm*: a method by which a process waits for a synchronization to become available when it is not. For example, if a process tries to acquire a lock but the lock is not free (or proceed past an event but the event has not yet occurred), it must somehow wait until the lock becomes free.

a *release method*: a method for a process to enable other processes to proceed past a synchronization event; for example, an implementation of the UNLOCK operation, a method for the last processes arriving at a barrier to release the waiting processes, or a method for notifying a process waiting at a point-to-point event that the event has occurred.

The choice of waiting algorithm is quite independent of the type of synchronization: What should a processor that has reached the acquire point do while it waits for the release to happen? There are two choices here: *busy-waiting* and *blocking*. Busy-waiting means that the process spins in a loop that repeatedly tests for a variable to change its value. A release of the synchronization event by another processor changes the value of the variable, allowing the process to proceed. Under blocking, the process does not spin but simply blocks (suspends) itself and releases the processor if it finds that it needs to wait. It will be awoken and made ready to run again when the release it was waiting for occurs. The tradeoffs between busy-waiting and blocking are clear. Blocking has higher overhead, since suspending and resuming a process involves the operating system, and suspending and resuming a thread involves the runtime system of a threads package, but it makes the processor available to other threads or processes with useful work to do. Busy-waiting avoids the cost of suspension, but consumes the processor and memory system bandwidth while waiting. Blocking is strictly more powerful than busy waiting, because if the process or thread that is being waited upon is not allowed to run, the busy-wait will never end.[1] Busy-waiting is likely to be better when the waiting period is short, whereas, blocking is likely to be a better choice if the waiting period is long and if there are other processes to run. Hybrid waiting methods can be used, in which the process busy-waits for a while in case the waiting period is short, and if the waiting period exceeds a certain threshold, blocks allowing other processes to run. The difficulty in implementing high level synchronization operations in hardware is not the acquire and release components, but the waiting algorithm. Thus, it makes sense to provide hardware support for the critical aspects of the acquire and release and allow the three components to be glued together in software. However, there remains a more subtle but very important hardware/software interaction in how the spinning operation in the busy-wait component is realized.

5.6.2 Role of User, System Software and Hardware

Who should be responsible for implementing the internals of high-level synchronization operations such as locks and barriers? Typically, a programmer wants to use locks, events, or even higher level operations and not have to worry about their internal implementation. The implementation is then left to the system, which must decide how much hardware support to provide and how much of the functionality to implement in software. Software synchronization algorithms using simple atomic exchange primitives have been developed which approach the speed of full hardware implementations, and the flexibility and hardware simplification they afford are

1. This problem of denying resources to the critical process or thread is one problem that is actually made simpler in with more processors. When the processes are timeshared on a single processor, strict busy-waiting without preemption is sure to be a problem. If each process or thread has its own processor, it is guaranteed not to be a problem. Realistic multiprogramming environments on a limited set of processors fall somewhere in between.

very attractive. As with other aspects of the system design, the utility of faster operations depends on the frequency of the use of those operations in the applications. So, once again, the best answer will be determined by a better understanding of application behavior.

Software implementations of synchronization constructs are usually included in system libraries. Many commercial systems thus provide subroutines or system calls that implement lock, unlock or barrier operations, and perhaps some types of other event synchronization. Good synchronization library design can be quite challenging. One potential complication is that the same type of synchronization (lock, barrier), and even the same synchronization variable, may be used at different times under very different runtime conditions. For example, a lock may be accessed with low-contention (a small number of processors, maybe only one, trying to acquire the lock at a time) or with high-contention (many processors trying to acquire the lock at the same time). The different scenarios impose different performance requirements. Under high-contention, most processes will spend time waiting and the key requirement of a lock algorithm is that it provide high lock-unlock bandwidth, whereas under low-contention the key goal is to provide low latency for lock acquisition. Since different algorithms may satisfy different requirements better, we must either find a good compromise algorithm or provide different algorithms for each type of synchronization among which a user can choose. If we are lucky, a flexible library can at runtime choose the best implementation for the situation at hand. Different synchronization algorithms may also rely on different basic handware primitives, so some may be better suited to a particular machine than others. A second complication is that these multiprocessors are often used for multiprogrammed workloads where process scheduling and other resource interactions can change the synchronization behavior of the processes in a parallel program. A more sophisticated algorithm that addresses multiprogramming effects may provide better performance in practice than a simple algorithm that has lower latency and higher bandwidth in the dedicated case. All of these factors make synchronization a critical point of hardware/software interaction.

5.6.3 Mutual Exclusion

Mutual exclusion (lock/unlock) operations are implemented using a wide range of algorithms. The simple algorithms tend to be fast when there is little contention for the lock, but inefficient under high contention, whereas sophisticated algorithms that deal well with contention have a higher cost in the low contention case. After a brief discussion of hardware locks, the simplest algorithms for memory-based locks using atomic exchange instructions are described. Then, we discuss how the simplest algorithms can be implemented by using the special load locked and conditional store instruction pairs to synthesize atomic exchange, in place of atomic exchange instructions themselves, and what the performance tradeoffs are. Next, we discuss more sophisticated algorithms that can be built using either method of implementing atomic exchange.

Hardware Locks

Lock operations can be supported entirely in hardware, although this is not popular on modern bus-based machines. One option that was used one some older machines was to have a set of lock lines on the bus, each used for one lock at a time. The processor holding the lock asserts the line, and processors waiting for the lock wait for it to be released. A priority circuit determines which gets the lock next when there are multiple requestors. However, this approach is quite inflexible since only a limited number of locks can be in use at a time and waiting algorithm is fixed (typically busy-wait with abort after timeout). Usually, these hardware locks were used only by the operating system for specific purposes, one of which was to implement a larger set of software

locks in memory, as discussed below. The Cray xMP provided an interesting variant of this approach. A set of registers were shared among the processors, including a fixed collection of lock registers[**XMP**]. Although the architecture made it possible to assign lock registers to user processes, with only a small set of such registers it was awkward to do so in a general purpose setting and, in practice, the lock registers were used primarily to implement higher level locks in memory.

Simple Lock Algorithms on Memory Locations

Consider a lock operation used to provide atomicity for a critical section of code. For the acquire method, a process trying to obtain a lock must check that the lock is free and if it is then claim ownership of the lock. The state of the lock can be stored in a binary variable, with 0 representing free and 1 representing busy. A simple way of thinking about the lock operation is that a process trying to obtain the lock should check if the variable is 0 and if so set it to 1 thus marking the lock busy; if the variable is 1 (lock is busy) then it should wait for the variable to turn to 0 using the waiting algorithm. An unlock operation should simply set the variable to 0 (the release method). Assembly-level instructions for this attempt at a lock and unlock are shown below (in our pseudo-assembly notation, the first operand always specifies the destination if there is one).

```
lock:   ld   register, location     /* copy location to register */
        cmp  location, #0           /* compare with 0 */
        bnz  lock                   /* if not 0, try again */
        st   location, #1           /* store 1 into location to mark it locked */
        ret                         /* return control to caller of lock */
```

and

```
unlock: st location, #0             /* write 0 to location */
        ret                         /* return control to caller */
```

The problem with this lock, which is supposed to provide atomicity, is that it needs atomicity in its own implementation. To illustrate this, suppose that the lock variable was initially set to 0, and two processes P0 and P1 execute the above assembly code implementations of the lock operation. Process P0 reads the value of the lock variable as 0 and thinks it is free, so it enters the critical section. Its next step is to set the variable to 1 marking the lock as busy, but before it can do this process P1 reads the variable as 0, thinks the lock is free and enters the critical section too. We now have two processes simultaneously in the same critical section, which is exactly what the locks were meant to avoid. Putting the store of 1 into the location just after the load of the location would not help. The two-instruction sequence—reading (testing) the lock variable to check its state, and writing (setting) it to busy if it is free—is not atomic, and there is nothing to prevent these operations from different processes from being interleaved in time. What we need is a way to atomically test the value of a variable *and* set it to another value if the test succeeded (i.e. to atomically read and then conditionally modify a memory location), and to return whether the atomic sequence was executed successfully or not. One way to provide this atomicity for user processes is to place the lock routine in the operating system and access it through a system call, but this is expensive and leaves the question of how the locks are supported for the system itself. Another option is to utilize a hardware lock around the instruction sequence for the lock routine, but this also tends to be very slow compared to modern processors.

Hardware Atomic Exchange Primitives

An efficient, general purpose solution to the lock problem is to have an *atomic read-modify-write* instruction in the processor's instruction set. A typical approach is to have and atomic exchange instruction, in which a value at a location specified by the instruction is read into a register, and another value—that is either a function of the value read or not—is stored into the location, all in an atomic operation. There are many variants of this operation with varying degrees of flexibility in the nature of the value that can be stored. A simple example that works for mutual exclusion is an atomic *test&set* instruction. In this case, the value in the memory location is read into a specified register, and the constant 1 is stored into the location atomically if the value read is 0 (1 and 0 are typically used, though any other constants might be used in their place). Given such an instruction, with the mnemonic t&s, we can write a lock and unlock in pseudo-assembly language as follows:

```
lock:   t&s  register, location   /* copy location to reg, and if 0 set location to 1 */
        bnz  register, lock       /* compare old value returned with 0 */
                                  /* if not 0, i.e. lock already busy, try again */
        ret                       /* return control to caller of lock */
```

and

```
unlock: st location, #0           /* write 0 to location */
        ret                       /* return control to caller */
```

The lock implementation keeps trying to acquire the lock using test&set instructions, until the test&set returns zero indicating that the lock was free when tested (in which case the test&set has set the lock variable to 1, thus acquiring it). The unlock construct simply sets the location associated with the lock to 0, indicating that the lock is now free and enabling a subsequent lock operation by any process to succeed. A simple mutual exclusion construct has been implemented in software, relying on the fact that the architecture supports an atomic test&set instruction.

More sophisticated variants of such atomic instructions exist, and as we will see are used by different software synchronization algorithms. One example is a *swap* instruction. Like a test&set, this reads the value from the specified memory location into the specified register, but instead of writing a fixed constant into the memory location it writes whatever value was in the register to begin with. That is, it atomically exchanges or swaps the values in the memory location and the register. Clearly, we can implement a lock as before by replacing the test&set with a swap instruction as long as we ensure that the value in the register is 1 before the swap instruction is executed.

Another example is the family of so-called *fetch&op* instructions. A fetch&op instruction also specifies a location and a register. It atomically reads the value of the location into the register, and writes into the location the value obtained by applying to the current value of the location the operation specified by the fetch-and-op instruction. The simplest forms of fetch&op to implement are the fetch&increment and fetch&decrement instructions, which atomically read the current value of the location into the register and increment (or decrement) the value in the location by one. A fetch&add would take another operand which is a register or value to add into the previous value of the location. More complex primitive operations are possible. For example, the *compare&swap* operation takes two register operands plus a memory location; it compares the

value in the location with the contents of the first register operand, and if the two are equal it swaps the contents of the memory location with the contents of the second register.

Performance Issues

Figure 5-29 shows the performance of a simple test&set lock on the SGI Challenge.[1] Performance is measured for the following pseudocode executed repeatedly in a loop:

```
lock(L); critical-section(c); unlock(L);
```

where `c` is a delay parameter that determines the size of the critical section (which is only a delay, with no real work done). The benchmark is configured so that the same total number of locks are executed as the number of processors increases, reflecting a situation where there is a fixed number of tasks, independent of the number of processors. Performance is measured as the time per lock transfer, i.e., the cumulative time taken by all processes executing the benchmark divided by the number of times the lock is obtained. The uniprocessor time spent in the critical section itself (i.e. c times the number of successful locks executed) is subtracted from the total execution time, so that only the time for the lock transfers themselves (or any contention caused by the lock operations) is obtained. All measurements are in microseconds.

The upper curve in the figure shows the time per lock transfer with increasing number of processors when using the test&set lock with a very small critical section (ignore the curves with "backoff" in their labels for now). Ideally, we would like the time per lock acquisition to be independent of the number of processors competing for the lock, with only one uncontended bus transaction per lock transfer, as shown in the curve labelled ideal. However, the figure shows that performance clearly degrades with increasing number of processors. The problem with the test&set lock is that every attempt to check whether the lock is free to be acquired, whether successful or not, generates a write operation to the cache block that holds the lock variable (writing the value to 1); since this block is currently in the cache of some other processor (which wrote it last when doing its test&set), a bus transaction is generated by each write to invalidate the previous owner of the block. Thus, all processors put transactions on the bus repeatedly. The resulting contention slows down the lock considerably as the number of processors, and hence the frequency of test&sets and bus transactions, increases. The high degree of contention on the bus and the resulting timing dependence of obtaining locks causes the benchmark timing to vary sharply across numbers of processors used and even across executions. The results shown are for a particular, representative set of executions with different numbers of processors.

The major reason for the high traffic of the simple test&set lock above is the waiting method. A processor waits by repeatedly issuing test&set operations, and every one of these test&set operations includes a write in addition to a read. Thus, processors are consuming precious bus bandwidth even while waiting, and this bus contention even impedes the progress of the one process that is holding the lock (as it performs the work in its critical section and attempts to release the

1. In fact, the processor on the SGI Challenge, which is the machine for which synchronization performance is presented in this chapter, does not provide a test&set instruction. Rather, it uses alternative primitives that will be described later in this section. For these experiments, a mechanism whose behavior closely resembles that of test&set is synthesized from the available primitives. Results for real test&set based locks on older machines like the Sequent Symmetry can be found in the literature [GrT90, MCS87].

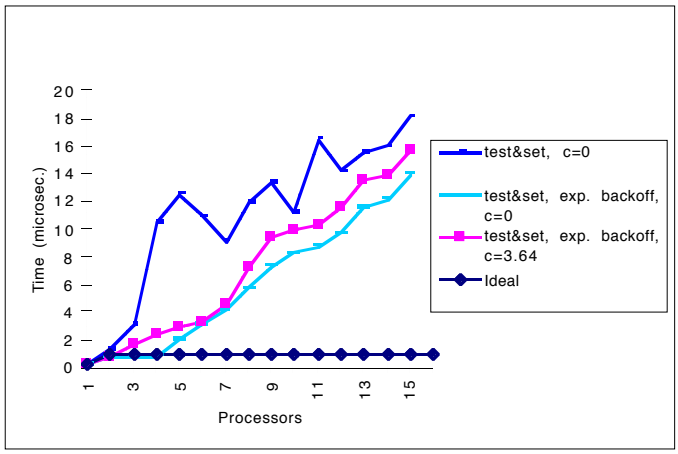


Figure 5-29 Performance of test&set locks with increasing number of competing processors on the SGI Challenge.

The Y axis is the time per lock-unlock pair, excluding the critical section of size c microseconds. The "exp. backoff" refers to exponential backoff, which will be discussed shortly. The irregular nature of the top curve is due to the timing-dependence of the contention effects caused. Note that since the processor on the SGI Challenge does not provide anatomic read-modify-write primitive but rather more sophisticated primitives discussed later in this section, the behavior of a test&set is simulated using those primitives for this experiment. Performance of locks that use test&set and test&set with backoff on older systems can be found in the literature [GrT90, MCS91].

lock).There are two simple things we can do to alleviate this traffic. First, we can reduce the frequency with which processes issue test&set instructions while waiting; second, we can have processes busy-wait only with read operations so they do not generate invalidations and misses until the lock is actually released. Let us examine these two possibilities, called the test&set lock with backoff and the test-and-test&set lock.

Test&set Lock with Backoff. The basic idea with backoff is to insert a delay after an unsuccessful attempt to acquire the lock. The delay between test&set attempts should not be too long, otherwise processors might remain idle even when the lock becomes free. But it should be long enough that traffic is substantially reduced. A natural question is whether the delay amount should be fixed or should vary. Experimental results have shown that good performance is obtained by having the delay vary “exponentially”; i.e. the delay after the first attempt is a small constant k, and then increases geometrically so that after the i^{th} iteration it is $k*c^i$ where c is another constant. Such a lock is called a test&set lock with exponential backoff. Figure 5-29 also shows the performance for the test&set lock with backoff for two different sizes of the critical section and the starting value for backoff that appears to perform best. Performance improves, but still does not scale very well. Performance results using a real test&set instruction on older machines can be found in the literature [GrT90, MCS91]. See also Exercise 5.6, which discusses why the performance with a null critical section is worse than that with a non-zero critical section when backoff is used.

Test-and-test&set Lock. A more subtle change to the algorithm is have it use instructions that do not generate as much bus traffic while busy-waiting. Processes busy-wait by repeatedly *reading* with a standard load, not a test&set, the value of the lock variable until it turns from 1 (locked) to 0 (unlocked). On a cache-coherent machine, the reads can be performed in-cache by all processors, since each obtains a cached copy of the lock variable the first time it reads it. When the lock is released, the cached copies of all waiting processes are invalidated, and the next

read of the variable by each process will generate a read miss. The waiting processes will then find that the lock has been made available, and will only then generate a test&set instruction to actually try to acquire the lock.

Before examining other lock algorithms and primitives, it is useful to articulate some performance goals for locks and to place the above locks along them. The goals include:

Low latency If a lock is free and no other processors are trying to acquire it at the same time, a processor should be able to acquire it with low latency.

Low traffic Suppose many or all processors try to acquire a lock at the same time. They should be able to acquire the lock one after the other with as little generation of traffic or bus transactions as possible. High traffic can slow down lock acquisitions due to contention, and can also slow down unrelated transactions that compete for the bus.

Scalability Related to the previous point, neither latency nor traffic should scale quickly with the number of processors used. Keep in mind that since the number of processors in a bus-based SMP is not likely to be large, it is not asymptotic scalability that is important.

Low storage cost The information needed for a lock should be small and should not scale quickly with the number of processors.

Fairness Ideally, processors should acquire a lock in the same order as their requests are issued. At least, starvation or substantial unfairness should be avoided.

Consider the simple atomic exchange or test&set lock. It is very low-latency if the same processor acquires the lock repeatedly without any competition, since the number of instructions executed is very small and the lock variable will stay in that processor's cache. However, as discussed earlier it can generate a lot of bus traffic and contention if many processors compete for the lock. The scalability of the lock is poor with the number of competing processors. The storage cost is low (a single variable suffices) and does not scale with the number of processors. The lock makes no attempt to be fair, and an unlucky processor can be starved out. The test&set lock with backoff has the same uncontended latency as the simple test&set lock, generates less traffic and is more scalable, takes no more storage, and is no more fair. The test-and-test&set lock has slightly higher uncontended overhead than the simple test&set lock (it does a read in addition to a test&set even when there is no competition), but generates much less bus traffic and is more scalable. It too requires negligible storage and is not fair. Exercise 5.6 asks you to count the number of bus transactions and the time required for each type of lock.

Since a test&set operation and hence a bus transaction is only issued when a processor is notified that the lock is ready, and thereafter if it fails it spins on a cache block, there is no need for backoff in the test-and-test&set lock. However, the lock does have the problem that all processes rush out and perform both their read misses and their test&set instructions at about the same time when the lock is released. Each of these test&set instructions generates invalidations and subsequent misses, resulting in $O(p^2)$ bus traffic for p processors to acquire the lock once each. A random delay before issuing the test&set could help to stagger at least the test&set instructions, but it would increase the latency to acquire the lock in the uncontended case.

Improved Hardware Primitives: Load-locked, Store-conditional

Several microprocessors provide a pair of instructions called load locked and store conditional to implement atomic operations, instead of a atomic read-modify-write instructions like test&set. Let us see how these primitives can be used to implement simple lock algorithms and improve

performance over a test-and-test&set lock. Then, we will examine sophisticated lock algorithms that tend to use more sophisticated atomic exchange operations, which can be implemented either as atomic instructions or with load locked and store conditional.

In addition to spinning with reads rather than read-modify-writes, which test-and-test&set accomplishes, it would be nice to implement read-modify-write operations in such a way that failed attempts to complete the read-modify-write do not generate invalidations. It would also be nice to have a single primitive that allows us to implement a range of atomic read-modify-write operations—such as test&set, fetch&op, compare&swap—rather than implement each with a separate instruction. Using a pair of special instructions rather than a single instruction to implement atomic access to a variable, let's call it a synchronization variable, is one way to achieve both goals. The first instruction is commonly called *load-locked* or *load-linked* (LL). It loads the synchronization variable into a register. It may be followed by arbitrary instructions that manipulate the value in the register; i.e. the modify part of a read-modify-write. The last instruction of the sequence is the second special instruction, called a *store-conditional* (SC). It writes the register back to the memory location (the synchronization variable) *if and only if* no other processor has written *to that location* since this processor completed its LL. Thus, if the SC succeeds, it means that the LL-SC pair has read, perhaps modified in between, and written back the variable atomically. If the SC detects that an intervening write has occurred to the variable, it fails and does not write the value back (or generate any invalidations). This means that the atomic operation on the variable has failed and must be retried starting from the LL. Success or failure of the SC is indicated by the condition codes or a return value. How the LL and SC are actually implemented will be discussed later; for now we are concerned with their semantics and performance.

Using LL-SC to implement atomic operations, lock and unlock subroutines can be written as follows, where reg1 is the register into which the current value of the memory location is loaded, and reg2 holds the value to be stored in the memory location by this atomic exchange (reg2 could simply be 1 for a lock attempt, as in a test&set). Many processors may perform the LL at the same time, but only the first one that manages to put its store conditional on the bus will succeed in its SC. This processor will have succeeded in acquiring the lock, while the others will have failed and will have to retry the LL-SC.

```
lock:   ll   reg1, location     /* load-linked the location to reg1 */
        bnz  reg1, lock         /* if location was locked (nonzero), try again*/
        sc   location, reg2     /* store reg2 conditionally into location*/
        beqz  lock              /* if SC failed, start again*/
        ret                     /* return control to caller of lock */
```

and

```
unlock: st location, #0         /* write 0 to location */
        ret                     /* return control to caller */
```

If the location is 1 (nonzero) when a process does its load linked, it will load 1 into reg1 and will retry the lock starting from the LL without even attempt the store conditional.

It is worth noting that the LL itself is not a lock, and the SC itself is not an unlock. For one thing, the completion of the LL itself does not guarantee obtaining exclusive access; in fact LL and SC are used together to implement a lock operation as shown above. For another, even a successful LL-SC pair does not guarantee that the instructions between them (if any) are executed atomi-

cally with respect to those instructions on other processors, so in fact those instructions do not constitute a critical section. All that a successful LL-SC guarantees is that no conflicting writes *to the synchronization variable* accessed by the LL and SC themselves intervened between the LL and SC. In fact, since the instructions between the LL and SC are executed but should not be visible if the SC fails, it is important that they do not modify any important state. Typically, they only manipulate the register holding the synchronization variable—for example to perform the op part of a fetch&op—and do not modify any other program variables (modification of the register is okay since the register will be re-loaded anyway by the LL in the next attempt). Microprocessor vendors that support LL-SC explicitly encourage software writers to follow this guideline, and in fact often provide guidelines on what instructions are possible to insert with guarantee of correctness given their implementations of LL-SC. The number of instructions between the LL and SC should also be kept small to reduce the probability of the SC failing. On the other hand, the LL and SC can be used directly to implement certain useful operations on shared data structures. For example, if the desired function is a shared counter, it makes much more sense to implement it as the natural sequence (LL, register op, SC, test) than to build a lock and unlock around the counter update.

Unlike the simple test&set, the spin-lock built with LL-SC does not generate invalidations if either the load-linked indicates that the lock is currently held or if the SC fails. However, when the lock is released, the processors spinning in a tight loop of load-locked operations will miss on the location and rush out to the bus with read transactions. After this, only a single invalidation will be generated for a given lock acquisition, by the processor whose SC succeeds, but this will again invalidate all caches. Traffic is reduced greatly from even the test-and-test&set case, down from $O(p^2)$ to $O(p)$ per lock acquisition, but still scales quickly with the number of processors. Since spinning on a locked location is done through reads (load-locked operations) there is no analog of a test-and-test&set to further improve its performance. However, backoff can be used between the LL and SC to further reduce bursty traffic.

The simple LL-SC lock is also low in latency and storage, but it is not a fair lock and it does not reduce traffic to a minimum. More advanced lock algorithms can be used that both provide fairness and reduce traffic. They can be built using both atomic read-modify-write instructions or LL-SC, though of course the traffic advantages are different in the two cases. Let us consider two of these algorithms.

Advanced Lock Algorithms

Especially when using a test&set to implement locks, it is desirable to have only one process attempt to obtain the lock when it is released (rather than have them all rush out to do a test&set and issue invalidations as in all the above cases). It is even more desirable to have only one process even incur a read miss when a lock is released. The ticket lock accomplishes the first purpose, while the array-based lock accomplishes both goals but at a little cost in space. Both locks are fair, and grant the lock to processors in FIFO order.

Ticket Lock. The ticket lock operates just like the ticket system in the sandwich line at a grocery store, or in the teller line at a bank. Every process wanting to acquire the lock takes a number, and busy-waits on a global *now-serving* number—like the number on the LED display that we watch intently in the sandwich line—until this *now-serving* number equals the number it obtained. To release the lock, a process simply increments the *now-serving* number. The atomic primitive needed is a fetch&increment, which a process uses to obtain its ticket number from a shared counter. It may be implemented as an atomic instruction or using LL-SC. No test&set is needed

to actually obtain the lock upon a release, since only the unique process that has its ticket number equal *now-serving* attempts to enter the critical section when it sees the release. Thus, the atomic primitive is used when a process first reaches the lock operation, not in response to a release. The acquire method is the fetch&increment, the waiting algorithm is busy-waiting for *now-serving* to equal the ticket number, and the release method is to increment *now-serving*. This lock has uncontended overhead about equal to the test-and-test&set lock, but generates much less traffic. Although every process does a fetch and increment when it first arrives at the lock (presumably not at the same time), the simultaneous test&set attempts upon a release of the lock are eliminated, which tend to be a lot more heavily contended. The ticket lock also requires constant and small storage, and is fair since processes obtain the lock in the order of their fetch&increment operations. However, like the simple LL-SC lock it still has a traffic problem. The reason is that all processes spin on the same variable (now-serving). When that variable is written at a release, all processors' cached copies are invalidated and they all incur a read miss. (The simple LL-SC lock was somewhat worse in this respect, since in that case another invalidation and set of read misses occurred when a processor succeeded in its SC.) One way to reduce this bursty traffic is to introduce a form of backoff. We do not want to use exponential backoff because we do not want all processors to be backing off when the lock is released so none tries to acquire it for a while. A promising technique is to have each processor backoff from trying to read the now-serving counter by an amount proportional to when it expects its turn to actually come; i.e. an amount proportional to the difference in its ticket number and the *now-serving* counter it last read. Alternatively, the array-based lock eliminates this extra read traffic upon a release completely, by having every process spin on a distinct location.

Array-based Lock. The idea here is to use a fetch&increment to obtain not a value but a unique location to busy-wait on. If there are p processes that might possibly compete for a lock, then the lock contains an array of p locations that processes can spin on, ideally each on a separate memory block to avoid false-sharing. The acquire method then uses a fetch&increment operation to obtain the next available location in this array (with wraparound) to spin on, the waiting method spins on this location, and the release method writes a value denoting "unlocked" to the next location in the array after the one that the releasing processor was itself spinning on. Only the processor that was spinning on that location has its cache block invalidated, and its consequent read miss tells it that it has obtained the lock. As in the ticket lock, no test&set is needed after the miss since only one process is notified when the lock is released. This lock is clearly also FIFO and hence fair. It's uncontended latency is likely to be similar to that of the test-and-test&set lock (a fetch&increment followed by a read of the assigned array location), and it is more scalable than the ticket lock since only one process incurs the read miss. It's only drawback for a bus-based machine is that it uses $O(p)$ space rather than $O(1)$, but with both p and the proportionality constant being small this is usually not a very significant drawback. It has a potential drawback for distributed memory machines, but we shall discuss this and lock algorithms that overcome this drawback in Chapter 7.

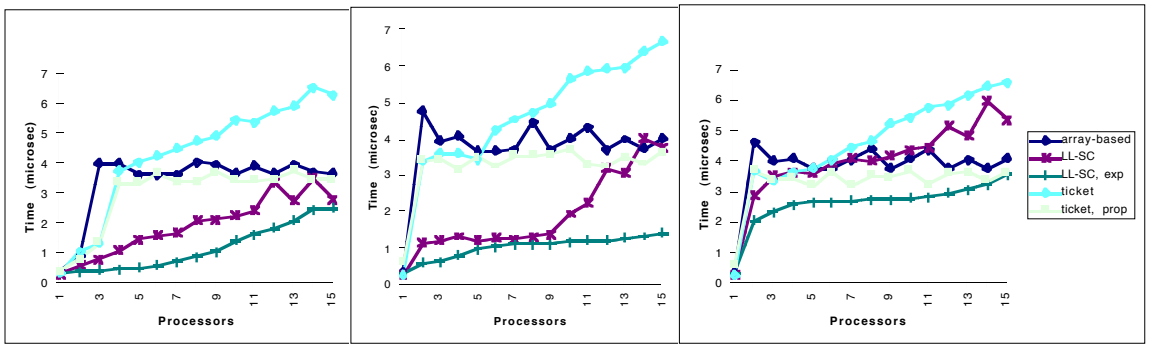
Performance

Let us briefly examine the performance of the different locks on the SGI Challenge, as shown in Figure 5-30. All locks are implemented using LL-SC, since the Challenge provides only these and not atomic instructions. The test&set locks are implemented by simulating a test&set using LL-SC, just as they were in Figure 5-29, and are shown as leaping off the graph for reference[1]. In particular, every time an SC fails a write is performed to another variable on the same cache block, causing invalidations as a test&set would. Results are shown for a somewhat more param-

eterized version of the earlier code for test&set locks, in which a process is allowed to insert a delay between its release of the lock and its next attempt to acquire it. That is, the code is a loop over the following body:

```
lock(L); critical_section(c); unlock(L); delay(d);
```

Let us consider three cases—(i) c=0, d=0, (ii) c=3.64 µs, d=0, and (iii) c=3.64 µs, d=1.29 µs—called *null*, *critical-section*, and *delay*, respectively. The delays c and d are inserted in the code as round numbers of processor cycles, which translates to these microsecond numbers. Recall that in all cases `c` and `d` (multiplied by the number of lock acquisitions by each processor) are subtracted out of the total time, which is supposed to measure the total time taken for a certain number of lock acquisitions and releases only (see also Exercise 5.6).



(a) null (c=0, d=0) (b) critical-section (c=3.64 µs, d=0) (c) delay (c=3.64 µs, d=1.29 µs)

Figure 5-30 Performance of locks on the SGI Challenge, for three different scenarios.

Consider the null critical section case. The first observation, comparing with Figure 5-29, is that all the subsequent locks we have discussed are indeed better than the test&set locks as expected. The second observation is that the simple LL-SC locks actually perform better than the more sophisticated ticket lock and array-based lock. For these locks, that don't encounter so much contention as the test&set lock, with reasonably high-bandwidth busses the performance of a lock is largely determined by the number of bus transactions between a release and a successful acquire. The reason that the LL-SC locks perform so well, particularly at lower processor counts, is that they are not fair, and the unfairness is exploited by architectural interactions! In particular, when a processor that does a write to release a lock follows it immediately with the read (LL) for its next acquire, it's read and SC are likely to succeed in its cache before another processor can read the block across the bus. (The bias on the Challenge is actually more severe, since the releasing processor can satisfy its next read from its write buffer even before the corresponding read-exclusive gets out on the bus.) Lock transfer is very quick, and performance is good. As the number of processors increases, the likelihood of self-transfers decreases and bus traffic due to invalidations and read misses increases, so the time per lock transfer increases. Exponential backoff helps reduce the burstiness of traffic and hence slows the rate of scaling.

1. This method of simulating test&set with LL-SC may lead to somewhat worse performance than a true test&set primitive, but it conveys the trend.

At the other extreme (c=3.64, d=1.29), we see the LL-SC lock doing not quite so well, even at low processor counts. This is because a processor waits after its release before trying to acquire the lock again, making it much more likely that some other waiting processor will acquire the lock before it. Self-transfers are unlikely, so lock transfers are slower even at two processors. It is interesting that performance is particularly worse for the backoff case at small processor counts when the delay d between unlock and lock is non-zero. This is because with only a few processors, it is quite likely that while a processor that just released the lock is waiting for d to expire before doing its next acquire, the other processors are in a backoff period and not even trying to acquire the lock. Backoff must be used carefully for it to be successful.

Consider the other locks. These are fair, so every lock transfer is to a different processor and involves bus transactions in the critical path of the transfer. Hence they all start off with a jump to about 3 bus transactions in the critical path per lock transfer even when two processors are used. Actual differences in time are due to what exactly the bus transactions are and how much of their latency can be hidden from the processor. The ticket lock without backoff scales relatively poorly: With all processors trying to read the now-serving counter, the expected number of bus transactions between the release and the read by the correct processor is $p/2$, leading to the observed linear degradation in lock transfer critical path. With successful proportional backoff, it is likely that the correct processor will be the one to issue the read first after a release, so the time per transfer does not scale with p. The array-based lock also has a similar property, since only the correct processor issues a read, so its performance also does not degrade with more processors.

The results illustrate the importance of architectural interactions in determining the performance of locks, and that simple LL-SC locks perform quite well on busses that have high enough bandwidth (and realistic numbers of processors for busses). Performance for the unfair LL-SC lock scales to become as bad as or a little worse than for the more sophisticated locks beyond 16 processors, due to the higher traffic, but not by much because bus bandwidth is quite high. When exponential backoff is used to reduce traffic, the simple LL-SC lock delivers the best average lock transfer time in all cases. The results also illustrate the difficulty and the importance of sound experimental methodology in evaluating synchronization algorithms. Null critical sections display some interesting effects, but meaningful comparison depend on what the synchronization patterns look like in practice in real applications. An experiment to use LL-SC but guarantee round-robin acquisition among processors (fairness) by using an additional variable showed performance very similar to that of the ticket lock, confirming that unfairness and self-transfers are indeed the reason for the better performance at low processor counts.

Lock-free, Non-blocking, and Wait-free Synchronization

An additional set of performance concerns involving synchronization arise when we consider that the machine running our parallel program is used in a multiprogramming environment. Other processes run for periods of time or, even if we have the machine to ourselves, background daemons run periodically, processes take page faults, I/O interrupts occur, and the process scheduler makes scheduling decisions with limited information on the application requirements. These events can cause the rate at which processes make progress to vary considerably. One important question is how the program as a whole slows down when one process is slowed. With traditional locks the problem can be serious because if a process holding a lock stops or slows while in its critical section, all other processes may have to wait. This problem has received a good deal of attention in work on operating system schedulers and in some cases attempts are made to avoid preempting a process that is holding a lock. There is another line of research that takes the view

that lock-based operations are not very robust and should be avoided. If a process dies while holding a lock, other processes hang. It can be observed that many of the lock/unlock operations are used to support operations on a data structure or object that is shared by several processes, for example to update a shared counter or manipulate a shared queue. These higher level operation can be implemented directly using atomic primitives without actually using locks.

A shared data structure is *lock-free* if its operations do not require mutual exclusion over multiple instructions. If the operations on the data structure guarantee that some process will complete its operation in finite amount of time, even if other processes halt, the data structure is *non-blocking*. If the data structure operations can guantee that every (non-faulty) process will complete its operation in a finite amount of time, then the data structure is *wait-free*.[Her93]. There is a body of literature that investigates the theory and practice of such data structures, including requirements on the basic atomic primitives[Her88], general purpose techniques for translating sequential operations to non-blocking concurrent operations[Her93], specific useful lock-free data structures [Val95,MiSc96], operating system implementations [MaPu91,GrCh96] and proposals for architectural support[HeMo93]. The basic approach is to implement updates to a shared object by reading a portion of the object to make a copy, updating the copy, and then performing an operation to commit the change only if no conflicting updates have been made. As a simple example, consider a shared counter. The counter is read into a register, a value is added to the register copy and the result put in a second register, then a compare-and-swap is performed to update the shared counter only if its value is the same as the copy. For more sophisticated data structures a linked structure is used and typically the new element is linked into the shared list if the insert is still valid. These techniques serve to limit the window in which the shared data structure is in an inconsistent state, so they improve the robustness, although it can be difficult to make them efficient.

Having discussed the options for mutual exclusion on bus-based machines, let us move on to point-to-point and then barrier event synchronization.

5.6.4 Point-to-point Event Synchronization

Point-to-point synchronization within a parallel program is often implemented using busy-waiting on ordinary variables as flags. If we want to use blocking instead of busy-waiting, we can use semaphores just as they are used in concurrent programming and operating systems [TaW97].

Software Algorithms

Flags are control variables, typically used to communicate the occurrence of a synchronization event, rather than to transfer values. If two processes have a producer-consumer relationship on the shared variable *a*, then a flag can be used to manage the synchronization as shown below:

P1		P2	
a = f(x);	/* set a */	while (flag is 0) do nothing;	
flag = 1;		b = g(a);	/* use a */

If we know that the variable *a* is initialized to a certain value, say 0, which will be changed to a new value we are interested in by this production event, then we can use *a* itself as the synchronization flag, as follows:

```
            P1                                  P2
a = f(x);          /* set a */      while (a is 0) do nothing;
                                    b = g(a);          /* use a */
```

This eliminates the need for a separate flag variable, and saves the write to and read of that variable.

Hardware Support: Full-empty Bits

The last idea above has been extended in some research machines—although mostly machines with physically distributed memory—to provide hardware support for fine-grained producer-consumer synchronization. A bit, called a full-empty bit, is associated with every word in memory. This bit is set when the word is "full" with newly produced data (i.e. on a write), and unset when the word is "emptied" by a processor consuming those data (i.e. on a read). Word-level producer-consumer synchronization is then accomplished as follows. When the producer process wants to write the location it does so only if the full-empty bit is set to empty, and then leaves the bit set to full. The consumer reads the location only if the bit is full, and then sets it to empty. Hardware preserves the atomicity of the read or write with the manipulation of the full-empty bit. Given full-empty bits, our example above can be written without the spin loop as:

```
            P1                                  P2
a = f(x);          /* set a */      b = g(a);          /* use a */
```

Full-empty bits raise concerns about flexibility. For example, they do not lend themselves easily to single-producer multiple-consumer synchronization, or to the case where a producer updates a value multiple times before a consumer consumes it. Also, should all reads and writes use full-empty bits or only those that are compiled down to special instructions? The latter requires support in the language and compiler, but the former is too restrictive in imposing synchronization on all accesses to a location (for example, it does not allow asynchronous relaxation in iterative equation solvers, see Chapter 2). For these reasons and the hardware cost, full-empty bits have not found favor in most commercial machines.

Interrupts

Another important kind of event is the interrupt conveyed from an I/O device needing attention to a processor. In a uniprocessor machine there is no question where the interrupt should go, but in an SMP any processor can potentially take the interrupt. In addition, there are times when one processor may need to issue an interrupt to another. In early SMP designs special hardware was provided to monitor the priority of the process on each processor and deliver the I/O interrupt to the processor running at lowest priority. Such measures proved to be of small value and most modern machines use simple arbitration strategies. In addition, there is usually a memory mapped interrupt control region, so at kernel level any processor can interrupt any other by writing the interrupt information at the associated address.

5.6.5 Global (Barrier) Event Synchronization

Software Algorithms

Software algorithms for barriers are typically implemented using locks, shared counters and flags. Let us begin with a simple barrier among p processes, which is called a centralized barrier since it uses only a single lock, a single counter and a single flag.

Centralized Barrier

A shared counter maintains the number of processes that have arrived at the barrier, and is therefore incremented by every arriving process. These increments must be mutually exclusive. After incrementing the counter, the process checks to see if the counter equals p, i.e. if it is the last process to have arrived. If not, it busy waits on the flag associated with the barrier; if so, it writes the flag to release the waiting processes. A simple barrier algorithm may therefore look like:

```
struct bar_type {
     int counter;
     struct lock_type lock;
     int flag = 0;
} bar_name;

BARRIER (bar_name, p)
{
     LOCK(bar_name.lock);
     if (bar_name.counter == 0)
          bar_name.flag = 0;          /* reset flag if first to reach*/
     mycount = bar_name.counter++;  /* mycount is a private variable*/
     UNLOCK(bar_name.lock);
     if (mycount == p) {             /* last to arrive */
          bar_name.counter = 0;      /* reset counter for next barrier */
          bar_name.flag = 1;         /* release waiting processes */
     }
     else
          while (bar_name.flag == 0) {};/* busy wait for release */
}
```

Centralized Barrier with Sense Reversal

Can you see a problem with the above barrier? There is one. It occurs when the same barrier (barrier bar_name above) is used consecutively. That is, each processor executes the following code:

```
some computation...
BARRIER(bar1, p);
some more computation...
BARRIER(bar1, p);
```

The first process to enter the barrier the second time re-initializes the barrier counter, so that is not a problem. The problem is the flag. To exit the first barrier, processes spin on the flag until it is set to 1. Processes that see flag change to one will exit the barrier, perform the subsequent computation, and enter the barrier again. Suppose one processor P_x gets stuck while in the spin loop; for example, it gets swapped out by the operating system because it has been spinning too long. When it is swapped back in, it will continue to wait for the flag to change to 1. However, in the meantime other processes may have entered the second instance of the barrier, and the first of these will have reset the flag to 0. Now the flag will only get set to 1 again when all p processes have registered at the new instance of the barrier, which will never happen since P_x will never leave the spin loop and get to this barrier instance.

How can we solve this problem? What we need to do is prevent a process from entering a new instance of a barrier before all processes have exited the previous instance of the same barrier. One way is to use another counter to count the processes that leave the barrier, and not let a process enter a new barrier instance until this counter has turned to p for the previous instance. However, manipulating this counter incurs further latency and contention. A better solution is the following. The main reason for the problem in the previous case is that the flag is reset before all processes reach the next instance of the barrier. However, with the current setup we clearly cannot wait for all processes to reach the barrier before resetting the flag, since that is when we actually set the flag for the release. The solution is to have processes wait for the flag to obtain a different value in consecutive instances of the barrier, so for example processes may wait for the flag to turn to 1 in one instance and to turn to 0 in the next instance. A private variable is used per process to keep track of which value to wait for in the current barrier instance. Since by the semantics of a barrier a process cannot get more than one barrier ahead of another, we only need two values (0 and 1) that we toggle between each time, so we call this method sense-reversal. Now the flag need not be reset when the first process reaches the barrier; rather, the process stuck in the old barrier instance still waits for the flag to reach the old release value (sense), while processes that enter the new instance wait for the other (toggled) release value. The value of the flag is only changed when all processes have reached the barrier instance, so it will not change before processes stuck in the old instance see it. Here is the code for a simple barrier with sense-reversal.

```
BARRIER (bar_name, p)
{
     local_sense = !(local_sense);          /* toggle private sense variable */
     LOCK(bar_name.lock);
     mycount = bar_name.counter++;          /* mycount is a private variable*/
     if (bar_name.counter == p) {           /* last to arrive */
          UNLOCK(bar_name.lock);
          bar_name.counter = 0;             /* reset counter for next barrier */
          bar_name.flag = local_sense;      /* release waiting processes */
     }
     else {
          UNLOCK(bar_name.lock);
          while (bar_name.flag != local_sense) {};/* busy wait for release */
     }
}
```

The lock is not released immediately after the increment of the counter, but only after the condition is evaluated; the reason for this is left as an exercise (see Exercise 5.7) We now have a correct barrier that can be reused any number of times consecutively. The remaining issue is performance, which we examine next. (Note that the LOCK/UNLOCK protecting the increment of the counter can be replaced more efficiently by a simple LL-SC or atomic increment operation.

Performance Issues

The major performance goals for a barrier are similar to those for locks:

Low latency (small critical path length) Ignoring contention, we would like the number of operations in the chain of dependent operations needed for p processors to pass the barrier to be small.

Low traffic Since barriers are global operations, it is quite likely that many processors will try to execute a barrier at the same time. We would like the barrier algorithm to reduce the number of bus transactions and hence the possible contention.

Scalability Related to traffic, we would like to have a barrier that scales well to the number of processors we may want to support.

Low storage cost We would of course like to keep the storage cost low.

Fairness This is not much of an issue since all processors get released at about the same time, but we would like to ensure that the same processor does not always become the last one to exit the barrier, or to preserve FIFO ordering.

In a centralized barrier, each processor accesses the lock once, so the critical path length is at least p. Consider the bus traffic. To complete its operation, a centralized barrier involving p processors performs $2p$ bus transactions to get the lock and increment the counter, two bus transactions for the last processor to reset the counter and write the release flag, and another $p-1$ bus transactions to read the flag after it has been invalidated. Note that this is better than the traffic for

even a test-and-test&set lock to be acquired by p processes, because in that case each of the p releases causes an invalidation which results in $O(p)$ processes trying to perform the test&set again, thus resulting in $O(p^2)$ bus transactions. However, the contention resulting from these competing bus transactions can be substantial if many processors arrive at the barrier simultaneously, and barriers can be expensive.

Improving Barrier Algorithms for a Bus

Let us see if we can devise a better barrier for a bus. One part of the problem in the centralized barrier is that all processors contend for the same lock and flag variables. To address this, we can construct barriers that cause less processors to contend for the same variable. For example, processors can signal their arrival at the barrier through a software combining tree (see Chapter 3, Section 3.4.2). In a binary combining tree, only two processors notify each other of their arrival at each node of the tree, and only one of the two moves up to participate at the next higher level of the tree. Thus, only two processors access a given variable. In a network with multiple parallel paths, such as those found in larger-scale machines, a combining tree can perform much better than a centralized barrier since different pairs of processors can communicate with each other in different parts of the network in parallel. However, with a centralized interconnect like a bus, even though pairs of processors communicate through different variables they all generate transactions and hence contention on the same bus. Since a binary tree with p leaves has approximately $2p$ nodes, a combining tree requires a similar number of bus transactions to the centralized barrier. It also has higher latency since it requires $\log p$ steps to get from the leaves to the root of the tree, and in fact on the order of p serialized bus transactions. The advantage of a combining tree for a bus is that it does not use locks but rather simple read and write operations, which may compensate for its larger uncontended latency if the number of processors on the bus is large. However, the simple centralized barrier performs quite well on a bus, as shown in Figure 5-31. Some of the other, more scalable barriers shown in the figure for illustration will be discussed, along with tree barriers, in the context of scalable machines in Chapter 7.

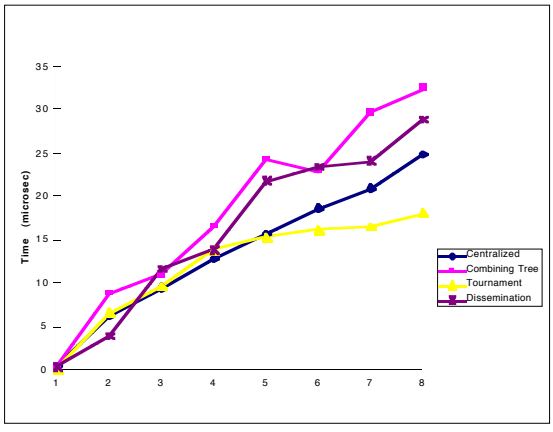


Figure 5-31 Performance of some barriers on the SGI Challenge.

Performance is measured as average time per barrier over a loop of many consecutive barriers. The higher critical path latency of the combining barrier hurts it on a bus, where it has no traffic and contention advantages.

Hardware Primitives

Since the centralized barrier uses locks and ordinary reads/writes, the hardware primitives needed depend on what lock algorithms are used. If a machine does not support atomic primitives well, combining tree barriers can be useful for bus-based machines as well.

A special bus primitive can be used to reduce the number of bus transactions in the centralized barrier. This optimization takes advantage of the fact that all processors issue a read miss for the same value of the flag when they are invalidated at the release. Instead of all processors issuing a separate read miss bus transaction, a processor can monitor the bus and abort its read miss before putting it on the bus if it sees the response to a read miss to the same location (issued by another processor that happened to get the bus first), and simply take the return value from the bus. In the best case, this "piggybacking" can reduce the number of read miss bus transactions from p to one.

Hardware Barriers

If a separate synchronization bus is provided, it can be used to support barriers in hardware too. This takes the traffic and contention off the main system bus, and can lead to higher-performance barriers. Conceptually, a wired-AND is enough. A processor sets its input to the gate high when it reaches the barrier, and waits till the output goes high before it can proceed. (In practice, reusing barriers requires that more than a single wire be used.) Such a separate hardware mechanism for barriers is particularly useful if the frequency of barriers is very high, as it may be in programs that are automatically parallelized by compilers at the inner loop level and need global synchronization after every innermost loop. However, it can be difficult to manage when not all processors on the machine participate in the barrier. For example, it is difficult to dynamically change the number of processors participating in the barrier, or to adapt the configuration when processes are migrated among processors by the operating system. Having multiple participating processes per processor also causes complications. Current bus-based multiprocessors therefore do not tend to provide special hardware support, but build barriers in software out of locks and shared variables.

5.6.6 Synchronization Summary

While some bus-based machines have provided full hardware support for synchronization operations such as locks and barriers, issues related to flexibility and doubts about the extent of the need for them has led to a movement toward providing simple atomic operations in hardware and synthesizing higher level synchronization operations from them in software libraries. The programmer can thus be unaware of these low-level atomic operations. The atomic operations can be implemented either as single instructions, or through speculative read-write instruction pairs like load-locked and store-conditional. The greater flexibility of the latter is making them increasingly popular. We have already seen some of the interplay between synchronization primitives, algorithms, and architectural details. This interplay will be much more pronounced when we discuss synchronization for scalable shared address space machines in the coming chapters. Theoretical research has identified the properties of different atomic exchange operations in terms of the time complexity of using them to implement synchronized access to variables. In particular, it is found that simple operations like test&set and fetch&op are not powerful enough to guarantee that the time taken by a processor to access a synchronized variable is independent of the number

of processors, while more sophisticated atomic operations like compare&swap and an memory-to-memory swap are [Her91].

5.7 Implications for Software

So far, we have looked at architectural issues and how architectural and protocol tradeoffs are affected by workload characteristics. Let us now come full circle and examine how the architectural characteristics of these small-scale machines influence parallel software. That is, instead of keeping the workload fixed and improving the machine or its protocols, we keep the machine fixed and examine how to improve parallel programs. Improving synchronization algorithms to reduce traffic and latency was an example of this, but let us look at the parallel programming process more generally, particularly the data locality and artifactual communication aspects of the orchestration step.

Of the techniques discussed in Chapter 3, those for load balance and inherent communication are the same here as in that general discussion. In addition, one general principle that is applicable across a wide range of computations is to try to assign computation such that as far as possible only one processor writes a given set of data, at least during a single computational phase. In many computations, processors read one large shared data structure and write another. For example, in Raytrace processors read a scene and write an image. There is a choice of whether to partition the computation so the processors write disjoint pieces of the destination structure and read-share the source structure, or so they read disjoint pieces of the source structure and write-share the destination. All other considerations being equal (such as load balance and programming complexity), it is usually advisable to avoid write-sharing in these situations. Write-sharing not only causes invalidations and hence cache misses and traffic, but if different processes write the same words it is very likely that the writes must be protected by synchronization such as locks, which are even more expensive.

The structure of communication is not much of a variable: With a single centralized memory, there is little incentive to use explicit large data transfers, so all communication is implicit through loads and stores which lead to the transfer of cache blocks. DMA can be used to copy large chunks of data from one area of memory to another faster than loads and stores, but this must be traded off against the overhead of invoking DMA and the possibility of using other latency hiding techniques instead. And with a zero-dimensional network topology (a bus) mapping is not an issue, other than to try to ensure that processes migrate from one processor to another as little as possible, and is invariably left to the operating system. The most interesting issues are related to temporal and spatial locality: to reduce the number of cache misses, and hence reduce both latency as well as traffic and contention on the shared bus.

With main memory being centralized, temporal locality is exploited in the processor caches. The specialization of the working set curve introduced in Chapter 3 is shown in Figure 5-32. All capacity-related traffic goes to the same local bus and centralized memory. The other three kinds of misses will occur and generate bus traffic even with an infinite cache. The major goal is to have working sets fit in the cache hierarchy, and the techniques are the same as those discussed in Chapter 3.

For spatial locality, a centralized memory makes data distribution and the granularity of allocation in main memory irrelevant (only interleaving data among memory banks to reduce conten-