The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors

THOMAS E. ANDERSON

Abstract—**Most shared-memory multiprocessor architectures provide hardware support for making mutually exclusive accesses to shared data structures. This support usually consists of instructions that atomically read and then write a single memory location. These atomic instructions are used to manipulate locks; when a processor is accessing a data structure, its lock is busy, and other processors needing access must wait.**

For small critical sections, spinning (or "busy-waiting") for a lock to be released is more efficient than relinquishing the processor to do other work. Unfortunately, spin-waiting can slow other processors by consuming communication bandwidth.

This paper examines the question: are there efficient algorithms for software spin-waiting given hardware support for atomic instructions, or are more complex kinds of hardware support needed for performance?

We consider the performance of a number of software spin-waiting algorithms. Arbitration for control of a lock is in many ways similar to arbitration for control of a network connecting a distributed system. We apply several of the static and dynamic arbitration methods originally developed for networks to spin locks.

We also propose a novel method for explicitly queueing spinning processors in software by assigning each a unique sequence number when it arrives at the lock. Control of the lock can then be passed to the next processor in line with minimal effect on other processors.

Finally, we examine the performance of several hardware solutions that reduce the cost of spin-waiting.

Index Terms—**Architecture, cache coherence, locking, multiprocessor, and performance.**

I. Introduction

MANY shared-memory multiprocessors have been designed in the past few years. The Sequent Symmetry [18], Alliant FX [20], and the BBN Butterfly [6] are among the more commercially successful; research vehicles include the DEC SRC Firefly [24], Illinois Cedar [10], IBM RP3 [22], and the Wisconsin Multicube [11].

In shared-memory multiprocessors, each processor can directly address memory that can also be addressed by all other processors. This uniform access requires some method for ensuring mutual exclusion: the logically atomic execution of operations (critical sections) on a shared data structure. Consistency of the data structure is guaranteed by serializing the operations done on it.

Manuscript received April 21, 1989; revised August 25, 1989. This work was supported by the National Science Foundation Grants CCR-8619663, CCR-8703049, and CCR-870010L, the Naval Ocean Systems, U.S. West Advanced Technologies, the Washington Technology Center, and Digital Equipment Corporation (the Systems Research Center and the External Research Program).

The author is with the Department of Computer Science and Engineering FR-35, University of Washington, Seattle, WA 98195.

IEEE Log Number 8931909.

Since pure software mutual exclusion is expensive [17], virtually all shared-memory multiprocessors provide some form of hardware support for making mutually exclusive accesses to shared data structures. This support usually consists of instructions that atomically read and then write a single memory location. All of the multiprocessors mentioned above support atomic instructions, although some, most notably the Multicube, also provide other mechanisms [12].

Atomic instructions serve two purposes. First, if the operations on the shared data are simple enough, they can be encapsulated into single atomic instructions. (Herlihy [14] discusses the computational power of atomic instructions for building parallel algorithms.) Mutual exclusion is directly guaranteed in hardware. If a number of processors simultaneously attempt to update the same location, each waits its turn without returning control back to software.

A *lock* is needed for critical sections that take more than one instruction. Atomic instructions are used to arbitrate between simultaneous attempts to acquire the lock, but if the lock is busy, waiting is done in software. When a lock is busy, the waiting process can either block, relinquishing the processor to do other work, or spin ("busy-wait") until the lock is released. Even though spin-waiting wastes processor cycles, it is useful in two situations: if the critical section is small, so that the expected wait is less than the cost of blocking and resuming the process, or if no other work is available.

This paper examines the question: are there efficient algorithms for software spin-waiting for busy locks given hardware support for atomic instructions, or are more complex kinds of hardware support needed for performance? (Jayasimha [15] and Agarwal and Cherian [2] have looked at the related issue of efficient spin-waiting for data dependencies.)

We show that the simple approaches to spin-waiting for busy locks have poor performance [3]. Spinning processors can slow processors doing useful work, including the one holding the lock, by consuming communication bandwidth. This performance penalty occurs if processors spin by continuously trying to acquire the lock; it also occurs for small critical sections if processors spin reading the (cached) lock value and try to acquire the lock only when it is released.

We consider the performance of several software spin-waiting alternatives. Although the analogy is not perfect, arbitration for control of a lock is in many ways similar to arbitration for permission to transmit on carrier-sense multiple-access (CSMA) networks. In both there is a cost when either zero or more than one waiting processor attempts to

1045-9219/90/0100-0006$01.00

acquire the resource. A number of arbitration mechanisms have been proposed for CSMA networks, including statically assigned slots (BRAM [9]), static delays (Aloha [8]), and dynamic backoff (Ethernet [19]); we discuss the performance of these methods when applied to spin-waiting.

We propose a novel method for explicitly queueing spinning processors. As processors arrive at a lock, they each acquire a unique sequence number specifying the order that they will execute the critical section. When the lock is released, control can be directly passed to the next processor in line with no further synchronization and minimal effect on other processors.

We also examine the performance of several hardware solutions. We propose an addition to snoopy cache protocols that exploits the semantics of spin lock requests to obtain better performance.

The remainder of this paper discusses these issues in more detail. Section II outlines the range of architectures that we will consider and how these systems commonly support mutual exclusion. Section III analyzes the performance problems of simple spin-waiting. Section IV presents new software alternatives; Section V considers hardware solutions. Section VI summarizes our conclusions.

II. Range of Multiprocessor Architectures Considered

While spinning processors can slow busy processors on any multiprocessor where spin-waiting consumes communication bandwidth, the precise performance of spin-waiting varies along several architectural dimensions: how processors are connected to memory, whether or not each processor has a hardware-managed coherent private cache, and if so, the coherence protocol. This paper will consider six types of architectures from within this design space:

- multistage interconnection network without coherent private caches
- multistage interconnection network with invalidation-based cache coherence using remote directories
- bus without coherent private caches
- bus with snoopy write-through invalidation-based cache coherence
- bus with snoopy write-back invalidation-based cache coherence
- bus with snoopy distributed-write cache coherence

(We assume for all of these that processors block when making a read request to memory.) While there are clearly some shared-memory architectures that are not represented in this list, these sample architectures expose most of the interesting issues in the performance of spin-waiting.

A. Common Hardware Support for Mutual Exclusions

Most architectures support mutual exclusion by providing instructions that atomically read, modify, and write memory. These atomic instructions are straightforward to implement. Conceptually, they require four services that might need interprocessor communication: the read and write, some method of arbitration between simultaneous requests, and some state that prevents further accesses from being granted while the instruction is being executed. Most multiprocessors are able to collapse these services into one or two bus or network transactions.

Multistage networks connect multiple processors to multiple memory modules. Memory requests are forwarded through a series of switches to the correct memory module. When a value is read from memory as part of an atomic instruction, any cached copies of the location (recorded in the directory associated with the memory module) must be invalidated and subsequent accesses to that memory module or at least to that location must be delayed (or refused and retired) while the new value is being computed. To minimize this delay, the computation can be done remotely by an ALU attached to each memory module. The Butterfly [6] and RP3 [22] implement this kind of remote "fetch and op."

In single bus multiprocessors, the bus can be used for arbitration between simultaneous atomic instructions. Before starting an atomic instruction, a processor acquires the bus and raises a line (*the atomic bus line*). This line is held while the new memory value is being computed to prevent further atomic requests from being started, but the bus can be released to allow other normal memory requests to proceed. Waiting atomic requests delay and only re-arbitrate for the bus when the line is dropped.

In systems that do not cache shared data, the bus transaction used to acquire the atomic bus line can be overlapped with the read request for the data. Similarly, with invalidation-based coherence [4], even if the lock value is cached, acquiring the atomic bus line can be overlapped with the signal to invalidate other cache copies. Note that normally the invalidation occurs even if the instruction does not change the value of the location, because it is done before the instruction executes.

Write-back invalidation-based coherence avoids an extra bus transaction to write the data. In this protocol, the new value is temporarily stored in the processor's cache. When another processor needs the value (for instance, as part of an atomic instruction), it gets the value at the same time it invalidates the first processor's copy.

With distributed-write write-back coherence, the initial read is usually not needed. Because copies in all caches are updated instead of invalidated when a processor changes a memory value, the cache block needed by an atomic instruction will often already be in the cache. In this case it would be wasteful of bus cycles to piggy-back the arbitration mechanism for the atomic bus line on top of the arbitration for the bus. For this reason, the Firefly, which implements distributed-write cache coherence, has a separate arbitration mechanism for its atomic bus line [24]. A bus cycle is still usually needed at the end of the atomic instruction to update copies in other caches.

III. The Performance of Simple Approaches to Spin-Waiting

Given atomic read-modify-write instructions, it is relatively straightforward to develop a correct spin lock. For instance, each processor can execute an atomic test-and-set instruction to acquire the lock; this instruction reads the old value of the lock and sets it to busy. If the read returns that the lock was free, the processor has the lock; if the lock was busy, the

processor must try again. The lock is released by (atomically) clearing the lock value.

It is more difficult to devise an efficient spin lock; this requires balancing several apparently opposing concerns. Performance when there is contention for the lock depends on minimizing the communication bandwidth used by spinning processors, since this can slow processors doing useful work; the delay between when a lock is released and when it is reacquired by a spinning processor must also be minimized, since no processor is executing the critical section during this time. This appears to pose a tradeoff: the more frequently a processor tries to acquire a lock, the faster it will be acquired, but the more other processors will be disrupted.

Latency, the time for a processor to acquire a lock in the absence of contention, is also important, for instance to applications with frequent locking, yet containing no bottleneck lock. A complex algorithm that reduces the cost of spin-waiting could degrade overall performance if it takes longer to acquire the lock when there is no contention.

It might seem that the behavior of multiprocessors when there is contention for a spin lock is not important. A highly parallel application will by definition have no lock with significant amounts of contention, since that would imply a sequential component. If an application has a lock that is a bottleneck, the best alternative would be to redesign the application's algorithms to eliminate the contention. In no case does it make sense to add processors to an application if they end up only spin-waiting.

There are, however, several situations where spin lock performance when there is contention is important. Poor contention performance may prevent an application with a heavily utilized lock from reaching its peak performance, because the average number of spin-waiting processors will become nontrivial as the lock approaches saturation. Furthermore, if processors arrive at a lock in a burst, queue lengths can be temporarily long, resulting in bad short-term performance, without the lock being a long-term bottleneck.

Alternately, it may not always be possible to tune a program to use the optimal number of processors. An operating system, for instance, has little control over the rate at which users make operating system calls. At high load, locks that are normally not a problem could become sources of contention. Similarly, on a multiprogrammed multiprocessor, a naive user can inadvertently ruin performance for all other users by combining a bottleneck critical section, lots of processors, and an inefficient spin lock.

In this section, we analyze the performance of two simple spin-waiting algorithms; combined measurement results are presented at the end of the section.

A. Spin on Test-and Set

The simplest spin-waiting algorithm is for each processor to repeatedly execute a test-and-set instruction until it succeeds at acquiring the lock. Table I lists sample code for this approach. Not surprisingly, the performance of spinning on test-and-set degrades badly as the number of spinning processors increases.

Two factors cause this degradation. First, in order to release the lock, the lock holder must contend with spinning processors for exclusive access to the lock location. Most multiprocessor architectures have no way of giving priority to the clear request of the lock holder, requiring it to wait behind test-and-sets of spinning processors, even though these cannot succeed until the lock is released.

TABLE I
SPIN ON TEST-AND-SET

```
Init      lock := CLEAR;
Lock      while (TestAndSet (lock) = BUSY);
Unlock    lock := CLEAR;
```

TABLE II
SPIN ON READ (TEST-AND-TEST-AND-SET)

```
Lock      while (lock = BUSY or TestAndSet (lock) = BUSY)
```

Furthermore, on architectures where test-and-set requests share the same bus or network as normal memory references, the requests of spinning processors can slow accesses to other locations by the lock holder or by other busy processors. On multistage network architectures, spin-waiting can cause a "hot-spot," delaying accesses to the memory module containing the lock location as well as to other modules [21]. On bus-structured multiprocessors, each test-and-set consumes at least one bus transaction, regardless of whether the lock value is changed; these can saturate the bus.

B. Spin on Read (Test-and-Test-and-Set)

Intuitively, coherent caches should be able to reduce the cost of spin-waiting. Segall and Rudolph [23] propose that spinning processors loop reading the value of the lock, and only when the lock is free, execute a test-and-set instruction; this eliminates the need to repeatedly test-and-set while the lock is held. They call this spinning on test-and-test-and-set; code for it is listed in Table II. (We assume that Boolean expressions are evaluated only if needed; the test-and-set is only executed if the lock is not busy.)

While the lock is busy, spinning is done in the cache without consuming bus or network cycles. When the lock is released, each copy is updated to the new value (distributed-write) or invalidated, causing a cache read miss that obtains the new value. The waiting processor sees the change in state and performs a test-and-set; if someone acquired the lock in the interim, the processor can resume spinning in its cache.

When the critical section is small, however, spinning on a read has almost as much effect on busy processors as spinning directly on a test-and-set instruction. The reason is that transient behavior can dominate; when the lock is released and reacquired by one of the waiting processors, it takes some time for the remaining processors to resume looping in their caches. During this time, most spinning processors have pending memory requests, delaying requests by busy processors during this interim. This behavior is most pronounced for systems with invalidation-based cache coherence, but it also occurs with distributed-write.

Suppose a number of processors are spinning reading the lock value in their caches. When the lock is released, these cache copies will all be invalidated; each processor will then incur a read miss to fetch the new value back into its cache. These read misses will be satisfied serially. Each processor to get the new value will then try to execute a test-and-set; these requests must compete for the bus or memory module with any remaining processors doing read misses.

The first processor to test-and-set will acquire the lock. Any processor who completed its read miss before this, however, will have seen the lock as free, proceed to do a test-and-set itself, fail, and go back to spinning reading the lock value. Unfortunately, each failing test-and-set instruction, because it is treated as a memory write, invalidates all cache copies of the lock, forcing any processors that had resumed spinning to miss again.

Thus, once the lock has been reacquired, some processors have passed the barrier and have a pending test-and-set request; the remainder have pending reads, trying to fill their cache after the original read miss. (The number of processors who have seen the lock as free will be worse on systems with multistage networks, because of the greater distance between the processors and memory.) Each read miss that is satisfied decreases the number of pending requests; that processor obtains a cache copy of the lock and resumes looping. Each test-and-set request that is satisfied decreases the number of processors waiting to test-and-set; however, it also invalidates all existing cache copies of the lock, forcing those processors that had been spinning in their cache to read miss again. After each test-and-set, every processor but the one that did the test-and-set must contend for memory. Eventually, the last spinning processor does a test-and-set, allowing every other spinning processor to do a read miss and then quiesce.

Before quiescence, each spinning processor spends most of its time contending for the bus or memory. After quiescence, spinning processors consume no communication resources. Thus, a normal memory request will be slowed dramatically if it occurs before quiescence and not at all if it occurs afterwards. For long critical sections, this initial slowdown is less significant, but for short critical sections, it dominates performance.

Our discussion so far has assumed random arbitration among memory requests. It might seem that spinning on a cache copy would perform well given fixed priority bus arbitration, as for instance on the Firefly. When a lock is released, the highest priority processor will acquire the lock. Even if it takes some time for the other processors to quiesce, the lock holder would not be slowed since it has higher priority than the other processors. However, if the lock is released before quiescence, a low priority processor with a pending test-and-set could acquire the lock before higher priority processors looping on read. The lock holder might then be delayed by these higher priority processors.

C. Reasons for the Poor Performance of Spin on Read

There are several factors that cause the performance of spinning on a memory read to be worse than expected.

- There is a separation between detecting that the lock has been released and attempting to acquire it with a test-and-set instruction. This separation allows more than one processor to notice that the lock has been released, pass by that test, and proceed to try a test-and-set. Ideally, if one processor could notice the change and acquire the lock before any other processor committed to doing a test-and-set, the performance would be better.
- Cache copies of the lock value are invalidated by a test-and-set instruction even if the value is not changed. If this were not the case, invalidations would occur only when the lock is released and then again when it is reacquired.
- Invalidation-based cache-coherence requires $O(P)$ bus or network cycles to broadcast a value to P waiting processors. This occurs despite the fact that, after an invalidation, they each request exactly the same data.

While a solution to any of these three problems by itself would result in better performance, any single solution would still require bus activity that grows linearly with the number of processors.

For example, distributed-write cache coherence eliminates invalidations; each processor directly receives all updates to the lock value. All reads can therefore be done locally; only test-and-sets still require bus traffic. The Sequent Balance [7] and the Silicon Graphics 4D-MP [5] both use a separate bus for test-and-set variables for just this reason; the bus implements distributed-write coherence to reduce bus traffic due to spin-waiting. Unfortunately, broadcasting updates makes the separation between the test and the test-and-set worse: all processors receive the updated lock value at the same time, and all therefore proceed to try the test-and-set. The result is that P test-and-sets must be performed before quiescence. It is unclear whether either the Balance or the 4D-MP has special hardware to avoid this problem.

D. Measurement Results

To demonstrate the performance of simple spin-waiting, we implemented both approaches on a Sequent Symmetry Model B shared-memory multiprocessor with 20 80386 (approximately 2 MIP) processors. The Symmetry has a shared bus and write-back invalidation-based cache coherence; unlike the Balance, test-and-set variables are handled on the same bus as normal memory references [18]. Acquiring and releasing a lock on the Symmetry normally takes 5.6 μs, less if the cache block containing the lock is initially private to the locking processor.

Fig. 1 is the principal performance comparison: the elapsed time for various number of processors to cooperatively execute a critical section one million times, for the two alternatives. Each processor loops: wait for the lock, do the critical section once, release the lock, and delay for a time randomly selected from a uniform distribution. The mean delay is equal to five times the size of the critical section. The wait in the loop eliminates any locality effect: each iteration, the lock and the shared data accessed by the critical section move between caches. The lock and shared data are placed so as to fall in separate cache blocks.

This benchmark simulates the performance of an application with a small central critical section. (Similar curves have been

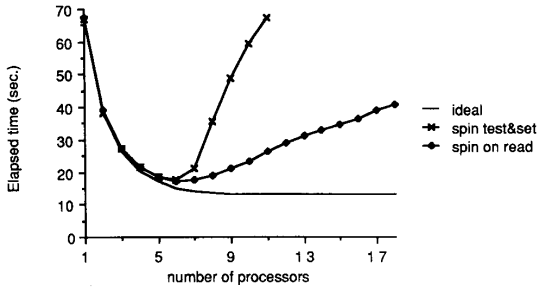


Fig. 1. Principal performance comparison: elapsed time (second) to execute benchmark (measured). Each processor loops one million/P times: acquire lock, do critical section, release lock, and compute.

measured using a fixed delay between lock accesses.) It also shows spin lock latency and performance with small and large amounts of contention. Ideally, performance initially improves as processors are added, due to increased parallelism, but as the critical section becomes a bottleneck, performance levels out. The ideal curve in Fig. 1 is the time the test would have taken, given free spin-waiting; this was determined by simulation from the time to execute the critical section and the mean delay between lock accesses.

Fig. 1 confirms our analysis. Performance degrades badly as processors spin on test-and-set; spinning on a read is better, but it still has disappointing performance. As the critical section becomes a bottleneck, the average number of spin-waiting processors increases, significantly slowing the processor executing the critical section. As a result, peak ideal performance is never reached. Performance with these alternatives is very sensitive to the exact number of processors given to an application; adding even a few processors beyond where the lock saturates worsens overall performance considerably.

This behavior can be degenerative [3]. Critical sections, since their purpose is to manipulate shared data structures, typically have higher memory access rates than noncritical sections. As a critical section becomes a bottleneck, the spinning processors slow the lock holder's execution, both in absolute terms and relative to noncritical sections, making it more of a bottleneck, resulting in more spinning processors.

As we noted, there is a difference in the effect on memory accesses before and after quiescence when processors spin on a read. This two-phase behavior allows us to measure the time to quiesce on the Symmetry. We construct a critical section whose behavior mirrors that of the bus, but in reverse. The critical section begins by delaying for some amount of time without using the bus at all, then proceeds to use the bus heavily before releasing the lock. If the initial delay is longer than the time to quiesce the spinning processors, then the critical section will run as fast on P processors as on one. If the heavy bus usage begins before quiescence, the critical section will run slower on P processors. We vary the length of the initial delay to find this performance knee; in practice, this knee was quite sharp.

Fig. 2 shows the results of this test. The time to quiesce grows steeply but linearly with the number of processors. As a result, even a few spinning processors can adversely impact the execution speed of a moderate-sized critical section.

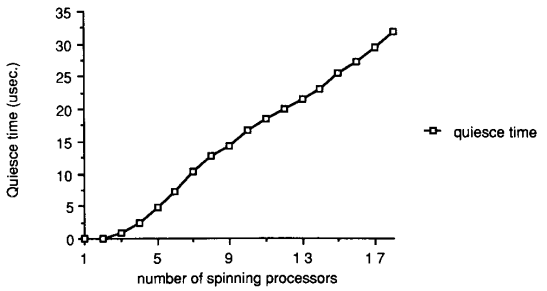


Fig. 2. Time to quiesce, spin on read (microseconds).

TABLE III
DELAY AFTER SPINNER NOTICES RELEASED LOCK

```
Lock        while (lock = BUSY or TestAndSet (Lock) = BUSY)
              begin
              while (lock = BUSY) ;
              Delay ();
              end;
```

TABLE IV
DELAY BETWEEN EACH REFERENCE

```
Lock        while (lock = BUSY or TestAndSet (lock) = BUSY)
              Delay ();
```

IV. New Software Alternatives

In this section, we first describe five software spin-waiting approaches, four based on CSMA network protocols, and one using explicit queueing, leaving until afterwards the presentation of their combined measurement results.

A. Delay Alternatives

We consider four ways of inserting delays into the spin-wait loop, defined by two dimensions: where the delay is inserted and whether the size of the delay is set statically or dynamically. A delay can be inserted after the lock has been released or alternatively after every separate access to the lock; code for these approaches is listed in Tables III and IV. Because processors first try to acquire the lock before delaying, lock latency is unaffected.

1) *Delay after Spinning Processor Notices Lock has been Released:* We can reduce the number of unsuccessful test-and-sets when spinning on a read by inserting a delay between when a processor reads that the lock is released and when it commits to trying the test-and-set. If some other processor acquires the lock during this delay, then the processor can resume spinning; if not, then the processor can try the test-and-set, with a greater likelihood that the lock will be acquired. In this way, the number of unsuccessful test-and-sets, and thus invalidations, can be reduced.

Each processor can be statically assigned a separate slot, or amount of time to delay, from 0 to $P - 1$ where P is the number of processors. The spinning processor with the smallest assigned delay checks the lock, sees that it is free, and acquires it. Processors with longer delays then time out, see that the lock is busy (enduring another cache miss), and resume spinning. By statically assigning delays, we can ensure

that at most one processor times out at any instant. Chlamtac *et al.* [9] propose a similar method to arbitrate access to a CSMA network. (Slots can also be used to implement priority access to the critical section, by assigning lower delays to higher priority processes.)

This algorithm performs well when there are many spinning processors. It is likely that some spinning processor will have a short delay; when the lock is released, some processor will quickly reacquire it. When there is only one spinning processor, however, it is unlikely to have a short delay, leaving the lock unacquired for a relatively long time, harming performance.

The number of slots can be varied to trade off performance between these two cases. When there are few spinning processors, using fewer slots improves performance by reducing the time to pass control of the lock to a waiting processor. When there are many spinning processors, using fewer slots worsens performance since more than one processor would simultaneously time out and attempt to test-and-set, requiring longer to quiesce.

By varying spinning behavior based on the number of waiting processors, we can have good performance in both situations. Such an algorithm has already been devised for CSMA networks: Ethernet's exponential backoff [19]. In a CSMA network, each processor can detect when the network is being used. When the network is unused, a processor can acquire the network by beginning to transmit, but if another processor simultaneously begins transmitting (''collides''), they both fail and must retry. The idea is for each processor to use the number of collisions it has experienced to estimate the number of spinning processors.

Initially, an arriving processor assumes that there are no other processors waiting to use the network and chooses a random delay with a small mean. Whenever it times out, tries to acquire the network and fails because some other processor timed out at the same time, then, assuming random arrivals, there are likely to be many more waiting processors that did not collide. Collisions are unlikely if the average delay is at least half the number of spinning processors. In Ethernet, then, each processor doubles its mean delay after each collision.

Analogously, a processor trying to acquire a spin lock could begin by assuming there were no other waiting processors. Each time it times out, sees the lock is still free, tries to test-and-set and fails, it has ''collided'' with at least one other processor. There are likely to be many other spinning processors it did not collide with, and thus it should double its mean delay, up to some limit.

Although Ethernet's backoff has been shown to have good performance [19], the performance of backoff for spin locks will not be the same as for networks. In a network, a collision aborts all processors; there is an equal cost to a collision among any number of processors as there is to an empty slot. By contrast, a test-and-set collision allows one processor to proceed, and the cost depends on how many processors collide. The more processors that try to acquire the lock and fail, the longer it will take them to quiesce, and the more that other processors, including the lock holder, will be slowed.

In designing a backoff scheme for spin locks, there are a number of details that affect performance. Our first implementation got most of these wrong.

- When a processor detects that the lock has been acquired, it should not increase (or decrease) its mean delay. The fact that some other processor had a shorter delay does not imply much about how many other spinning processors there are.
- There needs to be a maximum bound on the mean delay. Otherwise, if a processor backs off a number of times and then becomes the only waiting processor, it will take a long time for it to acquire the lock. This bound should be equal to the number of proessors, so that backoff has the same performance as statically assigned slots when there are many spinning processors.
- The initial delay of an arriving processor should be some fraction of its delay the last time at the lock. In a CSMA network, an arriving processor can efficiently reestimate the number of spinning proessors because collisions are not unduly costly. For spin locks, however, the learning curve can be expensive. There is no more reason to assume initially that there are no other spinning processors than that the number is related to past experience. For our measurements, we set the initial delay to be half the previous delay. Note that in Table III if the lock if free when the processor arrives, it will immediately acquire it; backoff is only used if the lock is initially busy.

While the justification for backoff assumes random arrivals at the lock, it performs well compared to using static slots even when this is not the case. If processors execute for a fixed amount of time between lock accesses, they will tend to self-schedule so that either there is no contention for the lock or there are always the same number of spinning processors. In the latter case, backoff would increase the delays until there were few collisions, and then the hysteresis would help maintain those delays.

Similarly, both backoff and static slots have performance problems when processors repeatedly arrive at a lock in a burst. The first time the lock is accessed, all processors choose a small delay, time out together, and take a long time to quiesce. Eventually the mean delays are increased enough to avoid collisions; this initial degradation is largely avoided the next iteration. Using static slots also avoids this initial performance degradation. However, since the number of waiting processors decreases as more acquire the lock, both alternatives are finally left with processors with inappropriately long delays, making it take longer to pass control of the lock.

Polling for the lock release is only practical for systems with per-processor coherent caches. On other systems, processors would consume communication bandwidth if they were to spin reading memory waiting for the lock to be released. Some multistage network multiprocessors with caches based on remote directories limit the number of outstanding copies of a location in order to limit the size of the directories [1]; if the number of spinning processors exceeds this number, spinning on a read degenerates to spinning across the network.

Even for multiprocessors with snoopy or complete directory invalidation-based caches, using exponential backoff or static

TABLE V
QUEUE USING ATOMIC READ-AND-INCREMENT

```
Init    flags[0] : = HAS_LOCK;
        flags[1..P-1] : = MUST_WAIT;
        queueLast : = 0;
Lock    myPlace : = ReadAndIncrement (queueLast);
        while (flags[myPlace mod P] = MUST_WAIT)
            ;
        flags[myPlace mod P] : = MUST_WAIT;
Unlock  flags[(myPlace + 1) mod P] : = HAS_LOCK;
```

slots solves only one of the problems with spinning on a memory read. Each spinning processor still requires at least two cache read misses per execution of the critical section, one when the lock is released and one when the lock is acquired. For sufficient numbers of spinning processors, this read miss activity can saturate the bus or network.

Exponential backoff after the lock is released does, however, provide scalable performance for multiprocessors with distributed-write cache coherence. In these systems, each test-and-set requires a single bus cycle to broadcast the new value, independent of the number of spinning processors. Exponential backoff, in turn, limits the number of unsuccessful test-and-sets.

2) *Delay Between Each Memory Reference:* An alternative approach to reducing the cost of spin-waiting is to insert a delay between each memory reference. This can be used on architectures without coherent caches or with invalidation-based coherence to limit the communication bandwidth consumed by spinning processors. In the code in Table IV, we check if the lock is free before trying to test-and-set since we assume that a test-and-set instruction consumes more bandwidth than a simple read.

The mean delay between each reference can be set statically or dynamically, analogous to the Aloha [8] and Ethernet network protocols. Most of the tradeoffs outlined above apply to these alternatives: more frequent polling improves performance when there are few spinning processors and worsens performance when there are many. Exponential backoff can be used to dynamically adapt to varying conditions.

Delaying between each reference poses special problems, however. For instance, the performance of backoff is bad when there is a single spinning processor for a moderate-sized critical section. The processor will continue to back off the delay as long as the lock is held. When the lock is released, the spinning processor will be in the midst of a long delay that must finish before it noties the change.

B. Queueing in Shared Memory

It might seem that shared memory could be used to store state to control the activity of spinning processors. This is less easy than it appears. For instance, a shared counter could be used to directly keep track of the number of spinning processors, instead of relying on a backoff algorithm to estimate that number. Given atomic increment and decrement instructions, the apparent cost of maintaining this state is the execution of two extra atomic instructions per critical section. However, each spinning processor must read this data to compute its delay; on systems without distributed-write coherence, this would consume as much bandwidth as directly polling the lock.

Another approach would be to maintain an explicit queue of spinning processors. Each arriving processor enqueues itself and then spins on a separate flag. When the processor finishes with the critical section, it dequeues itself and sets the flag of the next processor in the queue. This approach can reduce invalidations: if each processor's flag is kept in a separate cache block, then only one cache read miss is needed to notify the next processor. Maintaining queues, however, is expensive; the enqueue and dequeue operations must themselves be locked. (Even if there are atomic enqueue and dequeue instructions, as on the VAX, these operations are likely to be slow since they must modify more than one location.) The result is a much worse performance for small critical sections. For instance, it would not be reasonable to do this if the critical section itself was a queue operation.

We have developed a method of queueing spin-waiting processors that requires only a single atomic operation per execution of the critical section. Each arriving processor does an atomic read-and-increment to obtain a unique sequence number. When a processor finishes with the lock, it taps the processor with the next highest sequence number; that processor now owns the lock. Since processors are sequenced, no atomic read-modify-write instruction is needed to pass control of the lock. Table V lists the code for this approach ("myPlace" is a location private to each processor). Sequent [13] has independently devised a similar algorithm.

The best implementation varies somewhat among architectures. With distributed-write coherence, processors can all spin on a single counter. To release the lock, a processor simple writes its sequence number into the counter; each processor's cache is updated, directly notifying the next processor in line with a single bus transaction.

With invalidation-based coherence, each processor should wait on a flag in a separate cache block. Only two bus or network transactions (an invalidation and a read miss) are needed to signal the next processor. Similarly, on a multistage network without coherent caches, each flag should be placed in a separate memory module. Even though processors must poll to learn when it is their turn, there can be no more than P such polling requests outstanding at a time among $P \times \log P$ switches and P memory modules.

This approach is less valuable in a system with a bus but no cache coherence. Processors must still poll to find out if it is their turn; the bus can easily be swamped with this polling. To be effective, a delay can be inserted between each poll that depends on how close the processor is to the front of the queue and on how long it takes to execute the critical section. This indicates one way of using fewer than P separate memory locations in Table V: if a processor is farther from the front than the number of flags, it can poll (rarely) to find out when it is close enough to spin on its own flag.

If an architecture does not support an atomic read-and-increment instruction, this operation can itself be locked. Since the operation would take at most a few instructions, it would not normally become a bottleneck except when used with the most trivial of critical sections. When processors

arrive in a burst, however, there can be short-term contention for this lock. In this case, one of the delay alternatives from Section IV-A should be chosen to minimize bus traffic. The tradeoffs are slightly different here; a delay in passing control over the read-and-increment operation need not impact overall performance, provided some backlog of spinning processors have already obtained a number. Interestingly, the Symmetry supports an atomic increment but not an atomic read-and-increment instruction (the original value is not saved).

Because a spinning processor automatically gets control of the critical section when its bit is set, the time between when one processor finishes and the next processor starts executing the critical section is reduced. In some sense, this exploits parallelism: the spinning processor does the time-consuming work of the atomic operation before the lock is released, decreasing the amount of serial work required to pass control. Thus, throughput actually increases as a critical section becomes a bottleneck.

Unfortunately, queueing has some bad aspects. It increases lock latency. Each processor must increment a counter, check a location, zero that location, and set another location; in the other methods, when there is no contention, the first test-and-set acquires the lock. Thus, when there is contention, queueing is better; when there is no contention, backoff or simple spin-waiting is better.

While processor preemption can yield bad spin-locking performance [25], queueing makes this problem more severe. Normally, if a process holding a lock is preempted, every process spinning on that lock must wait for it to be rescheduled. Good performance requires lock holders to not be preempted. With queueing, however, preempting any spin-waiting process forces all behind it to wait if it reaches the front of the queue before being rescheduled. This can cause lock-step behavior if a small critical section is accessed frequently. When a process in line is preempted, other processes queue up behind it; when it is rescheduled, it uses the lock once, but may then have to wait when it reaccesses the lock for other processes that have been preempted in the interim. One way of avoiding this problem, if a process can be notified before it is preempted, is for it to remove itself from the queue by notifying the next process in line of that event (e.g., by setting a bit).

Another problem with queueing is that it makes it more difficult to wait for multiple events. As the number of processors increases, any centralized resource can become a bottleneck. One way of increasing throughput is to divide control over a resource among several critical sections, so that a spinning processor need access only one of the locks to get service. It is easy to see how delays could be used in this case; each waiting processor could randomly poll a server, and if busy, delay before polling another server. It is hard to see how queueing could be adapted, however, since a processor would only be able to wait in one queue at a time.

C. Measurement Results

We implemented the five software alternatives we have described on the Symmetry Model B with 20 processors. The static and dynamic delays varied from 0 to 15 μs; it takes approximately one microsecond on the Symmetry to execute the test-and-set instruction. Since the Symmetry does not support an atomic read-and-increment instruction, queueing uses an explicit lock (with backoff after each memory reference) to access the sequence number.

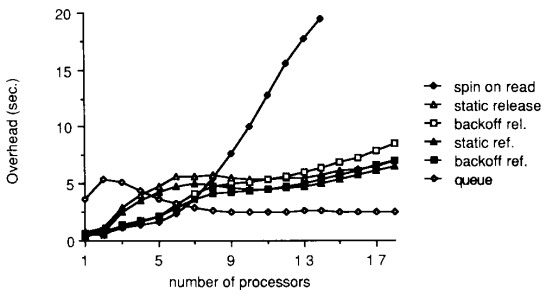


Fig. 3. Principal performance comparison: spin-waiting overhead (seconds) in executing the benchmark (measured). Each processor loops one million/P times: acquire lock, do critical section, release lock, and compute.

Fig. 3 is the principal performance comparison: spin-waiting overhead to execute the benchmark used for Fig. 1, as a function of the number of processors. To isolate the effect of spinning, we subtract from the elapsed time to execute the benchmark the "ideal" curve, the time the test would have taken given free spin-waiting. This leaves just the component due to spin-waiting overhead. We include spin on read for comparison.

Fig. 3 confirms our analysis. Although performance varies, all five methods described in this section have reasonable performance across the range of conditions. The one processor time reflects lock latency; queueing has high latency, while all other alternatives have low latency. Queueing would have better latency on systems with an atomic read-and-increment instruction. As the lock approaches saturation, the static delay alternatives have worse performance, because the delays are inappropriate for small numbers of spinning processors. The performance of the backoff alternatives remains close to the simple spin-waiting methods by adapting to the number of spinning processors.

When there are high numbers of spinning processors, backoff performs slightly worse than static delays; some collisions are necessary to maintain appropriate delays. Queueing performs best in this case by parallelizing the lock handoff. Across the entire spectrum, because of the Symmetry's invalidation-based coherence, delaying after each reference is slightly better than delaying after the lock is released.

To demonstrate the potential benefit of backoff relative to static delays, Fig. 4 compares spin-waiting overhead for the benchmark as a function of the number of static slots. As can be seen, small numbers of slots perform better when there are few spinning processors, while larger numbers of slots perform better when there are many. This tradeoff becomes harsher as the maximum number of spinning processors increases: 64 slots has much worse low load performance than direct spinning on read, yet that number of slots might be necessary to avoid poor high load performance in systems with large numbers of processors. Backoff avoids the tradeoff by performing well in both situations.

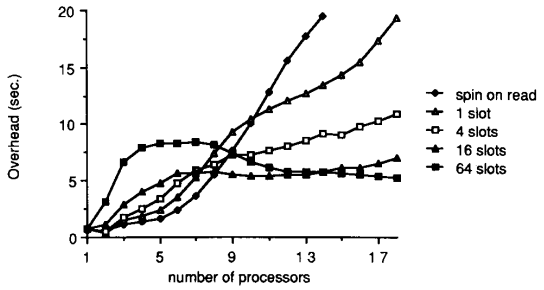


Fig. 4. Spin-waiting overhead (seconds) versus number of slots.

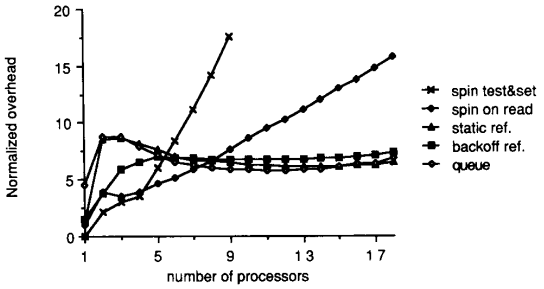


Fig. 5. Spin-waiting overhead in achieving barrier, normalized by the number of processors (microseconds per processor).

Fig. 5 shows spin-waiting overhead when processors arrive at a spin lock at the same time. A timestamp is taken before the processors are released from a barrier; each processor then acquires the lock and bumps a counter; and another timestamp is taken when the last processor acquires the lock. As in Fig. 3, we subtract the time to execute this test given free spin-waiting. This result is then normalized by the number of processors, to yield the average spin-waiting overhead per execution of the critical section. For clarity, we omit the curves for static and dynamic delays after the lock is released, as these are everywhere slightly worse than delaying between each reference.

The results in Fig. 5 are similar to that of Fig. 3. Queueing has bad latency in the one processor case. When two processors arrive together, using a static mean delay performs worst, but all alternatives perform badly because of the initial contention. As the number of processors increases, the behavior becomes similar to that of Fig. 3, except that queueing does not perform well with high numbers of processors because it uses backoff to arbitrate for the lock protecting its sequence number.

V. Hardware Solutions

In this section, we consider hardware changes to improve spin lock performance. As for the software alternatives, implementing solutions in hardware also poses tradeoffs. For example, the best cache coherence mechanism for spin locks may not be the best for normal memory references; some systems, such as the Balance and the 4D-MP, try to avoid this dilemma by using one bus with invalidation-based coherence for normal requests and a separate one with distributed-write coherence for test-and-set variables. Unfortunately, this duplication adds expense that is of little benefit to applications that do not spend significant amounts of time spin-waiting. Furthermore, if this separate bus is slower than the normal bus, as on the Balance, lock latency will suffer.

We consider the question of hardware solutions separately for multistage network and single bus multiprocessors.

A. Multistage Interconnection Network Multiprocessors

Combining networks, by providing parallel access to a single memory location [21], can improve the performance of spinning directly on test-and-set. Requests to the same location that arrive at the same network switch are combined and forwarded as a single request; the result is the same as if the two requests were made sequentially at the memory module. For example, two test-and-set requests would result in one request being forwarded and one request returning immediately with the value as set; no matter what the current value, only one will succeed if the two requests are made sequentially. Similarly, a test-and-set and a clear (to release the lock) would be combined to forward the set, while the test-and-set request returns having acquired the lock.

Assuming the cycle time of the combining network is the same as a normal network, combining has good performance for any number of spinning processors. When there is no or little contention, there is little combining, and performance is similar to normal spinning on test-and-set. As more processors spin-wait, combining reduces congestion due to duplicate test-and-sets, and since the request to release the lock is likely to be combined with a test-and-set at an earlier stage of the network, the time to pass control of the lock would be reduced. However, since the complexity of combining switches is likely to increase their latency, better performance might be obtained by a normal network with backoff or queueing.

Hardware queueing at the memory module, like software queueing, can eliminate polling across the network; it can also speed passing control of the lock. For this, processors would issue explicit "enter" and "exit" critical section instructions to the memory module, which would maintain queues of the processors waiting for each lock. When a processor's "enter" request returns, it has the lock; no polling across the network is necessary. With software queueing on a system with coherent caches, the processor releasing the lock notifies the next processor by writing its flag; an invalidation followed by a read miss is needed before the spinning processor can start executing the critical section. By specially handling critical section requests, hardware queueing eliminates one network round trip to pass control of the lock. Perhaps most importantly, lock latency is likely to be better with hardware than with software queueing; even though hardware queueing increases complexity at the memory module, it reduces the number of instructions needed to acquire the lock.

Goodman *et al.* [12], albeit for a different architecture, have proposed using caches to hold queue links. Their approach stores the name of the next processor in the queue directly in each processor's cache; when the lock is released, the next processor can be notified without going through the original memory module. To enhance flexibility, they have also proposed that control return to software after the processor is put on the queue for a critical section; the processor is then

separately notified by the hardware when it gets to the front of the queue.

B. Single Bus Multiprocessors

One obvious solution to reducing the number of invalidations caused by spinning on a read would be to invalidate only if the lock value changes. Before starting an atomic instruction, a processor would acquire the bus and raise a line to prevent other processors from accessing their potentially incorrect cache copies. These copies would then be invalidated only if the value changes. Unfortunately, this solves only one of the problems with spinning on a read. When the lock is released, there will still an invalidation, a cache miss by each spinning processor, followed by some number of failing test-and-sets; each of these consumes bus bandwidth. The time to quiesce is reduced but not eliminated. Unlike software queueing or backoff, performance degrades as more processors spin.

Rather, we note that more intelligent snooping of bus activity can reduce the cost of spin-waiting. We have already seen this in practice. If hardware keeps caches coherent, processors can spin on a cache copy instead of repeatedly reading from memory. Similarly, invalidation-based coherence can result in a cascade of read misses, which do not occur given write-broadcast coherence.

We will present two ways of improving performance by using information transmitted over the bus. One eliminates duplicate read requests; the other eliminates redundant test-and-sets. Simple spin-waiting is expensive because all spinning processors make bus requests to do the same thing, read or test-and-set, at the same time. This fact can be used to advantage.

Read broadcast [23], [16] can eliminate duplicate read miss requests. Each processor's cache controller monitors the bus; if a read occurs corresponding to an invalid block in its cache, it takes the data off the bus and sets the block to valid. Thus, whenever the cache copies of spinning processors are invalidated, the first read will fill all caches. Some spinning processors, however, will have already seen the cache as invalid and will be waiting at the bus to do the read; if a controller with a pending read observes the bus grant a read on the same location to some other processor, it should simply wait and take the data returning for that request. This eliminates the cascade of read misses when spinning on a read, without implementing full distributed-write coherence.

By specially handling test-and-set requests in the cache and bus controllers, we can eliminate the need for failing test-and-sets to use the bus. This way, processors can spin on test-and-set, acquiring the lock quickly when it is free, without consuming bus bandwidth when the lock is busy. Provided that specially handling test-and-sets does not increase the bus or cache cycle time, its performance would be better than software backoff or queueing. Fig. 3 shows that neither of these achieves ideal performance on the Symmetry. As the critical section becomes a bottleneck, backoff performance degrades slightly because of the overhead of computing random delays; the complexity of queueing similarly increases lock latency.

The idea is to not commit to doing the test-and-set over the bus so long as there is the possibility that it might fail (return that the lock is busy), and to return immediately without using the bus whenever the test-and-set would fail if it were the next to execute.

When a processor issues a test-and-set request, it first checks the cache. If the lock is not in the cache (because it was replaced or invalidated), a read miss occurs. Duplicate read misses can be eliminated using read broadcast. Once the lock value is in the cache, the test-and-set can return immediately if the lock is busy. If the lock is free, the controller can then try to acquire the bus to get the mutual exclusion needed by the atomic instruction.

While the controller is waiting for the bus, it must monitor the bus activity to determine if it should continue waiting. With distributed-write coherence, if some other processor acquires the bus to do a test-and-set, it will broadcast the new lock value, and all pending test-and-set requests can be aborted. If the lock value is invalidated, the processor must convert the test-and-set request back to a read request to see if the lock is now busy.

Typically, cache and bus controllers do not know the type of atomic instruction making a request, since the ALU is responsible for performing the logic of the instruction. This information is needed for the cache to be able to abort pending test-and-sets. When the cache returns control to the processor, the processor can proceed as if it had exclusive access, whether or not the test-and-set actually acquired the bus. In one case, it really has the exclusive access needed to acquire the lock; in the other, it can proceed because its actions will be consistent with some serial ordering of atomic instructions.

VI. Conclusions

In this paper, we have shown that simple methods of spin-waiting for mutually exclusive access to shared data structures degrade overall performance as the number of spinning processors increases. We have proposed and analyzed the performance of several hardware and software solutions to this problem.

For multiprocessors without special support for spin-waiting beyond implementing atomic instructions, we have shown that software queueing and a variant of Ethernet backoff have good performance even for large numbers of spinning processors. Because it is simpler, backoff has better performance when there is no contention for the lock; queueing, by parallelizing the lock handoff, performs best when there are waiting processors.

We have also shown that performance can be further improved by specially handling spin lock requests. On multiprocessors with multistage interconnection networks, explicit hardware queueing of spin-waiting processors, whether at the memory module or in each cache, can reduce the time to pass control of the lock to a waiting processor. On shared bus multiprocessors, failing test-and-sets can be handled with no bus traffic given more intelligent snooping. Whether real workloads will have significant enough amounts of spin-waiting to make such additional hardware support worthwhile remains an open question.

ACKNOWLEDGMENT

The author would like to thank M. Donner, J. Goodman, D. Keppel, E. Lazowska, D. Wagner, J. Zahorjan, and the referees for helpful discussions of the issues presented in this paper.

REFERENCES

[1] A. Agarwal, R. Simoni, J. Hennessey, and M. Horowitz, "An evaluation of directory schemes for cache coherence," in *Proc. 15th Intern. Symp. Comput. Architect.*, June 1988, pp. 280–289.

[2] A. Agarwal and M. Cherian, "Adaptive backoff synchronization techniques," in *Proc. 16th Intern. Symp. Comput. Architect.*, June 1989, pp. 396–406.

[3] T. E. Anderson, E. D. Lazowska, and H. M. Levy, "The performance implications of thread management alternatives for shared-memory multiprocessors," presented at 1989 ACM SIGMETRICS and Performance '89 Conf. Measurement Modeling Comput. Syst., pp. 49–60, May 1989.

[4] J. Archibald and J.-L. Baer, "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Trans. Comput. Syst.*, vol. 4, no. 4, Nov. 1986.

[5] F. Baskett, T. Jermoluk, and D. Solomon, "The 4D-MP graphics superworkstation: Computing + Graphics = 40 MIPS + 40 MFLOPS and 100000 lighted polygons per second, *IEEE Spring COMPCON*, pp. 468–471, 1988.

[6] BBN Laboratories, Butterfly parallel processor overview, 1985.

[7] B. Beck, B. Kasten, and S. Thakkar, "VLSI assist for a multiprocessor," in *Proc. Second Intern. Conf. Archit. Support for Programm. Languages and Operating Systems* (ASPLOS-II), Oct. 1987, pp. 10–20.

[8] R. Binder, N. Abrahamson, F. Kuo, A. Okinawa, and D. Wax, "Aloha packet broadcasting—A retrospective," *AFIPS Conf. Proc.*, 1975.

[9] I. Chlamtac, W. Franta, and D. Levin, "BRAM: The broadcast recognizing access method," *IEEE Trans. Commun.*, vol. 27, pp. 1183–1190, Aug. 1987.

[10] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh, "CEDAR—A large scale multiprocessor," in *Proc. 1983 Intern. Conf. Parallel Process.*, Aug. 1983, pp. 524–529.

[11] J. Goodman and P. Woest, "The Wisconsin Multicube: A new large-scale cache-coherent multiprocessor," in *Proc. 15th Annu. Intern. Symp. Comput. Architect.*, June 1988, pp. 442–431.

[12] J. Goodman, M. Vernon, and P. Woest, "A set of efficient synchronization primitives for a large-scale shared-memory multiprocessor," in *Proc. Third Intern. Conf. Architect. Support for Programm. Languages and Operating Syst. (ASPLOS-III)*, Apr. 1989.

[13] G. Graunke, Personal communication, 1988.

[14] M. Herlihy, "Impossibility and universality results for wait-free synchronization," in *Proc. Seventh Annual ACM Symp. Principles of Distributed Comput.*, 1988, pp. 276–291.

[15] D. N. Jayasimha, "Parallel access to synchronization variables," in *Proc. 1987 Intern. Conf. Parallel Process.*, Aug. 1987, pp. 97–100.

[16] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator, "Competitive snoopy caching," in *Proc. 27th Annu. Symp. Foundatons of Comput. Sci.*, Oct. 1986, pp. 244–254.

[17] L. Lamport, "A fast mutual exclusion algorithm," *ACM Trans. Comput. Syst.*, vol. 5, no. 1, 1987.

[18] T. Lovett and S. Thakkar, "The Symmetry multiprocessor system," in *Proc. 1988 Intern. Conf. Parallel Process.*, Aug. 1988, pp. 303–310.

[19] R. Metcalfe and D. Boggs, "Ethernet: Distributed packet switching for local computer networks," *Commun. ACM*, vol. 19, no. 7, pp. 395–404, July 1976.

[20] R. Perron and C. Mundie, "The architecture of the Alliant FX/8 computer," *IEEE COMPCON*, 1986.

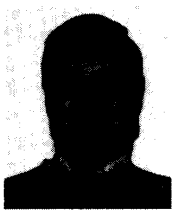
[21] G. Pfister and V. Norton, "Hot-spot contention and combining in multistage interconnection networks," *ACM Trans. Comput. Syst.*, vol. 3, no. 4, Oct. 1985.

[22] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weise, "The IBM research parallel processor prototype (RP3): Introduction and architecture," in *Proc. 1985 Intern. Conf. Parallel Process.*, Aug. 1985.

[23] Z. Segall and L. Rudolph, "Dynamic decentralized cache schemes for an MIMD parallel processor," in *Proc. 11th Annu. Intern. Symp. Comput. Architect.*, June 1984, pp. 340–347.

[24] C. Thacker, L. Stewart, and E. Satterthwaite, Jr., "Firefly: A multiprocessor workstation," *IEEE Trans. Comput.*, vol. 37, no. 8, pp. 909–920, Aug. 1988.

[25] J. Zahorjan, E. Lazowska, and D. Eager, "Spinning versus blocking in parallel systems with uncertainty," in *Proc. Intern. Seminar Performance of Distributed and Parallel Syst.*, North Holland, Dec. 1988.



Thomas E. Anderson received the A.B. degree in 1983 from Harvard University, Cambridge, MA.

Since 1987, he has pursued the doctoral degree in the Department of Computer Science, University of Washington, Seattle. His research interests include multiprocessor operating systems, architecture, and performance modeling.

Mr. Anderson won an IBM Graduate Fellowship in 1989.