URCS TR 342

Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors

John M. Mellor-Crummey* Michael L. Scott†

April 1990

Abstract

Busy-wait techniques are heavily used for mutual exclusion and barrier synchronization in shared-memory parallel programs. Unfortunately, typical implementations of busy-waiting tend to produce large amounts of memory and interconnect contention, introducing performance bottlenecks that become markedly more pronounced as applications scale. We argue in this paper that this problem is not fundamental, and that one can in fact construct busy-wait synchronization algorithms that induce no memory or interconnect contention. The key to these algorithms is for every processor to spin on a separate location in *local* memory, and for some other processor to terminate the spin with a single remote write operation at an appropriate time. Locations on which to spin may be local as a result of coherent caching, or by virtue of static allocation in the local portion of physically distributed shared memory.

We present a new scalable algorithm for spin locks that generates $O(1)$ remote references per lock acquisition, independent of the number of processors attempting to acquire the lock. Our algorithm provides reasonable latency in the absence of contention, requires only a constant amount of space per lock, and requires no hardware support other than a swap-with-memory instruction. We also present a new scalable barrier algorithm that generates $O(1)$ remote references per processor reaching the barrier, and observe that two previously-known barriers can likewise be cast in a form that spins only on local locations. None of these barrier algorithms requires hardware support beyond the usual atomicity of memory reads and writes.

We compare the performance of our scalable algorithms with other software approaches to busy-wait synchronization on both a Sequent Symmetry and a BBN Butterfly. Our principal conclusion is that *contention due to synchronization need not be a problem in large-scale shared-memory multiprocessors*. The existence of scalable algorithms greatly weakens the case for costly special-purpose hardware support for synchronization, and provides a case against so-called "dance hall" architectures, in which shared memory locations are equally far from all processors.

---

*Center for Research on Parallel Computation, Rice University, P.O. Box 1892, Houston, TX 77251-1892. Internet address: `johnmc@rice.edu`. Supported in part by the National Science Foundation under research grant CCR-8809615.

†Computer Science Department, University of Rochester, Rochester, NY 14627. Internet address: `scott@cs.rochester.edu`. Supported in part by the National Science Foundation under Institutional Infrastructure grant CDA-8822724.

1 Introduction

Techniques for efficiently coordinating parallel computation on MIMD, shared-memory multiprocessors are of growing interest and importance as the scale of parallel machines increases. On shared-memory machines, processors communicate by sharing data structures. To ensure the consistency of shared data structures, processors perform simple operations by using hardware-supported atomic primitives and coordinate complex operations by using synchronization constructs and conventions to protect against overlap of conflicting operations.

Two of the most widely used synchronization constructs are busy wait spin locks and barriers. Spin locks provide a means for achieving mutual exclusion (ensuring that only one processor can access a particular shared data structure at a time) and are a basic building block for synchronization constructs with richer semantics, such as semaphores and monitors. Spin locks are ubiquitously used in the implementation of parallel operating systems and in application programs. Barriers provide a means of ensuring that no processes advance beyond a particular point in a computation until all have arrived at that point. They are typically used to separate "phases" of an application program. A barrier might guarantee, for example, that all processes have finished initializing the values in a shared matrix before any processes use the values as input.

The performance of locks and barriers is a topic of great importance. Spin locks are generally employed to protect very small critical sections, and may be executed an enormous number of times in the course of a computation. Barriers, likewise, are frequently used between brief phases of data-parallel algorithms (*e.g.*, successive relaxation), and may be a major contributor to run time. Minimizing the cost of synchronization mechanisms is more complicated than simply shortening code paths. Performance may improve, for example, if one re-writes code to allow part of a `release_lock` operation to overlap execution of an `acquire_lock` operation on another processor, even if the total number of instructions executed on each processor increases. More important, by reducing the frequency with which synchronization variables are accessed by more than one processor, one may greatly reduce memory and interconnect contention, which slows down execution. On machines with coherent caches, it is particularly important to reduce the number of writes to shared locations, because these induce interconnect traffic for invalidations and subsequent cache-line reloads. Reduced contention speeds up not only the synchronization primitives themselves, but also any concurrent activity on the machine that requires bandwidth from conflicting portions of memory or the interconnect.

The execution overhead associated with synchronization in multiprocessor systems has been widely regarded as a serious performance problem [1, 5, 13, 11, 36, 38]. As part of a larger study, Agarwal and Cherian [1] investigated the impact of synchronization on overall program performance. Their simulations of benchmarks on a cache-coherent multiprocessor indicate that memory references due to synchronization cause cache line invalidations much more often than non-synchronization references. In simulations of the benchmarks on a 64-processor "dance hall" machine (in which each access to a shared variable traverses the processor-memory interconnection network), they observed that synchronization accounted for as much as 49% of total network traffic.

In response to performance concerns, the history of synchronization techniques has displayed a trend toward increasing hardware support. Early algorithms assumed only the ability to read and write individual memory locations atomically. They tended to be subtle, and costly in time and space, requiring both a large number of shared variables and a large number of operations to coordinate concurrent invocations of synchronization primitives [12, 23, 29]. Modern multiprocessors generally include more sophisticated atomic operations, permitting simpler and faster coordination strategies. Particularly common are various `fetch_and_`Φ operations [20], which atom-

ically read, modify, and write a memory location. `Fetch_and_Φ` operations include `test_and_set`, `fetch_and_store` (`swap`), `fetch_and_add`, and `compare_and_swap`.

More recently, there have been proposals for multistage networks that combine concurrent accesses to the same memory location [16, 30, 32], multistage networks that have special synchronization variables embedded in each stage of the network [19], and special-purpose cache hardware to maintain a queue of processors waiting for the same lock [13, 24, 28]. The principal purpose of these primitives is to reduce the impact of busy waiting. Before adopting them, it is worth considering the extent to which software techniques can achieve a similar result.

We contend that appropriate design of spin locks and barriers can lead to busy-wait synchronization mechanisms in which contention is extremely low. Specifically, by distributing data structures appropriately, we can ensure that each processor busy waits only on statically-allocated variables on which no other processor spins. On a machine with coherent caches, processors spin only on locations in their caches. On a machine in which shared memory is distributed (*e.g.*, the BBN Butterfly [8], the IBM RP3 [30], or a shared-memory hypercube [10]), processors spin only on locations in the local portion of the shared memory. The implication of this result is that busy waiting generates no memory or interconnect contention.

We discuss the implementation of spin locks in section 2, presenting both existing approaches and a new algorithm of our own design. (The correctness of our algorithm is discussed in an appendix.) In section 3 we turn to the issue of barrier synchronization, explaining how existing approaches can be adapted to eliminate spinning on remote locations, and introducing a new design that achieves both a short critical path and the theoretical minimum total number of remote references. We present performance results in section 4 for a variety of spinlock and barrier implementations, and discuss the implications of these results for software and hardware designers. Our conclusions are summarized in section 5.

2 Spin Locks

In this section we describe a series of four implementations for a mutual-exclusion spin lock. The first three are optimized versions of locks appearing in the literature. The fourth is a lock of our own design. Each lock can be seen as an attempt to eliminate some deficiency in the previous design. We describe each lock using pseudo-code and prose; hand-optimized assembly language versions can be retrieved via anonymous ftp from titan.rice.edu (/public/scalable_synch).

Our pseudo-code notation is meant to be more-or-less self explanatory. We have used line breaks to terminate statements, and indentation to indicate nesting in control constructs. The keyword `shared` indicates that a declared variable is to be shared by all processors. The declaration implies no particular physical location for the variable, but we will often specify locations in comments and/or accompanying text. The keywords `processor private` indicate that each processor is to have a separate, independent copy of a declared variable.

2.1 The Simple `Test_and_set` Lock

The simplest mutual exclusion lock, found in all operating system textbooks and widely used in practice, is the straightforward `test_and_set` loop:

```
type lock = (free, locked) := free

procedure acquire_lock (L : ^lock)
    repeat while test_and_set (L) = locked      // spin
        // test_and_set sets L^ = locked and returns old value

procedure release_lock (L : ^lock)
    lock^ := free
```

The principal shortcoming of the `test_and_set` lock is contention. Each waiting processor generates references as fast as it can to a common memory location. The resulting contention degrades the performance of the entire memory bank in which the lock resides. On distributed-memory machines it may also degrade the performance of the processor in whose memory the lock resides, by stealing memory cycles. In any event, remote memory references create contention on the interconnection network, and the arbitration of memory requests delays the `release_lock` operation, increasing the time required to pass the lock on to another processor.

A further weakness of the `test_and_set` lock is that each reference to shared memory employs a read-modify-write operation. Such operations are typically significantly more expensive than a simple read or write, particularly on a machine with coherent caches, since each such operation may cause a remote invalidation. This latter form of overhead can be reduced by replacing the `test_and_set` operation with a so-called test-and–`test_and_set` [33]:

```
procedure acquire_lock (L : ^lock)
    loop
        repeat while L^ = locked            // spin
        if test_and_set (L) = free
            return
```

The advantage of this formulation is that waiting processors generate only read requests, not read-modify-write requests, during the time that the lock is held. Once the lock becomes available, some fraction of the waiting processors will detect that the lock is free and will perform a `test_and_set` operation, exactly one of which will succeed, but each of which will still cause a remote invalidation on a coherent cache machine.

The total amount of remote traffic for a simple lock can be reduced further by introducing delay on each processor between consecutive probes of the lock. The simplest approach employs a constant delay; more elaborate schemes use some sort of backoff on unsuccessful probes. Anderson [4] reports the best performance with exponential backoff; our experiments confirm this result. Pseudo-code for a simple lock with exponential backoff appears in algorithm 1.

```
procedure acquire_lock (L : ^lock)
    delay : integer := 1
    while test_and_set (L) = locked     // returns old value
        pause (delay)                   // consume this many units of time
        delay := delay * 2
```

Algorithm 1: Simple `test_and_set` lock with exponential backoff.

2.2 The Ticket Lock

In a test-and–**test_and_set** lock, the number of read-modify-write operations is substantially less than for a simple **test_and_set** lock, but still potentially large. Specifically, it is possible for every waiting processor to perform a **test_and_set** operation every time the lock becomes available, even though only one can actually acquire the lock. The ticket lock reduces the number of **fetch_and_Φ** operations to one per lock acquisition. It also ensures FIFO service by granting the lock to processors in the same order in which they first requested it.[1] A ticket lock is therefore fair in a strong sense; it eliminates the possibility of starvation.

```
type lock = record
    next_ticket : unsigned integer := 0
    now_serving : unsigned integer := 0

procedure acquire_lock (L : ^lock)
    var my_ticket : unsigned integer
    my_ticket := fetch_and_increment (&L->next_ticket)
        // returns old value; arithmetic overflow is harmless
    repeat until L->now_serving = my_ticket     // spin

procedure release_lock (L : ^lock)
    L->now_serving := L->now_serving + 1
```

Though it probes with read operations only (and thus avoids the overhead of unnecessary invalidations in coherent cache machines), the ticket lock still incurs substantial contention through polling of a common location. As with the simple lock, this contention can be reduced by introducing delay on each processor between consecutive probes of the lock. In this case, however, exponential backoff is clearly a bad idea. Since processors acquire the lock in FIFO order, overshoot in backoff by the first processor in line will delay all others as well, causing them to back off even farther. Our experiments suggest that a good backoff scheme can be obtained by using information not available with a simple lock: namely, the number of processors already waiting for the lock. This number can be computed as the difference between a newly-obtained ticket and the value of the **now_serving** variable.

Delaying for an appropriate amount of time requires an estimate of how long it will take each processor to execute its critical section and pass the lock to its successor. If this time is known exactly, it is in principle possible to acquire the lock with only two probes, one to determine the number of processors already in line (if any), and another (if necessary) to verify that one's predecessor in line has finished with the lock. This sort of accuracy is not likely in practice, however, since critical sections do not in general take identical, constant amounts of time. Moreover, delaying based on the expected *average* time to hold the lock is risky: if the processors already in line average less than the expected amount the waiting processor will delay too long and slow the entire system. A more appropriate basis for delay is the *minimum* time that a processor can hold the lock. Pseudocode for a ticket lock with proportional backoff appears in algorithm 2.

[1]The ticket lock is an optimization of Lamport's bakery lock [22], which was designed for fault-tolerance rather than performance. Instead of inspecting a `now_serving` variable, processors using a bakery lock repeatedly examine the `my_ticket` variables of all their peers. The optimization has probably occurred to many other researchers. It appears, for example, in chapter 4 of [6].

```
type lock = record
    next_ticket : unsigned integer := 0
    now_serving : unsigned integer := 0

procedure acquire_lock (L : ^lock)
    var my_ticket : unsigned integer
    my_ticket := fetch_and_increment (&L->next_ticket)
        // returns old value; arithmetic overflow is harmless
    loop
        pause (L->now_serving - my_ticket)
            // consume this many units of time
            // on most machines, subtraction works correctly despite overflow
        if L->now_serving = my_ticket
            return

procedure release_lock (L : ^lock)
    L->now_serving := L->now_serving + 1
```

Algorithm 2: Ticket lock with proportional backoff.

2.3 Anderson's Array-Based Queueing Lock

Even using a ticket lock with proportional backoff, it is not possible to obtain a lock with an expected constant number of remote memory operations, due to the unpredictability of the length of critical sections. Anderson [4] has proposed a locking algorithm that achieves the constant bound on cache-coherent multiprocessors that support atomic `fetch_and_increment`. The trick is for each processor to use `fetch_and_increment` to obtain the address of a location on which to spin. Each processor spins on a different location, in a different cache line. Anderson's experiments indicate that his queuing lock outperforms a simple lock with exponential backoff when contention for the lock is high [4] (on the Sequent Symmetry, when more than six processors are waiting). Pseudo-code for Anderson's lock appears in algorithm 3.[2]

Anderson did not include the ticket lock with proportional backoff in his experiments. In qualitative terms, both locks guarantee FIFO ordering of requests. Both are written to use an atomic `fetch_and_increment` instruction. The ticket lock with proportional backoff is likely to require more remote references on a cache-coherent multiprocessor, but less on a multiprocessor without a coherent cache for shared variables. Anderson's lock requires space *per lock* linear in the number of processors, whereas the ticket lock requires only a small constant amount of space. We provide quantitative comparisons of the locks' performance in section 4.3.

[2] Anderson's original pseudo-code did not address the issue of overflow, which causes his algorithm to fail unless `numprocs` $= 2^k$. Our variant of his algorithm addresses this problem.

```
type lock = record
    slots : array [0..numprocs -1] of (has_lock, must_wait)
        := (has_lock, must_wait, must_wait, ..., must_wait)
        // each element of slots allocated in a different memory module
        // or cache line
    next_slot : integer := 0
processor private my_place : integer

procedure acquire_lock (L : ^lock)
    my_place := fetch_and_increment (&L->next_slot)
        // returns old value
    if my_place mod numprocs = 0
        atomic_add (&L->next_slot, -numprocs)
        // avoid problems with overflow; return value ignored
    my_place := my_place mod numprocs
    repeat while L->slots[myplace] = must_wait      // spin
    L->slots[myplace] := must_wait                  // init for next time

procedure release_lock (L : ^lock)
    L->slots[(my_place + 1) mod numprocs] := has_lock
```

Algorithm 3: Anderson's array-based queueing lock.

2.4 A New List-Based Queueing Lock

We have devised a new locking mechanism that

- guarantees FIFO ordering of lock acquisitions;
- spins on local memory only;[3]
- requires a small constant amount of space per lock; and
- works equally well on machines with and without coherent caches.

Our lock was inspired by the QOSB (Queue On Synch Bit) primitive proposed for the cache controllers of the Wisconsin Multicube [13], but is implemented entirely in software. It requires an atomic `fetch_and_store` (`swap`) instruction, and benefits from the availability of `compare_and_swap`. (`Fetch_and_store` exchanges a register with memory; `compare_and_swap` compares the contents of a memory location against a given value, and if equal exchanges the memory with a register.) Without `compare_and_swap` we lose the guarantee of FIFO ordering and introduce the theoretical possibility of starvation, though lock acquisitions are likely to remain very nearly FIFO in practice.

Pseudo-code for our lock appears in algorithm 4. Every processor using the lock allocates a record containing a queue link and a Boolean flag. Each processor employs one additional temporary variable during the `acquire_lock` operation. Processors holding or waiting for the lock are chained together by the links. Each processor spins on its own local flag. The lock itself contains a pointer to

[3]More precisely, on statically-allocated, processor-specific memory locations, which will be local on any machine in which shared memory is distributed or coherently cached.

the record for the processor at the tail of the queue, or a `nil` if the lock is not held. Each processor in the queue holds the address of the record for the processor behind it—the processor it should resume after acquiring and releasing the lock. `Compare_and_swap` enables a processor to determine whether it is the only processor in the queue, and if so remove itself correctly, as a single atomic action. The spin in `acquire_lock` waits for the lock to become free. The spin in `release_lock` compensates for the timing window between the `fetch_and_store` and the assignment to `predecessor->next` in `acquire_lock`. Both spins are local.

```
type qlink = record
    next : ^qlink
    locked : Boolean
type lock = ^qlink

processor private I : ^qlink
    // initialized to point to a queue link record
    // in the local portion of shared memory

procedure acquire_lock (L : ^lock)
    var predecessor : ^qlink
    I->next := nil
    predecessor := fetch_and_store (L, I)
    if predecessor != nil      // queue was non-empty
        I->locked := true
        predecessor->next := I
        repeat while I->locked             // spin

procedure release_lock (L : ^lock)
    if I->next = nil       // no known successor
        if compare_and_swap (L, I, nil)
            return
            // assuming compare_and_swap returns true iff it swapped
        repeat while I->next = nil          // spin
    I->next->locked := false
```

Algorithm 4: The MCS list-based queueing lock.

Figure 1, parts (a) through (e), illustrates a series of `acquire_lock` operations. In (a) the lock is free. In (b), processor 1 has acquired the lock. It is running (indicated by the 'R'), though its locked flag is irrelevant (indicated by putting the 'R' in parentheses). In (c), two more processors have entered the queue while the lock is still held by processor 1. They are blocked spinning on their locked flags (indicated by the 'B's). In (d), processor 1 has completed, and has changed the locked flag of processor 2 so that it is now running. In (e), processor 2 has completed, and has similarly unblocked processor 3. If no more processors enter the queue in the immediate future, the lock will return to the situation in (a) when processor 3 completes its critical section.

Alternative code for the `release_lock` operation, without `compare_and_swap`, appears in algorithm 5. Like the code in algorithm 4, it spins on static, processor-specific memory locations only, requires constant space per lock, and works well regardless of whether the machine provides coherent caches. Its disadvantages are extra complexity and the loss of strict FIFO ordering.

Parts (e) through (h) of figure 1 illustrate the subtleties of the alternative code for `release_lock`. In the original version of the lock, `compare_and_swap` ensures that updates to the tail of the queue

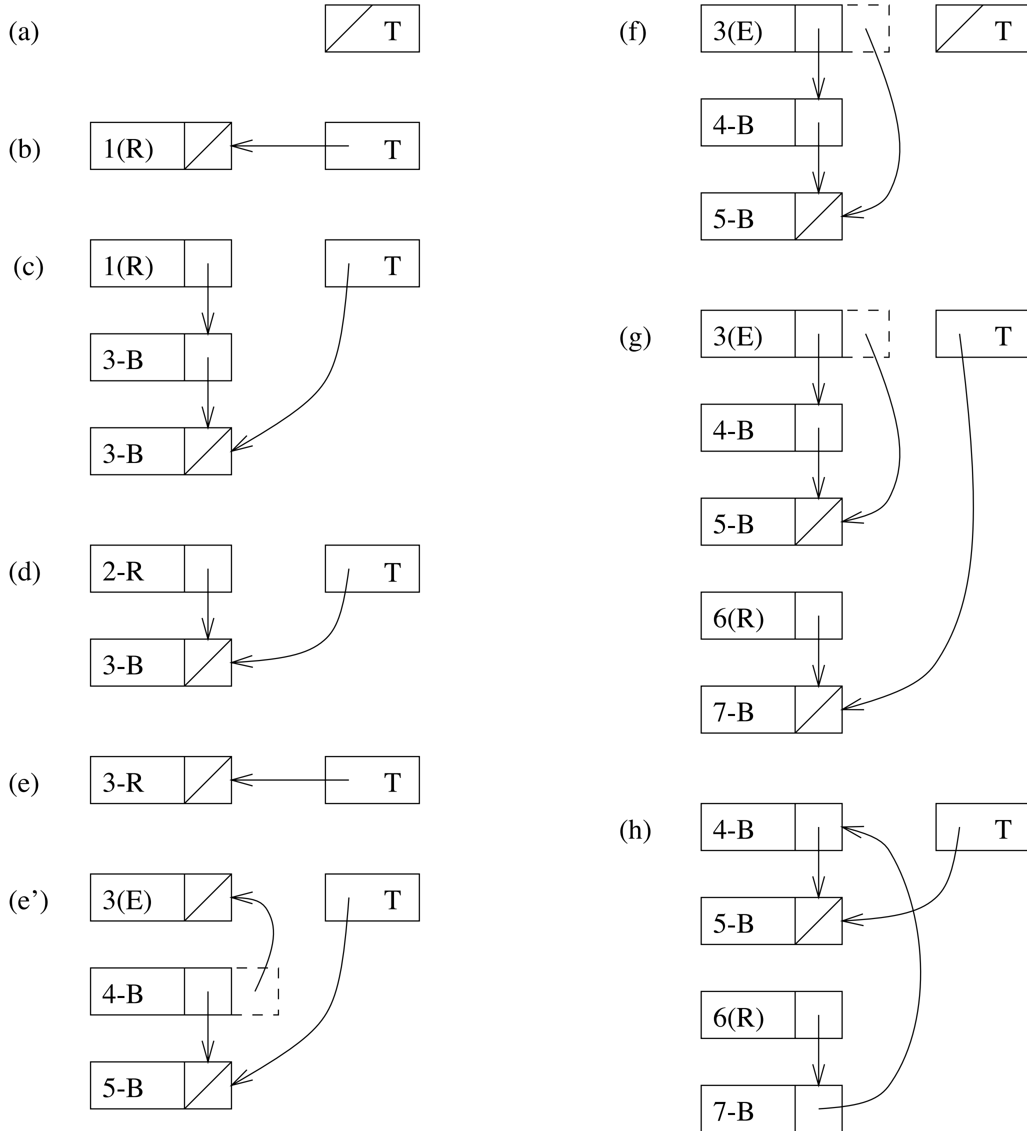

Figure 1: Pictorial version of MCS locking protocol in the presence of competition.

happen atomically. There are no processors waiting in line if and only if the tail pointer of the queue points to the processor releasing the lock. Inspecting the processor's next pointer is solely an optimization, to avoid unnecessary use of a comparatively expensive atomic instruction. Without `compare_and_swap`, inspection and update of the tail pointer cannot occur atomically. When processor 3 is ready to release its lock, it assumes that no other processor is in line if its next pointer is `nil`. In other words, it assumes that the queue looks the way it does in (e). (It could equally well make this assumption after inspecting the tail pointer and finding that it points to itself, but the next pointer is local and the tail pointer is probably remote.) This assumption may be incorrect because other processors may have linked themselves into the queue between processor 3's inspection and its subsequent update of the tail. The queue may actually be in the state shown in (e′), with one or more processors in line behind processor 3, the first of which has yet to update 3's next pointer. (The 'E' in parentheses on processor 3 indicates that it is exiting its critical section; the value of its locked flag is irrelevant.)

```
// I points to a queue link record in the local portion of shared memory

procedure release_lock (L : ^lock)
    var old_tail : ^qlink
    if I->next = nil        // no known successor
        old_tail := fetch_and_store (L, nil)
        if old_tail = I     // I really had no successor
            return
        // We have accidentally removed some processors from the queue.
        // We have to put them back.
        usurper := fetch_and_store (L, old_tail)
        repeat while I->next = nil          // spin
        if usurper != nil
            // somebody got into the queue ahead of our victims
            usurper->next := I->next
        else
            I->next->locked := false
    else
        I->next->locked := false
```

Algorithm 5: Code for `release_lock`, without `compare_and_swap`.

When new processors enter the queue during this timing window, the data structure temporarily takes on the form shown in (f). The return value of processor 3's `fetch_and_store` (shown in the extra dotted box) is the tail pointer for a list of processors that have accidentally been linked out of the queue. By waiting for its next pointer to become non-`nil`, processor 3 obtains a head pointer for this "victim" list. It can patch the victim processors back into the queue, but before it does so additional processors ("usurpers") may enter the queue with the first of them acquiring the lock, as shown in (g). Processor 3 puts the tail of the victim list back into the tail pointer of the queue with a `fetch_and_store`. If the return value of the `fetch_and_store` is `nil`, processor 3 unblocks its successor. Otherwise, as shown in (h), processor 3 inserts the victim list behind the usurpers by writing its next pointer (the head of the victim list) into the next pointer of the tail of the usurper list. In either case, the structure of the queue is restored.

3 Barriers

Barriers have also received a great deal of attention in the literature, and the many published algorithms differ significantly in notational conventions and architectural assumptions. We focus in this section on three distinct approaches to barrier synchronization, examining the more important variants of each, and casting all the code in similar form.[4] We note those places in which we have made substantive changes to code from cited sources.

The barriers in our first class are centralized in the sense that they spin on one or more shared global flags. The simplest algorithm generates significant amounts of interconnect traffic, which can be reduced by reversing the sense of variables in successive barriers (much the same as input and output arrays are commonly reversed in successive rounds of numerical computations), or by introducing backoff.

[4]Optimized C versions of our code are available via anonymous ftp from titan.rice.edu (/public/scalable_synch).

```
shared count : integer := 0
shared flag : enumeration {ENTER, EXIT} := ENTER

procedure naive_central_barrier
    repeat until flag = ENTER
    if fetch_and_increment (&count) = P - 1
        flag := EXIT
    else
        repeat until flag = EXIT
    if fetch_and_decrement (&count) = 1
        flag := ENTER
```

Algorithm 6: A naive, centralized barrier.

The second class of barrier algorithm uses a tree to distribute the data accessed during synchronization, thereby reducing contention. We describe the "software combining tree" approach of Yew, Tzeng, and Lawrie, and present a new tree-based approach of our own that performs fewer remote operations, spins on local memory only, and requires no atomic instructions other than read and write. We also describe tree-based "tournament barriers" developed by Hensgen, Finkel, and Manber and by Lubachevsky, which have properties similar to those of our tree-based algorithm, but which in their original form would scale well only on cache-coherent multiprocessors.

The third class of algorithm employs a symmetric pattern of pairwise synchronizations between processors to achieve global synchronization. We present the "dissemination barrier" developed by Hensgen, Finkel, and Manber, and observe that its data structures can be distributed in such a way that processors spin on local memory only.

3.1 Centralized Barriers

In a centralized implementation of barrier synchronization, each processor updates a small amount of shared barrier state to indicate its arrival, and then polls that state to determine when all of the processors have arrived. Once all of the processors have arrived, each processor is permitted to continue past the barrier. Like simple spin locks, centralized barriers are of obscure origin. Essentially equivalent algorithms have undoubtedly been invented by numerous individuals.

The Naive Barrier

A centralized P-processor barrier can be implemented in software using two globally shared variables: `count` and `flag` (see algorithm 6). Initially, `count` = 0 and `flag` = `ENTER`.

As processors arrive at the barrier, they spin until `flag` = `ENTER` (initially true), and then increment `count`.[5] The first $P-1$ processors to increment `count` spin until `flag` = `EXIT`. The last processor to arrive at the barrier sets `flag` to `EXIT`, releasing all busy-waiting processors. Each processor then decrements `count`. The first $P-1$ processors to do so leave the barrier immediately. The last sets `flag` to `ENTER`, re-initializing the barrier for its next use. The two states of `flag` prevent the overlap of multiple barriers in a sequence. If the first repeat loop were omitted, a

[5] On a machine that does not support some form of atomic increment and decrement, a spin lock would be needed to preserve the integrity of `count`.

```
shared count : integer := P
shared sense : Boolean := true
processor private local_sense : Boolean := true

procedure central_barrier
    local_sense := not local_sense   // each processor toggles its own sense
    if fetch_and_decrement (&count) = 1
        count := P
        sense := local_sense         // last processor toggles global sense
    else
        repeat until sense = local_sense
```

Algorithm 7: A sense-reversing centralized barrier.

processor leaving the current barrier could arrive at and pass through the next barrier before the last processor had left the current barrier. A pair of barrier algorithms proposed by Tang and Yew (the first algorithm appears in [35, Algorithm 3.1] and [36, p. 3]; the second algorithm appears in [36, Algorithm 3.1]) suffer from this type of flaw.

The Sense-Reversing Barrier

For P processors to achieve a barrier using algorithm 6, a minimum of $4P+1$ operations on shared variables are necessary.[6] We can reduce this number by reversing the sense of `flag` in consecutive barriers [17].[7] The improved code is shown in algorithm 7. Arriving processors simply decrement `count` and then wait until `sense` has a different value than it did in the previous barrier. The last arriving processor resets `count` and reverses `sense`. Consecutive barriers cannot interfere with each other, because all operations on `count` occur before `sense` is toggled to release the waiting processors. This sense-reversal technique reduces the minimum number of operations on shared variables to $2P+1$.

Lubachevsky [25] presents a similar barrier algorithm that uses two shared counters and a processor private two-state flag. Without a shared flag variable, two counters are needed to prevent conflicts between processors in consecutive barrier episodes. The private flag selects which counter to use; consecutive barriers use alternate counters. A similar algorithm can be found in library packages distributed by Sequent for the Symmetry. Arriving processors read the current value of a shared epoch number, update the shared counter, and spin until the epoch number changes. The last arriving processor reinitializes the counter and advances the epoch number.

The potential drawback of centralized barriers is the spinning that occurs on a single, shared location. Because processors do not in practice arrive at a barrier simultaneously, the number of busy-wait accesses will in general be far above the minimum.[8] On broadcast-based cache-coherent multiprocessors, these accesses may not be a problem. The shared flag or sense variable is replicated into the cache of every waiting processor so subsequent busy-wait accesses can be satisfied without

[6] P tests of `flag`, P increments of `count`, $P-1$ tests of `flag`, 2 assignments to `flag`, and P decrements of `count`.

[7] A similar technique appears in [3, p. 445], where it is credited to Isaac Dimitrikovsky.

[8] Commenting on Tang and Yew's barrier algorithm (algorithm 3.1 in [35]), Agarwal and Cherian [1] show that on a machine in which contention causes memory accesses to be aborted and retried, the expected number of memory accesses initiated by each processor to achieve a single barrier is linear in the number of processors participating, even if processors arrive at the barrier at approximately the same time.

any network traffic. This shared variable is written only when the barrier is achieved, causing a single broadcast invalidation of all cached copies.[9] All busy-waiting processors then acquire the new value of the variable and are able to proceed. On machines without coherent caches, however, or on machines with directory-based caches that limit the degree of replication, busy-wait references to a shared location may generate unacceptable levels of memory and interconnect contention.

Adaptive Backoff Schemes

To reduce the interconnection network traffic caused by busy waiting on a barrier flag, Agarwal and Cherian [1] investigated the utility of adaptive backoff schemes. They arranged for processors to delay between successive polling operations for geometrically-increasing amounts of time. Their results indicate that in many cases backoff can substantially reduce the amount of network traffic required to achieve a barrier. However, with this reduction in network traffic often comes an increase in latency at the barrier. Processors in the midst of a long delay do not immediately notice when all other processors have arrived. Their departure from the barrier is therefore delayed, which in turn delays their arrival at subsequent barriers.

Agarwal and Cherian also note that for systems with more than 256 processors, for a range of arrival intervals and delay ratios, backoff strategies are of limited utility for barriers that spin on a single flag [1]. In such large-scale systems, the number of network accesses per processor increases sharply as collisions in the interconnection network cause processors to repeat accesses. These observations imply that centralized barrier algorithms will not scale well to large numbers of processors, even using adaptive backoff strategies. Our experiments (see section 4.4) confirm this conclusion.

3.2 Distributed Barriers

The Software Combining Tree Barrier

To reduce hot-spot contention for synchronization variables, Yew, Tzeng, and Lawrie [38] have devised a data structure known as a software combining tree. Like hardware combining in a multistage interconnection network [16], a software combining tree serves to collect multiple references to the same shared variable into a single reference whose effect is the same as the combined effect of the individual references. A shared variable that is expected to be the target of multiple concurrent accesses is represented as a tree of variables, with each node in the tree assigned to a different memory module. Processors are divided into groups, with one group assigned to each leaf of the tree. Each processor updates the state in its leaf. If it discovers that it is the last processor in its group to do so, it continues up the tree, updating its parent to reflect the collective updates to the child. Proceeding in this fashion, late-coming processors eventually propagate updates to the root of the tree.

Combining trees are presented as a general technique that can be used for several purposes. At every level of the tree, atomic instructions are used to combine the arguments to write operations or split the results of read operations. In the context of this general framework, Tang and Yew [36] describe how software combining trees can be used to implement a barrier. Writes into one tree are used to determine that all processors have reached the barrier; reads out of a second are used to allow them to continue. Algorithm 8 shows an optimized version of the combining tree barrier. We

[9] This differs from the situation in simple spin locks, where a waiting processor can expect to suffer an invalidation for every contending processor that acquires the lock before it.

```
type node = record
    k : integer             // fan-in of this node
    count : integer         // initialized to k
    locksense : Boolean     // initially false
    parent : ^node          // pointer to parent node; nil if root

shared nodes : array [0..P-1] of node
    // each element of nodes allocated in a different memory module or cache line
processor private sense : Boolean  := true
processor private mynode : ^node    // my group's leaf in the combining tree

procedure combining_barrier ()
    combining_barrier_aux (mynode)      // join the barrier
    sense := not sense                  // for next barrier

procedure combining_barrier_aux (nodepointer: ^node)
    with nodepointer^ do
        if fetch_and_decrement (&count) = 1     // last one to reach this node
            if parent != nil
                combining_barrier_aux (parent)
            count := k                          // prepare for next barrier
            locksense := not locksense          // release waiting processors
        repeat until locksense = sense
```

Algorithm 8: A software combining tree barrier with optimized wakeup.

have used the sense-reversing technique to avoid overlap of successive barriers without requiring two spinning episodes per barrier, and have replaced the atomic instructions of the second combining tree with simple reads, since no real information is returned.

Each processor begins at a leaf of the combining tree, and decrements its leaf's `count` variable. The last processor to reach each node in the tree continues up to the next level. The processor that reaches the root of the tree begins a reverse wave of updates to `locksense` flags, with each processor unblocking its siblings as soon as it awakes. Simulations by Yew, Tzeng, and Lawrie [38] show that a software combining tree can significantly decrease memory contention and prevent tree saturation (a form of network congestion that delays the response of the network to all references [31]) in multistage interconnection networks, by distributing accesses across the memory modules of the machine.

A New Tree-Based Barrier Algorithm

The principal shortcoming of the combining tree barrier, from our point of view, is that is requires processors to spin on memory locations that cannot be statically determined, and on which other processors also spin. On broadcast-based cache-coherent machines, processors may obtain local copies of the tree nodes on which they spin, but on other machines (including the Cedar machine which Yew, Tzeng, and Lawrie simulated), processors will spin on remote locations, leading to unnecessary contention for interconnection network bandwidth. We have developed a tree-based barrier algorithm in which each processor spins on its own unique location, statically allocated and thus presumably local. Our algorithm uses no atomic instructions other than read and write,

requires less space than the combining tree barrier, and performs the minimum possible number of remote memory operations.

In our approach, a barrier to synchronize P processors is based on a pair of P-node trees. Each processor is assigned a unique tree node, which is linked into an arrival tree by a parent link, and into a wakeup tree by a set of child links. It is useful to think of these as separate trees, because the fan-in in the arrival tree differs from the fan-out in the wakeup tree.[10] A processor does not examine or modify the state of any other nodes except to signal its arrival at the barrier by setting a flag in its parent's node, and when notified by its parent that the barrier has been achieved, to notify each of its children by setting a flag in each of their nodes. Each processor spins only on state information in its own tree node. To achieve a barrier, each processor executes the code shown in algorithm 9.

Data structures for the tree barrier are initialized so that each node's `parentpointer` variable points to the appropriate `childnotready` flag in the node's parent, and the `childpointers` variables point to the `parentsense` variables in each of the node's children. Child pointers of leaves and the parent pointer of the root are initialized to reference pseudo-data. The `havechild` flags indicate whether a parent has a particular child or not. Initially, and after each barrier episode, each node's `childnotready` flags are set to the value of the node's respective `havechild` flags.

Upon arrival at a barrier, a processor tests to see if the `childnotready` flag is clear for each of its children. For leaf nodes, these flags are always clear, so deadlock cannot result. After a node's associated processor sees its `childnotready` flags are clear, it immediately re-initializes them for the next barrier. Since a node's children do not modify its `childnotready` flags again until they arrive at the next barrier, there is no potential for conflicting updates. After all of a node's children have arrived, the node's associated processor clears its `childnotready` flag in the node's parent. All processors other than the root then spin on their `parentsense` flag. When the root node's associated processor arrives at the barrier and notices that all of the root node's `childnotready` flags are clear, then all of the processors are waiting at the barrier. The processor at the root node then toggles the `parentsense` flag in each of its children to release them from the barrier. At each level in the tree, newly released processors release all of their children before leaving the barrier, thus ensuring that all processors are eventually released. Consecutive barrier episodes do not interfere since, as described earlier, the `childnotready` flags used during arrival are re-initialized before wakeup occurs.

This tree barrier implementation differs from the combining tree barrier in several important respects:

- Since each processor spins on a unique (fixed) node in the wakeup tree, we can embed the nodes in memory so that no two spinning processors compete for the same memory module or path through the interconnection network. In multiprocessors with coherent caches or distributed shared memory, we can exploit this locality to ensure that a busy waiting processor spins locally (either on a cache copy or on a tree node allocated in its local shared memory) and spinning will consume no network cycles.
- We exploit the atomicity of write operations, rather than `fetch_and_add`, to coordinate processors as they arrive at and subsequently depart from the barrier. Write instructions have

[10] We use a fan-out of 2 because it results in the the shortest critical path required to resume P spinning processors for a tree of uniform degree. We use a fan-in of 4 (1) because it produced the best performance in Yew, Tzeng, and Lawrie's experiments with software combining, and (2) because the ability to pack four bytes in a word permits an optimization on many machines in which a parent can inspect status information for all of its children simultaneously at the same cost as inspecting the status of only one.

```
type treenode = record
    parentsense : Boolean
    parentpointer : ^Boolean
    childpointers : array [0..1] of ^Boolean
    havechild : array [0..3] of Boolean
    childnotready : array [0..3] of Boolean
    dummy : Boolean     // pseudo-data

shared nodes : array [0..P-1] of treenode
    // nodes[vpid] is allocated in the portion of the shared memory
    // local to processor vpid
processor private vpid : integer     // a unique ``virtual processor'' index
processor private sense : Boolean

// on processor i, sense is initially true
// in nodes[i]:
//    havechild[j] = true if 4*i+j < P; otherwise false
//    parentpointer = &nodes[floor((i-1)/4)].childnotready[(i-1) mod 4],
//       or &dummy if i = 0
//    childpointer[1] = &nodes[2*i+1].parentsense, or &dummy if 2*i+1 >= P
//    childpointer[2] = &nodes[2*i+2].parentsense, or &dummy if 2*i+2 >= P
//    initially childnotready = havechild and parentsense = false

procedure tree_barrier
    with nodes[vpid] do
        repeat until childnotready = {false, false, false, false}
        childnotready := havechild    // prepare for next barrier
        parentpointer^ := false       // let parent know I'm ready
        // if not root, wait until my parent signals wakeup
        if vpid != 0
            repeat until parentsense = sense
        // signal children in wakeup tree
        childpointers[0]^ := sense
        childpointers[1]^ := sense
        sense := not sense
```

Algorithm 9: A scalable, distributed, tree-based barrier with only local spinning.

lower overhead than `fetch_and_add`, and are supported on all shared-memory multiprocessors, while `fetch_and_add` is not.

- Our algorithm requires $O(P)$ space, rather than $O(P \log P)$.
- Our algorithm reduces the number of remote memory operations that processors must perform to coordinate their arrival at the barrier. A barrier count implemented using software combining trees with fan-in k requires $\sum_{i=0}^{\log_k P-1} P/k^i = \frac{(P-1)k}{k-1}$ `fetch_and_add` operations to determine that all processors have reached the barrier, and a minimum of $\frac{(P-1)(k+2)}{k-1}$ other accesses to shared and potentially remote locations to allow them to proceed again.[11] Our tree barrier algorithm requires only $P-1$ remote write operations to determine that all processors have reached the barrier, and $P-1$ remote writes to allow them to proceed again.

Our tree barrier achieves the theoretical lower bound on the number of remote operations needed to achieve a barrier on machines that lack broadcast and that distinguish between local and remote memory. At least $P-1$ processors must signal their arrival to some other processor, requiring $P-1$ remote operations, and must then be informed of wakeup, requiring another $P-1$ remote operations. The length of the critical path in our algorithm is proportional to $\lceil \log_4 P \rceil + \lceil \log_2 P \rceil$. The first term is the time to propagate arrival up to the root, and the second term is the time to propagate wakeup back down to all of the leaves. On a machine with coherent caches and unlimited replication, we could replace the wakeup phase of our algorithm with a spin on a global flag. We explore this alternative on the Sequent in section 4.4.

Tournament Barriers

Hensgen, Finkel, and Manber [17] and Lubachevsky [26] have devised tree-style "tournament" barriers. Conceptually, to achieve a barrier using these tournament algorithms, processors start at the leaves of a binary tree, as in a combining tree barrier with fan-in two. One processor from each node continues up the tree to the next "round" of the tournament. The "winning" processor is statically determined in a tournament barrier, however, and as in our tree-based barrier there is no need for atomic `fetch_and_Φ` instructions. Also, both proposals for tournament barriers use a fan-in of 2, for $\lceil \log_2 P \rceil$ rounds of synchronization.

In round k (counting from zero) of Hensgen, Finkel, and Manber's barrier, processor i sets a flag awaited by processor j, where $i \equiv 2^k \pmod{2^{k+1}}$ and $j = i - 2^k$. Processor i then drops out of the tournament and waits on a global flag for notice that the barrier has been achieved. Processor j participates in the next round of the tournament. Processor 0 sets a global flag when the tournament is over.

Lubachevsky [26] presents a CREW (concurrent read, exclusive write) tournament barrier that uses a global flag for wakeup, similar to that of Hensgen, Finkel, and Manber. He also presents an EREW (exclusive read, exclusive write) tournament barrier in which each processor spins on separate flags in a binary wakeup tree, similar to the one in our modified version of the combining tree barrier in algorithm 8. Like our tree-based barrier, Lubachevsky's second tournament requres $O(P)$ space, compared to $O(P \log P)$ space for the Hensgen, Finkel, and Manber tournament.

Because all processors busy wait on a single global flag, Hensgen, Finkel, and Manber's tournament barrier and Lubachevsky's CREW barrier are appropriate for multiprocessors that use broadcast to maintain cache consistency. They will cause heavy interconnect traffic, however, on

[11] An assignment to `count`, an assignment to `locksense`, and k reads of `locksense` for each of $\frac{(P-1)}{k-1}$ nodes.

machines that lack coherent caches, or that limit the degree of cache line replication. Lubachevsky's EREW tournament could be used on any multiprocessor with coherent caches, including those that use limited-replication directory-based caching without broadcast. Unfortunately, in Lubachevsky's EREW barrier algorithm, each processor spins on a non-contiguous set of elements in an array, and no simple scattering of these elements will suffice to eliminate spinning-related network traffic on a machine without coherent caches.

By modifying Hensgen, Finkel, and Manber's tournament barrier to use a wakeup tree as in Lubachevsky's EREW tournament barrier, we have constructed an EREW tournament barrier in which each processor spins on its own set of contiguous, statically allocated flags (see algorithm 10). As in our tree-based barrier, the resulting code is able to avoid spinning on any remote locations, both on cache-coherent machines and on distributed shared memory multiprocessors. In addition to employing a wakeup tree, we have modified Hensgen, Finkel, and Manber's algorithm to use sense reversal to avoid re-initializing flag variables in each round.

The Dissemination Barrier

Besides the tree-style barriers, a second style of distributed barrier has evolved in which processors act in a homogeneous manner with no distinguished member (*e.g.*, the root or tournament champion in the tree barrier algorithms). Brooks [9] proposed a "butterfly barrier" in which each processor participates in a sequence of $\sim \log_2 P$ pairwise synchronizations. In round k (counting from zero) of a butterfly barrier, processor i synchronizes with processor $i \oplus 2^k$. If the number of processors is not a power of 2, then existing processors stand in for the missing ones, thereby participating in as many as $2\lceil \log_2 P \rceil$ pairwise synchronizations. The code for an individual synchronization is shown in algorithm 11.

Hensgen, Finkel, and Manber [17] describe a "dissemination barrier" that improves on Brooks's algorithm by employing a more efficient pattern of pairwise synchronizations and by reducing the cost of each synchronization. They provide performance figures for the Sequent Balance multiprocessor, comparing their algorithm against Brooks's, and against their own tournament barrier. They report that the tournament barrier outperforms the dissemination barrier when $P > 16$. The dissemination barrier requires $O(P \log P)$ remote operations, but the tournament barrier requires only $O(P)$. Beyond 16 processors, the additional factor of $\log P$ in bus traffic for the dissemination barrier algorithm dominates the higher constant of the tournament barrier. However, on scalable multiprocessors with multi-stage interconnection networks, many of the remote operations required by the dissemination barrier algorithm can proceed in parallel without interference.

The dissemination barrier takes its name from an algorithm developed to disseminate information among a set of processes. In round k, processor i synchronizes with processor $(i + 2^k) \bmod P$. This pattern does not require existing processes to stand in for missing ones, and therefore requires only $\lceil \log_2 P \rceil$ synchronization operations on its critical path, regardless of P. For a more detailed description of the synchronization pattern and a proof of its correctness, see [17].

For each pairwise synchronization of the dissemination barrier, Hensgen, Finkel, and Manber use alternating sets of variables in consecutive barrier episodes, avoiding interference without requiring two separate spins in each pairwise synchronization. They also use sense reversal to avoid resetting variables after every barrier. These two changes serve to eliminate the first and final lines of algorithm 11. The first change also serves to eliminate remote spinning. The authors motivate their algorithmic improvements in terms of reducing the number of instructions executed in the course of a pairwise synchronization, but we consider the elimination of remote spinning to be an even more important benefit. The flags on which each processor spins are statically determined,

```
type round_t = record
    role: enumeration {winner, loser, bye, champion, dropout}
    opponent: ^Boolean
    flag: Boolean
shared rounds : array [0..P-1][0..LogP] of round_t
    // row vpid of rounds is allocated in the portion of the shared memory
    // local to processor vpid
private sense : Boolean := true
processor private vpid : integer // a unique virtual processor index

// initially
//    rounds[i][j].flag = false for all i,j
// rounds[i][k].role =
//    winner if k > 0, i mod 2^k = 0, and i + 2^(k-1) < P
//    bye if k > 0, i mod 2^k = 0, and i + 2^(k-1) >= P
//    loser if k > 0 and i mod 2^k = 2^(k-1)
//    champion if k > 0, i = 0, and 2^k >= P
//    dropout otherwise
// rounds[i][k].opponent points to
//    rounds[i-2^(k-1)][k].flag if rounds[i][k].role = loser
//    rounds[i+2^(k-1)][k].flag if rounds[i][k].role = winner or champion
//    nil otherwise

procedure tournament_barrier
    var round : integer := 1
        local_sense : Boolean
    loop                // arrival
        case rounds[vpid][round].role of
            loser:
                rounds[vpid][round].opponent^ := sense
                repeat until rounds[vpid][round].flag = sense
                exit loop
            winner:
                repeat until rounds[vpid][round].flag = sense
            bye:        // do nothing
            champion:
                repeat until rounds[vpid][round].flag = sense
                rounds[vpid][round].opponent^ := sense
                exit loop
             dropout:   // impossible
        round := round + 1
    loop                // wakeup
        round := round - 1
        case rounds[vpid][round].role of
            loser:      // impossible
            winner:
                rounds[vpid][round].opponent^ := sense
            bye:        // do nothing
            champion:   // impossible
            dropout:
                exit loop
    sense := not sense
```

Algorithm 10: A scalable, distributed tournament barrier with only local spinning.

```
   Process 1                          Process 2
repeat while flag1                 repeat while flag2
flag1 := true                      flag2 := true
repeat until flag2                 repeat until flag1
flag2 := false                     flag1 := false
```

Algorithm 11: Brook's pairwise synchronization.

```
type flags = record
    myflags : array [0..1] of array [0..LogP-1] of Boolean
    partnerflags : array [0..1] of array [0..LogP-1] of ^Boolean

processor private parity : integer := 0
processor private sense : Boolean := true
processor private localflags : ^flags
shared allnodes : array [0..P-1] of flags
    // allnodes[i] is allocated in the portion of the shared memory
    // local to processor i

// on processor i, localflags points to allnodes[i]
// initially allnodes[i].myflags[r][k] is false for all i, r, k
// if j = (i+2^k) mod P, then for r = 0, 1:
//    allnodes[i].partnerflags[r][k] points to allnodes[j].myflags[r][k]

procedure dissemination_barrier
    var instance: integer
    for instance := 0 to LogP -1
        localflags^.partnerflags[parity][instance]^ := sense
        repeat until localflags^.myflags[parity][instance] = sense
    if parity = 1
        sense := not sense
    parity := 1 - parity
```

Algorithm 12: The scalable, distributed dissemination barrier with only local spinning.

and no two processors spin on the same flag. Each flag can therefore be located near the processor that reads it, leading to local-only spinning on any machine with local shared memory or coherent caches.

Algorithm 12 presents the dissemination barrier. The `parity` variable controls the use of alternating sets of flags in successive barrier episodes. On a machine with distributed shared memory and without coherent caches, the shared `allnodes` array would be scattered statically across the memory banks of the machine, or replaced by a scattered set of variables.

4 Performance Measurements

We have measured the performance of various spin lock and barrier algorithms on the BBN Butterfly 1, a distributed shared memory multiprocessor, and the Sequent Symmetry Model B, a cache-

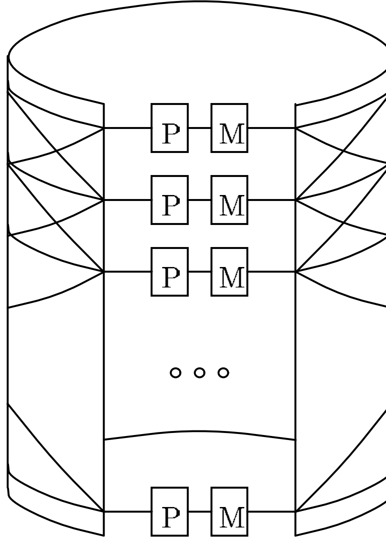

Figure 2: The BBN Butterfly 1.

coherent, shared-bus multiprocessor. Anyone wishing to reproduce our results or extend our work to other machines can obtain copies of our C and assembler source code via anonymous ftp from titan.rice.edu (directory /public/scalable_synch).

4.1 Hardware Description

BBN Butterfly

The BBN Butterfly 1 is a shared-memory multiprocessor that can support up to 256 processor nodes. Each processor node contains an 8 MHz MC68000 that uses 24-bit virtual addresses, and one to four megabytes of memory (one on our machine). Each processor can access its own memory directly, and can access the memory of any node through a $\log_4$-depth switching network (see figure 2). Transactions on the network are packet-switched and non-blocking. If collisions occur at a switch node, one transaction succeeds and all of the others are aborted, to be retried at a later time (in hardware) by the processors that initiated them. In the absence of contention, a remote memory reference (read) takes about 4 μs, roughly 5 times as long as a local reference.

The Butterfly 1 supports two 16-bit atomic operations: `fetch_and_clear_then_add` and `fetch_and_clear_then_xor`. Each operation takes three arguments: the address of the 16-bit destination operand, a 16-bit mask, and the value of the 16-bit source operand. The value of the destination operand is `and`ed with the one's complement of the mask, and then `add`ed or `xor`ed with the source operand. The resulting value replaces the original value of the destination operand. The previous value of the destination operand is the return value for the atomic operation. Using these two primitives, one can perform a variety of atomic operations, including `fetch_and_add`, `fetch_and_store` (`swap`), and `fetch_and_or` (which, like `swap`, can be used to perform a `test_and_set`).

Sequent Symmetry

The Sequent Symmetry Model B is a shared-bus multiprocessor that supports up to 30 processor nodes. Each processor node consists of a 16 MHz Intel 80386 processor equipped with a 64 KB

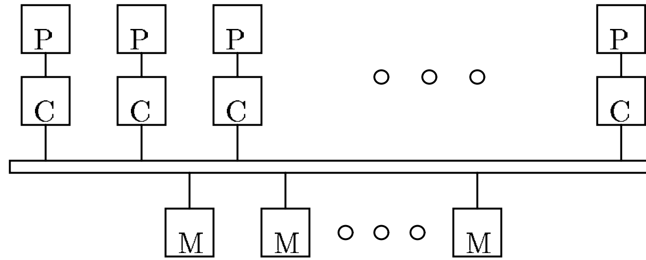

Figure 3: The Sequent Symmetry Model B.

two-way set-associative cache. All caches in the system are kept coherent by snooping on the bus (see figure 3). Each cache line is accompanied by a tag that indicates whether the data is replicated in any other cache. Writes are passed through to the bus only for replicated cache lines, invalidating all other copies. Otherwise the caches are write-back.

The Symmetry provides an atomic `fetch_and_store` operation, and allows various logical and arithmetic operations to be applied atomically as well. Each of these operations can be applied to any 1, 2, or 4 byte quantity. The logical and arithmetic operations do not return the previous value of the modified location; they merely update the value in place and set the processor's condition codes. This limitation makes them significantly less useful than the `fetch_and_Φ` operations of the Butterfly.

4.2 Measurement Technique

Our results were obtained by embedding lock acquisitions or barrier episodes inside a loop and averaging over a large number of operations. In the spin lock graphs, each data point (P,T) represents the average time T to acquire and release the lock with P processors competing for the lock. On the Butterfly, the average is over 10^5 lock acquisitions. On the Sequent, the average is over 10^6 lock acquisitions. For an individual test of P processors collectively executing K lock acquisitions, we required that each processor acquire and release the lock $\lfloor K/P \rfloor$ times. In the barrier graphs, each data point (P,T) represents the average time T for P processors to achieve a barrier. On both the Sequent and the Butterfly, the average is over 10^5 barriers.

This averaging technique introduces a significant anomaly into the data points near the left edge of the spin lock graphs. When $P = 1$, T represents the latency on one processor of the `acquire_lock` and `release_lock` operations in the absence of competition. When P is moderately large, T represents the time between lock acquisitions on successive competing processors. This *passing time* is a very different quantity from the latency measured on one processor; significant amounts of computation in `acquire_lock` prior to actual acquisition, and in `release_lock` after actual release, may be overlapped with work on other processors. When P is 2 or 3, T may represent either latency or passing time, depending on the relative amounts of overlapped and non-overlapped computation.[12] Moreover, in several cases (Anderson's lock, the MCS lock, and the locks incorporating backoff) the actual series of instructions executed for small values of P depends on how many other processors are competing for the lock, and which operations they have so far performed.

[12]Since T may represent either latency or passing time when more than one processor is competing for a lock, it is difficult to factor out overhead due to the timing loop for timing tests with more than one processor. For consistency, we included loop overhead in all of the average times reported.

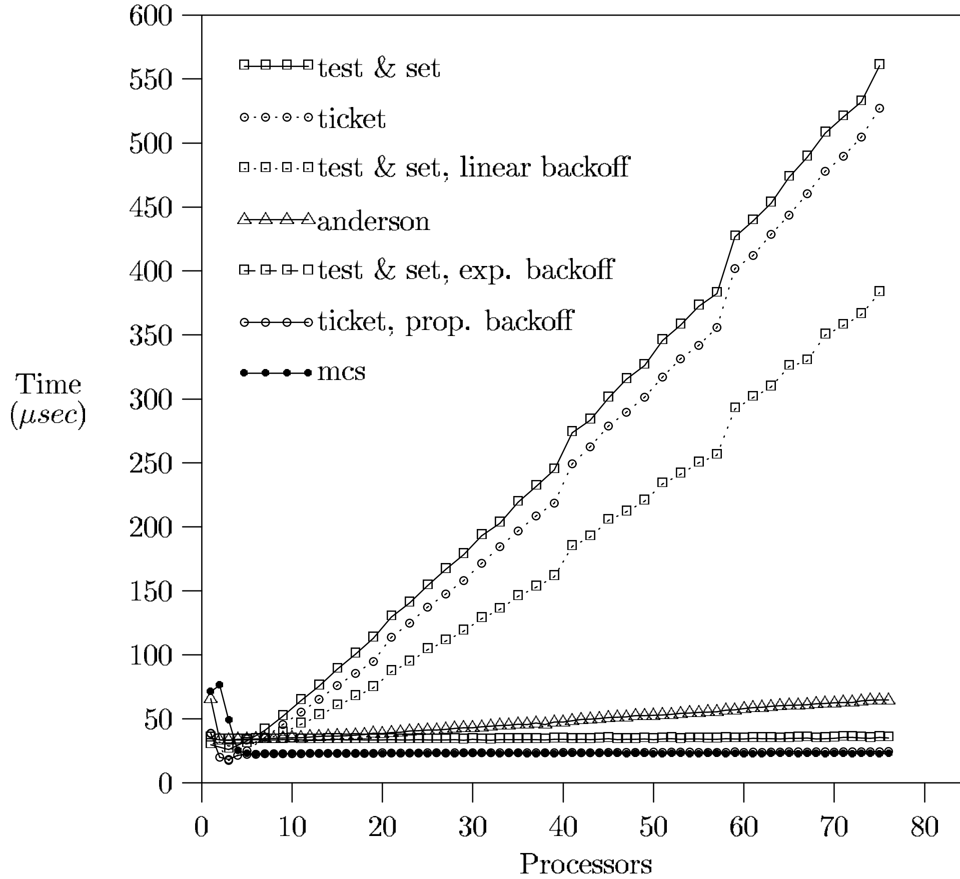

Figure 4: Performance of spin locks on the Butterfly (empty critical section).

Unless otherwise noted, all measurements on the Butterfly were performed with interrupts disabled. Similarly, on the Sequent, the `tmp_affinity()` system call was used to bind processes to processors for the duration of our experiments. These measures were taken to provide repeatable timing results.

4.3 Spin Locks

Figure 4 shows the performance on the Butterfly of the spin lock algorithms described in section 2. The top curve is a simple `test_and_set` lock, and displays the poorest scaling behavior.[13] As expected, the ticket lock is slightly faster, due to polling with a simple read instead of a `fetch_and_Φ`. We would expect a test-and–`test_and_set` lock to perform similarly. A linear least squares regression on the timings shows that the time to acquire and release the simple lock increases 7.4 μs per additional processor. The time for a ticket lock increases 7.0 μs per processor.

By analogy with the exponential backoff scheme described in section 2, we investigated the effect of having each processor delay between polling operations for a period of time directly proportional to the number of unsuccessful `test_and_set` operations. This change reduces the slope of the simple lock graph to 5.0 μs per processor, but performance degradation is still linear in the number of competing processors.

[13] We implement `test_and_set` using the hardware primitive `fetch_and_clear_then_add` with a mask that specifies to clear the lowest bit, and an addend of 1. This operation returns the old value of the lock and leaves the lowest bit in the lock set.

The time to acquire and release the simple lock, the ticket lock, and the simple lock with linear backoff does not actually increase linearly as more processors compete for the lock, but rather in a piecewise linear fashion. This behavior is a function of the interconnection network topology and the order in which we add processors to the test. For the tests shown in figure 4, our Butterfly was configured as an 80 processor machine with 5 switch cards in the first column of the interconnection network, supporting 16 processors each. Processors were added to the test in a round robin fashion, one from each card. The breaks in the performance graph occur as each group of 20 processors is added to the test. These are the points at which we have included an additional 4 processors from each switch card, leading to the use of one more 4-input, 4-output switch node on each card. What we see in the performance graphs is that the additional node causes behavior that is qualitatively different from that obtained by including another processor attached to a switch node already in use. This difference is likely related to the fact that additional processors attached to a switch node already in use add contention in the first level of the interconnection network. An additional switch node adds contention in the second level of the network. In a fully configured machine with 256 processors attached to 16 switch cards in the first column of the interconnection network, we would expect the breaks in spin lock performance to occur every 64 processors.

Figure 5 provides an expanded view of performance results for the more scalable algorithms, whose curves are grouped together near the bottom of figure 4. In this expanded graph, it is apparent that the time to acquire and release the lock in the single processor case is often much larger than the the time required when multiple processors are competing for the lock. As noted above, parts of each acquire/release protocol can execute in parallel when multiple processors compete. What we are measuring in our trials with many processors is not the time to execute an acquire/release pair from start to finish, but rather the length of time between a pair of lock acquisitions. Complicating matters is that the time required to release an MCS lock depends on whether another processor is waiting.

The top curve in figure 5 shows the performance of Anderson's array-based queueing algorithm, modified to scatter the slots of the queue across the available processor nodes. This modification distributes traffic evenly in the interconnection network, by allowing each processor to spin on a location in a different memory bank. Because the Butterfly lacks coherent caches, however, and because processors spin on statically unpredictable locations, it is not in general possible with Anderson's lock to spin on *local* locations. Linear regression yields a slope for the performance graph of 0.4 μs per processor.

Three algorithms—the simple lock with exponential backoff, the ticket lock with proportional backoff, and the MCS lock—all scale extremely well. Ignoring the data points below 10 processors (which helps us separate throughput under heavy competition from latency under light competition), we find slopes for these graphs of 0.025, 0.021, and 0.00025 μs per processor, respectively. Since performance does not degrade appreciably for any of these locks within the range of our tests, we would expect them to perform well even with thousands of processors competing.

Figure 6 shows performance results for several spin lock algorithms on the Sequent. We adjusted data structures in minor ways to avoid the unnecessary invalidations that would result from placing unrelated data items in the same cache line. The test-and-`test_and_set` algorithm showed the poorest scaling behavior, with the time to acquire and release the lock increasing dramatically even over this small range of processors. A simple `test_and_set` would perform even worse.

Because the atomic add instruction does not return the old value of its target location, implementation of the ticket lock is not possible on the Sequent, nor is it possible to implement Anderson's lock directly. In his implementation [4], Anderson (who worked on a Sequent) intro-

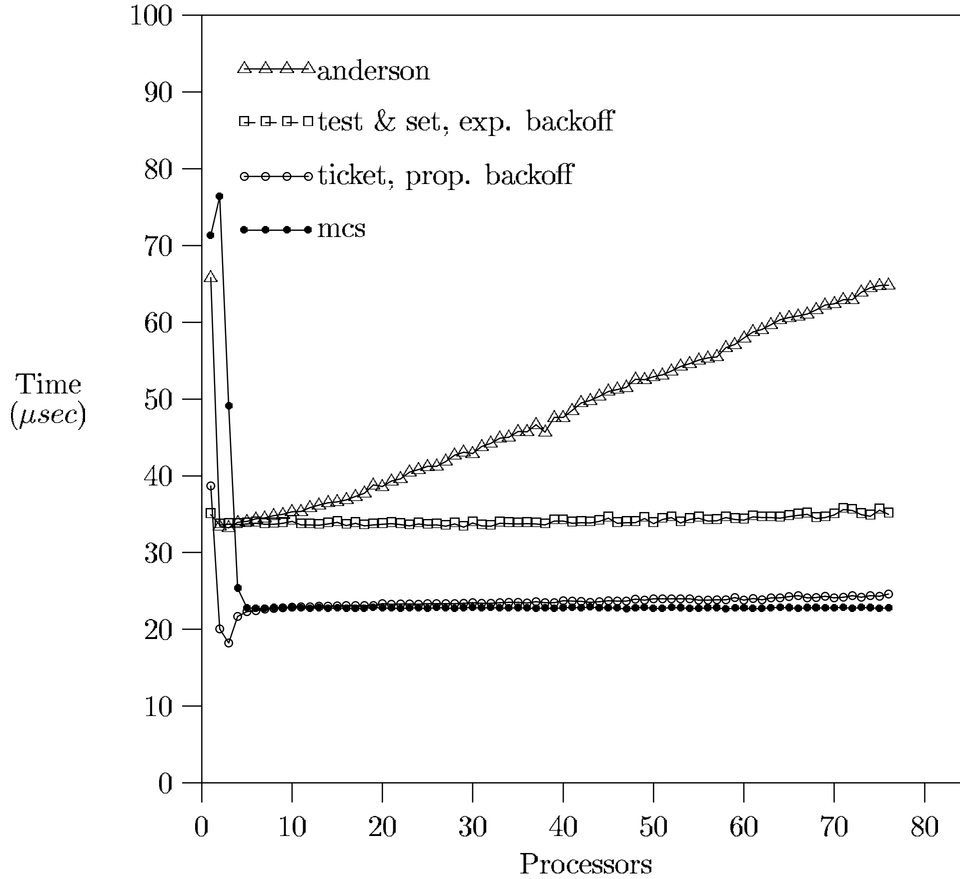

Figure 5: Performance of selected spin locks on the Butterfly (empty critical section).

duced an outer **test_and_set** lock with randomized backoff to protect the state of his queue.[14] This strategy is reasonable when the critical section protected by the outer lock (namely, acquisition or release of the inner lock) is substantially smaller than the critical section protected by the inner lock. This was not the case in our initial test, so the graph in figure 6 actually results from processors contending for the *outer* lock, instead of the inner, queue-based lock. To eliminate this anomaly, we repeated our tests with a non-empty critical section, as shown in figure 7. With a sufficiently long critical section (6.48 μs in our tests), processors have a chance to queue up on the inner lock, eliminating competition for the outer lock, and allowing the inner lock to eliminate bus transactions due to spinning. The time spent in the critical section has been factored out of the timings.

In addition to Anderson's lock, our experiments indicate that the MCS lock and the simple lock with exponential backoff also scale extremely well. All three of the scalable algorithms have comparable absolute performance. Anderson's lock has a small edge for non-empty critical sections, but requires statically-allocated space per lock linear in the number of processors. The other two algorithms need only constant space, and do not require coherent caches to work well. The simple lock with exponential backoff shows a slight increasing trend, and might not do as well as the others on a very large machine.

The peak in the cost of the MCS lock on two processors reflects the lack of **compare_and_swap**.

[14]Given that he required the outer lock in any case, Anderson also replaced the **nextslot** index variable with a pointer, to save time on address arithmetic.

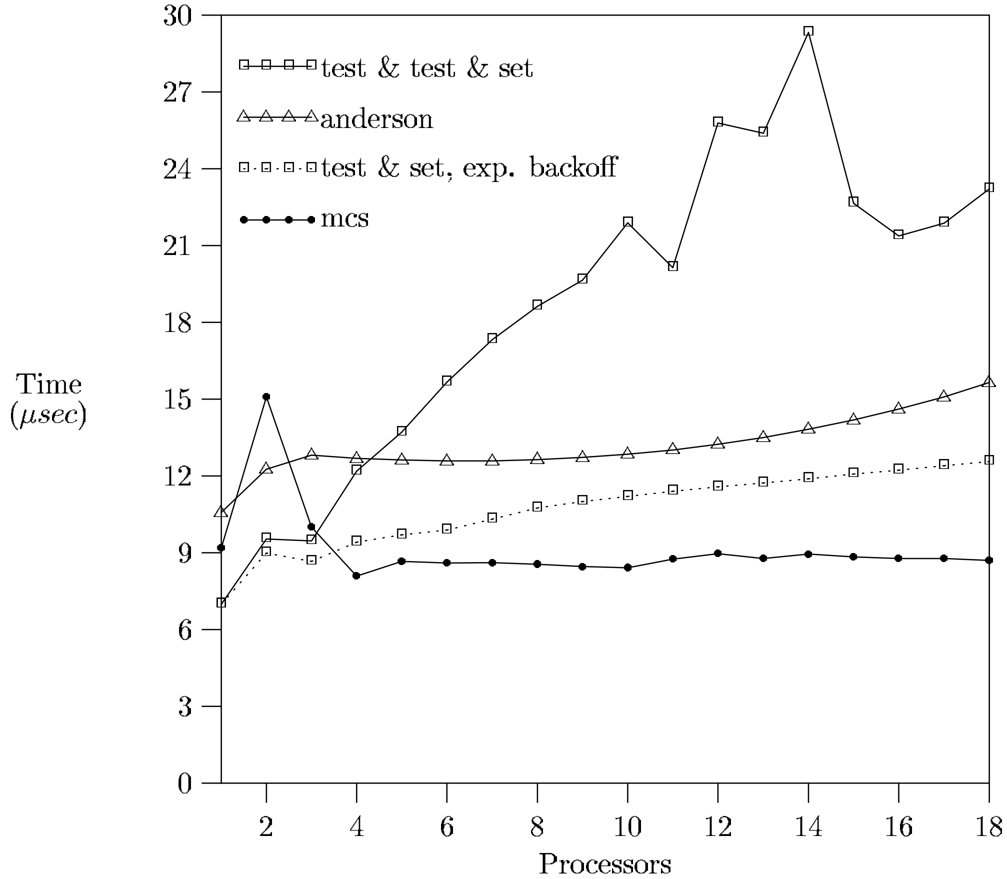

Figure 6: Performance of spin locks on the Sequent (empty critical section).

	`test_and_set`	ticket	Anderson	MCS
Butterfly	29.8 μs	38.6 μs	65.7 μs	71.3 μs
Sequent	7.0 μs	NA	10.6 μs	9.2 μs

Table 1: Time for an acquire/release pair in the single processor case.

Some fraction of the time, a processor releasing the lock finds that its `next` variable is `nil` but then discovers that it has a successor after all when it performs its `fetch_and_store` on the lock's tail pointer. Entering this timing window necessitates an additional `fetch_and_store` to restore the state of the queue, with a consequent drop in performance. The non-empty critical sections of figure 7 reduce the likelihood of hitting the window, thereby reducing the size of the two-processor peak. With `compare_and_swap` that peak would disappear altogether.

Latency and Impact on Other Operations

In addition to performance in the presence of many competing processors, an important criterion for any lock is the time it takes to acquire and release it in the absence of competition. Table 1 shows this measure for representative locks on both the Butterfly and the Sequent. Times for the simple lock are without backoff of any kind; times for the ticket lock are with code in place for proportional backoff. The simple lock is cheapest on both machines in the single processor case;

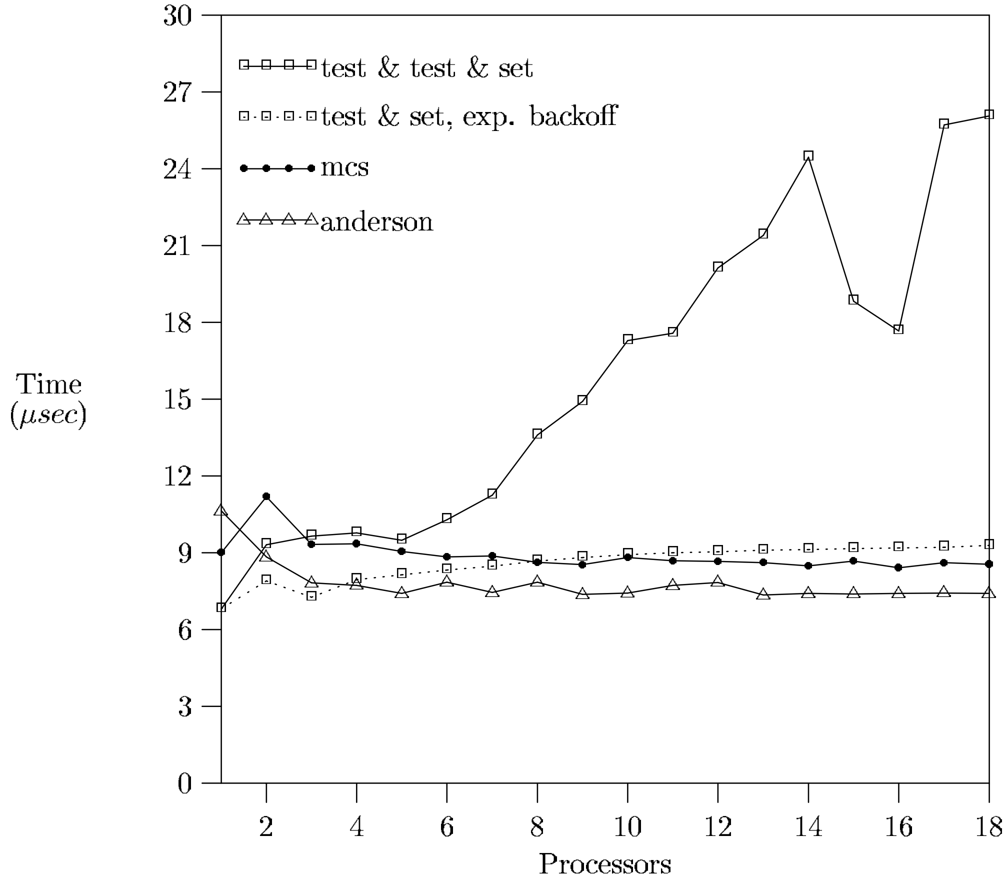

Figure 7: Performance of spin locks on the Sequent (small critical section).

it has the shortest code path. On the Butterfly, the ticket lock, Anderson's lock, and the MCS lock are 1.29, 2.20, and 2.39 times as costly, respectively. On the Sequent, Anderson's lock and the MCS lock are 1.51 and 1.31 times as costly as the simple lock.

Several factors skew the absolute numbers on the Butterfly, making them somewhat misleading. First, our simple lock is implemented completely in line, while the other locks are complicated enough to warrant subroutine calls. Ideally, the ticket lock with proportional backoff would call a subroutine only when backoff is required. Difficulties with our compiler made this approach impractical, but we expect that it would have reduced the latency of the ticket lock to nearly the same as the simple lock (an empty subroutine call on the Butterfly takes almost 10 μs). Second, the atomic operations on the Butterfly are inordinately expensive in comparison to their non-atomic counterparts. Most of the latency of each of the locks is due to the cost of setting up a parameter block and executing an atomic operation. This affects the performance of the MCS lock in particular, since it requires at least two `fetch_and_store` operations (one to acquire the lock and another to release it), and possibly a third (if we hit the timing window). Third, the 16-bit atomic primitives on the Butterfly cannot manipulate 24-bit pointers atomically. To implement the MCS algorithm, we were forced to replace the pointers with indices into a replicated, statically-allocated array of pointers to `qlink` records.

Absolute performance for all the algorithms is much better on the Sequent than on the Butterfly. The Sequent's clock runs twice as fast, and its caches make memory in general appear significantly faster. The differences between algorithms are also smaller on the Sequent, mainly because of the lower difference in cost between atomic and non-atomic instructions, and also because the Sequent's

at rest	test_and_set	ticket	ticket w/ prop. backoff	test_and_set w/ exp. backoff	Anderson	MCS
14.6 μs	251 μs	184 μs	25 μs	39 μs	27 μs	15 μs

Table 2: Measurement of busy-waiting impact on network load.

32-bit `fetch_and_store` instruction allows the MCS lock to use pointers.

A final important measure of spin lock performance is the amount of interconnection network traffic caused by busy-waiting processors, and the impact of this traffic on other activity on the machine. We used BBN's "probe" utility on the Butterfly to obtain an indirect measure of these quantities. Every three seconds, probe accesses memory through the switch interface of every processor in the machine, and reports the total time required. Individual reports are not particularly meaningful, but the ratios of reports obtained at different times are a good indication of the relative level of load on the network. In table 2 we compare the "at rest" probe value with the values obtained when 75 processors are competing for each of several types of lock. With no user programs active, an average over 109 reports by the probe command showed a probe of each processor's switch interface to take 14.6 μs with a standard deviation of 1.4 μs. Since the MCS locking algorithm uses no network bandwidth for busy waiting, it is no surprise that with 75 processors using the algorithm to compete for a lock, our measured value for the probe time is within one standard deviation of the at rest value. In contrast, using the simple and ticket locks without backoff, the probe time increased by factors of 17 and 12.6, respectively. Probe times for the remaining algorithms (the ticket lock with proportional backoff, the simple lock with exponential backoff, and Anderson's lock), increased by factors of 1.7, 2.7, and 1.9, respectively. One defficiency of the probe utility is that it provides no indication as to whether contention is uniform, or isolated to a relatively small number of "hot" memory modules and switch nodes.

Discussion and Recommendations

Spin lock algorithms can be evaluated on the basis of several criteria:

- scalability and induced network load
- one-processor latency
- space requirements
- fairness/sensitivity to preemption
- implementability with given atomic operations

The MCS lock and, on cache-coherent machines, the Anderson lock are the most scalable algorithms we studied. The simple and ticket locks also scale well with appropriate backoff, but induce more network load. The simple and ticket locks have the lowest single-processor latency, but with good implementations of `fetch_and_`Φ instructions the MCS and Anderson locks are reasonable as well (on the Sequent, the single-processor latency of the MCS lock is only 31% higher than that of the simplest `test_and_set` lock). The space needs of Anderson's lock are likely to be

prohibitive when a large number of locks is needed—in the internals of an operating system, for example.

The ticket lock, Anderson's lock, and the MCS lock all guarantee that processors attempting to acquire a lock will succeed in FIFO order. This guarantee of fairness is likely to be considered an advantage in many environments, but will be a distinct *disadvantage* if spinning processes may be preempted. A simple lock with exponential backoff will allow latecomers to acquire the lock when processes that arrived earlier are not running, and may therefore be preferred. In any event, it may be important to introduce mechanisms to ensure that a process is not preempted while actually holding a lock [34, 39].

All of the spin lock algorithms we have considered require some sort of `fetch_and_Φ` instructions. The simple lock requires `test_and_set`. The ticket lock requires `fetch_and_increment`. The MCS lock requires `fetch_and_store`,[15] and benefits from `compare_and_swap`. Anderson's lock benefits from `fetch_and_add`. Of these requirements, those of the ticket and MCS locks are most likely to be a problem on currently-available machines.

For cases in which competition is expected, the MCS lock is clearly the implementation of choice. On average it takes the smallest amount of time to pass on a lock, provides FIFO ordering, scales almost perfectly, and requires only a small, constant amount of space per lock. It also induces the least amount of interconnect contention. On a machine with fast `fetch_and_Φ` operations (particularly if `compare_and_swap` is available), its one-processor latency will be competitive with all the other algorithms.

The ticket lock with proportional backoff is an attractive alternative if one-processor latency is an overriding concern, or if `fetch_and_store` is not available. Although our experience with it is limited to a distributed-memory multiprocessor without cache coherence, our expectation is that the ticket lock with proportional backoff would perform well on cache-coherent multiprocessors as well. The simple lock with exponential backoff is an attractive alternative if preemption is possible while spinning, or if neither `fetch_and_store` nor `fetch_and_increment` is available. It is 47% slower than the ticket lock with proportional backoff in the presence of heavy competition, and results in significantly more network load.

4.4 Barriers

Figure 8 shows the performance on the Butterfly of the barrier algorithms described in section 3. The top three curves are all sense-reversing, counter-based barriers as in algorithm 7, with various backoff strategies. The slowest performs no backoff. The next uses exponential backoff. We obtained the best performance with an initial delay of 10 iterations through an empty loop, with a backoff base of 2. Our results suggest that it may be necessary to limit the maximum backoff in order to maintain stability. When a large number of barriers are executed in sequence, the skew of processors arriving at the barriers is magnified by the exponential backoff strategy. As the skew between arriving processors increases, processors back off farther. With a backoff base larger than 2, we had to cap the maximum delay in order for our experiments to finish. Even with a backoff base of 2, a delay cap improved performance. Our best results were obtained with a cap of $8P$ delay loop iterations.

In our final experiment with a centralized, counter-based barrier, we used a variant of the proportional delay idea employed in the ticket lock. After incrementing the barrier count to signal

[15] It could conceivably be used with only `compare_and_swap`, simulating `fetch_and_store` in a loop, but at a significant loss in scalability.

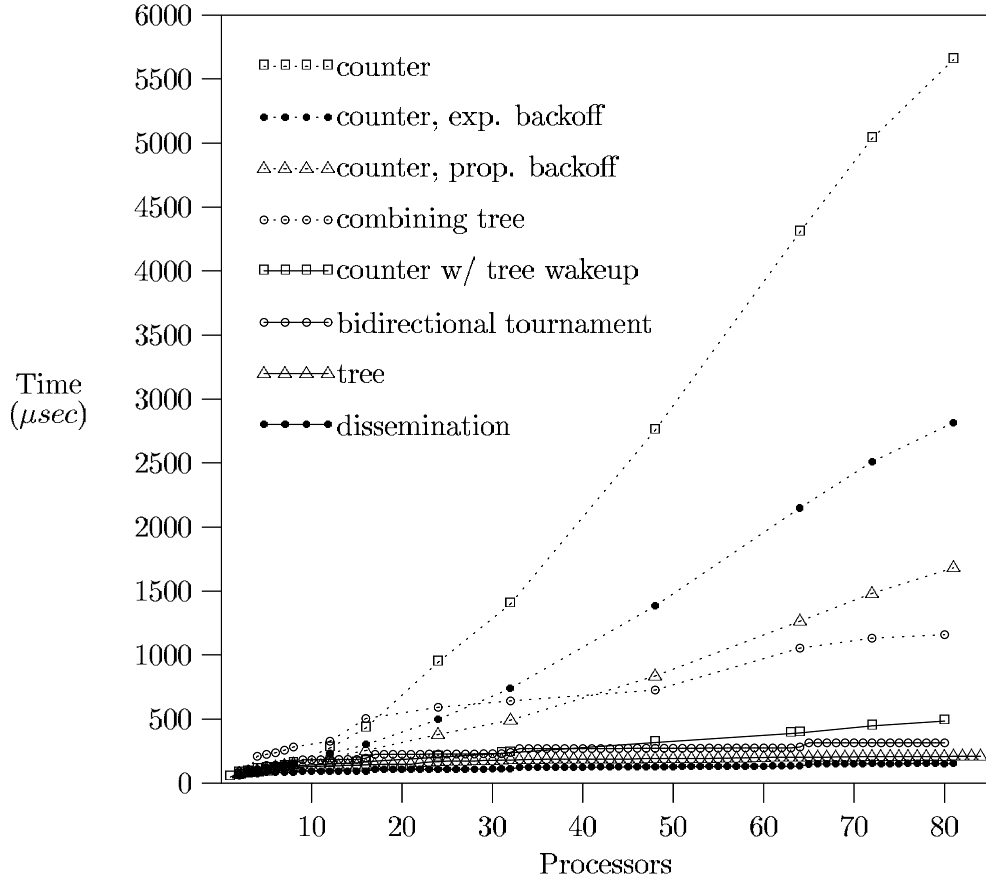

Figure 8: Performance of barriers on the Butterfly.

its arrival, each processor participating in the barrier delays a period of time proportional to the total number of participants (*not* the number yet to arrive), prior to testing the sense variable for the first time. The rationale for this strategy is to timeslice available interconnect bandwidth between the barrier participants. Since the Butterfly network does not provide hardware combining, at least $2P-1$ accesses to the barrier state are required (P to signal processor arrivals, and $P-1$ to discover that all have arrived). Each processor delays long enough for later processors to indicate their arrival, and for earlier processors to notice that all have arrived. As shown in figure 8, this strategy outperforms both the naive central barrier and the central barrier with exponential backoff. At the same time, all three counter-based algorithms lead to curves of similar shape. The time to achieve a barrier appears to increase more than linearly in the number of participants. The best of these purely centralized algorithms (the proportional backoff strategy) requires over 1.4 *ms* for an 80 processor barrier.

The fourth curve in figure 8 is the combining tree barrier of algorithm 8. Though this algorithm scales better than the centralized approaches (in fact, it scales roughly logarithmically with P, although the constant is large), it still spins on remote locations, and encounters increasing interconnect contention as the number of processors grows.

Figure 9 provides an expanded view of performance results for the algorithms with the best performance, whose curves are grouped together near the bottom of figure 8. The code for the upper curve uses a central counter to tally arrivals at the barrier, but employs a binary tree for wakeup, as in algorithm 9. The processor at the root of the tree spins on the central counter. Other processors spin on flags in their respective nodes of the tree. All spins are local, but the tallying of

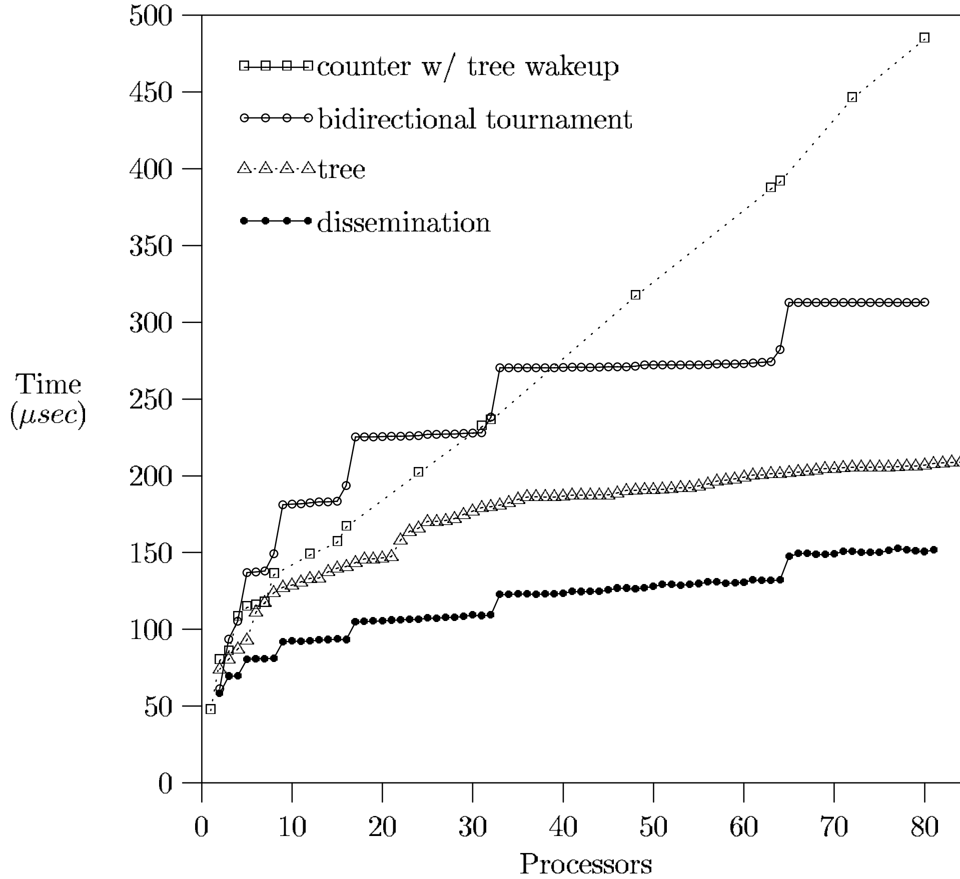

Figure 9: Performance of selected barriers on the Butterfly.

arrivals is serialized. We see two distinct sections in the resulting performance curve. With fewer than 10 processors, the time for wakeup dominates, and the time to achieve the barrier is roughly logarithmic in the number of participants. With more than 10 processors, the time to access the central counter dominates, and the time to achieve the barrier is roughly linear in the number of participants.

The three remaining curves in figure 9 are for our tree barrier, the dissemination barrier, and a modified version of the tournament barrier of Hensgen, Finkel and Manber. The time to achieve a barrier with each of these algorithms scales logarithmically with the number of processors participating. The tournament and dissemination barriers proceed through $O(\lceil \log P \rceil)$ rounds of synchronization, leading to a stair-step curve. Since the Butterfly does not provide coherent caches, the tournament barrier employs a binary wakeup tree, as shown in algorithm 10. It requires $2\lceil \log_2 P \rceil$ rounds of synchronization, compared to only $\lceil \log_2 P \rceil$ rounds in the dissemination barrier, resulting in a roughly two-fold difference in performance. Our tree-based barrier lies between the other two; its critical path comprises approximately $\log_4 P + \log_2 P$ pairwise synchronizations. The lack of clear-cut rounds in our barrier explains its smoother performance curve: each additional processor adds another level to some path through the tree, or becomes the second child of some node in the wakeup tree, delayed slightly longer than its sibling.

Figure 10 shows the performance on the Sequent Symmetry of several different barriers. Results differ sharply from those on the Butterfly for two principal reasons. First, it is acceptable on the Sequent for more than one processor to spin on the same location; each obtains a copy in its cache. Second, no significant advantage arises from distributing writes across the memory modules of

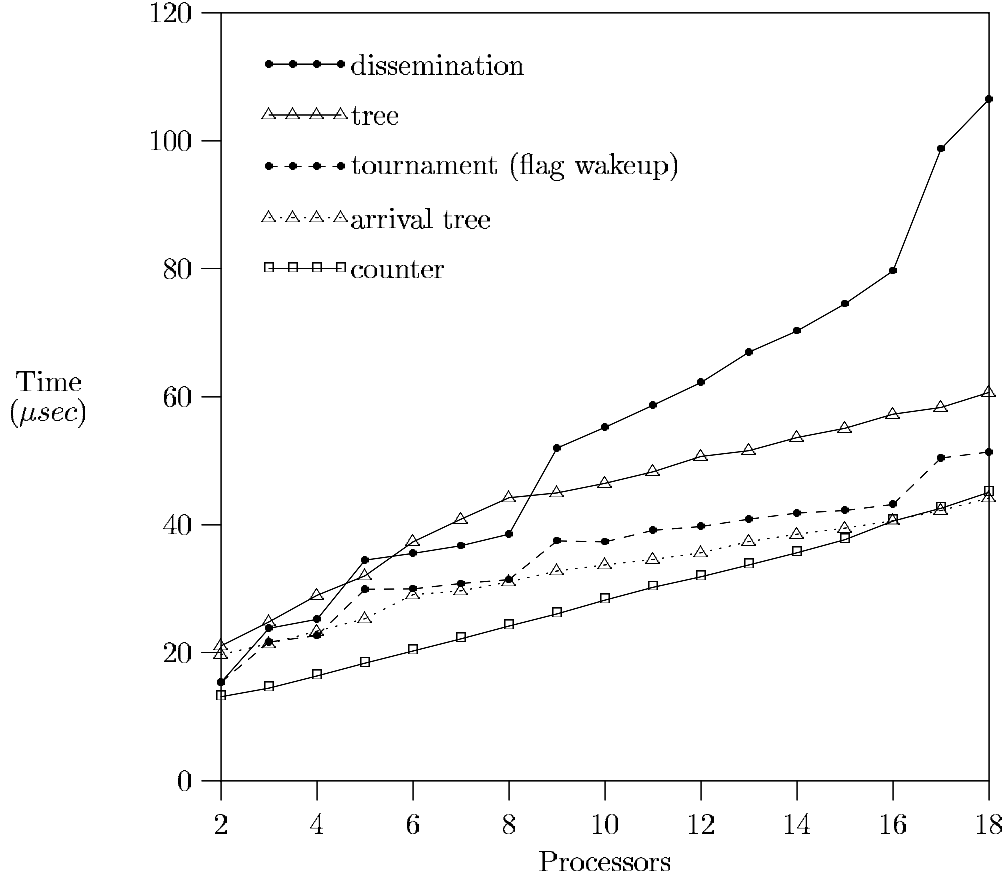

Figure 10: Performance of barriers on the Sequent.

the machine; the shared bus enforces an overall serialization. The dissemination barrier requires $O(P \log P)$ bus transactions to achieve a P-processor barrier. The other four algorithms require $O(P)$ transactions, and all perform better than the dissemination barrier for $P > 8$.

Below the maximum number of processors in our tests, the fastest barrier on the Sequent used a centralized counter with a sense-reversing wakeup flag (from algorithm 7). P bus transactions are required to tally arrivals, 1 to toggle the sense-reversing flag (invalidating all the cached copies), and $P-1$ to effect the subsequent re-loads. Our tree barrier generates $2P-2$ writes to flag variables on which other processors are waiting, necessitating an additional $2P-2$ re-loads. By using a central sense-reversing flag for wakeup (instead of the wakeup tree), we can eliminate half of this overhead. The resulting algorithm is identified as "arrival tree" in figure 10. Though the arrival tree barrier has a larger startup cost, its $P-1$ writes are cheaper than the P read-modify-write operations of the centralized barrier, so its slope is lower. For large values of P, the arrival tree with wakeup flag is the best performing barrier, and should become clearly so on larger machines.

The tournament barrier on the Sequent uses a central wakeup flag. It roughly matches the performance of the arrival tree barrier for $P = 2^i$, but is limited by the length of the execution path of the tournament champion, which grows suddenly by one each time that P exceeds a power of 2.

Discussion and Recommendations

The dissemination barrier appears to be the most suitable algorithm for distributed memory machines without broadcast. It has a shorter critical path than the tree and tournament barriers (by a constant factor), and is therefore faster. The class of machines for which the dissemination barrier should outperform all other algorithms includes the BBN Butterfly [8], the IBM RP3 [30], Cedar [37], the BBN Monarch [32], the NYU Ultracomputer [16], and proposed large-scale multiprocessors with directory-based cache coherence [2]. Our tree-based barrier will also perform well on these machines. It induces less network load, and requires total space proportional to P, rather than $P \log P$, but its critical path is longer by a factor of about 1.5.

Our tree-based barrier with wakeup flag should be the fastest algorithm on large-scale multiprocessors that use broadcast to maintain cache coherence (either in snoopy cache protocols [14] or in directory-based protocols with broadcast [7]). It requires only $O(P)$ updates to shared variables in order to tally arrivals, compared to $O(P \log P)$ for the dissemination barrier. Its updates are simple writes, which are cheaper than the read-modify-write operations of a centralized counter-based barrier. Its space needs are lower than those of the tournament barrier ($O(P)$ instead of $O(P \log P)$), its code is simpler, and it performs slightly less local work when P is not a power of 2. Our results are consistent with those of Hensgen, Finkel, and Manber [17], who showed their tournament barrier to be faster than their dissemination barrier on the Sequent Balance multiprocessor. They did not compare their algorithms against a centralized barrier because the lack of a hardware atomic increment instruction on the Balance precludes efficient atomic update of a counter.

4.5 Architectural Implications

Many different shared memory architectures have been proposed. From the point of view of synchronization, the two relevant issues seem to be (1) whether shared memory can be accessed locally by some processor (instead of through the interconnection network), and (2) whether broadcast is available for cache coherency. The first issue is crucial; it determines whether busy waiting can be eliminated as a cause of memory and interconnect contention. The second issue determines whether barrier algorithms can efficiently employ a centralized flag for wakeup.

The most scalable synchronization algorithms (the MCS spin lock and the tree, bidirectional tournament, and dissemination barriers) are designed in such a way that each processor spins on statically-determined location(s) on which no other processor spins. On a distributed shared memory machine, variables can be located explicitly at the processor that spins on them. On a cache-coherent machine, they migrate to the correct location automatically, provided that flag variables used for busy-waiting by different processors are in separate cache lines. In either case, remote operations are used only to update a location on which some processor is waiting. For the MCS spin lock the number of remote operations per lock acquisition is constant. For the tree and tournament barriers, the number of remote operations per barrier is linear in the number of processors involved. For the dissemination barrier, the number of remote operations is $O(P \log P)$, but $O(\log P)$ on the critical path. No remote operations are due to spinning, so interconnect contention is not a problem.

On "dance hall" machines, in which shared memory must always be accessed through a shared processor-memory interconnect, there is no way to eliminate synchronization-related interconnect contention. Nevertheless, the algorithms we have described are useful since they minimize memory contention and hot spots caused by synchronization. The structure of these algorithms makes it

at rest	tree barrier		dissemination barrier	
	w/ local polling	w/ network polling	w/ local polling	w/ network polling
14.5 μs	15 μs	32.9 μs	14.4 μs	32.5 μs

Table 3: Effect of barrier algorithms on network load, with and without local access to shared memory.

easy to assign each processor's busy-wait flag variables to a different memory bank so that the load induced by spinning will be distributed evenly throughout memory and the interconnect, rather than being concentrated in a single spot. Unfortunately, on dance hall machines the load will still consume interconnect bandwidth, degrading the performance not only of synchronization operations but also of all other activity on the machine, severely constraining scalability.

Dance hall machines include bus-based multiprocessors without coherent caches, and multistage network architectures such as Cedar [37], the BBN Monarch [32], and the NYU Ultracomputer [16]. Both Cedar and the Ultracomputer include processor-local memory, but only for private code and data. The Monarch provides a small amount of local memory as a "poor man's instruction cache." In none of these machines can local memory be modified remotely. We consider the lack of local shared memory to be a significant architectural shortcoming; the inability to take full advantage of techniques such as those described in this paper is a strong argument against the construction of dance hall machines.

To assess the importance of local shared memory, we used our Butterfly 1 to simulate a machine in which all shared memory is accessed through the interconnection network. By flipping a bit in the segment register for the synchronization variables on which a processor spins, we can cause the processor to go out through the network to reach these variables (even though they are in its own memory), without going through the network to reach code and private data. This trick effectively flattens the two-level shared memory hierarchy of the Butterfly into a single level organization similar to that of Cedar, the Monarch, or the Ultracomputer.

Figure 11 compares the performance of the dissemination and tree barrier algorithms for one and two level memory hierarchies. All timing measurements in the graph were made with interrupts disabled, to eliminate any effects due to timer interrupts or scheduler activity. The bottom two curves are the same as in figures 8 and 9. The top two curves show the corresponding performance figures when all accesses to shared memory are forced to go through the interconnection network. When busy-waiting accesses traverse the interconnect, the time to achieve a barrier using the tree and dissemination algorithms increases linearly with the number of processors participating. A least squares fit shows the additional cost per processor to be 27.8 μs and 9.4 μs, respectively. For an 84-processor barrier, the lack of local spinning increases the cost of the tree and dissemination barriers by factors of 11.8 and 6.8, respectively.

In a related experiment, we measured the impact on network latency of executing the dissemination or tree barriers with and without local access to shared memory. The results appear in table 3. As in table 2, we used the Butterfly "probe" utility to compare network latency at rest with latency during a barrier. We find that network latency is virtually unaffected when processors are able to spin on shared locations without going through the interconnect. With only remote access to shared memory, latency more than doubles.

Studies by Pfister and Norton [31] show that hot-spot contention can lead to tree saturation in multistage interconnection networks with blocking nodes and distributed routing control, inde-

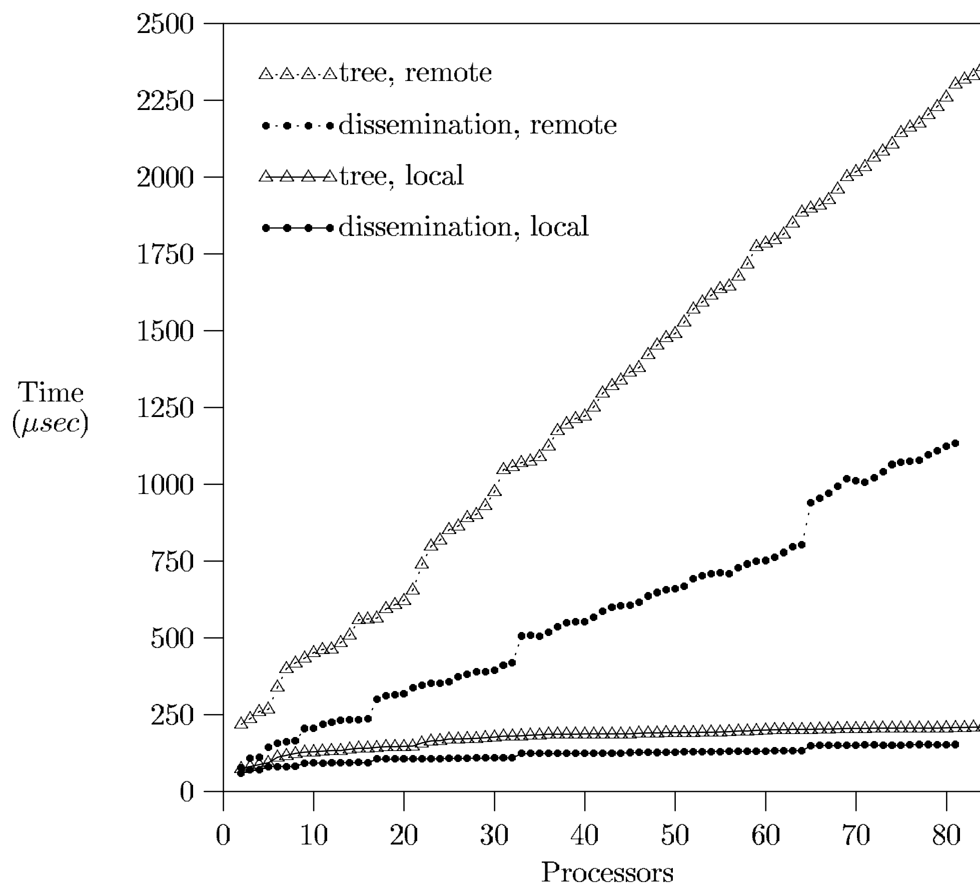

Figure 11: Performance of tree and dissemination barriers with and without local access to shared memory.

pendent of the network topology. A study by Kumar and Pfister [21] shows the onset of hot-spot contention to be rapid. Pfister and Norton argue for hardware message combining in interconnection networks to reduce the impact of hot spots. They base their argument primarily on anticipated contention for locks, noting that they know of no quantitative evidence to support or deny the value of combining for general memory traffic. Our results indicate that the cost of synchronization in a system without combining, and the impact that synchronization activity will have on overall system performance, is much less than previously thought (provided that the architecture incorporates a shared memory hierarchy of two or more levels). Although the scalable algorithms presented in this paper are unlikely to match the performance of hardware combining, they will come close enough to provide an extremely attractive alternative to complex, expensive hardware.[16]

Other researchers have suggested building special-purpose hardware mechanisms solely for synchronization, including synchronization variables in the switching nodes of multistage interconnection networks [19] and lock queueing mechanisms in the cache controllers of cache-coherent multiprocessors [13, 24, 28]. Our results suggest that simple exploitation of a multi-level memory hierarchy, *in software*, provides a more cost-effective means of avoiding lock-based contention.

The algorithms we present in this paper require no hardware support for synchronization other than commonly-available atomic instructions. The scalable barrier algorithms rely only on atomic read and write. The MCS spin lock algorithm uses `fetch_and_store` and maybe `compare_and_-`

[16]Pfister and Norton estimate that message combining will increase the size and possibly the cost of an interconnection network 6- to 32-fold. Gottlieb [15] indicates that combining networks are difficult to bit-slice.

swap. Anderson's lock benefits from fetch_and_increment, and the ticket lock requires it. All of these instructions have uses other than the construction of busy-wait locks.[17] Because of their general utility, they are substantially more attractive than special-purpose synchronization primitives. Future designs for shared memory machines should include a full set of fetch_and_Φ operations, including compare_and_swap.

5 Conclusion

The principal conclusion of our work is that memory and interconnect contention due to busy wait synchronization in shared-memory multiprocessors need not be a problem. This conclusion runs counter to widely-held beliefs. We have studied the behavior of mutual exclusion and barrier synchronization mechanisms based on busy waiting, with a particular eye toward minimizing the remote references that lead to contention. We have demonstrated that appropriate algorithms using simple and widely-available atomic instructions can reduce synchronization contention effectively to zero.

For spin locks on a shared-memory multiprocessor, regardless of architectural details, we suggest:

1. If the hardware provides an efficient fetch_and_store instruction (and maybe compare_and_swap), then use the MCS lock. One-processor latency will be reasonable, and scalability will be excellent.

2. If fetch_and_store is not available, or if atomic operations are very expensive relative to non-atomic instructions and one-processor latency is an overwhelming concern, then use the ticket lock with proportional backoff (assuming the hardware supports fetch_and_increment). The code for such a lock is typically more complicated than code for the MCS lock, and the load on the processor-memory interconnect will be higher in the presence of competition for the lock, but speed on a single processor will be slightly better and scalability will still be reasonable.

3. Use the simple lock with exponential backoff (with a cap on the maximum delay) if processes might be preempted while spinning, or if one-processor latency is an overwhelming concern and the hardware does not support fetch_and_increment (assuming of course that it *does* support test_and_set).

For barrier synchronization we suggest:

1. On a broadcast-based cache-coherent multiprocessor (with unlimited replication), use a barrier based on our 4-ary arrival tree and a central sense-reversing wakeup flag.

2. On a multiprocessor without coherent caches, or with directory-based coherency that limits the degree of replication, use either the dissemination barrier (with data structures distributed to respect locality) or our tree-based barrier with tree wakeup. The critical path through the dissemination barrier algorithm is about a third shorter than that of the tree barrier, but the total amount of interconnect traffic is $O(P \log P)$ instead of $O(P)$. The dissemination barrier will outperform the tree barrier on machines such as the Butterfly, which allow remote references from many different processors to proceed in parallel.

[17] Fetch_and_store and compare_and_swap, for example, are essential for manipulating pointers to build concurrent data structures [18, 27].

For the designers of large-scale shared-memory multiprocessors, our results argue in favor of providing distributed memory or coherent caches, rather than dance-hall memory without caches. Our results also indicate that combining networks must be justified on grounds other than the reduction of synchronization overhead. We strongly suggest that future multiprocessors include both `compare_and_swap` and a variety of `fetch_and_Φ` operations (especially `fetch_and_store`). We see no need for additional, special-purpose instructions designed for synchronization. Straightforward software techniques can eliminate contention due to busy-wait synchronization; hardware techniques are not required.

Acknowledgments

Tom LeBlanc and Evangelos Markatos provided helpful comments on an earlier version of this paper.

References

[1] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proc. of International Symposium on Computer Architecture*, pages 396–406, Jerusalem, Israel, May 1989.

[2] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proc. of International Symposium on Computer Architecture*, pages 280–289, NY, June 1988.

[3] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing.* Benjamin/Cummings, Redwood City, CA, 1989.

[4] T. E. Anderson. The performance implications of spin-waiting alternatives for shared memory multiprocessors. In *Proc. 1989 International Conference on Parallel Processing*, pages II–170–II–174, Aug. 1989.

[5] T. E. Anderson, E. D. Lazowska, and H. M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *Performance Evaluation Review*, 17(1):49–60, May 1989. SIGMETRICS '89 Conference Paper.

[6] G. R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, Redwood City, CA, 1990.

[7] J. Archibald and J.-L. Baer. An economical solution to the cache coherence problem. In *Proc. of International Symposium on Computer Architecture*, pages 355–362, 1984.

[8] BBN Laboratories. Butterfly parallel processor overview. Technical Report 6148, Version 1, BBN Laboratories, Cambridge, MA, Mar. 1986.

[9] E. D. Brooks III. The butterfly barrier. *International Journal of Parallel Programming*, 15(4):295–307, 1986.

[10] E. D. Brooks III. The shared memory hypercube. *Parallel Computing*, 6:235–245, 1988.

[11] H. Davis and J. Hennessy. Characterizing the synchronization behavior of parallel programs. In *Proc. of the ACM/SIGPLAN PPEALS 1988 Conference on Parallel Programming: Experience with Applications, Languages and Systems*, pages 198–211, New Haven, CT, July 1988.

[12] E. Dijkstra. Solution of a problem in concurrent programming and control. *Communications of the ACM*, 8(9):569, Sept. 1965.

[13] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proc. of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, Boston, MA, Apr. 1989.

[14] J. R. Goodman and P. J. Woest. The Wisconsin Multicube: A new large-scale cache coherent multiprocessor. In *The 15th Annual International Symposium on Computer Architecture*, pages 422–431, Honolulu, HI, May 1988.

[15] A. Gottlieb. Scalability, Combining and the NYU Ultracomputer. Ohio State University Parallel Computing Workshop, Mar. 1990. Invited Lecture.

[16] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer — designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, Feb. 1983.

[17] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988.

[18] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 197–206, Seattle, WA, Mar. 1990.

[19] D. N. Jayasimha. Distributed synchronizers. In *Proc. of the 1988 International Conference on Parallel Processing*, pages 23–27, St. Charles, IL, Aug. 1988.

[20] C. P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization on multiprocessors with shared memory. In *Proc. of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 218–228, 1986.

[21] M. Kumar and G. F. Pfister. The onset of hot spot contention. In *Proc. of the 1986 International Conference on Parallel Processing*, pages 28–34, 1986.

[22] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, Aug. 1974.

[23] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, Feb. 1987.

[24] J. Lee and U. Ramachandran. Synchronization with multiprocessor caches. In *Proc. of International Symposium on Computer Architecture*, Seattle, WA, May 1990.

[25] B. Lubachevsky. An approach to automating the verification of compact parallel coordination programs. I. *Acta Informatica*, pages 125–169, 1984.

[26] B. Lubachevsky. Synchronization barrier and related tools for shared memory parallel programming. In *Proc. 1989 International Conference on Parallel Processing*, pages II–175–II–179, Aug. 1989.

[27] J. M. Mellor-Crummey. Concurrent queues: Practical fetch-and-Φ algorithms. Technical Report 229, Computer Science Department, University of Rochester, Nov. 1987.

[28] P1596 Working Group of the IEEE Computer Society Microprocessor Standards Committee. SCI (scalable coherent interface): An overview of extended cache-coherence protocols, Feb. 5, 1990. Draft 0.59 P1596/Part III-D.

[29] G. L. Peterson. A new solution to Lamport's concurrent programming problem using small shared variables. *ACM Transactions on Programming Languages and Systems*, 5(1):56–65, Jan. 1983.

[30] G. Pfister et al. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proc. of the 1985 International Conference on Parallel Processing*, pages 764–771, St. Charles, Illinois, Aug. 1985.

[31] G. F. Pfister and V. A. Norton. "Hot spot" contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(10):943–948, Oct. 1985.

[32] R. D. Rettberg, W. R. Crowther, P. P. Carvey, and R. S. Tomlinson. The Monarch parallel processor hardware design. *Computer*, 23(4):18–30, Apr. 1990.

[33] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *Proc. of International Symposium on Computer Architecture*, pages 340–347, 1984.

[34] M. L. Scott, T. J. LeBlanc, and B. D. Marsh. Multi-model parallel programming in Psyche. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 70–78, Seattle, WA, Mar. 1990.

[35] P. Tang and P.-C. Yew. Processor self-scheduling for multiple-nested parallel loops. In *Proc. of the 1986 International Conference on Parallel Processing*, pages 528–535, St. Charles, IL, Aug. 1986.

[36] P. Tang and P.-C. Yew. Algorithms for distributing hot-spot addressing. CSRD report 617, Center for Supercomputing Research and Development, University of Illinois Urbana-Champaign, Jan. 1987.

[37] P.-C. Yew. Architecture of the Cedar parallel supercomputer. CSRD report 609, Center for Supercomputing Research and Development, University of Illinois Urbana-Champaign, Aug. 1986.

[38] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, Apr. 1987.

[39] J. Zahorjan, E. D. Lazowska, and D. L. Eager. The effect of scheduling discipline on spin overhead in shared memory parallel processors. Technical Report TR-89-07-03, Computer Science Department, University of Washington, July 1989.

A Derivation and Correctness of the MCS Spin Lock

To derive the MCS spin lock, we begin with a correct but impractical algorithm, then refine it through a series of correctness-preserving transformations into something that we can actually implement with existing hardware primitives. Throughout, we will use angle brackets (`<...>`) to enclose actions that are to be performed atomically. The idiom "`><`" within a atomic action will indicate that the action is to be broken into pieces at that point—*i.e.* that the current state will be externally visible and that atomic actions by other processors may occur before execution of the current processor continues. Assertions appear in braces.

The purpose of the atomic actions is to ensure that the lock data structure (*i.e.*, the queue) is always changed from one consistent state to another. In other words, consistent queue states can be characterized by invariants that are maintained by all atomic actions. Implicit in all the following code fragments is the statement that queue structure invariants hold between all atomic actions.

We begin with a version of the algorithm in which the entire `acquire_lock` and `release_lock` procedures are atomic actions, broken only where a processor waits (in `acquire_lock`) for the lock to become available. We assume that processors call `acquire_lock` and `release_lock` in strict alternation, that they call `acquire_lock` first, and that any processor that calls `acquire_lock` will eventually call `release_lock`.

```
type qlink = record
    next : ^qlink
    state : (outside, inside, waiting)

shared tail : ^qlink := nil
shared head : ^qlink := nil
processor private I : ^qlink
    // initialized to point to a queue link record
    // in the local portion of shared memory


procedure acquire_lock ()
{I->state = outside}
<
    I->next := nil
    predecessor := tail; tail := I
    if predecessor != nil
        I->state := waiting
        predecessor->next := I
        repeat >< until head = I        // spin
    else head := I
    I->state := inside
>
{I->state = inside}
```

```
procedure release_lock ()
{I->state = inside}
<
    if tail = I
        tail := nil
        head := nil
    else head := I->next
    I->state := outside
>
{I->state = outside}
```

The invariants on the queue data structure are: The lock is either free (`tail = head = nil`), or there is a linked list of processors from `head^` through `tail^`, arranged in the order in which they entered `acquire_lock`. Each processor on the list, other than the head, has a unique predecessor, and `tail->next` = `nil`. These invariants can be formalized and proven for the code above, but such an exercise adds little to the current discussion; people know how to write queues.

Since `I->state` is modified only by processor `I`, it cannot change between the return from `acquire_lock` and a subsequent call to `release_lock`. To demonstrate mutual exclusion, it therefore suffices to note that `I->state = inside` $\Longrightarrow$ `head = I` in every observable state of the system (that is, every state outside an atomic action). This implication follows from the fact that `acquire_lock` sets `I->state` to `inside` only if `head = I`, and `release_lock` merely sets `I->state` to `outside`.

To demonstrate lack of deadlock, we must show that `I->state` cannot equal `waiting` forever. We can argue this by induction on the distance between processor `I` and the head of the queue. If the distance is zero we have `head = I`, and the while loop in `acquire_lock` will terminate, allowing processor `I` to continue. If the distance is N, processor `I` will enter the while loop immediately after linking itself behind the $N-1$st processor, whom we can assume will enter state `inside` by inductive hypothesis. By assumption this $N-1$st processor will eventually call `release_lock`, at which point it will set `head` to `I`, and processor `I` will proceed.

These arguments can be made more rigorous, but we do not find the resulting sea of notation any more convincing. We proceed, therefore, with a modified version of the algorithm.

To eliminate spinning on the globally-shared head pointer, we introduce a new field in each `qlink`:

```
locked : Boolean
```

We set `I->locked = true` immediately before the while loop in `acquire_lock`. Thus when `I` falls into the loop we have `I->locked = true` and `head != I`. We also change `release_lock` so that it sets `I->next->locked = false` at the same time that it sets `head = I->next`. Since this code in `release_lock` is the only place that head can change when `tail != nil`, we have `(I->state = waiting)` $\Longrightarrow$ `(head = I <=> I->locked = false)`. We can therefore change the condition on the while loop in `acquire_lock` from `head != I` to `I->locked`. But this condition was the only place in the code where the value of head was inspected! With the changes just described, `head` becomes a useless variable and can be eliminated. Our code is now as follows:

```
procedure acquire_lock ()
{I->state = outside}
<
    I->next := nil
    predecessor := tail; tail := I
    if predecessor != nil
        I->state := waiting
        I->locked := true
        predecessor->next := I
        repeat >< while I->locked       // spin
    I->state := inside
>
{I->state = inside}


procedure release_lock ()
{I->state = inside}
<
    if tail = I
        tail := nil
    else I->next->locked := false
    I->state := outside
>
{I->state = outside}
```

In order to implement this algorithm on practical machines, we will need to break up the atomic actions (It makes no sense to use a lock to implement atomicity for the purpose of building a lock). We cannot, for example, link a newly-arrived processor simultaneously into both the tail of the queue and the processor's predecessor with standard atomic instructions. This means we are talking about breaking atomicity when the queue is in an inconsistent state—one in which processor `I` is not the head of the queue and yet is not pointed at by its predecessor. We note that the only way this state can arise is for processor `I` to be in the gap between the two halves of the atomic action in `acquire_lock`, in which case it will fix its predecessor's pointer before spinning. Let us represent this point in the code by letting `I->state = linking`.

When `I->state = linking`, `I` will already have linked itself into the tail of the queue, so when its predecessor calls `release_lock`, it will see that it is not at the tail of the queue and realize that it must have a successor. All that is required to restore the correctness of the algorithm is to (1) relax the invariant on queue structure to permit a non-tail element to have a `nil` next pointer, so long as its successor's state is `linking`, and (2) introduce a spin in `release_lock` to force the successor to wait for its next pointer to be updated before using it. Our code now looks like this:

```
procedure acquire_lock ()
{I->state = outside}
<
    I->state := linking
    I->next := nil
    predecessor := tail; tail := I
>
<
    if predecessor != nil
        I->state := waiting
        I->locked := true
        predecessor->next := I
        repeat >< while I->locked       // spin
    I->state := inside
>
{I->state = inside}


procedure release_lock ()
{I->state = inside}
<
    if tail = I
        tail := nil
    else                 // we have a successor
        repeat >
            { I->next = nil ==> (exists J:
                (J->state = linking AND J's predecessor variable is I)) }
            < while I->next = nil       // spin
        I->next->locked := false
    I->state := outside
>
{I->state = outside}
```

The new spin in `release_lock` cannot introduce deadlock because it happens only when `tail != I` and `I->state = inside`, but `I->next = nil`, that is, when I's successor's state is `linking`. Since I's successor is not blocked, it will set `I->next` to point to itself, and the spin in `release_lock` will terminate.

To reduce the size of the remaining atomic actions, we first note that `I->next` is irrelevant when `I->state = outside`, so the assignment `I->next := nil` can be moved outside the first atomic action in `acquire_lock`. If we interpret the release of atomicity in the waiting loop in `acquire_lock` to allow other atomic actions to occur before examining the loop condition for the first time (a reasonable interpretation) then the second atomic action in `acquire_lock` cannot interfere with other actions even if not atomic—it sets only one variable (`predecessor->next`) that is read by another processor, and reads no variables that are set by other processors. The bodies of the repeat loops in both routines are equally interference free. Finally, the `state` field of a `process_block` serves only to facilitate discussion, and is never inspected; it can be deleted. After deleting unnecessary atomicity and assignments to `I->state`, our code now looks like this:

```
procedure acquire_lock ()
    I->next := nil
<
    predecessor := tail; tail := I
>
    if predecessor != nil
        I->locked := true
        predecessor->next := I
        repeat while I->locked          // spin


procedure release_lock ()
<
    if tail = I
        tail := nil
>
    else                // we have a successor
        repeat while I->next = nil      // spin
        I->next->locked := false
```

We observe that the remaining atomic action in `acquire_lock` is a simple `fetch_and_store` operation, and the atomic action in `release_lock` is a `compare_and_swap`. We can re-write the code as follows:

```
procedure acquire_lock ()
    I->next := nil
    predecessor := fetch_and_store (tail, I)
    if predecessor != nil
        I->locked := true
        predecessor->next := I
        repeat while I->locked          // spin


procedure release_lock ()
    if not compare_and_swap (tail, I, nil)
        repeat while I->next = nil      // spin
        I->next->locked := false
```

To avoid unnecessary use of the `compare_and_swap` instruction in `release_lock`, we can note that `I->next != nil` $\Longrightarrow$ `tail != I`, so we can skip both the `compare_and_swap` and the while loop, and proceed directly to the update of `I->next->locked`, whenever `I->next != nil`. Making this change and modifying the calling sequence to permit the lock to be specified as a parameter results in the code of algorithm 4.

If a `compare_and_swap` instruction is not available, we will be unable to implement the remaining atomic action in `release_lock` as written. Assuming that `fetch_and_store` *is* available, it is tempting to use it to simulate `compare_and_swap` by reserving one bit pattern as an "invalid" value:[18]

[18] `Compare_and_swap` can also be used in a loop to simulate `fetch_and_store`, without the need for a special invalid value. A survey of current architectures reveals that each of these instructions is often provided without the other, but that `compare_and_swap` is the one more frequently omitted.

```
procedure fake_compare_and_swap (location, old_value, new_value)
    repeat
        current_value := fetch_and_store (location, invalid_value)
    until (current_value != invalid_value)
    if (current_value = old_value)
        location^ := new_value
        return true
    return false
```

This approach suffers from two problems, however. First, it includes a loop that would perform a potentially unbounded number of remote `fetch_and_store` operations in `release_lock`. Second, it requires the use of a similar loop in `acquire_lock`. Because of the possibility of finding an invalid value in the tail pointer, the line in `acquire_lock` that reads

```
predecessor := fetch_and_store (tail, I)
```

must be changed to

```
repeat
    predecessor = fetch_and_store (tail, invalid_value)
until (predecessor != invalid_value)
lock := I
```

Since the goal of the MCS lock is to eliminate spinning on remote locations, this "solution" is unacceptable. Instead, we pursue an optimistic implementation of `release_lock`, in which the calling processor `I` assumes that it never has a successor whose state is `linking`. When `I` finds that its next pointer is `nil`, it uses a single `fetch_and_store` operation to set the tail pointer to `nil`, thereby releasing the lock. If optimism was unwarranted, the return value from `fetch_and_store` will not be the expected value `I`, and recovery will be necessary. Fortunately, the queue can be repaired in constant time, at the expense of FIFO ordering of lock acquisitions.[19]

Code for the modified version of `release_lock`, without `compare_and_swap`, appeared in algorithm 5. To understand its recovery actions, consider the state of the queue in the event that the `fetch_and_store` reveals that `I` was not in fact the tail of the queue, so that a `compare_and_swap` operation would have failed. `Tail` will be `nil`. The processor to which `tail` *should* point will have been returned by `fetch_and_store`, and saved in local variable `old_tail`. `I`'s successor will eventually update `I->next`. To an independent caller of `acquire_lock`, the queue will appear to be empty, the lock unheld. The processor or processors from `I->next` through `old_tail` will have been mistakenly removed from the queue.

Recovery begins with a second `fetch_and_store` operation, to restore the tail pointer. If this second `fetch_and_store` returns `nil`, then the queue is again intact (though some independent processors may have acquired and released the lock in the meantime, out of FIFO order). If the second `fetch_and_store` returns a pointer other than `nil`, then one or more independent processors have queued up while the lock appeared to be free, and the first of them may be inside its critical section. We will permit all of these processors to acquire the lock before any of the processors we mistakenly removed from the queue. It is conceivable (though presumably very unlikely) that the

[19]In the symmetric case, where `compare_and_swap` is provided without `fetch_and_store`, we are unaware of any alternative to simulating `fetch_and_store` with a loop that busy-waits on the remote location.

last of the usurpers will attempt to release the lock before we have updated its next pointer. In this event the above scenario can repeat, and starvation is theoretically possible. The performance results in section 4.3 were obtained with the alternative version of the MCS lock, without `compare_and_swap`.