Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors

John Mellor-Crummey and Michael Scott

CRPC-TR91129
April 1991

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors*

John M. Mellor-Crummey†
(johnmc@rice.edu)
Center for Research on Parallel Computation
Rice University, P.O. Box 1892
Houston, TX 77251-1892

Michael L. Scott‡
(scott@cs.rochester.edu)
Computer Science Department
University of Rochester
Rochester, NY 14627

April 1991

Abstract

Reader-writer synchronization relaxes the constraints of mutual exclusion to permit more than one process to inspect a shared object concurrently, as long as none of them changes its value. On uniprocessors, mutual exclusion and reader-writer locks are typically designed to de-schedule blocked processes; however, on shared-memory multiprocessors it is often advantageous to have processes busy wait. Unfortunately, implementations of busy-wait locks on shared-memory multiprocessors typically cause memory and network contention that degrades performance. Several researchers have shown how to implement scalable mutual exclusion locks that exploit locality in the memory hierarchies of shared-memory multiprocessors to eliminate contention for memory and for the processor-memory interconnect. In this paper we present reader-writer locks that similarly exploit locality to achieve scalability, with variants for reader preference, writer preference, and reader-writer fairness. Performance results on a BBN TC2000 multiprocessor demonstrate that our algorithms provide low latency and excellent scalability.

1 Introduction

Busy-wait synchronization is fundamental to parallel programming on shared-memory multiprocessors. Busy waiting is generally preferred over scheduler-based blocking when scheduling overhead exceeds expected wait time, or when processor resources are not needed for other tasks (so that the lower wake-up latency of busy waiting need not be balanced against an opportunity cost).

*This paper appeared in *Proc. of the 3rd ACM Symposium on Principles and Practice of Parallel Programming*, pages 106–113, Williamsburg, VA, April 1991.

†Supported in part by the National Science Foundation under Cooperative Agreement number CCR-8809615.

‡Supported in part by the National Science Foundation under Institutional Infrastructure grant CDA-8822724.

Because busy-wait mechanisms are often used to protect very small, frequently-executed critical sections, their performance is a matter of paramount importance. Unfortunately, typical implementations of busy waiting tend to produce large amounts of contention for memory and communication bandwidth, introducing performance bottlenecks that become markedly more pronounced in larger machines and applications. When many processors busy-wait on a single synchronization variable, they create a *hot spot* that gets a disproportionate share of the processor-memory bandwidth. Several studies [1, 4, 10] have identified synchronization hot spots as a major obstacle to high performance on machines with both bus-based and multi-stage interconnection networks.

Recent papers, ours among them [9], have addressed the construction of scalable, contention-free busy-wait locks for mutual exclusion. These locks employ atomic `fetch_and_`Φ instructions[1] to construct queues of waiting processors, each of which spins only on *locally-accessible* flag variables, thereby inducing no contention. In the locks of Anderson [2] and Graunke and Thakkar [5], which achieve local spinning only on cache-coherent machines, each blocking processor chooses a unique location on which to spin, and this location becomes resident in the processor's cache. Our MCS mutual exclusion lock (algorithm 1) exhibits the dual advantages of (1) spinning on locally-accessible locations even on distributed shared-memory multiprocessors without coherent caches, and (2) requiring only $O(P + N)$ space for N locks and P processors, rather than $O(NP)$.

```
type qnode = record
    next : ^qnode      // ptr to successor in queue
    locked : Boolean   // busy-waiting necessary
type lock = ^qnode     // ptr to tail of queue

// I points to a qnode record allocated
// (in an enclosing scope) in shared memory
// locally-accessible to the invoking processor

procedure acquire_lock(L : ^lock; I : ^qnode)
    var pred : ^qnode
    I->next := nil          // initially, no successor
    pred := fetch_and_store(L, I) // queue for lock
    if pred != nil          // lock was not free
        I->locked := true // prepare to spin
        pred->next := I   // link behind predecessor
        repeat while I->locked // busy-wait for lock

procedure release_lock(L : ^lock; I : ^qnode)
    if I->next = nil      // no known successor
        if compare_and_swap(L, I, nil)
            return        // no successor, lock free
        repeat while I->next = nil // wait for succ.
    I->next->locked := false       // pass lock
```

Algorithm 1: The MCS queue-based spin lock. An alternative version of `release_lock` can be written without `compare_and_swap`, but it sacrifices FIFO ordering under load.

Mutual exclusion is a sufficient mechanism for most forms of synchronization, but it introduces serialization that is not always necessary. Reader-writer synchronization, as described by Courtois, Heymans, and Parnas [3], relaxes the constraints of mutual exclusion to permit more than one process to inspect a shared data structure simultaneously, so long as none of them modifies it. Operations are separated into two classes: *writes*, which require exclusive access while modifying the data structure, and *reads*, which can be concurrent with one another (though not with writes) because they make no observable changes.

As recognized by Courtois, Heymans, and Parnas, different fairness properties are appropriate

[1] A `fetch_and_`Φ operation [7] reads, modifies, and writes a memory location atomically. Several common `fetch_and_`Φ operations are defined in appendix A.

for a reader-writer lock depending on the context in which it is used. A "reader preference" lock minimizes the delay for readers and maximizes total throughput by allowing a reading process to join a group of current readers even if a writer is waiting. A "writer preference" lock ensures that updates are seen as soon as possible by requiring readers to wait for any current or waiting writer, even if other processes are currently reading. Both of these options permit indefinite postponement and even starvation of non-preferred processes when competition for the lock is high. Though not explicitly recognized by Courtois, Heymans, and Parnas, it is also possible to construct a reader-writer lock (called a "fair" lock here) in which readers wait for any earlier writer and writers wait for any earlier process.

The reader and writer preference locks presented by Courtois, Heymans, and Parnas use semaphores for scheduler-based blocking. Most multiprocessor implementations of semaphores, in turn, depend on a busy-wait mutual exclusion lock. As noted above, there are circumstances in which scheduler-based blocking is inappropriate: specifically when the expected wait time is very short or when the processor has nothing else to do. Moreover on a multiprocessor when competition is high, even the serialization implied by a mutually-exclusive implementation of semaphores may itself be a performance problem. These observations suggest the need for reader-writer spin locks. Unfortunately, naive implementations of such locks are just as prone to contention as naive implementations of traditional spin locks.

Our contribution in this paper is to demonstrate that the local-only spinning property of our mutual exclusion spin lock can be obtained as well for reader-writer locks. All that our algorithms require in the way of hardware support is a simple set of `fetch_and_`Φ operations and a memory hierarchy in which each processor is able to read some portion of shared memory without using the interconnection network.

Section 2 discusses simple approaches to reader-writer spin locks, including a reader preference lock in which processors attempt to induce state changes in a central flag word, and a fair lock in which they wait for the appearance of particular values in a pair of central counters. Section 3 presents three new algorithms for reader-writer spin locks without embedded mutual exclusion, and with local-only spinning. One of these latter locks is fair; the others implement reader and writer preference. In section 4 we present performance results for our locks on the BBN TC2000, a distributed shared-memory multiprocessor with a rich set of `fetch_and_`Φ instructions. Our results indicate that reader-writer spin locks with local-only spinning can provide excellent performance in both the presence and absence of heavy competition, with no fear of contention on very large machines. We summarize our conclusions in section 5.

2 Simple Reader-Writer Spin Locks

In this section we present simple, centralized algorithms for busy-wait reader-writer locks with two different fairness conditions. The algorithms for these protocols are constructed using atomic operations commonly available on shared-memory multiprocessors. They provide a naive base against which to compare local-spinning reader-writer locks, much as a traditional `test_and_set` lock provides a base against which to compare local-spinning mutual exclusion locks. Our pseudo-code notation is meant to be more-or-less self explanatory. Line breaks terminate statements (except in obvious cases of run-on), and indentation indicates nesting in control constructs. Definitions of atomic operations appear in appendix A.

2.1 A Reader Preference Lock

Algorithm 2 implements a simple reader preference spin lock. It uses an unsigned integer to represent the state of the lock. The lowest bit indicates whether a writer is active; the upper bits contain a count of active or interested readers. When a reader arrives, it increments the reader count (atomically) and waits until there are no active writers. When a writer arrives, it attempts to acquire the lock by using `compare_and_swap`. (`Compare_and_swap` tests if the value in a memory location is equal to a given 'old' value and sets a Boolean condition code; if the two values are equal, a specified 'new' value is written into the memory location.) The writer succeeds, and proceeds, only when all bits were clear, indicating that no writers were active and that no readers were active or interested.

```
type lock = unsigned integer
  // layout of lock
  //  31          ...          1         0
  // +-------------------------+----------------+
  // | interested reader count | active writer? |
  // +-------------------------+----------------+

const WAFLAG = 0x1  // writer active flag
const RC_INCR = 0x2 // to adjust reader count

procedure start_write(L : ^lock)
    repeat until compare_and_swap(L, 0, WAFLAG)

procedure start_read(L : ^lock)
    atomic_add(L, RC_INCR)
    repeat until (L^ & WAFLAG) = 0

procedure end_write(L : ^lock)
    atomic_add(L, -WAFLAG)

procedure end_read(L : ^lock)
    atomic_add(L, -RC_INCR)
```

Algorithm 2: A simple reader preference lock.

Reader preference makes sense when competition for a lock is bursty, but not continuous. It allows as many processors as possible to proceed in parallel at all times, thereby maximizing throughput. If heavy, continuous competition for a reader-writer lock is expected, and it is not possible to redesign the application program to avoid this situation, then a fair reader-writer lock must be used to eliminate the possibility of starvation of one class of operations. We present such a lock in the following section.

2.2 A Fair Lock

Algorithm 3 implements fair, simple, reader-writer synchronization by generalizing the concept of a ticket lock [9] to the reader-writer case. Like Lamport's Bakery lock [8], a mutual-exclusion ticket lock is based on the algorithm employed at service counters to order arriving customers. Each customer obtains a number larger than that of previous arriving customers, and waits until no waiting customer has a lower number. The ticket lock generates numbers with `fetch_and_add` operations, copes with arithmetic wrap-around, and compares tickets against a central `now_serving` counter. In our simple, fair reader-writer lock, separate counters track the numbers of read and write requests.

We use `fetch_clear_then_add` and `clear_then_add` operations to cope with arithmetic overflow. Applied to the value of a memory location, both operations first clear the bits specified by a mask word, then add to the results the value of an addend. The sum replaces the old value in the

memory location. `Fetch_clear_then_add` returns the value the memory location had prior to the operation; `clear_then_add` does not. Ordinary `fetch_and_add` operations can be used to build a fair reader-writer lock, but the code is slightly more complicated.

We represent the state of our lock with a pair of 32-bit unsigned integers. Each integer represents a pair of counters: the upper 16 bits count readers; the lower 16 bits count writers. The request counters indicate how many readers and writers have requested the lock. The completion counters indicate how many have already acquired and released the lock. Before each increment of one of the 16 bit counts, the top bit is cleared to prevent it from overflowing its allotted 16 bits. Tickets are strictly ordered, beginning with the values in the completion counters, proceeding upward to 2^{15}, and cycling back to 1. With fewer than 2^{15} processes, all outstanding tickets are guaranteed to be distinct. Readers spin until all earlier write requests have completed. Writers spin until all earlier read and write requests have completed.

```
type counter = unsigned integer
   // layout of counter
   //  31    ...   16 15    ...     0
   // +-----------------------------+
   // | reader count | writer count  |
   // +-----------------------------+

const RC_INCR = 0x10000 // to adjust reader count
const WC_INCR = 0x1     // to adjust writer count
const W_MASK  = 0xffff  // to extract writer count

// mask bit for top of each count
const WC_TOPMSK = 0x8000
const RC_TOPMSK = 0x80000000

type lock = record
    requests : unsigned integer := 0
    completions : unsigned integer := 0

procedure start_write(L : ^lock)
    unsigned integer prev_processes :=
        fetch_clear_then_add(&L->requests,
            WC_TOPMSK, WC_INCR)
    repeat until completions = prev_processes

procedure start_read(L : ^lock)
    unsigned integer prev_writers :=
        fetch_clear_then_add(&L->requests, RC_TOPMSK,
            RC_INCR) & W_MASK
    repeat until (completions & W_MASK) = prev_writers

procedure end_write(L: ^lock)
    clear_then_add(&L->completions, WC_TOPMSK, WC_INCR)

procedure end_read(L : ^lock)
    clear_then_add(&L->completions, RC_TOPMSK, RC_INCR)
```

Algorithm 3: A simple fair reader-writer lock.

The principal drawback of algorithms 2 and 3 is that they spin on shared locations. References generated by a large number of competing processors will lead to a hot spot that can severely degrade performance. Interference from waiting processors increases the time required to release the lock when the current holder is done, and also degrades the performance of any process that needs to make use of the memory bank in which the lock resides, or of conflicting parts of the interconnection network. In the next section, we present more complex reader-writer locks that eliminate this problem via local-only spinning.

3 Locks with Local-Only Spinning

In order to implement reader-writer spin locks without inducing contention for memory or communication bandwidth, we need to devise algorithms that spin on local variables. One obvious approach is to use the MCS spin lock (algorithm 1) to protect critical sections that manipulate a second layer of reader-writer queues. There are two principal drawbacks to this approach. First, solutions based on mutual exclusion introduce non-trivial amounts of serialization for concurrent readers. Second, even in the absence of lock contention the two-level structure of the resulting locks leads to code paths that are unnecessarily long.

A more attractive approach is to design single-level locks that implement reader-writer control with lower latency and less serialization. We present three versions here: one that provides fair access to both readers and writers, one that grants preferential access to readers, and one that grants preferential access to writers. All three avoid the memory and interconnection network contention common to centralized busy waiting strategies by spinning only on locally accessible per-processor variables, allocated in a scope that encloses their use.

3.1 A Fair Lock

Our fair reader-writer lock (algorithm 4) requires atomic `fetch_and_store` and `compare_and_swap` instructions. A read request is granted when all previous write requests have completed. A write request is granted when all previous read and write requests have completed.

As in the MCS spin lock, we use a linked list to keep track of requesting processors. In this case, however, we allow a requestor to read and write fields in the link record of its predecessor (if any). To ensure that the record is still valid (and has not been deallocated or reused), we require that a processor access its predecessor's record *before* initializing the record's `next` pointer. At the same time, we force every processor that has a successor to wait for its `next` pointer to become non-`nil`, even if the pointer will never be used. As in the MCS lock, the existence of a successor is determined by examining `L->tail`.

A reader can begin reading if its predecessor is a reader that is already active, but it must first unblock its successor (if any) if that successor is a waiting reader. To ensure that a reader is never left blocked while its predecessor is reading, each reader uses `compare_and_swap` to *atomically* test if its predecessor is an active reader, and if not, notify its predecessor that it has a waiting reader as a successor.

Similarly, a writer can proceed if its predecessor is done and there are no active readers. A writer whose predecessor is a writer can proceed as soon as its predecessor is done, as in the MCS lock. A writer whose predecessor is a reader must go through an additional protocol using a count of active readers, since some readers that started earlier may still be active. When the last reader of a group finishes (`reader_count = 0`), it must resume the writer (if any) next in line for access. This may require a reader to resume a writer that is not its direct successor. When a writer is next in line for access, we write its name in a global location. We use `fetch_and_store` to read and erase this location atomically, ensuring that a writer proceeds on its own if and only if no reader is going to try to resume it. To make sure that `reader_count` never reaches zero prematurely, we increment it *before* resuming a blocked reader, and before updating the `next` pointer of a reader whose reading successor proceeds on its own.

```
type qnode = record
    class : (reading, writing)
    next : ^qnode
    state : record
        blocked : Boolean    // need to spin
        successor_class : (none, reader, writer)
type lock = record
    tail : ^qnode := nil
    reader_count : integer := 0
    next_writer : ^qnode := nil

// I points to a qnode record allocated (in an enclosing scope) in shared
// memory locally-accessible to the invoking processor

procedure start_write(L : ^lock; I : ^qnode)
    with I^, L^
        class := writing; next := nil; state := [true, none]
        pred : ^qnode := fetch_and_store(&tail, I)
        if pred = nil
            next_writer := I
            if reader_count = 0 and fetch_and_store(&next_writer,nil)=I
                // no reader who will resume me
                blocked := false
        else
            // must update successor_class before updating next
            pred->successor_class := writer
            pred->next := I
        repeat while blocked

procedure end_write(L: ^lock; I : ^qnode)
    with I^, L^
        if next != nil or not compare_and_swap(&tail, I, nil)
            repeat while next = nil   // wait until succ inspects my state
            if next->class = reading
                atomic_increment(&reader_count)
            next->blocked := false

procedure start_read(L : ^lock; I : ^qnode)
    with I^, L^
        class := reading; next := nil; state := [true,none]
        pred : ^qnode := fetch_and_store(&tail, I)
        if pred = nil
            atomic_increment(&reader_count)
            blocked := false  // for successor
        else
            if pred->class = writing or
                    compare_and_swap(&pred->state, [true,none], [true,reader])
                // pred is a writer, or a waiting reader. pred will increment
                // reader_count and release me
                pred->next := I
                repeat while blocked
            else // increment reader_count and go
                atomic_increment(&reader_count)
                pred->next := I
                blocked := false
        if successor_class = reader
            repeat while next = nil
            atomic_increment(&reader_count)
            next->blocked := false

procedure end_read(L : ^lock; I : ^qnode)
    with I^, L^
        if next != nil or not compare_and_swap(&tail, I, nil)
            repeat while next = nil     // wait until succ inspects my state
            if successor_class = writer
                next_writer := next
        if fetch_and_decrement(&reader_count) = 1
            // I'm last reader, wake writer if any
            w : ^qnode := fetch_and_store(&next_writer, nil)
            if w != nil
                w->blocked := false
```

Algorithm 4: A fair reader-writer lock with local-only spinning.

3.2 A Reader Preference Lock

In our reader preference lock (algorithm 5) we separate a list of waiting readers from the queue of waiting writers. We also employ a flag word that allows processes to discover and, if appropriate, modify the state of the lock atomically. The queues are manipulated with `fetch_and_store` and `compare_and_swap` instructions; the flag word is manipulated with `fetch_and_or`, `fetch_and_and`, and `fetch_and_add` instructions. It contains one bit to indicate that one or more writers are waiting (but that none is currently active), another bit to indicate that a writer is currently active, and a count of the number of readers that are either waiting or active. The active and waiting bits for writers are mutually exclusive. A reader cannot proceed if the active writer bit is set. A writer cannot proceed if the reader count is non-zero.

To avoid race conditions between modification of the flag word and manipulation of the reader list, each reader must double-check the flag word after adding itself to the list. Writers insert themselves in their queue before inspecting the flag word. A process can (and must!) unblock appropriate processes (possibly including itself) when it causes the following transitions in `rdr_cnt_and_flags`:

reader count = 0 and writer interested
  → writer active;
(reader count > 0 and writer active) or
(reader count = 0 and writer not active)
  → reader count > 0 and writer not active.

```
type qnode = record
    next : ^qnode
    blocked : Boolean
type RPQlock = record
    reader_head : ^qnode := nil
    writer_tail : ^qnode := nil
    writer_head : ^qnode := nil
    rdr_cnt_and_flags : unsigned integer := 0

// layout of rdr_cnt_and_flags:
//  31     ...       2        1              0
// +-----------------+------------+-----------------+
// | interested rdrs | active wtr? | interested wtr? |
// +-----------------+------------+-----------------+

const WIFLAG = 0x1  // writer interested flag
const WAFLAG = 0x2  // writer active flag
const RC_INCR = 0x4 // to adjust reader count

// I points to a qnode record allocated
// (in an enclosing scope) in shared memory
// locally-accessible to the invoking processor

procedure start_write (L : ^RPQlock, I : ^qnode)
    with I^, L^
        blocked := true; next := nil
        pred: ^qnode := fetch_and_store(&writer_tail,I)
        if pred = nil
            writer_head := I
            if fetch_and_or(&rdr_cnt_and_flag, WIFLAG)
                    = 0
                if compare_and_swap(&rdr_cnt_and_flag,
                        WIFLAG, WAFLAG)
                    return
            // else readers will wake up the writer
        else
            pred->next := I
        repeat while blocked
```

```
procedure end_write(L: ^RPQlock, I : ^qnode)
    with I^, L^
        writer_head := nil
        // clear wtr flag and test for waiting rdrs
        if fetch_and_and(&rdr_cnt_and_flag, ~WAFLAG)
                != 0
            // waiting readers exist
            head : ^qnode :=
                fetch_and_store(&reader_head, nil)
            if head != nil
                head->blocked := false
        // testing next is strictly an optimization
        if next != nil or not
                compare_and_swap(&writer_tail, I, nil)
            repeat while next = nil  // resolve succ
            writer_head := next
            if fetch_and_or(&rdr_cnt_and_flag,WIFLAG)
                    = 0
                if compare_and_swap(&rdr_cnt_and_flag,
                        WIFLAG, WAFLAG)
                    writer_head->blocked := false
            // else readers will wake up the writer

procedure start_read(L : ^RPQlock, I : ^qnode)
    with I^, L^
        // incr reader count, test if writer active
        if fetch_and_add(&rdr_cnt_and_flag, RC_INCR) &
                WAFLAG
            blocked := true
            next := fetch_and_store(&reader_head, I)
            if (rdr_cnt_and_flag & WAFLAG) = 0
                // writer no longer active
                // wake any waiting readers
                head : ^qnode :=
                    fetch_and_store(&reader_head, nil)
                if head != nil
                    head->blocked := false
            repeat while blocked            // spin
            if next != nil
                next->blocked := false

procedure end_read(L : ^RPQlock, I : ^qnode)
    with I^, L^
        // if I am the last reader, resume the first
        // waiting writer (if any)
        if fetch_and_add(&rdr_cnt_and_flag, -RC_INCR) = (RC_INCR + WIFLAG)
            if compare_and_swap(&rdr_cnt_and_flag,
                    WIFLAG, WAFLAG)
                writer_head->blocked := false
```

Algorithm 5: A reader preference lock with local-only spinning.

3.3 A Writer Preference Lock

Like the reader preference lock, our writer preference lock (algorithm 6) separates a list of waiting readers from the queue of waiting writers, and uses a combination flag and count word to keep track of the state of the lock. Where the reader preference lock counted interested readers and kept flags for active and interested writers, the writer preference lock counts *active* readers, and keeps flags for interested readers and active or interested writers. This change reflects the fact that readers in a writer preference lock cannot indiscriminately join a current reading session, but must wait for any earlier-arriving writers. To avoid a timing window in `start_read`, active and interested writers are indicated by a *pair* of flags, the first of which is always set by the first arriving writer, and the second of which is set only when no reader is in the process of joining a current reading session. A reader cannot proceed if either of the writer flags are set when it sets the reader flag. A writer cannot proceed if the reader count is non-zero or if a reader has set the reader flag when no writers

were active or waiting.

The `fetch_and_or` in `start_read` allows a reader to simultaneously indicate interest in the lock and check for active writers. If there are no active writers, the reader is considered to have begun a reading session; it can safely use a `fetch_and_add` to clear the reader interest bit and increment the active reader count. A writer that arrives after the reader's first atomic operation will enqueue itself and wait, even though there are no writers in line ahead of it. A reader that joins a current reading session sets the second writer flag on behalf of any writer that may have attempted to acquire the lock while the reader was joining the session.

A reader that finds no other readers waiting (`reader_head = nil`) uses `fetch_and_store` to obtain a pointer to the most recent arriving reader, whom it unblocks. In most cases it will unblock itself, but if two readers arrive at approximately the same time this need not be the case. A reader that is not the first to arrive, or that discovers active or interested writers, always waits to be resumed.

```
type qnode : record
    next : ^qnode
    blocked : Boolean

type WPQlock = record
    reader_head : ^qnode := nil
    writer_tail : ^qnode := nil
    writer_head : ^qnode := nil
    rdr_cnt_and_flags : unsigned integer := 0

// layout of rdr_cnt_and_flags:
//  31   ...    3          2          1          0
// +-------------+----------+----------+------+
// | active rdrs | int. rdr? | wtr & no rdr? | wtr? |
// +-------------+----------+----------+------+

const WFLAG1 = 0x1  // writer interested or active
const WFLAG2 = 0x2  // writer, no entering rdr
const RFLAG = 0x4   // rdr int. but not active
const RC_INCR = 0x8 // to adjust reader count

// I points to a qnode record allocated (in an enclosing scope) in shared
// memory locally-accessible to the invoking processor

procedure start_write(L : ^WPQlock, I : ^qnode)
    with I^, L^
        blocked := true; next := nil
        pred : ^qnode := fetch_and_store(&writer_tail, I)
        if pred = nil
            set_next_writer(L, I)
        else
            pred->next := I
        repeat while blocked

procedure set_next_writer(L : ^WPQlock, W : ^qnode)
    with L^
        writer_head := W
        if not (fetch_and_or(&rdr_cnt_and_flags,WFLAG1)
                & RFLAG)
            // no reader in timing window
            if not (fetch_and_or(&rdr_cnt_and_flags, WFLAG2) >= RC_INCR)
                // no readers are active
                W->blocked := false
```

```
procedure start_read(L : ^WPQlock, I : ^qnode)
    with I^, L^
        blocked := true
        next := fetch_and_store(&reader_head, I)
        if next = nil
            // first arriving reader in my group
            // set rdr interest flag, test writer flag
            if not (fetch_and_or(&rdr_cnt_and_flags, RFLAG) & (WFLAG1 + WFLAG2))
                // no active or interested writers
                unblock_readers(L)
        repeat while blocked
        if next != nil
            atomic_add(&rdr_cnt_and_flags, RC_INCR)
            next->blocked := false // wake successor

procedure unblock_readers(L : ^WPQlock)
    with L^
        // clear rdr interest flag, increment rdr count
        atomic_add(&rdr_cnt_and_flags, RC_INCR - RFLAG)
        // indicate clear of window
        if (rdr_cnt_and_flags & WFLAG1) and not (rdr_cnt_and_flags & WFLAG2)
            atomic_or(&rdr_cnt_and_flags, WFLAG2)
        // unblock self and any other waiting rdrs
        head : ^qnode :=
            fetch_and_store(&reader_head, nil)
        head->blocked := false

procedure end_write(L : ^WPQlock, I : ^qnode)
    with I^, L^
        if next != nil
            next->blocked := false
        else
            // clear wtr flag, test rdr interest flag
            if fetch_and_and(&rdr_cnt_and_flags, ~(WFLAG1 + WFLAG2)) & RFLAG
                unblock_readers(L)
            if compare_and_swap(&writer_tail, I, nil)
                return
            else
                repeat while next = nil
                set_next_writer(L, next)

procedure end_read(L : ^WPQlock, I : ^qnode)
    with I^, L^
        if (fetch_and_add(&rdr_cnt_and_flags, -RC_INCR)
                & ~RFLAG) = (RC_INCR + WFLAG1 + WFLAG2)
            // last active rdr must wake waiting writer
            writer_head->blocked := false
        // if only WFLAG1 is set and not WFLAG2, then
        // the writer that set it will take care of
        // itself
```

Algorithm 6: A writer preference lock with local-only spinning.

4 Empirical Performance Results

We have measured the performance of C language versions of our reader-writer algorithms on a BBN Butterfly TC2000, a distributed shared-memory multiprocessor based on the MC88100 microprocessor. Atomic read, write, and **fetch_and_store** instructions are triggered directly by the instruction set of the processor; other atomic operations are triggered by kernel calls, which use a privileged architectural feature to provide extended atomicity.

The implemented versions of the local-spin reader preference and writer preference algorithms differ slightly from those presented in section 3. Adjustments to the algorithms were made to improve the single processor latency, generally by means of an initial **compare_and_swap** that bypasses large parts of the protocol when it determines that no interfering operations are in progress. Although these adjustments improve the single processor latency, they add to the cost in the case

Lock type	start_read/ end_read	start_write/ end_write
simple reader pref.	22.3 μs	20.2 μs
simple fair	22.3 μs	22.3 μs
local-spin fair	44.5 μs	29.6 μs
local-spin reader pref.	20.3 μs	22.5 μs
local-spin writer pref.	20.2 μs	36.6 μs

Table 1: Single processor latencies for each pair of lock operations.

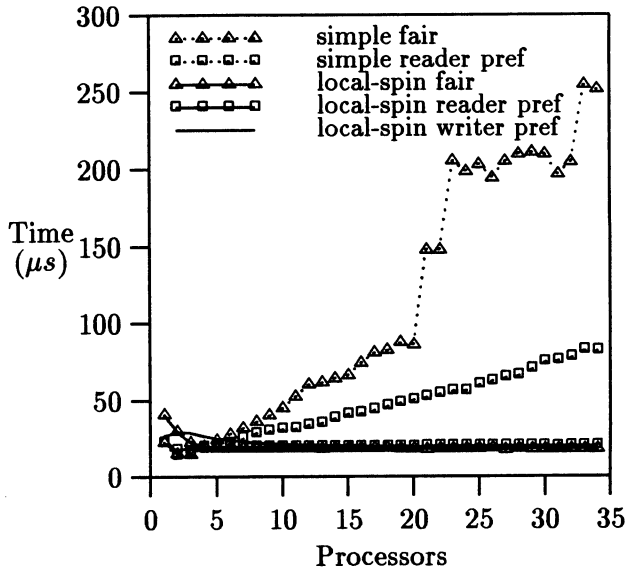

Figure 1: Timing results for reader-writer spin locks on the BBN TC2000.

that multiple processes are competing for the lock. Anyone wishing to reproduce our results or extend our work to other machines can obtain copies of our source code via anonymous ftp from titan.rice.edu (/public/scalable_synch/TC2000/read_write).

Our results were obtained by embedding a lock acquisition/release pair inside a loop and averaging over 10^5 operations. Table 1 reports the single processor latency for each of the lock operations in the absence of competition. Reported values are slightly pessimistic in that the overhead of the test loop was not factored out of the timings.

Figure 1 compares the performance of the simple reader-writer protocols with centralized busy waiting versus their local-spin counterparts. Each data point (P,T) represents the average time T for an individual processor to acquire and release the lock once, with P processors competing for access. Each processor performs a mixture of read and write requests. We used a pseudo-random number generator (off-line) to generate request sequences in which reads outnumber writes 3–1. As expected, the queue based locks show excellent performance. The single processor latencies of the local-spin protocols are comparable to those of the far simpler centralized protocols. As the number of processors competing for a lock increases, the advantage of the local-spin protocols becomes more apparent. In the simple centralized protocols, waiting processors continuously poll the lock status generating intense memory and network contention that slows the progress of active processors. In the local-spin protocols, waiting processors generate no network traffic, keeping them out of the way of active processors. Figure 2 shows an expanded view of the performance of the local-spin algorithms. Initially, the average time for an acquire and release pair drops as more processors

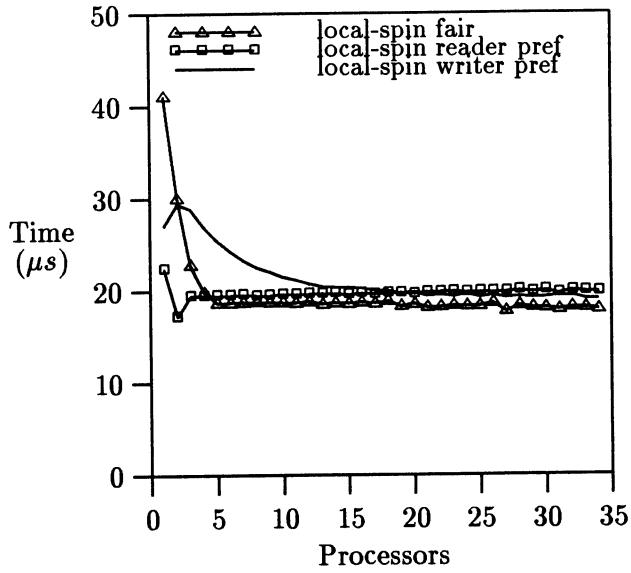

Figure 2: Expanded timing results for scalable reader-writer spin locks on the BBN TC2000.

compete for the lock. This effect results from overlapped execution of portions of the entry and exit protocols and from increases in the average number of simultaneous readers (particularly in the writer preference lock).

5 Discussion and Conclusions

We have demonstrated in this paper that simple `fetch_and_`Φ operations, in conjunction with local access to shared memory, can lead to reader-writer spin locks in which memory and interconnect contention due to busy waiting is non-existent. Our queue-based algorithms provide excellent single-processor latency in the absence of competition, and work well on distributed shared memory multiprocessors. Together with similar findings for mutual exclusion spin locks and barriers [9], this result indicates that contention due to busy-wait synchronization is much less a problem than has generally been thought.

Our algorithms require two forms of hardware support. First, they require a modest set of `fetch_and_`Φ operations. (Our fair queue-based lock requires `compare_and_swap` and `fetch_and_store`. Our queue-based reader and writer preference locks also require `fetch_and_or`, `fetch_and_and`, and `fetch_and_add`.) Second, they require that for every processor there be some region of shared memory that can be inspected locally, without going through the interconnection network, but which nonetheless can be modified remotely. This requirement is satisfied by all cache-coherent architectures, as well as by architectures in which shared memory is distributed. Other things being equal, the efficiency and scalability of our algorithms suggest that machines be constructed with these attributes—that they not skimp on the atomic operations, nor adopt "dance-hall" architectures in which all processors access shared locations through a common global interconnect. Our experience with `fetch_and_`Φ operations, particularly `fetch_and_store` and `compare_and_swap`, is consistent with the results of Herlihy [6] and of Kruskal, Rudolph, and Snir [7], who have found them valuable in the construction of a wide range of concurrent data structures.

Reader or writer preference locks are likely to be the mechanism of choice in many applications. While more complex than mutual exclusion locks, our algorithms have similar latency in the single-processor case, and admit more parallelism under load. Reader preference locks maximize

throughput; writer preference locks prevent readers from seeing outdated information. Fair locks ensure that neither readers nor writers are locked out when competition remains high for an extended period of time, though an application in which this situation can arise is probably poorly designed.

References

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

[2] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, Jan. 1990.

[3] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with 'readers' and 'writers'. *Communications of the ACM*, 14(10):667–668, Oct. 1971.

[4] H. Davis and J. Hennessy. Characterizing the synchronization behavior of parallel programs. In *Proceedings of the ACM Conference on Parallel Programming: Experience with Applications, Languages and Systems*, pages 198–211, July 1988.

[5] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *Computer*, 23(6):60–69, June 1990.

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

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

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

[9] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, to appear. Earlier version published as TR 342, Computer Science Department, University of Rochester, April 1990, and COMP TR90-114, Department of Computer Science, Rice University, May 1990.

[10] 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.

A Definitions of Atomic Operations

Our algorithms rely on the atomicity of **reads** and **writes** of 8, 16 and 32 bit quantities as well as the following atomic operations:

```
atomic_add(p: ^word, i: word): void
    p^ := p^ + i

atomic_decrement(p: ^word): void
    p^ := p^ - 1

atomic_increment(p: ^word): void
    p^ := p^ + 1

clear_then_add(p: ^word; m, i: word): void
    p^ := (p^ & ~m) + i

compare_and_swap(p: ^word; o, n: word): boolean
    cc: boolean := (p^ = o)
    if cc
        p^ := n
    return cc

fetch_and_add(p: ^word, i: word): word
    temp: word := p^
    p^ := p^ + i
    return temp

fetch_and_and(p: ^word, i: word): word
    temp: word := p^
    p^ := p^ & i
    return temp

fetch_and_decrement(p: ^word): word
    temp: word := p^
    p^ := p^ - 1
    return temp

fetch_and_or(p: ^word, i: word): word
    temp: word := p^
    p^ := p^ | i
    return temp

fetch_and_store(p: ^word, i: word): word
    temp: word := p^
    p^ := i
    return temp

fetch_clear_then_add(p: ^word; m, i: word): word
    temp: word := p^
    p^ := (p^ & ~m) + i
    return temp
```

The atomic operations described above execute indivisibly with respect to each other and with respect to reads and writes. Our algorithms use the simplest operation that provides the necessary functionality. For example, we use **atomic_add** rather than **fetch_and_add** whenever the return value is not needed. With the exception of **compare_and_swap**, the functionality of **fetch_clear_then_add** subsumes that of all of the other atomic operations listed.