A Fair Fast Scalable Reader-Writer Lock

Orran Krieger, Michael Stumm, Ron Unrau, and Jonathan Hanna
Department of Electrical and Computer Engineering
University of Toronto, Toronto, Canada, M5S 1A4

1 INTRODUCTION

A reader-writer (RW) lock allows either multiple readers to inspect shared data or a single writer exclusive access for modifying that data. On shared memory multiprocessors, the cost of acquiring and releasing these locks can have a large impact on the performance of parallel applications. A major problem with naive implementations of these locks, where processors *spin* on a global lock variable waiting for the lock to become available, is that the memory containing the lock and the interconnection network to that memory will also become contended when the lock is contended.

Several researchers have shown how to implement scalable exclusive locks, that is, exclusive locks that can become contended without resulting in memory or interconnection network contention [1, 2, 5]. These algorithms depend either on cache hardware support or on the existence of *local* memory, where accesses to local memory involve lower latency than accesses to *remote* memory (and involve no network traffic).

Mellor-Crummey and Scott[6] recognized the need for scalable RW locks and developed an implementation for the BBN TC2000. Their results indicate that their algorithm performs well, however, it depends on the rich set of atomic operations provided by the BBN TC2000. In particular: 1) atomic write operations for 8, 16 and 32 bit quantities, 2) atomic `fetch_and_store` instructions, 3) atomic `compare_and_swap` instructions, and 4) `atomic_increment` and `atomic_decrement` instructions.

In this paper we describe a new fair RW locking algorithm with similar goals to that developed by Mellor-Crummey and Scott. However, our algorithm has three major advantages over their algorithm. First, in the common case of an uncontended lock, our algorithm is faster since it requires fewer atomic operations and memory references. Second, their algorithm depends on three global variables that must be accessed atomically while our algorithm depends on only one such variable. Therefore, we believe that our algorithm will scale to larger numbers of processors. Finally, the only atomic operation required by our algorithm is `fetch_and_store`. Therefore, it can be used on multiprocessors that do not support the rich set of atomic operations provided by the BBN TC2000.

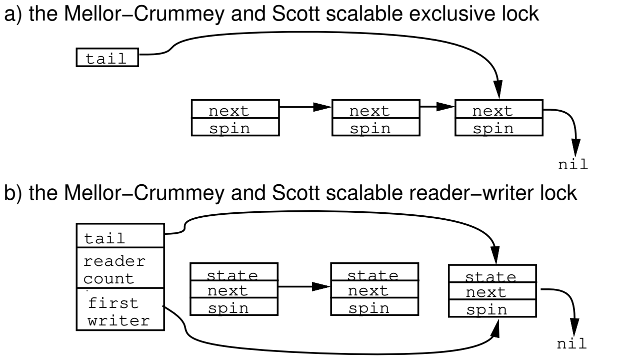

Figure 1: The Mellor-Crummey and Scott scalable locks

2 BACKGROUND

Mellor-Crummey and Scott's scalable RW lock is derived from their exclusive lock [6], which uses atomic operations to build a singly linked list of waiting processors (Fig. 1a). The processor at the list head has the lock and new processors add themselves to the list tail. Rather than spinning on a global lock variable, each processor spins on a variable in its local memory. A processor releases the lock by zeroing the variable on which the next processor in the queue is spinning.

For the RW variant of this algorithm, each queue element contains an additional variable to maintain the state of the request. When a new reader request arrives, the state of the previous element in the queue is examined to determine if the new request must block.

With a RW lock, readers must be able to release

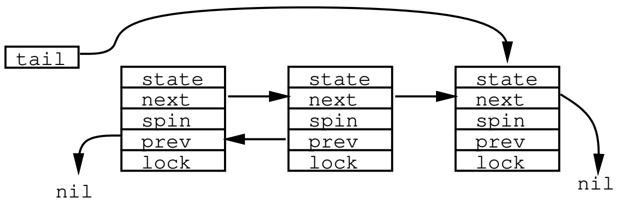

Figure 2: Our RW lock with two active readers and a single blocked writer.

the lock in any order. Hence, the singly linked list used in the Mellor-Crummey and Scott algorithm becomes discontinuous as readers dequeue. To allow for this, two global variables were added to their exclusive lock, namely: 1) a count of the number of active readers, and 2) a pointer to the first writer in the queue. As readers acquire and release the lock they keep the global count of active readers up to date. When releasing the lock, if a reader discovers that the reader count is zero, it unblocks the writer pointed to by the global variable. The structure of a list with two readers and a single blocked writer is shown in Fig. 1b.

3 OUR ALGORITHM

We have developed a new fair scalable RW locking algorithm which is also derived from Mellor-Crummey and Scott's exclusive locking algorithm. The key advantage of this new algorithm is that, rather than adding more global state (that can become contended), we distribute the extra state needed for a RW lock across the list associated with the lock. In particular, readers are maintained in a doubly linked list (Fig. 2).

With a doubly linked list, instead of synchronizing on a global variable, a reader that is releasing the lock can synchronize with its nearest neighbors to remove itself from the queue. This allows readers to dequeue in any order without the list becoming discontinuous. Hence, it is not necessary to keep either a global pointer to the first writer or a global count of the number of active readers.

We have developed two versions of this algorithm. The simpler (and more efficient) version requires that the hardware support atomic `compare_and_swap` operations. The more complicated version requires only `fetch_and_store` operations.

The simple version of our algorithm is shown in Figures 3 to 5. The per-processor list element structure used by our algorithm contains: 1) a state variable that indicates if the processor is an active reader, an intended reader, or a writer, 2) a local spin variable, 3) pointers to the next and previous queue elements, and 4) a spin lock used for dequeuing read requests.

```
type Lelem = record      // list element
   state : (READER, WRITER, ACTIVE_READER)
   spin  : int           // a local spin variable
   next, prev : ^Lelem   // neighbor pointers
   EL    : lock          // a spin lock

type RWlock : ^Lelem     // list tail pointer

procedure writerLock( L : ^RWlock, I : ^Lelem )
    var pred : ^Lelem
    I->state := WRITER
    I->spin := 1
    I->next := 0
    pred := fetch_and_store( I, L )
    if pred != nil
        pred->next := I
        repeat until I->spin = 0

procedure writerUnlock( L : ^RWlock, I : ^Lelem )
    var pred : ^Lelem

    if I->next = nil and
       compare_and_swap( 0, I, L ) == I
         return
    repeat until I->next != nil
    I->next->prev := 0
    I->next->spin := 0
```

Figure 3: Routines for write lock and unlock

The `writerLock` and `writerUnlock` operations (Fig. 3) are nearly identical to the `acquire_lock` and `release_lock` operations of Mellor-Crummey and Scott's exclusive lock algorithm. The only difference is that `writerUnlock` zeroes the next element's previous pointer field to signal that processor that it is now at the head of the queue. This is needed if the next processor is a reader.

On executing `readerLock`, the requesting processor constructs a doubly linked list by saving the pointer to the previous element (in the list) into its local structure and then placing a pointer to its local structure in the previous element's next pointer. After enqueueing

```
procedure readerLock( L : ^RWlock, I : ^Lelem )
    var pred : ^Lelem
    I->state := READER
    I->spin  := 1
    I->next  := I->prev = 0
    pred := fetch_and_store( I, L )
    if pred != nil
        I->prev := pred
        pred->next := I
        if pred->state != ACTIVE_READER
           repeat until I->spin = 0
    if I->next != nil and I->next->state = READER
        I->next->spin := 0
    I->state := ACTIVE_READER
```

Figure 4: The read lock routine

```
procedure readerUnlock( L : ^RWlock, I : ^Lelem )
    var prev : ^Lelem := I->prev
    if prev != nil
        exclusiveLock( &prev->EL )
        repeat until prev == I->prev
            exclusiveUnlock( &prev->EL )
            prev := I->prev
            if prev = nil break
            exclusiveLock( &prev->EL )
        if prev != nil
            exclusiveLock( &I->EL )
            prev->next := nil
            if I->next = nil and
               compare_and_swap(I->prev,I,L) != I
               repeat until I->next != nil
            if I->next != nil
               I->next->prev = I->prev
               I->prev->next = I->next
            exclusiveUnlock( &I->EL )
            exclusiveUnlock( &prev->EL )
            return
    exclusiveLock( &I->EL )
    if I->next = nil and
       compare_and_swap( 0, I, L ) != I
       repeat until I->next != nil
    if I->next != nil
       I->next->spin = 0
       I->prev->prev = 0
    exclusiveUnlock( &I->EL )
```

Figure 5: The read unlock routine

itself, the requesting processor checks if its predecessor has acquired a reader lock, in which case it can also acquire the reader lock without having to block. After acquiring the lock (and before modifies its state variable to indicate that it has done so), the requester checks to see if it has a successor that is also a reader request and if so unblocks that processor (by zeroing its spin variable).

As with `writerUnlock`, `readerUnlock` releases the lock by removing its local structure from the queue. To remove itself from the queue, the releasing processor must synchronize with both its queue neighbors. Since the order of elements in the queue is unique, it is easy to do this in a deadlock free fashion by having the releasing processor first acquire its predecessor's lock and then its own. After both these locks are acquired, the releasing processor can simply dequeue itself by modifying the previous and next fields of its neighbors in the linked list. If the releasing reader processor is at the end of the queue it swaps the pointer to its predecessor into the lock structure. (Note that `readerUnlock` differs from `writerUnlock` in that it does not unblock the next processor unless the releasing processor is at the head of the queue.)

While the algorithm described above is simple and efficient, it is not portable to all hardware bases because it depends on `compare_and_swap` operations.

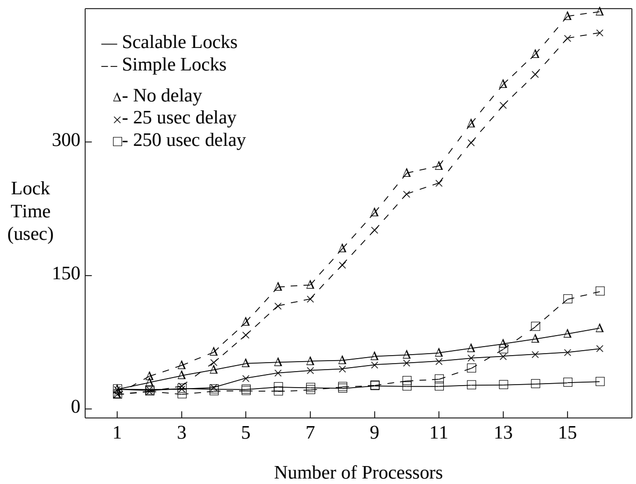

Figure 6: RW locks with only read requests.

Since most multiprocessors support `fetch_and_store` operations, we developed a more complicated version of the algorithm that depends only on these operations. The main added overhead with this version is that a global exclusive (spin) lock is required on unlock if no succeeding elements are in the linked list.

The complexity with this version arises from the fact that with only `fetch_and_store` there is no way to atomically dequeue an element at the tail of the list. That is, there is no way for an unlocking processor to atomically 1) detect that its local structure is the tail element and 2) dequeue that element. With the singly linked list used by Mellor-Crummey and Scott's exclusive lock, this problem can be solved at the cost of some requests occasionally being served out of order [5]. However, with our doubly linked list the problem becomes much more difficult to handle.

4 PERFORMANCE

The variant of our algorithm that uses only `fetch_and_store` was implemented in C code on the Hector multiprocessor [8]. The particular system used is a 16 processor system that runs at 16MHz and uses MC88100 processors. The experiments where performed as regular user programs running on a fully configured Hurricane operating system [7].

Figure 6 compares the performance of our scalable RW lock to a simple exponential backoff RW lock when p processors continuously acquire and release locks for reading. The different curves for each lock type show the performance when the locks are held for varying amounts of time. This *lock hold* time is subtracted away in the times presented. The scalable RW lock performs much better than the simple RW lock under

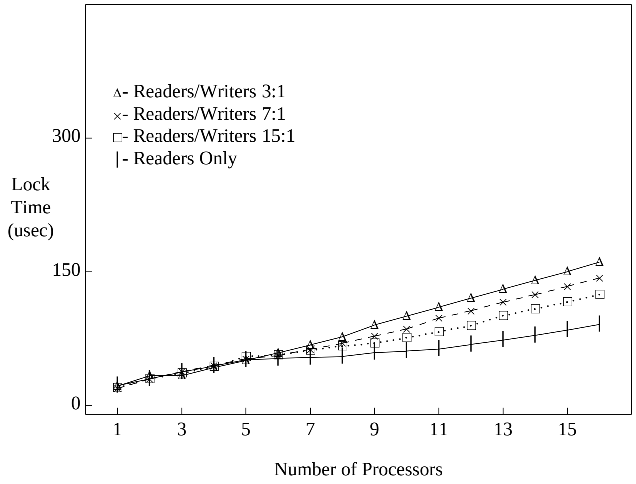

Figure 7: Scalable RW lock with various reader/writer ratios

any significant load. With a lock hold time of 0, the simple lock causes the memory containing the lock to saturate with just a small number of processors (i.e. 3 or 4) at which point the lock begins to perform very poorly. Even with a lock hold time of 250 usec, the performance of the spin lock begins to degenerate after about 11 processors.

For small numbers of processors and with a lock hold time of 0, as the number of requesting processors increases the scalable RW lock initially increases in cost quite quickly. This is because optimizations in our implementation for the uncontended lock no longer apply. However, the curve flattens as the number of processors increase further. The response time continues to increase, mainly because the memory that contains the pointer to the tail of the list becomes more contended. With a lock hold time of just 25 usec, the degradation of performance for the scalable lock is much more gradual. With a lock hold time of 250 usec, the curve for the scalable lock remains flat for all 16 processors.

The performance of the scalable RW lock with different ratios of read and write requests (and a lock hold time of 0) is shown in Figure 7.[1] Acquiring and releasing the lock for writing is (in the uncontended case) less expensive than for reading. The figure shows this, since for a small load performance is better if the ratio of readers to writers is low. However, after 5 processors contend for the lock, the greater the ratio of readers to writers the better the performance of the lock. This is expected, since the readers can acquire and release the lock concurrently. The advantage of a high ratio of readers to writers is even larger with some lock delay, hence this is a worst case experiment for readers.

[1] An off-line pseudo-random number generator was used to generate request sequences in which readers outnumbered writes according to the required ratio.

5 CONCLUDING REMARKS

We have developed a new scalable RW lock for shared memory multiprocessors. We believe that this lock is an improvement over Mellor-Crummey and Scott's RW lock in that 1) it involves fewer memory and atomic operations in the absence of contention; 2) it uses less global state that must be modified atomically; and 3) it requires only `fetch_and_store` operations, and hence can be used on most current multiprocessors. A full description of this algorithm is contained in [4].

To verify both versions of our algorithm, we used a state space searching tool [3] to do a full search of the state space for small numbers of requesters, and a partial search for larger numbers of requesters. Although partial searches cannot prove correctness, our tests have found all of our (previous) errors very quickly.

Acknowledgements

We would like to thank Benjamin Gamsa for his contribution in improving the presentation of this paper.

REFERENCES

[1] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Tran. on Par. and Dis. Sys.*, 1(1):6–16, 1990.

[2] G. Graunke and S. Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *IEEE Computer*, 23(6):60–69, June 1990. 1990.

[3] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[4] Orran Krieger, Michael Stumm, Ron Unrau, and Jonathan Hanna. A fair fast scalable reader-writer lock. CSRI, University of Toronto, 1993.

[5] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. on Comp. Sys.*, 9(1), Feb. 1991.

[6] J. M. Mellor-Crummey and M. L. Scott. Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors. In *Third ACM SIGPLAN Symp. on PPOPP*, 1991.

[7] M. Stumm, R. Unrau, and O. Krieger. Designing a Scalable Operating System for Shared Memory Multiprocessors. In *Usenix Workshop on Micro-kernels and Other Kernel Architectures*, 1992.

[8] Zvonko G. Vranesic, Michael Stumm, Ron White, and David Lewis. "The Hector Multiprocessor". *IEEE Computer*, 24(1), January 1991.