Scalable Queue-Based Spin Locks with Timeout

Michael L. Scott and William N. Scherer III

Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
{scott,scherer}@cs.rochester.edu*

ABSTRACT

Queue-based spin locks allow programs with busy-wait synchronization to scale to very large multiprocessors, without fear of starvation or performance-destroying contention. So-called *try locks*, traditionally based on non-scalable test-and-set locks, allow a process to abandon its attempt to acquire a lock after a given amount of time. The process can then pursue an alternative code path, or yield the processor to some other process.

We demonstrate that it is possible to obtain both scalability and bounded waiting, using variants of the queue-based locks of Craig, Landin, and Hagersten, and of Mellor-Crummey and Scott. A process that decides to stop waiting for one of these new locks can "link itself out of line" atomically. Single-processor experiments reveal performance penalties of 50–100% for the CLH and MCS try locks in comparison to their standard versions; this marginal cost decreases with larger numbers of processors.

We have also compared our queue-based locks to a traditional test-and-`test_and_set` lock with exponential backoff and timeout. At modest (non-zero) levels of contention, the queued locks sacrifice cache locality for fairness, resulting in a worst-case 3X performance penalty. At high levels of contention, however, they display a 1.5–2X performance *advantage*, with significantly more regular timings and significantly higher rates of acquisition prior to timeout.

1. INTRODUCTION

Spin locks are widely used for mutual exclusion on shared-memory multiprocessors. Traditional `test_and_set`-based spin locks are vulnerable to memory and interconnect contention, and do not scale well to large machines. Queue-based spin locks avoid contention by arranging for every waiting process to spin on a separate, local flag in memory.

*This work was supported in part by NSF grants numbers EIA-9972881, EIA-0080124, CCR-9705594, and CCR-9988361, and by DARPA/AFRL contract number F29601-00-K-0182.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPOPP'01, June 18-20, 2001, Snowbird, Utah, USA.
Copyright 2001 ACM 1-58113-346-4/01/0006 ...$5.00.

Several researchers (ourselves among them) have conjectured that any program that routinely encounters high levels of lock contention is unlikely to scale well to large machines. Conversely, in any program that scales well, the overhead of spin locks may not be a serious concern. The problem with this argument is that infrequent pathological events can be a serious problem to users. An improvement in *worst case* performance from, say, 2 to 10% efficiency may be enormously valuable, even if typical efficiency is more like 80%. With the increasing popularity of medium-scale (20–100 processor) server-class machines, we believe that scalable spin locks will prove increasingly important.

In a traditional `test_and_set` spin lock, processes are mutually anonymous. In a queue-based lock, they are not: each waiting process is linked to its predecessor and successor in the queue. In previous work [6], members of our group considered the possibility that a spinning process might be preempted, thereby unnecessarily delaying processes farther down the queue. In the current paper we consider the possibility that a spinning process may "become impatient" and wish to leave the queue before acquiring the lock.

Timeout-capable spin locks, sometimes referred to as "try-locks", are important for several reasons:

1. A process in a soft real-time application may need to bound the time it spends waiting for a lock. If the timeout expires, the process can choose to announce an error or to pursue an alternative code path that does not require the lock.

2. Absent special OS support, a user-level process in a multiprogrammed environment may be preempted while holding a lock. Timeout allows other processes waiting for the lock to give up, yield the processor, and try again when rescheduled.

3. In a parallel database system, timeout provides a viable strategy for deadlock recovery. A process that waits "too long" for a lock can assume that deadlock has occurred, abort the current transaction, and retry.

We are aware of commercially significant signal processing applications that use timeout for reason (1), and parallel database servers that use timeout for reasons (2) and (3). In the latter case, timeout may be *the* deciding factor in making spin locks usable in user-level code.

In contrast to the problem of preemption among spinning processes, timeout does not require interaction between the scheduler and the synchronization algorithm. It is complicated, however, by the fact that a timed-out process may

attempt to acquire the lock again, possibly many times, before the point at which it would have acquired the lock if it had stayed in line. The possibility of multiple aborted acquisition attempts suggests that a timed-out process must remove itself from the queue entirely, leaving nothing behind; otherwise we would be unable to bound the space or time requirements of the algorithm.

We have developed so-called "try lock" (timeout capable) versions of our MCS queue-based lock [8] and of the CLH queue-based lock of Craig [2] and Landin and Hagersten [7]. After presenting additional background information in section 2, we describe our new locks in section 3. Both new locks employ `swap` and `compare_and_swap` instructions, and can be implemented on any shared-memory machine, with or without cache coherence, that provides these operations or their equivalent. In section 4 we present performance results obtained on a 56-processor Sun Wildfire machine. We conclude with a summary of recommendations in section 5.

2. BACKGROUND

Programmers face several choices when synchronizing processes on a shared-memory multiprocessor. The most basic choice is between spinning (busy-waiting), in which processes actively poll for a desired condition, and blocking, in which processes yield the processor in expectation that they will be made runnable again when the desired condition holds. Spinning is the method of choice when the expected wait time is small, or when there is no other productive use for the processor.

The most basic busy-wait mechanism is a test-and-set lock, in which a process desiring entry to a critical section repeatedly attempts to change a "locked" flag from `false` to `true`, using an atomic hardware primitive. Unfortunately, test-and-set locks lead to increasing amounts of contention for memory and bus or interconnect bandwidth as the number of competing processors grows. This contention can be reduced somewhat by polling with ordinary read operations, rather than atomic `test_and_set` operations; polls are then satisfied from local caches during critical sections, with a burst of refill traffic whenever the lock is released. The cost of the burst can be further reduced using Ethernet-style exponential backoff [8]. Figure 1 shows code for a test-and-`test_and_set` (TATAS) lock with exponential backoff.

2.1 Queue-based locks

Even with exponential backoff, test-and-`test_and_set` locks still induce significant contention, leading to irregular timings and compromised memory performance on large machines. Queue-based spin locks eliminate these problems by arranging for every competing process to spin on a different memory location. To ensure scalability, each location must be local to the spinning process, either by virtue of hardware cache coherence or as a result of explicit local allocation on a non-cache-coherent machine.

Several researchers, working independently and roughly concurrently, developed queue-based spin locks in the late 1980s. Anderson [1] and Graunke and Thakkar [3] embed their queues in per-lock arrays, requiring space per lock proportional to the maximum number of processes that may compete for access concurrently. Anderson's lock uses atomic `fetch_and_increment` instructions to assign slots in the array to waiting processes; Graunke and Thakkar's lock uses `swap` instead. Our MCS lock, co-designed with John Mellor-Crummey [8], employs a linked list with pointers from each process to its successor; it relies on `swap` and `compare_and_swap`, and has the advantage of requiring total space linear in the number of locks and the number of competing processes. (The lock can be re-written to use only `swap`, but at the expense of FIFO ordering: in the `swap`-only version of `mcs_release` there is a timing window that may allow newly arriving processes to jump ahead of processes already in the queue.) The MCS lock is naturally suited to local-only spinning on both cache-coherent and non-cache-coherent machines. The Anderson and Graunke/Thakkar locks must be modified to employ an extra level of indirection on non-cache-coherent machines.

```
typedef unsigned long bool;
typedef volatile bool tatas_lock;

void tatas_acquire(tatas_lock *L) {
    if (tas(L)) {
        int b = BACKOFF_BASE;
        do {
            for (i = b; i; i--);    // delay
            b = min(b * BACKOFF_FACTOR, BACKOFF_CAP);
            if (*L) continue;       // spin with reads
        } while (tas(L));
    }
}

void tatas_release(tatas_lock *L) {
    *L = 0;
}
```

Figure 1: Test-and-`test_and_set` (TATAS) lock with exponential backoff. Parameters `BACKOFF_BASE`, `BACKOFF_FACTOR`, and `BACKOFF_CAP` must be tuned by trial and error for each individual machine architecture.

```
typedef struct mcs_qnode {
    volatile bool waiting;
    volatile struct mcs_qnode *volatile next;
} mcs_qnode;

typedef volatile mcs_qnode *mcs_qnode_ptr;
typedef mcs_qnode_ptr mcs_lock; // initialized to nil

void mcs_acquire(mcs_lock *L, mcs_qnode_ptr I) {
    I->next = nil;
    mcs_qnode_ptr pred = swap(L, I);
    if (pred == nil) return;    // lock was free
    I->waiting = true;          // word on which to spin
    pred->next = I;             // make pred point to me
    while (I->waiting);         // spin
}

void mcs_release(mcs_lock *L, mcs_qnode_ptr I) {
    mcs_qnode_ptr succ;
    if (!(succ = I->next)) {    // I seem to have no succ.
        // try to fix global pointer
        if (compare_and_store(L, I, nil)) return;
        do {
            succ = I->next;
        } while (!succ);        // wait for successor
    }
    succ->waiting = false;
}
```

Figure 2: The MCS queue-based spin lock. Parameter `I` points to a qnode record allocated (in an enclosing scope) in shared memory locally-accessible to the invoking processor.

```
typedef struct clh_qnode {
    volatile bool waiting;
    volatile struct clh_qnode *volatile prev;
} clh_qnode;

typedef volatile clh_qnode *clh_qnode_ptr;
typedef clh_qnode_ptr clh_lock;
    // initialized to point to an unowned qnode

void clh_acquire(clh_lock *L, clh_qnode_ptr I) {
    I->waiting = true;
    clh_qnode_ptr pred = I->prev = swap(L, I);
    while (pred->waiting);      // spin
}

void clh_release(clh_qnode_ptr *I) {
    clh_qnode_ptr pred = (*I)->prev;
    (*I)->waiting = false;
    *I = pred;                  // take pred's qnode
}
```

Figure 3: The CLH queue-based spin lock. Parameter `I` points to qnode record or, in `clh_release`, to a pointer to a qnode record. The qnode "belongs" to the calling process, but may be in main memory anywhere in the system, and will generally change identity as a result of releasing the lock.

The CLH lock, developed about three years later by Craig [2] and, independently, Landin and Hagersten [7], also employs a linked list, but with pointers from each process to its *predecessor*. The CLH lock relies on atomic `swap`, and may outperform the MCS lock on cache-coherent machines. Like the Anderson and Graunke/Thakkar locks, it requires an extra level of indirection to avoid spinning on remote locations on a non-cache-coherent machine [2]. Code for the MCS and CLH locks appears in Figures 2, 3, and 4.

```
typedef struct clh_numa_qnode {
    volatile bool *w_ptr;
    volatile struct clh_qnode *volatile prev;
} clh_numa_qnode;

typedef volatile clh_numa_qnode *clh_numa_qnode_ptr;
typedef clh_numa_qnode_ptr clh_numa_lock;
    // initialized to point to an unowned qnode

const bool *granted = 0x1;

void clh_numa_acquire(clh_numa_lock *L,
                      clh_numa_qnode_ptr I) {
    volatile bool waiting = true;
    I->w_ptr = nil;
    clh_numa_qnode_ptr pred = I->prev = swap(L, I);
    volatile bool *p = swap(&pred->w_ptr, &waiting);
    if (p == granted) return;
    while (waiting);            // spin
}

void clh_numa_release(clh_numa_qnode_ptr *I) {
    clh_numa_qnode_ptr pred = (*I)->prev;
    volatile bool *p = swap(&((*I)->w_ptr), granted);
    if (p) *p = false;
    *I = pred;                  // take pred's qnode
}
```

Figure 4: Alternative version of the CLH lock, with an extra level of indirection to avoid remote spinning on a non-cache-coherent machine.

```
// return value indicates whether lock was acquired
bool tatas_try_acquire(tatas_lock *L, hrtime_r T) {
    if (tas(L)) {
        hrtime_t start = gethrtime();
        int b = BACKOFF_BASE;
        do {
            if (gethrtime() - start > T) return false;
            for (i = b; i; i--);    // delay
            b = min(b * BACKOFF_FACTOR, BACKOFF_CAP);
            if (*L) continue;       // spin with reads
        } while (tas(L));
    }
}
```

Figure 5: The standard TATAS-try lock. Type definitions and `release` code are the same as in Figure 1.

2.2 Atomic primitives

In this paper we make reference to several atomic operations. `Swap`(address, value) atomically writes a memory location and returns its original contents. `Compare_and_swap`(address, expected_value, new_value) atomically checks the contents of a memory location to see if it matches an expected value and, if so, replaces it with a new value. In either event it returns the original contents. We also use an alternative form, `compare_and_store`, that instead returns a Boolean value indicating whether the comparison succeeded. `Fetch_and_increment`(address) atomically increments the contents of a memory location and returns the original contents.

`Compare_and_swap` first appeared in the IBM 370 instruction set. `Swap` and `compare_and_swap` are provided by Sparc V9. Several recent processors, including the Alpha, MIPS, and PowerPC, provide a pair of instructions, `load_linked` and `store_conditional`, that can be implemented naturally and efficiently as part of an invalidation-based cache-coherence protocol, and which together provide the rough equivalent of `compare_and_swap`. Both `compare_and_swap` and `load_linked/store_conditional` are *universal* atomic operations, in the sense that they can be used without locking (but at the cost of some global spinning) to implement any other atomic operation [5]. `Fetch_and_increment`, together with a host of other atomic operations, is supported directly but comparatively inefficiently on the x86.

3. TRY LOCKS

As noted in section 1, a process may wish to bound the time it may wait for a lock, in order to accommodate soft real-time constraints, to avoid waiting for a preempted peer, or to recover from transaction deadlock. Such a bound is easy to achieve with a `test_and_set` lock (see Figure 5): processes are anonymous and compete with one another chaotically. Things are not so simple, however, in a queue-based lock: a waiting process is linked into a data structure on which other processes depend; it cannot simply leave.

A similar problem occurs in multiprogrammed systems when a process stops spinning because it has been preempted. Our previous work in scheduler-conscious synchronization [6] arranged to mark the queue node of a preempted process so that the process releasing the lock would simply pass it over. Upon being rescheduled, a skipped-over process would have to reenter the queue. A process that had yet to reach the head of the queue when rescheduled would retain its original position.

Craig [2] proposes (in narrative form) a similar "mark the node" strategy for queue-based try locks. Specifically, he suggests that a timed-out process leave its queue node behind, where it will be reclaimed by another process when it reaches the head of the queue. Unfortunately, this strategy does not work nearly as well for timeout as it does for preemption. The problem is that a timed-out process is not idle: it may want to acquire other locks. Because we have no bound on how long it may take for a marked queue node to reach the head of the queue, we cannot guarantee that the same queue node will be available the next time the process tries to acquire a lock. As Craig notes, the total space required for P processes and L locks rises from $O(P + L)$ with the original CLH lock to $O(P \times L)$ in a mark-the-node try lock. We also note that the need to skip over abandoned queue nodes increases the worst case lock release time from $O(1)$ to $O(P)$.

We have addressed these space and time problems in new try-lock versions of both the CLH and MCS locks. In fairness to Craig, the code presented here requires a `compare_and_swap` operation; his work was predicated on the assumption that only `swap` was available. A different version of the MCS-try lock, also using only `swap`, has been developed by Vitaly Oratovsky and Michael O'Donnell of Mercury Computer Corp. [9]. Their lock has the disadvantage that newly arriving processes that have not specified a timeout interval (i.e. that are willing to wait indefinitely) will bypass any already-waiting processes whose patience is more limited. A process that specifies a timeout may thus fail to acquire a lock—may in fact never acquire a lock, even if it repeatedly becomes available before the expiration of the timeout interval—so long as processes that have not specified a timeout continue to arrive.

3.1 The CLH-try lock

In the standard CLH lock, a process leaves its queue node behind when releasing the lock. In its place it takes the node abandoned by its predecessor. For a try lock, we must arrange for a process that times out to leave with its own queue node. Otherwise, as noted above, we might need $O(P \times L)$ queue nodes in the system as a whole.

It is relatively easy for a process B to leave the middle of the queue. Since B's intended successor C (the process behind it in the queue) is already spinning on B's queue node, B can simply mark the node as "leaving". C can then dereference the node to find B's predecessor A, and mark B's node as "recycled", whereupon B can safely leave. There is no race between A and B because A never inspects B's queue node.

Complications arise when the departing process B is the last process in the queue. In this case B must attempt to modify the queue's tail pointer to refer to A's queue node instead of its own. We can naturally express the attempt with a `compare_and_swap` operation. If the `compare_and_swap` fails we know that another process C has arrived. At this point we might hope to revert to the previous (middle-of-the-queue) case. Unfortunately, it is possible that C may successfully leave the queue after B's `compare_and_swap`, at which point B may wait indefinitely for a handshake that never occurs. We could protect against the indefinite wait by repeatedly re-checking the queue's tail pointer, but that would constitute spinning on a non-local location, something we want to avoid.

Our solution is to require C to handshake with B in a way that prevents B from trying to leave the queue while C is in the middle of leaving. Code for this solution can be found on-line at `www.cs.rochester.edu/u/scott/synchronization/pseudocode/timeout.html#clh-try`. It comprises about 95 lines of C. The two principal cases (B in the middle of the queue and at the end) are illustrated in Figure 6.

Like the standard CLH lock, the CLH-try lock depends on cache coherence to avoid remote spinning. In the CLH-try lock it is actually possible for two processes to end up spinning on the same location. In the fourth line of the right-hand side of Figure 6, if process A calls `clh_release`, it will spin until the `transient` flag reverts to `waiting`. If a new process C arrives at about this time, it, too, will begin to spin on the flag in A's queue node. When B finally updates the flag, its write will terminate both spins.

3.2 The MCS-try lock

A feature of the standard MCS lock is that each process spins on its own queue node, which may be allocated in local memory even on a non-cache-coherent machine. To leave the queue, therefore, a process B must update the successor pointer in the queue node of its predecessor A so that it points to B's successor C, if any, rather than to B. If C later chooses to leave the queue as well, it will again need to update A's queue node, implying that B must tell it where A's queue node resides. Pointers to both predecessors and successors must therefore reside in the queue nodes in memory, where they can be read and written by neighboring processes. These observations imply that an MCS-try lock must employ a doubly linked queue, making it substantially more complex than the CLH-try lock. The benefit of the complexity is better memory locality on a non-cache-coherent machine. (We believe we could construct a version of the CLH-try lock, starting from the code in Figure 4, that would avoid remote spins on a non-cache-coherent machine, but it would require more atomic operations on its critical path, and would be unlikely to be competitive with the MCS-try lock.)

As in the CLH-try lock there are two principal cases to consider for the MCS-try lock, depending on whether the departing process B is currently in the middle or at the end of the queue. These cases are illustrated in Figure 7. While waiting to be granted the lock, a process ordinarily spins on its predecessor pointer. In the middle-of-the-queue case, departing process B first replaces the four pointers into and out of its queue node, respectively, with `leaving_other` and `leaving_self` flags (shown as LO and LS in the figure). It then updates C's predecessor pointer, and relies on C to update A's successor pointer. In the end-of-the-queue case, B "tags" A's nil successor pointer to indicate that additional changes are pending. Absent any race conditions, B eventually clears the tag using `compare_and_swap`.

Unfortunately, there are many potential races that have to be resolved. The process at the head of the queue may choose to grant the lock to its successor while the successor is attempting to leave the queue. Two neighboring processes may decide to leave the queue at approximately the same time. A process that is at the end of the queue in step 2 may discover in step 5 that it now has a successor. A complete discussion of these races and how we resolve them is too lengthy to include here. In general, the order of updates to pointers is chosen to ensure that (1) no process ever returns

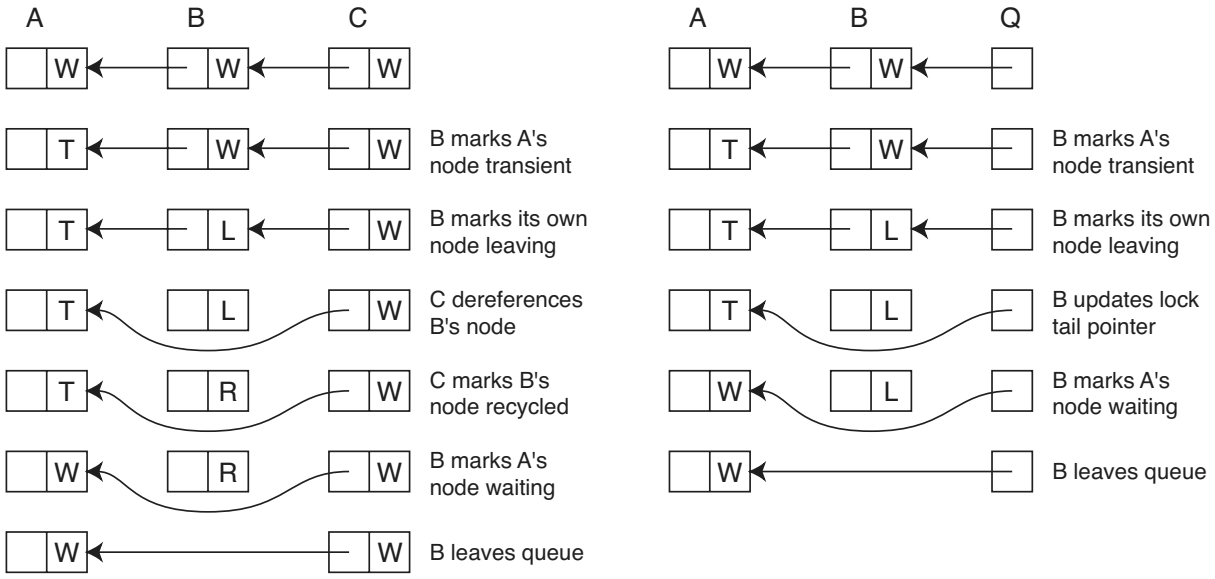

Figure 6: The two principal cases in the CLH-try lock. In the figure at left process B can leave the middle of the queue as soon as it receives confirmation from its successor, C, that no pointer to its queue node remains. In the figure at right, B can leave the end of the queue once it has updated the tail pointer, Q, using `compare_and_swap`. The transitions from `waiting` to `leaving` and from `waiting` to `available` (not shown) must also be made with `compare_and_swap`, to avoid overwriting a `transient` flag.

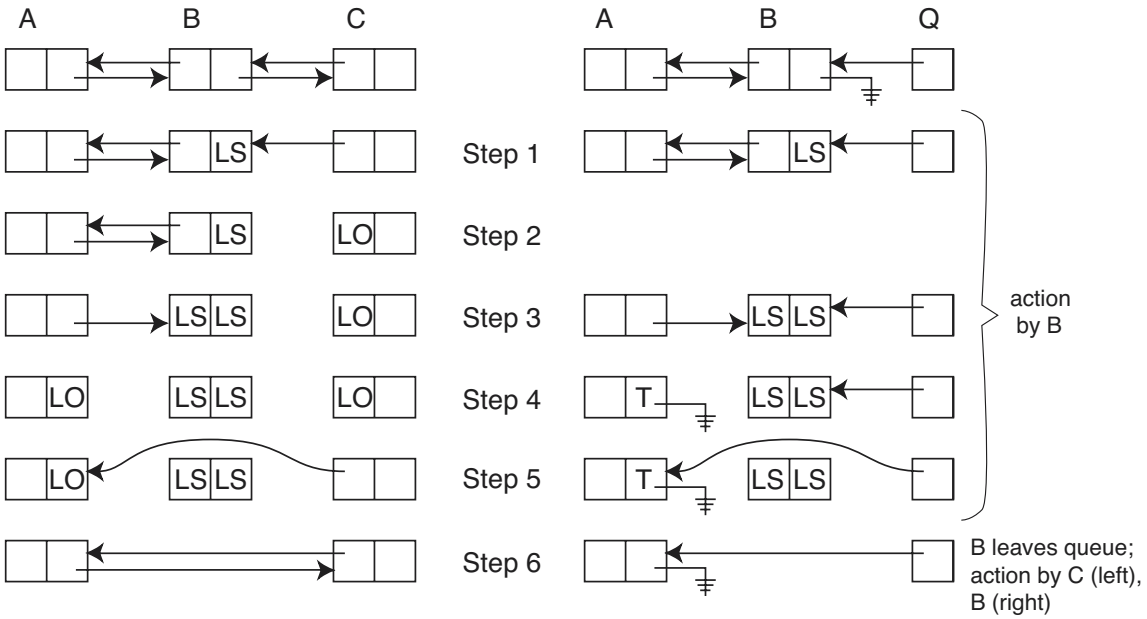

Figure 7: The two principal cases in the MCS-try lock. In the figure at left process B uses atomic `swap` operations to replace the pointers into and out of its queue node, in a carefully chosen order, with `leaving_other` and `leaving_self` flags. It then updates the pointer from C to A, and relies on C to update the pointer from A to C. In the figure at right, B has no successor at step 2, so it must perform the final update itself. In the meantime it leaves a "tag" on A's (otherwise nil) successor pointer to ensure that a newly arriving process will wait for B's departure to be finalized.

from `mcs_try_acquire` until we are certain that no pointers to its queue node remain, and (2) if two adjacent processes decide to leave concurrently, the one closer to the front of the queue leaves first. Code for the MCS-try lock is available online at `www.cs.rochester.edu/u/scott/synchronization/pseudocode/timeout.html#mcs-try`.

3.3 Correctness

Synchronization algorithms are notoriously subtle. Correctness proofs for such algorithms tend to be less subtle, but more tedious and significantly longer than the original code. We have not attempted to construct such a proof for either of our algorithms; for the MCS-try lock in particular, which runs to more than 360 lines of C code, it appears a daunting exercise.

Testing, of course, cannot demonstrate the absence of bugs, but it can significantly decrease their likelihood. We have tested both our algorithms extensively, under a variety of loads and schedules, and did indeed find and fix several bugs. We are reasonably confident of our code, but acknowledge the possibility that bugs remain.

There is a third possible approach to verification. We are in the process of constructing a simulation testbed that will allow us, for a given set of processes, to exhaustively explore all possible interleavings of execution paths. We plan to verify that these interleavings preserve all necessary correctness invariants for the operation of a queue-based lock, and to argue that they cover all the "interesting" states that could arise with any larger set of processes.

4. PERFORMANCE RESULTS

We have implemented the TATAS, CLH, and MCS locks, with and without timeout, using the `swap` and `compare_and_swap` operations available in the Sparc V9 instruction set. Initial testing and single-processor results employed a 336 MHz Sun Ultra 4500. Scalability tests were conducted on a 56-processor Sun Wildfire machine [4] (not to be confused with the Compaq product of the same name) with 250 MHz processors. Architecturally, the Wildfire machine consists of four banks of up to 16 processors each, connected by a central crossbar. Backoff constants for the TATAS lock were tuned separately for each machine.

Our tests employ a microbenchmark consisting of a tight loop containing a single acquire/release pair. Aside from counting the number of iterations and the number of successful acquires (these may be different in the case of a try lock), the loop does no useful work. Machines used for tests were otherwise unloaded.

4.1 Single-processor results

We can obtain an estimate of lock overhead in the absence of contention by running the microbenchmark on a single processor, and then subtracting the loop overhead. Results on on our Ultra 4500 are as follows:

TATAS	$137ns$
MCS	$172ns$
CLH	$137ns$
CLH-NUMA	$262ns$
MCS-try	$256ns$
CLH-try	$274ns$

In an attempt to avoid perturbation due to other activity on the machine, we have reported the minima over a series of several runs. As one might expect, none of the more complex locks is able to improve upon the time of the TATAS lock, though the CLH locks does tie it. The extra $35ns$ overhead in the MCS lock is primarily due to the `compare_and_swap` in `mcs_release`. The CLH-try and MCS-try locks pay an additional penalty for the extra argument to their `acquire` operations and, in the case of CLH-try, the `compare_and_swap` in `clh-release`. Neither of the try locks calls the Sun high-resolution timer if it is able to acquire the lock immediately. Each call to the timer consumes an additional $250ns$, which we would like to hide in wait time.

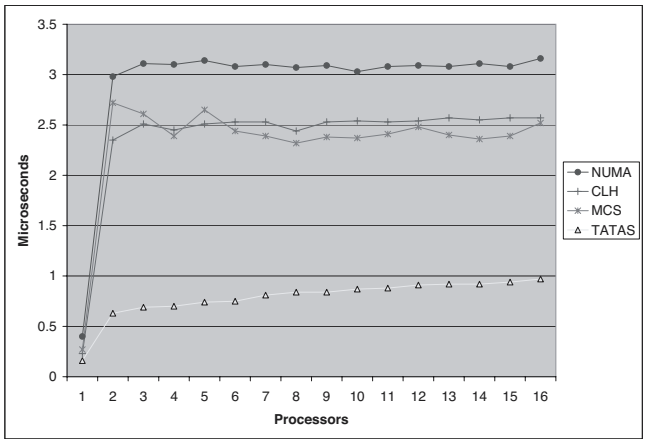

Figure 8: Microbenchmark iteration time for non-try locks. The very low overhead of the TATAS lock is an artifact of repeated acquisition by a single processor.

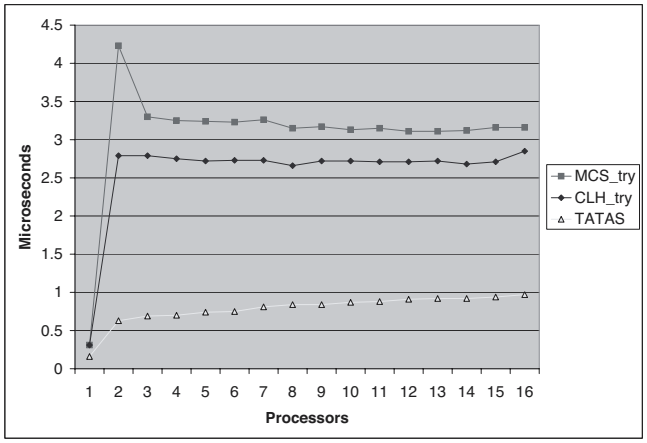

Figure 9: Microbenchmark iteration time for try locks, with patience (timeout interval) of $150\mu s$.

4.2 Lock passing time

We can obtain an estimate of the time required to pass a lock from one processor to another by running our microbenchmark on a large collection of processors. This *passing time* is not the same as total lock overhead: as discussed by Magnussen, Landin, and Hagersten [7], queue-based locks tend toward heavily pipelined execution, in which the initial cost of entering the queue and the final cost of leaving it are overlapped with the critical sections of other processors.

Figures 8 and 9 show the behaviors of all five queue-based locks on one bank of the Wildfire machine, with timeout values ("patience") set high enough that timeout never occurs in the queue-based try locks. All tests were run with a single process (pthread) on every processor. With only one active processor, the plotted value represents the sum of lock and loop overhead with perfect cache locality. The value for the queue-based locks jumps dramatically with two active processors, as a result of coherence misses. With three or more active processors, lock passing is fully pipelined, and the plotted value represents the time to pass the lock from one processor to the next.

Among the non-try locks (Figure 8), CLH-NUMA has a noticeably greater passing time ($3.1\mu s$) than either MCS or CLH. The passing times for MCS and CLH are just about the same, at $2.4\mu s$ and $2.5\mu s$ respectively. Both MCS and CLH are faster than either of their try lock counterparts, though at $2.7\mu s$, CLH-try beats out the CLH-NUMA lock. At $3.2\mu s$, MCS-try has the slowest passing time.

While the TATAS lock appears to be passing much faster than any of the other locks, this result is somewhat misleading. The queued locks are all fair: requests are granted in the order they were made. The TATAS lock, by contrast, is not fair: since the most recent owner of a lock has the advantage of cache locality, it tends to outrace its peers and acquire the lock repeatedly. (This effect would be reduced in a more realistic benchmark, with work outside the critical section.) In our experiments successive acquisitions of a queued lock with high patience occurred on different processors more than 99% of the time; successive acquisitions of a TATAS lock occurred on the *same* processor about 99% of the time. This unfairness has ramifications for timeout: even with $150\mu s$ patience (long enough for every processor, on average, to acquire and release the lock 10 times), TATAS still fails to acquire the lock some 4% of the time.

4.3 Bimodal behavior of try locks

Figure 10 plots the percentage of time that a processor in the microbenchmark succeed in acquiring a try lock. For this test the timeout interval (patience) has been set at only $25\mu s$. Figure 11 plots iteration time for the same experiment. With this lower patence level the the MCS-try and CLH-try locks exhibit distinctly bimodal behavior. With nine or fewer active processors, timeout almost never occurs, and behavior mimics that of the non-try locks. With ten or more active processors, timeouts begin to occur.

For higher processor counts, or for lower patience levels, the chance of a processor getting a lock is primarily a function of the number of processors that are in the queue ahead of it minus the number of those that time out and leave the queue before obtaining the lock. As is evident in these graphs, this chance drops off sharply with insufficient patience. The average time per iteration also drops, because giving up an attempt to acquire a lock is cheaper than waiting to acquire it.

4.4 Comparison of regular and try locks with TATAS

The tradeoff between MCS-try and plain MCS is as expected: at the cost of a higher average iteration time (per attempt), the plain MCS lock always manages to successfully acquire the lock. At the cost of greater complexity, the

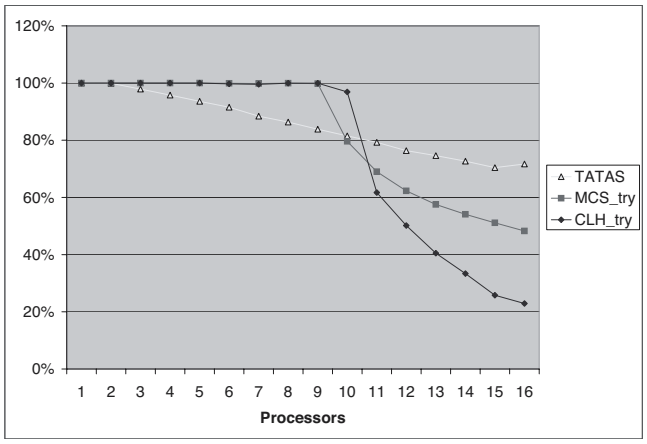

Figure 10: %Acquisition at $25\mu s$ patience.

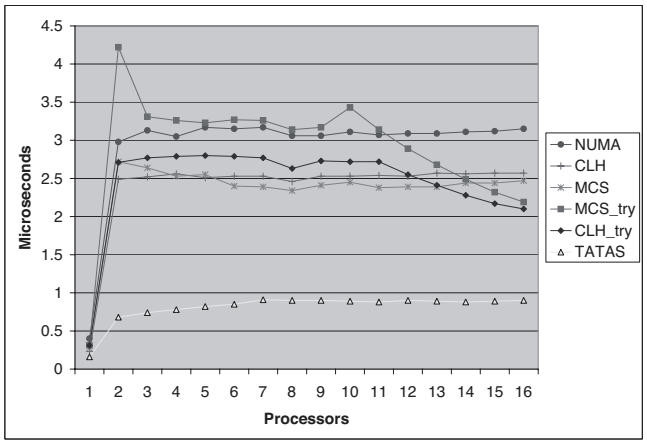

Figure 11: Iteration time at $25\mu s$ patience.

MCS-try lock provides the option of timing out. The same tradeoff holds between the CLH and CLH-try locks.

The tradeoffs between MCS-try or CLH-try and TATAS are more interesting. While the iteration time is consistently higher for the queue-based locks (Figure 11), the acquisition (success) rate depends critically on the ratio between patience and the level of competition for the lock. When patience is high, relative to competition, the queue-based locks are successful all the time. Once the expected wait time exceeds the timeout interval in the queue-based locks, however, the TATAS lock displays a higher acquisition rate. As we will see in Figures 14 and 15, TATAS is not able to maintain this advantage once we exceed the number of processors in a single bank of the Wildfire machine.

4.5 Results for high processor counts

Generally speaking, the results for larger numbers of processors are comparable to those seen within a single bank of the machine. Although crossing the interconnect between banks introduces a fair amount of noise into the timing results (see Figures 12 and 13), the MCS-try and CLH-try locks continue to have very similar iteration times, with MCS-try coming out somewhat slower than CLH-try.

The influence of the interconnect is particularly evident in the MCS-try iteration time in Figure 13: an additional

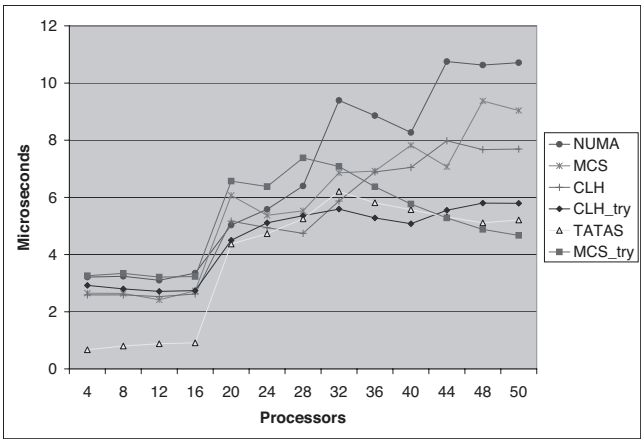

Figure 12: Iteration time at $200\mu s$ patience.

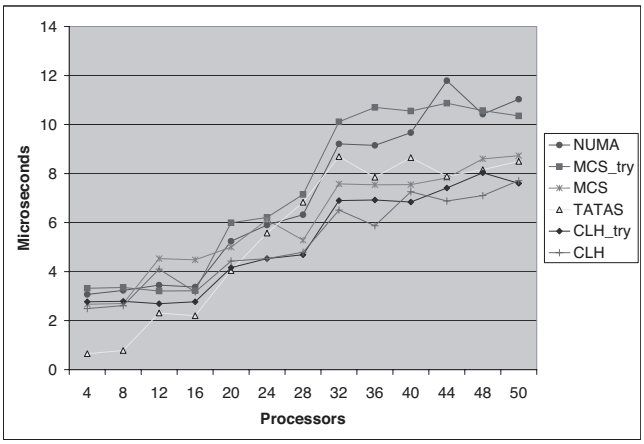

Figure 13: Iteration time at $500\mu s$ patience.

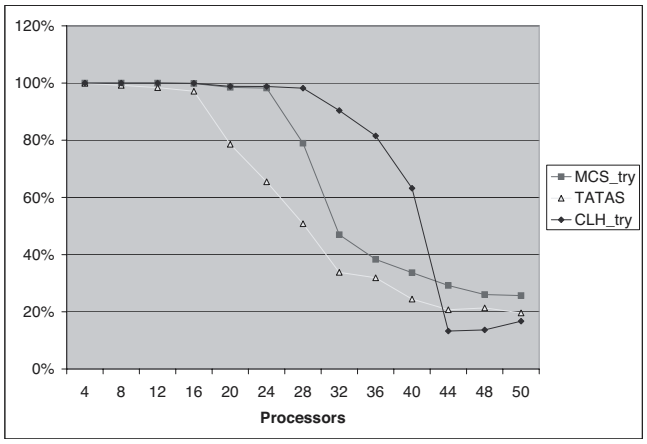

Figure 14: %Acquisition at $200\mu s$ patience.

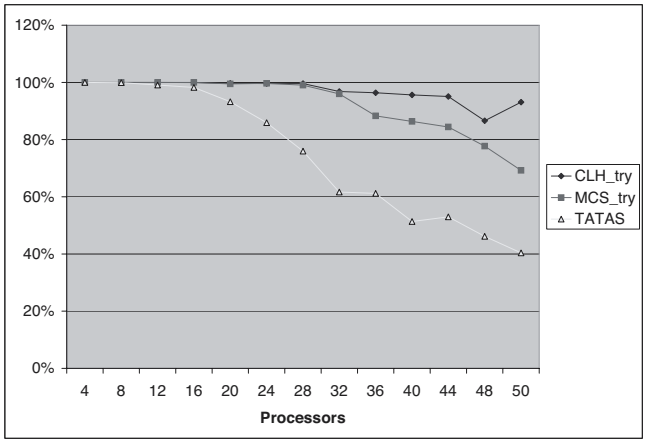

Figure 15: %Acquisition at $500\mu s$ patience.

bank of processors, requiring additional traversals of the interconnect, comes into play between 16 and 20 processors, and again between 28 and 32 processors. (A third transition point, between 40 and 44 processors, is not visible in the graph.)

Figures 14 and 15 show the establishment of a very long pipeline for lock acquisition. While the CLH-try lock sees a lower acquisition rate than the MCS-try lock at very high levels of competition relative to patience (Figure 14), there is a significant intermediate range where its acquisition rate is higher (Figure 15 and the center of Figure 14).

5. CONCLUSIONS

We have shown that it is possible, given standard atomic operations, to construct queue-based locks in which a process can time out and abandon its attempt to acquire the lock. Our algorithms provide reasonable performance on both small and large cache-coherent multiprocessors. Our future plans include tests on the non-cache-coherent Cray T3E; here we expect the MCS-try lock to perform particularly well.

In the decade since our original comparison of spin lock algorithms, ratios of single-processor latencies have remained remarkably stable. On a 16MHz Sequent Symmetry multiprocessor, a TATAS lock without contention consumed $7\mu s$ in 1991. The MCS lock consumed $9\mu s$, a difference of 28%. On a 336 MHz Sun Enterprise machine, a TATAS lock without contention takes $137ns$ today. The MCS lock takes $172ns$, a difference of 25%. The CLH lock, which we did not test in 1991, is tied with TATAS on a single processor.

With two or more processes competing for access to a lock, the numbers have changed more significantly over the years. In 1991 the TATAS lock (with backoff) ran slightly slower than the MCS lock at modest levels of contention. Today it appears to run in less than a third of the time of all the queue-based locks. Why then would one consider a queue-based lock?

The answer is three-fold. First, the performance advantage of the TATAS lock is exaggerated in our experiments by the use of back-to-back critical sections, allowing a single processor to leverage cache locality and acquire the lock repeatedly. Second, TATAS does not scale well. Once processes on the Wildfire machine are scattered across multiple banks, and must communicate through the central crossbar, iteration time for TATAS rises to 50–100% *above* that of the queue-based locks. Third, even with patience as high as $2ms$—200 times the average lock-passing time—the TATAS algorithm with timeout fails to acquire the lock about 20% of the time. This failure rate suggests that a regular TATAS

lock (no timeout) will see significant variance in acquisition times—in other words, significant unfairness. The queue based locks, by contrast, guarantee that processes acquire the lock in order of their requests.

For small-scale multiprocessors the TATAS lock remains the most attractive overall performer, provided that its back-off parameters have been tuned to the underlying hardware. Queue-based locks are attractive for larger machines, or for cases in which fairness and regularity of timing are particularly important. The CLH lock, both with and without timeout, has better overall performance on cache-coherent machines. The version with timeout is also substantially simpler than the MCS-try lock. One disadvantage of the CLH lock is its need for an initial queue node in each currently unheld lock. In a system with a very large number of locks this overhead might be an important consideration. Because the queue nodes of the MCS lock can be allocated statically in the local memory of the processors that spin on them, the MCS-try lock is likely to outperform the CLH-try lock on a non-cache-coherent machine. It is also likely, we believe, to be faster than a hypothetical try lock based on the CLH-NUMA lock.

6. ACKNOWLEDGMENTS

We are indebted to Vitaly Oratovsky and Michael O'Donnell of Mercury Computer Corp. for drawing our attention to the subject of queue-based try locks, and to Mark Hill, Alaa Alameldeen, and the Computer Sciences Department at the University of Wisconsin–Madison for the use of their Sun Wildfire machine.

7. REFERENCES

[1] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE TPDS*, 1(1):6–16, Jan. 1990.

[2] T. S. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, Dept. of Computer Science, Univ. of Washington, Feb. 1993.

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

[4] E. Hagersten and M. Koster. Wildfire: A scalable path for SMPs. In *5th HPCA*, pages 172–181, Orlando, FL, Jan. 1999.

[5] M. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, Jan. 1991.

[6] L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott. Scheduler-conscious synchronization. *ACM TOCS*, 15(1):3–40, Feb. 1997.

[7] P. Magnussen, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *8th IPPS*, pages 165–171, Cancun, Mexico, Apr. 1994. Expanded version available as "Efficient Software Synchronization on Large Cache Coherent Multiprocessors", SICS Research Report T94:07, Swedish Institute of Computer Science, Feb. 1994.

[8] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM TOCS*, 9(1):21–65, Feb. 1991.

[9] V. Oratovsky and M. O'Donnell. Personal communication, Feb. 2000. Mercury Computer Corp.