Non-Blocking Timeout in Scalable Queue-Based Spin Locks

Michael L. Scott

Technical Report #773

Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
`scott@cs.rochester.edu`[*]

February 2002

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. Timeout-capable spin locks allow a thread to abandon its attempt to acquire a lock; they are used widely in real-time systems to avoid overshooting a deadline, and in database systems to recover from transaction deadlock and to tolerate preemption of the thread that holds a lock.

In previous work we showed how to incorporate timeout in scalable queue-based locks. Technological trends suggest that this combination will be of increasing commercial importance. Our previous solutions, however, require a thread that is timing out to handshake with its neighbors in the queue, a requirement that may lead to indefinite delay in a preemptively multiprogrammed system.

In the current paper we present new queue-based locks in which the timeout code is non-blocking. These locks sacrifice the constant worst-case space per thread of our previous algorithms, but allow us to bound the time that a thread may be delayed by preemption of its peers. We present empirical results indicating that space needs are modest in practice, and that performance scales well to large machines. We also argue that constant per-thread space cannot be guaranteed together with non-blocking timeout in a queue-based lock.

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 [2, 5, 7, 13, 15] avoid contention by arranging for every waiting thread to spin on a separate, local flag in memory. Over the past ten years queue-based locks have been incorporated into a variety of academic and commercial operating systems,

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

including Compaq's Tru64, IBM's K42 and multiprocessor Linux systems, the Alewife [1] and Hurricane [18] systems, and parallel real-time software from Mercury Computer Systems.

Outside the operating system, non-scalable test-and-set locks have come to be widely used in commercially important applications, notably database systems such as Oracle's Parallel Server and IBM's DB2. Many of these applications depend critically on the ability of a thread that waits "too long" to time out and abandon its attempt to acquire a lock. Timeout-capable "try locks" allow a real-time application to signal an error condition or pursue an alternative code path. In a database system, they provide a simple means of recovering from transaction deadlock.

Unfortunately, until recently it was not clear how to combine scalability and timeout. The problem is that while threads competing for a test-and-set lock are mutually anonymous, and can abandon their spins without anyone being the wiser, threads in a queue-based lock are linked into an explicit data structure. A timed-out thread must somehow introduce its neighbors in the queue to one another, even in cases where the neighbors may also be timing out.

In a recent paper [17] we introduced timeout-capable queue-based try locks based on our MCS lock [15] and on the related CLH lock, due to Craig [5] and to Landin and Hagersten [13]. These locks perform well on large machines, and require only $O(L + T)$ total space for L locks and T threads. Unfortunately, they require that a departing thread "handshake" with its neighbors in the queue in order to determine when all the references to a queue node have been updated, and the node can be reclaimed. This handshaking is not a problem in a system with one thread per processor, but it fails on multiprogrammed systems, in which a neighbor thread may be preempted, and thus unable to cooperate.

The problem of preemption in critical sections has received considerable attention over the years. Alternative strategies include avoidance [6, 10, 14], recovery [3, 4], and tolerance [9, 16]. The latter approach is appealing for commercial applications because it does not require modification of the kernel interface: if a thread waits "too long" for a lock, it assumes that the lock holder has been preempted. It abandons its attempt, yields the processor to another thread (assuming there are plenty) and tries again at a later time. In database systems timeout serves the dual purpose of deadlock recovery and preemption tolerance.

In this paper we introduce a pair of queue-based spin locks—the CLH-NB try lock and the MCS-NB try lock—in which timeout is *non-blocking*: a thread that decides to abandon its attempt to acquire a lock can do so without waiting for activity by any other thread. In order to minimize space overhead, we attempt to reclaim queue nodes as soon as possible, but—the price of preemption safety—there are pathological schedules in which our algorithms may require unbounded space.

We introduce our algorithms in section 2. We also argue that it is impossible in any queue based lock to combine non-blocking timeout with an $O(L + T)$ space bound. In section 3 we compare the performance of our new locks to existing `test_and_set` and queue-based locks on large-scale and multiprogrammed multiprocessors. With threads on 64 processors attempting to acquire a lock simultaneously, we identify cases in which a traditional test-and-set lock (with backoff) is more than six times slower than our CLH-NB try lock, while failing (timing out) more than 22 times as often. In experiments with more threads than processors, we also demonstrate clearly the performance advantage of non-blocking timeout. We return in section 4 to a summary of conclusions and directions for future work.

2 Algorithms

In the subsections below we describe a pair of queue-based spin locks in which a waiting thread, once it decides to leave the queue, can do so within a bounded number of its own time steps. Code for both of these locks can be found in the appendix. The CLH-NB try lock is the simpler of the two, but relies on cache coherence. The MCS-NB try lock can be expected to scale better on non-cache-coherent machines. We have developed informal correctness arguments for each of the locks. We are currently working on a model-checking tool that should help to formalize these arguments.

In the original CLH [13] and MCS [15] locks, and in the CLH try and MCS try locks [17], space management for queue nodes is delegated to the callers of the `acquire` and `release` operations, and the queue node passed to `MCS_release` or returned from `CLH_release` is guaranteed to be available for immediate reuse once the `release` operation completes. For reasons discussed in section 2.4, no such guarantee can be made for locks with non-blocking timeout. We therefore choose in the CLH-NB try and MCS-NB try locks to perform dynamic space allocation within the `acquire` and `release` operations. To allow the `release` operation to find the queue node allocated by the `acquire` operation, we arrange for `acquire` to write a reference to that node into an extra field (a head pointer) of the lock variable itself, once the lock is held. A serendipitous side effect of this strategy is that the CLH-NB try and MCS-NB try locks can employ a standard API, making them suitable for linking with binary-only commercial applications.

2.1 CLH-NB try lock

Our first new algorithm is based on the lock of Craig [5] and of Landin and Hagersten [13]. A lock variable takes the form of a tail pointer for a singly linked list of queue nodes. A thread that wishes to acquire the lock allocates a node, swaps it into the tail pointer, and then spins on a flag in the node ahead of it in line, which was returned by the `swap`.

We have modified this lock to allow non-blocking timeout. In our version individual queue nodes contain only a single pointer. When nil this pointer indicates that the thread spinning on the node must wait. When set to `AVAILABLE` (a value we assume to be different from any valid reference), the pointer indicates that the lock is available to the thread spinning on the node. When neither nil nor `AVAILABLE`, the pointer contains a reference to the previous node in the list, and (in a departure from the original version of the lock) indicates that the thread that allocated the node containing the pointer has timed out. Up until its decision to time out, a thread maintains its reference to the node on which it is spinning in a local variable, rather than its queue node (indicated in the figure by starting the tail of an arrow in the empty space below a queue node).

In the event of timeout two principal cases arise, illustrated in figure 1. In the left-hand portion of the figure, departing thread B is in the middle of the queue, spinning on the pointer in the node allocated by thread A. When B times out it indicates its intent to leave by storing into its own queue node a reference to A's node. Thread C, which is spinning on B's node, notices this change. It updates its own local pointer to refer to A's node instead of B's, and then reclaims B's node.

Unfortunately, B cannot be certain that C exists. The case where it does not is illustrated in the right-hand side of figure 1. After writing the reference to A's queue node into its own queue node, B performs a `compare_and_swap` operation on the queue tail pointer, in an attempt to change it from a reference to B's node into a reference to A's node. In the middle-of-the-queue case (left)

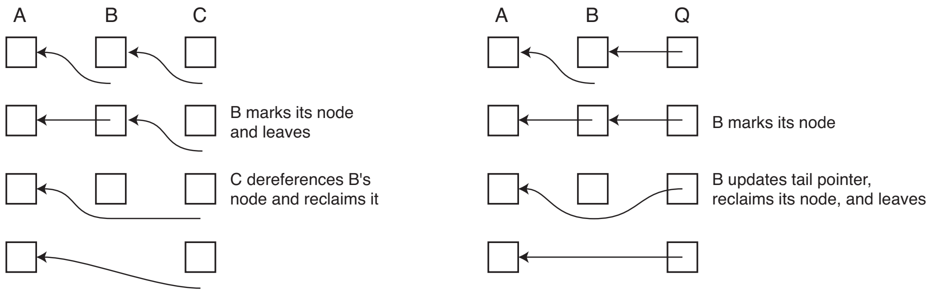

A
B
C
B marks its node and leaves
C dereferences B's node and reclaims it
A
B
Q
B marks its node
B updates tail pointer, reclaims its node, and leaves

Figure 1: Timeout in the CLH-NB try lock, with departing thread B in the middle (left) or at the end (right) of the queue.

this operation will fail. In the end-of-the-queue case (right) it succeeds, and B knows that it can reclaim its own queue node. In either case B can return as soon as it has attempted the `compare_and_swap`; it does not have to wait for C. If the `compare_and_swap` failed, B's queue node will not be available for reuse until it is reclaimed by C, or by some other, later thread, if C has also timed out.

In our realization of the CLH-NB try lock (see appendix) we have made one additional departure from the original CLH lock. By analogy to the end-of-queue case for timeout, we can eliminate the extra, "dummy" node in an uncontended lock by performing a `compare_and_swap` in the `release` operation. This extra atomic operation increases the overhead of every critical section, but reduces by one word the size of an unheld lock. Of course, we added a word to the lock in order to hold the head pointer that allows us to use a standard API; in effect, we have chosen to expend a bit of time in order to "buy back" this extra space.

2.1.1 Correctness sketch

We take as given that the plain CLH lock algorithm is correct: specifically, that its queueing discipline ensures mutual exclusion and liveness, and grants requests in FIFO order; and that it manages space correctly, never accessing a queue node after it has been returned from a `release` operation. We must argue that the CLH-NB try lock also maintains the queue and manages space correctly.

Consider first the issue of space management. Because queue nodes are allocated dynamically, we must show that all nodes are reclaimed, and that they are never again accessed once reclaimed. We do this by identifying a unique *responsible reference* (an "rref") to every queue node at every point in time. When a node Y is first allocated, its rref, R, resides in a local variable of the allocating thread, B. B moves R into the lock tail pointer (relinquishing responsibility for the node) when it performs the `swap` operation at the beginning of `acquire`. At the same time, if the `swap` does not return nil, B acquires a different rref, Q for its predecessor's queue node, X. Under normal (no timeout) circumstances, ownership of Q obligates B to eventually reclaim X. If B times out, however, it must delegate this responsibility. It does so by writing Q into its own queue node, Y, where it (Q) will become the property of whatever thread holds the rref for Y (R). The exception occurs when Y is the last node in the queue, in which case R resides in the lock's tail pointer. In this case B can reacquire R by performing a `compare_and_swap` that simultaneously writes Q into the lock's tail pointer.

Because a queue node has a unique rref at any given point in time, and because a node is reclaimed only by a thread that owns an rref (which it immediately discards), no node is reclaimed twice. Moreover since every rref is always either in the possession of a thread currently executing an `acquire` operation or else in memory in a node currently in the queue, every node is eventually reclaimed.

Because reclaimed queue nodes may be reused, we have to be careful to avoid the so-called *ABA problem*, in which a reference to a newly allocated node is mistaken for a reference to a previously reclaimed node. Specifically, once thread B writes rref Q into node Y, B's successor may reclaim Y. If Y's space is recycled quickly and used for some new queue node X, which is used in an attempt to acquire the same lock for which Y was used, B's `compare_and_swap` operation may succeed when it should not. We can eliminate this possibility, in this particular case, by using a memory allocator in which a given block of storage is always allocated by the same thread. Then Y's space, which was allocated by B, will be reused only by B, and only *after* B has attempted the `compare_and_swap` operation in which the ABA problem arises. Code for our space management routines appears in the appendix.

Mutual exclusion and liveness can be verified by envisioning access to the critical section as a mutex token that passes from thread to thread. The token takes three states: (1) In an unheld, uncontended lock, the token is embodied in the fact that the lock tail pointer is nil. (2) In a contended but momentarily available lock, the token takes the form of an `AVAILABLE` value in the `prev` pointer of a queue node. (3) When a lock is held, the token is embodied in the program counter of the thread that holds the lock.

When the `swap` operation in `acquire` returns nil, the mutex token changes from state 1 to state 3. When a thread writes `AVAILABLE` into its queue node, the token changes from state 3 to state 2. When a thread discovers `AVAILABLE` in a queue node and then reclaims that node, the token changes from state 2 to state 3. When the `compare_and_swap` operation in `release` succeeds, the token changes from state 3 to state 1. The use of atomic operations on the queue tail pointer ensures that only one thread can move the token from state 1 to state 3. The only remaining issues concern a token in state 2: we must verify that it can never be lost, and that it can be moved to state 3 by only one thread. For the former, we note that (a) every `release` operation moves the token to state 1 or 2, (b) a token in state 2 is never overwritten, because the only other code that writes to the pointer field of a queue node appears in the already-completed `acquire` operation, and (c) a node containing a token in state 2 is never reclaimed without recovering the token because, as noted with respect to space management above, a node is reclaimed only by the thread that holds the responsible reference, and only when that thread has either (i) just acquired the lock (token in state 3), or (ii) noticed that the lock contains an rref, placed there by the thread that allocated the node (and that has therefore timed out, and won't be obtaining the token at all).

2.2 MCS-NB try lock

Our second new algorithm is based on the lock of Mellor-Crummey and Scott [15]. As in the previous section, a lock variable takes the form of a tail pointer for a list of queue nodes, but where the CLH queue was linked from tail to head, the bulk of the MCS queue is linked from head to tail. After swapping a reference to its own queue node into the tail pointer, a thread writes an additional reference to its node into the node of its predecessor. It then proceeds to spin on its own node, rather

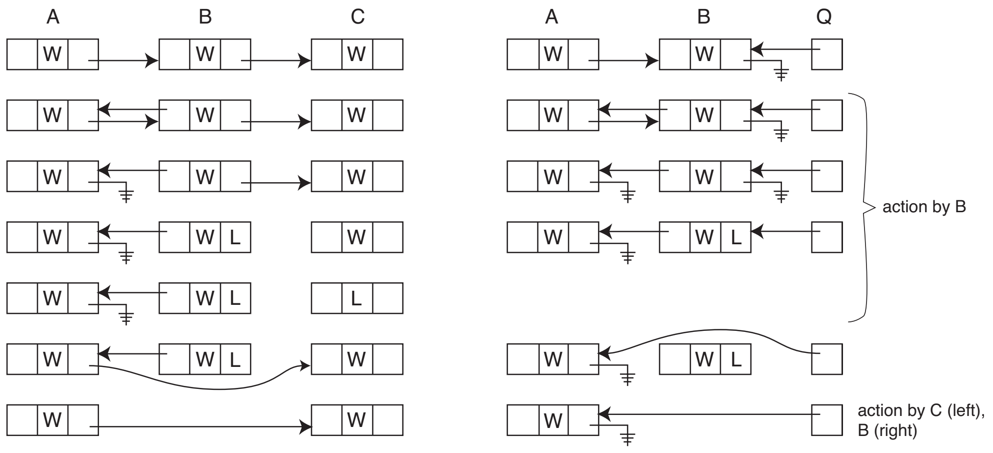


Figure 2: Timeout in the MCS-NB try lock, with departing thread B in the middle (left) or at the end (right) of the queue. For clarity, references to predecessor nodes that are held in local variables, rather than in qnodes, are not shown.

than the predecessor's node. This "backward" linking allows a thread to spin on a location that is guaranteed to be local even on a non-cache-coherent machine. Unfortunately, it also makes timeout significantly more complex.

In our timeout-capable MCS-NB try lock, illustrated in figure 2, queue nodes contain a status flag and a `pair` of pointers, used to link the queue in both directions. As in the plain MCS lock, the backward (`next`) pointer in node Y allows the thread B that allocated Y to find the node on which a successor thread is spinning. When nil, Y's `next` pointer indicates that no successor node is known. Three additional values, assumed not to be the same as any valid reference, correspond to special states. When set to `AVAILABLE`, Y's `next` pointer indicates that the lock is currently available. When set to `LEAVING`, it indicates that B has timed out and, further, that no `next` pointer anywhere refers to Y. When set to `TRANSIENT`, Y's `next` pointer also indicates that B has timed out, but that in doing so B was unable to break the reference to Y from its predecessor node.

The status word of a queue node has five possible values. Before linking its node into the queue, a thread initializes its status word to `waiting`. Once the link-in operation is complete, the thread will spin waiting for the value to change. Three possible values—`available`, `leaving`, and `transient`—mirror the special values of node `next` pointers described in the previous paragraph. The final value—`recycled`—accommodates race conditions in which two threads have references to a node that needs to be reclaimed. Whichever thread uses its pointer last will find the `recycled` flag, and know that it is responsible for reclamation.

When a thread C performs an initial `swap` operation on the tail pointer of a lock that is not currently available, it receives back a reference to the queue node Y allocated by C's predecessor, B. C swaps a reference to Z into Y's `next` pointer. By using a swap, rather than an ordinary write (as in the original MCS lock), C can recognize the timing window in which B decides to release the lock or to leave the queue when C has already swapped itself into the tail of the queue, but before C has updated Y's `next` pointer. Note that in the plain MCS lock a thread may spin in the `release` operation, waiting for its successor to update its `next` pointer. We must eliminate spins such as this in order to ensure non-blocking timeout.

If the swap on Y's `next` pointer returns `AVAILABLE`, C knows that it has the lock. Moreover B's `compare_and_swap` on the lock tail pointer is guaranteed to fail, because C's original swap on the tail pointer removed the reference to Y. C therefore knows that B will neither update Z nor reclaim Y, so C reclaims Y, writes a reference to Z into the head pointer field of the lock, and returns successfully.

If the swap on Y's `next` pointer returns `LEAVING`, C knows that B has timed out. It also knows, for reasons similar to those in the preceding paragraph, that B will neither update Z nor reclaim Y. C updates its local `pred` pointer to contain the reference found in Y's `prev` pointer, instead of a reference to Y. C then reclaims Y and tries again to link itself into line, using the updated `pred`.

Finally, if the swap on Y's `next` pointer returns `TRANSIENT`, C knows that B has timed out, but that B's predecessor, A, has a reference to Y, and is planning to use it. Whichever thread, A or C, accesses Y last will need to reclaim it. C swaps a `recycled` flag into Y's status word. If the return value of the swap is `waiting`, C knows that it has accessed Y before A, and that A will take responsiblity for reclaiming it. If the return value of the swap is `available`, `leaving`, or `transient`, however, C knows that A has already accessed Y. C therefore reclaims Y. In either case, C updates its local `pred` pointer and tries to link itself into line again, as in the preceding paragraph. Seen from A's perspective, any time we update the status word of a successor queue node we use a swap operation to do so, and reclaim the node if the return value is `recycled`.

Once successfully linked into the queue, thread C spins on the status word in its own queue node, Z. If that word changes to `available`, C writes a reference to Z into the head pointer field of the lock, and returns successfully. If Z's status word changes to `leaving` or `transient`, C resets it to `waiting` and then behaves as it would have in the preceding paragraphs, had it found `LEAVING` or `TRANSIENT` in the `next` pointer of its predecessor's queue node, Y.

If C times out in the the algorithm's inner loop, spinning on Z's status word, it first stores its local `pred` pointer into Z's `prev` pointer. It then attempts to erase the reference to Z found in Y's `next` pointer, using a `compare_and_swap` operation. If that attempt succeeds, C swaps `LEAVING` into Z's `next` pointer and, if necessary, swaps `leaving` into the status word of Z's successor node. As described above, C reclaims the successor node if the status word was already set to `recycled`. Finally, if Z appears to have no successor, C attempts to link it out of the end of the queue with a `compare_and_swap` and, if that operation succeeds, reclaims Z.

If C fails to erase the reference to Z found in Y's `next` pointer, then it knows its predecessor B will try to update Z's status word. It therefore swaps `TRANSIENT` into Z's `next` pointer and, if necessary, swaps `transient` into the status word of Z's successor node, reclaiming that node if its status word was already `recycled`. If Z appears to have no successor, then C must simply abandon it, to be reclaimed by some thread that calls the `acquire` operation at some point in the future.

If C times out in the algorithm's outer loop, while attempting to update a predecessor's `next` pointer, it mimics the case of timeout in the inner loop: it restores its predecessor's `next` pointer, sets Z's status to `leaving` or `transient`, as appropriate, and then takes the actions described in one of the preceding two paragraphs.

2.2.1 Correctness sketch

Again we take as given that the initial MCS lock algorithm is correct. As with the CLH-NB try lock, we must argue that the MCS-NB try lock maintains its queue and manages space correctly.

We again associate a unique responsible reference with every queue node at every point in time. This rref may reside in (a) the tail pointer of the queue, (b) the local `pred` pointer of some waiting thread, or (c) the `prev` pointer of some queue node. By tracing through the code we can identify points at which the rref for a given node moves from one of these places to another. The principal complication, in comparison to the CLH-NB try lock, is that an additional reference often exists in the `next` pointer of a predecessor node. Before reclaiming a node a thread must ensure that no such additional reference exists.

In the usual (no timeout) case, a node Y is reclaimed when thread C, spinning on successor node Z, realizes that the lock is available. At this point the predecessor of Y, if any, has already been reclaimed, and thus cannot hold a reference to Y. If thread B, which allocated Y, times out, it attempts, using a `compare_and_swap`, to destroy the reference to Y that resides in the `next` pointer of Y's predecessor, X, as outlined in the description of the algorithm above. If the `compare_and_swap` succeeds, B knows that only the responsible reference remains, and that Y can safely be reclaimed by C, if it exists, or by B itself if Y is at the tail of the queue. (As with the CLH-NB try lock, the use of pre-thread space management protects us from the ABA problem.)

If B's `compare_and_swap` on X's `next` pointer fails, B marks Y as `TRANSIENT`, rather than `LEAVING`, so that C will know that it is racing against A (the thread that allocated X) in an attempt to update Y's status word. If C finds `available`, `leaving`, or `transient` in Y's status word, rather than `waiting`, then it knows that A has already performed its update and has discarded its extra reference to Y; in this case C can reclaim Y. If on the other hand C finds `waiting` in Y's status word, we interpret C's swap of `recycled` into that word as a declaration that A's pointer is now the responsible reference. When A finds the `recycled` flag it will know that it shoud reclaim Y.

As in section 2.1.1, we verify mutual exclusion and liveness by modeling possession of the lock as a mutex token that passes from thread to thread. In the current algorithm, however, our model must encompass both `AVAILABLE` flags in `next` pointers and `available` flags in status words.

When thread A finishes its critical section and calls the `release` operation, it swaps `AVAIL-ABLE` into the `next` pointer of its queue node, X. If the return value of the swap is nil, then we say the token has moved to X. A then performs a `compare_and_swap` on the lock tail pointer in an attempt to change it from a reference to X to nil; if this operation succeeds, we say the token has moved to the lock tail pointer. If on the other hand A's original swap on X's `next` pointer returns a reference to a queue node, Y, we say that the token has *not* been placed in X, because A's successor, B, has already swapped its reference to Y into X's `next` pointer. It will not inspect that pointer again unless it times out, and even then it will use a `compare_and_swap` that leaves the pointer unmodified (and ignores its current value), if the pointer no longer refers to Y (as of course it won't, once A has swapped `AVAILABLE` into it).

In response to swapping a reference to Y out of X's `next` pointer, A swaps `available` into Y's status word. This swap moves the token into Y, where it will be found by B or, if B has timed out, by some successor C of B.

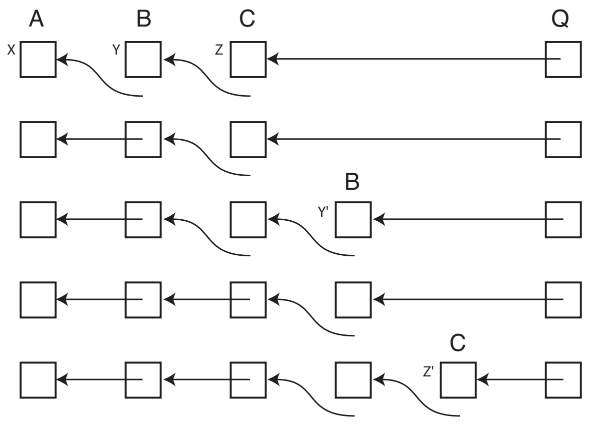


Figure 3: Worst-case scenario for space in the CLH-NB try lock.

2.3 Space requirements

Unfortunately, in order to avoid any spins in time-out code, we must generally return from an unsuccessful `acquire` operation without having reclaimed our queue node (that task having been left to some successor thread). As a result, we lose the $O(L + T)$ overall space bound of the CLH try lock and the MCS try lock, with L locks and T threads.

Perhaps the simplest pathological scenario occurs in either lock when the last thread in line is preempted. If the second-to-last thread then times out, its node may go unreclaimed for an arbitrarily long time. If the third-to-last thread subsequently times out its node may go unreclaimed as well, and so on.

Victor Luchangco of Sun Labs has observed [12] that worst-case space needs are in fact unbounded, with as few as three active threads in the CLH-NB try lock (see figure 3). Suppose initially that threads A, B, and C are waiting for the lock. Suppose then that B and C decide to leave at approximately the same time, and stop spinning on nodes X and Y. B then writes a reference to X into Y, but C is preempted before it can write a reference to Y into Z. B's `compare_and_swap` on the lock tail pointer will fail, because Z is in the way, and B will return from `acquire` without having reclaimed Y. If B requests the lock again it will get into line with a new queue node, call it Y'. Suppose that B then times out again, decides to leave the queue, and stops spinning on Z. Only now, let us suppose, does C wake up again and write a reference to Y into Z. C's `compare_and_swap` on the lock tail poitner will fail because Y' is in the way, and C will return from `acquire` without having reclaimed Z. This scenario can, in principle, repeat indefinitely. A similar scenario exists for the MCS-NB try lock.

2.4 Impossibility argument

Ideally, one might hope to design a queue-based spin lock with non-blocking timeout *and* an $O(L+T)$ space bound. We argue in this section that no such lock is possible.

Imagine a lock on which N threads are waiting (figure 4). Suppose now that $N - 2$ of these threads—all but the first and the last—decide to leave at essentially the same time. Imagine further

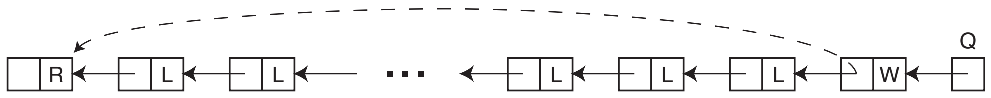


Figure 4: Impossible scenario for non-blocking timeout and constant space per thread.

that the last thread in line has been preempted, and that the first thread, which has the lock, is in a very long critical section. The departing threads would all like to complete their timeout operations in a bounded number of their own local time steps. In order to reclaim their space, however, we must arrange to introduce the remaining threads (the first and the last) to each other, in order to maintain the integrity of the queue. But because the queue embodies only local knowledge, we must perform $O(N)$ work in order to make this introduction. While a hypothetical highly clever algorithm might be able to perform this work in $O(\log N)$ time using a parallel prefix-like strategy [8], there is no way we can hope to do it in constant time.

It would be easy, of course, to obtain an $O(L \times T)$ overall space bound, by remembering the last queue node used by thread T in its attempt to acquire lock L. The next time T tried to acquire L it could check to see if the node were still in L's queue, in which case T could resume waiting where it was when it last timed out. This mechanism would have significant time cost, however, and seems unwarranted in practice.

3 Performance results

We have implemented eight different lock algorithms using the `swap` and `compare_and_swap` operations available in the Sparc V9 instruction set: TAS-B, TAS-B try, CLH, CLH try, CLH-NB try, MCS, MCS try, and MCS-NB try. Our tests employ a microbenchmark consisting of a tight loop containing a single acquire/release pair and optional timed "busywork" inside and outside the critical section.

`Acquire` and `release` operations are implemented as in-line subroutines wherever feasible. Specifically: for CLH and MCS we in-line both `acquire` and `release`. For TAS-B, TAS-B try, and CLH try we in-line `release` and the "fast path" of `acquire` (with an embedded call to a true subroutine if the lock is not immediately available). For MCS try we in-line the fast path of both `acquire` and `release`. For CLH-NB try and MCS-NB try the need for dynamic memory allocation forces us to implement both `acquire` and `release` as true subroutines.

Performance results were collected on an otherwise unloaded 64-processor Sun Enterprise 10000 multiprocessor, with 466MHz Ultrasparc 2 processors. Assignment of threads to processors was left to the operating system. Code was compiled with the –O3 level of optimization in gcc version 2.8.1, but was not otherwise hand-tuned.

3.1 Single-processor results

We can obtain an estimate of lock overhead in the absence of contention by running the microbenchmark on a single processor, with no critical or non-critical "busywork", and then subtracting out the loop overhead. Results appear in table 1. The first column gives measured processor cycles on the

	cycles	atomic ops	reads	writes
TAS-B	19	1	0	1
TAS-B try	19	1	0	1
CLH	35	1	3	4
CLH try	67	2	3	3
CLH-NB try	75	2	3	4
MCS	59	2	2	1
MCS try	59	2	2	1
MCS-NB try	91	3	3	4

Table 1: Single-processor (no-contention, in-cache) spin-lock overhead.

Enterprise 10000. In an attempt to avoid perturbation due to kernel activity, we have reported the minima over a series of 8 runs. The remaining columns indicate the number of atomic operations (`swap`s and `compare_and_swap`s), shared-memory reads, and shared-memory writes found in the fast path of each algorithm. The times for the CLH-NB and MCS-NB try locks include dynamic allocation and deallocation of queue nodes.

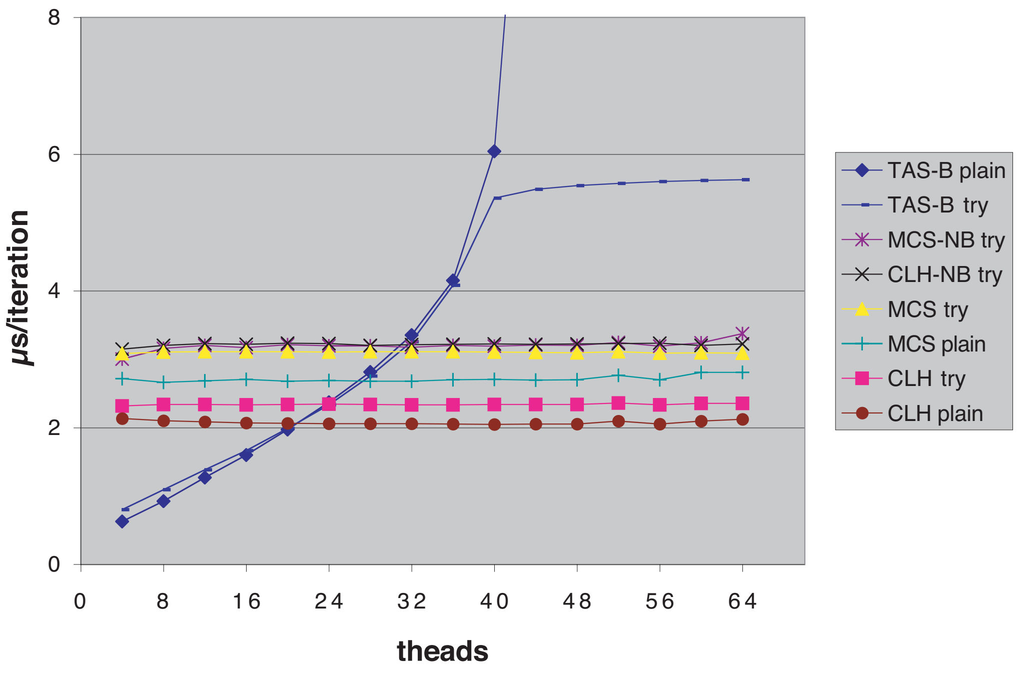
As one might expect, none of the queue-based locks is able to match the time of the TAS-B lock. The closest competitor, the plain CLH lock, takes nearly twice as long. Atomic operations are the single largest contributor to overhead. The CLH-NB try and MCS-NB try locks, which are not in-lined, also pay a significant penalty for subroutine linkage.

3.2 Overhead on multiple processors

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 [13], 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. In recognition of this difference, we subtract from microbenchmark iteration times only the critical section "busywork", not the loop overhead or other non-critical work.

Figure 5 shows the behaviors of all eight locks on the Enterprise 10000, with timeout threshold (patience) of $225\mu s$, non-critical busywork of $440ns$ (50 iterations of an empty loop), and critical section busywork of $229ns$ (25 iterations of the loop). With a lock-passing time of about $3.4\mu s$ in the MCS-NB try lock, there isn't quite enough time to finish 63 critical sections before the 64th thread times out ($(3400 + 229) \times 63 > 225,000$). As a result the success rate of the MCS-NB try lock drops sharply at the right end of the graph, and the CLH-NB try lock is just reaching the drop-off point. The TAS-B try lock, on the other hand, suffers a severe increase in passing time around 36 processors, with a corresponding drop-off in success rate. Passing time for the TAS-B lock without timeout has been omitted beyond 40 processors so as not to distort the scale of the graph. At 64 processors it is $45.0\mu s$.

Below about 20 processors the TAS-B locks appear to outperform all competitors, but this appearance 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



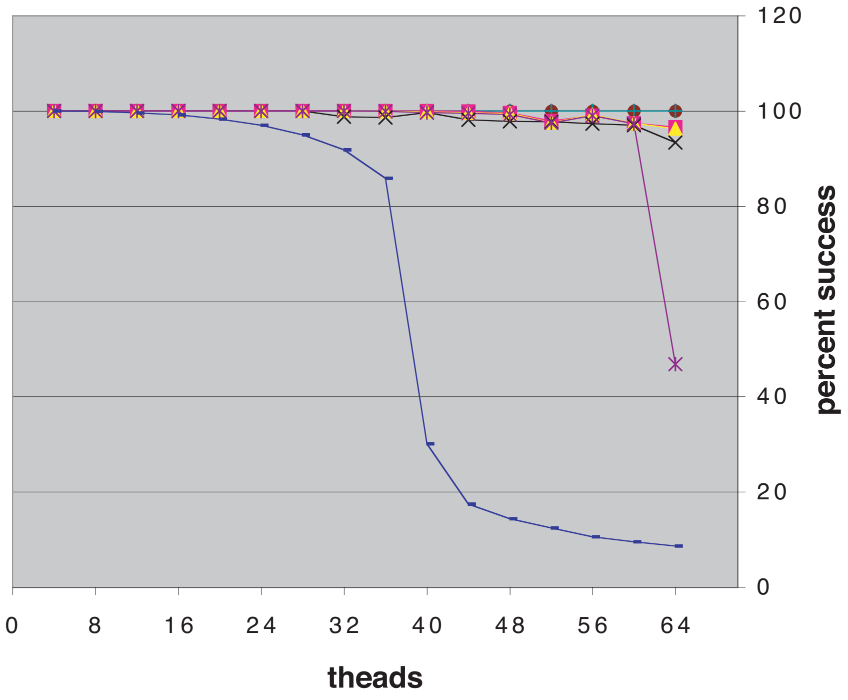


Figure 5: Microbenchmark net iteration time (left) and success rate (right) with patience of $225\mu s$, non-critical busywork of $440ns$ and critical section busywork of $229ns$. Below 36 processors the TAS locks are acquired consecutively by the same processor more than half the time.

has the advantage of cache locality, it tends to outrace its peers and acquire the lock repeatedly. At 20 processors, in fact, the TAS-B locks are "handed off" from one processor to another only about 30% of the time, despite the fact that each thread performs $440ns$ of busywork between its critical sections. Not until more than 36 processors are active does the handoff rate rise above 50%. System designers considering the use of a TAS-B lock may need to consider whether this unfairness is acceptable in the event of severe contention.

3.3 The impact of preemption

In an attempt to assess the benefits and cost of non-blocking timeout, we have also collected results on a preemptively scheduled system with more threads than processors. Specifically, we ran our microbenchmark with 8–16 threads on an 8-processor Sun Enterprise 4500, with 336MHz processors. With increasing numbers of threads comes an increasing chance of preemption, not only in the critical section, but while waiting in the queue. Under these circumstances we would expect the CLH-NB and MCS-NB try locks to outperform the handshake-based CLH and MCS try locks. Our results confirm this expectation.

Figure 6 plots iteration time and `acquire` success rate against number of threads for our preemption sensitivity experiment. Results were averaged over 16 runs, each of which performed 100,000 `acquire/release` pairs per thread. The timeout threshold (patience) was chosen to produce a modestly overloaded system when running with one thread on each of the machine's 8 processors. As discussed below, the meaning of "iteration time" is rather complicated in this experiment. The numbers plotted in the left side of the figure are T_s/ti, where T_s is total wall clock time, t is the number of threads, and i is the number of iterations performed by each thread.

As the number of threads exceeds the number of processors, the success rate plummets, due primarily to preemption of threads in their critical sections. The difference between blocking and non-blocking timeout then becomes sharply visible in the left-hand graph. The CLH-NB and MCS-NB try locks are able to bound the amount of time that a thread spends waiting for an unavailable lock; the CLH and MCS try locks cannot.

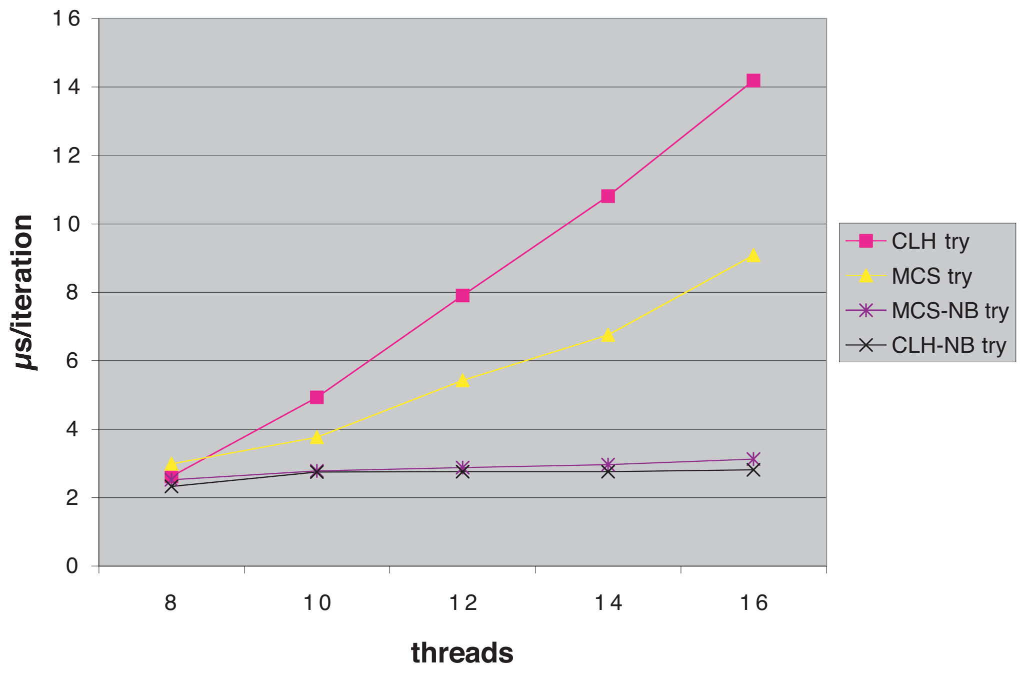
We can model iteration time in this experiment in two related ways. First, successful `acquire` operations introduce critical sections, which exclude one another in time. Total wall clock time should therefore equal the number of successful `acquire` operations times the average cost (passing time, critical section busywork, and time spent preempted) of a single critical section. Let T_a be lock passing time, T_c be critical section busywork, t again be the number of threads, and i again be the number of iterations executed by each thread. Now measure s, the `acquire` operation success rate, and T_s, the total wall clock time. We can estimate T_x, the average time per critical section spent preempted, via the following equations:

$$T_s = sti(T_a + T_c + T_x)$$

$$T_x = \frac{T_s}{sti} - (T_a + T_c)$$

Note that T_a can be estimated based on experiments with ample patience and a dedicated thread per processor.

Second, failed `acquire` operations and the busy-waiting prior to successful `acquire` operations occur more-or-less in parallel. Total wall clock time should therefore equal the total number



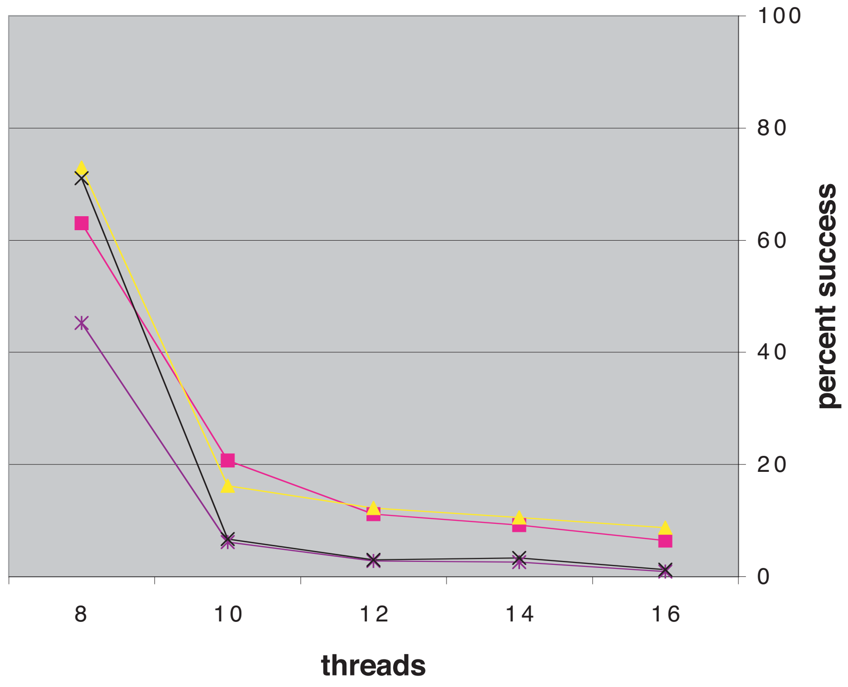


Figure 6: Microbenchmark iteration time (left) and success rate (right) with patience of $15\mu s$ and critical section busywork of $305ns$ (25 iterations of empty loop) on an overburdened 8-processor machine.

of unsuccessful `acquire` operations times the average cost (loop overhead, patience, and timeout [handshake] time) of a single failed `acquire`, plus the total number of successful `acquire` operations times the average wait time, all divided by the number of processors not busy on the critical path (i.e. one fewer than the total number of processors).

Let m be the number of processors in the machine, T_p be patience, and T_l be loop overhead. If we let T_w represent the average lock wait time, then we can estimate T_h, the time required for timeout (including handshaking if necessary) via the following equations:

$$\begin{aligned}
T_s &= \frac{ti}{m-1}[T_l + (1-s)(T_p + T_h) + sT_w] \\
&\leq \frac{ti}{m-1}[T_l + T_p + (1-s)T_h] \\
T_h &\geq \frac{(m-1)T_s}{(1-s)ti} - \frac{T_l + T_p}{1-s}
\end{aligned}$$

Here we have exploited the fact that $T_w \geq T_p$. T_l can be estimated based on single-processor experiments.

Figure 7 plots our estimates of T_x and T_h for the experiments depicted in figure 6, with $t > 8$ threads. Values for T_x vary greatly from run to run, reflecting the fact that preemption in a critical section is relatively rare, but very expensive. Variations among algorithms in preempted time per critical section can be attributed to the rate of success of `acquire` operations and, to a lesser extent, lock overhead. Higher rates of success and lower overhead increase the percentage of time that a thread is in its critical section, and thus the likelihood that it will be preempted there.

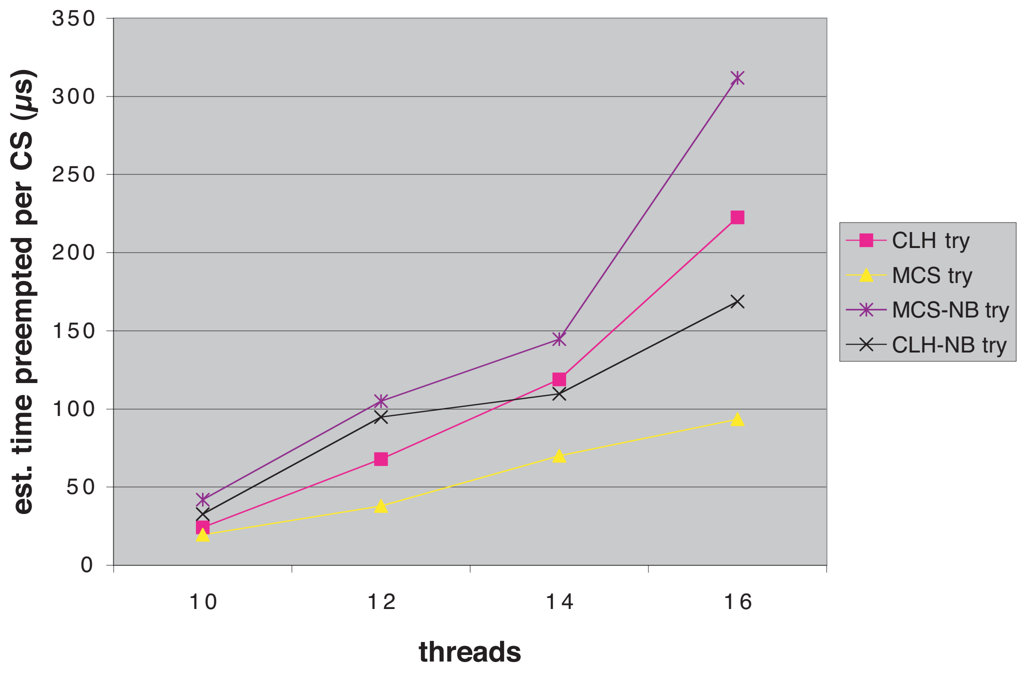
The right-hand side of figure 7 is in some sense the "punch line" of this paper: with the CLH-NB and MCS-NB try locks, a thread can leave the queue within a modest constant amount of time. In the CLH and MCS try locks it can be arbitrarily delayed by the preemption of a neighboring thread.

The careful reader will note that the times given in the right-hand side of figure 7 are significantly larger than the "times" given in the left-hand size of figure 6. By dividing wall clock time (T_s) by the total number of `acquire` attempts (ti), figure 6 effectively pretends that all those operations happen sequentially. The calculations behind figure 7 recognize that much of the work occurs in parallel.

3.4 Space overhead

As part of the experiments reported in the previous section, we instrumented our space management routines (see appendix) to remember the maximum number of queue nodes ever extant at any time. Across the sixteen measured runs, encompassing six million `acquire/release` pairs, the maximum number of allocated queue nodes was 84, or roughly 5 per thread. The CLH-NB and MCS-NB try locks appear to be roughly comparable in the number of nodes they require.

Given that our experiment was deliberately designed to induce an unreasonably high level of lock contention, and to maximize the chance of inopportune preemption, we find this relatively modest maximum number of queue nodes reassuring. We would not expect space overhead to be an obstacle to the use of non-blocking timeout in any realistic setting.



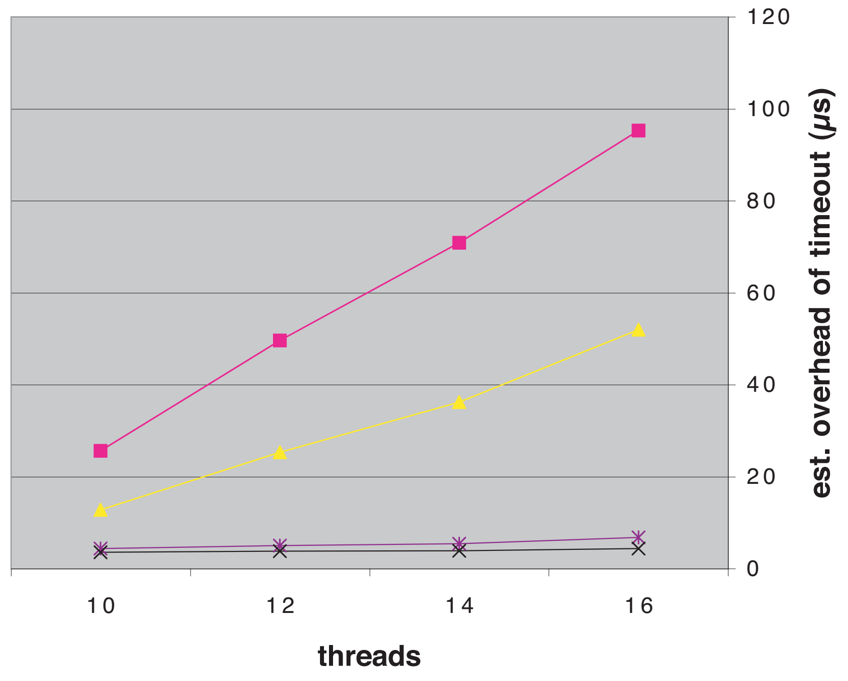


Figure 7: Calculated values of T_x (left) and T_h (right), with patience of $16.5\mu s$ and critical section busywork of $305ns$ on the 8-processor Enterprise machine.

4 Conclusions

We have shown that it is possible, given standard atomic operations, to construct queue-based locks in which a thread can time out and abandon its attempt to acquire the lock. Our previous algorithms guaranteed immediate reclamation of abandoned queue nodes, but required that a departing thread obtain the cooperation of its neighbors. Our new algorithms incorporate non-blocking timeout code, and can safely be used in the presence of preemption (assuming, of course, that the processor can be put to other use while waiting for the preempted lock holder to be rescheduled).

The price of non-blocking timeout is an unbounded worst-case requirement for space. We have argued, however, that large amounts of space are unlikely to be required in practice, and our experimental results support this argument.

Results obtained on a 64-processor Sun Enterprise 10000 indicate that traditional test-and-set locks, which support timeout trivially, do not scale to large machines, even when designed to back off in the presence of contention. Technological trends would appear to be making queue-based locks increasingly important, and a timeout mechanism significantly increases the scope of their applicability. On a single processor, without contention, the CLH-NB try lock takes about twice as long as the plain (no timeout) CLH lock, which in turn takes about twice as long as a conventional test-and-set lock (with or without timeout). With 64 processors attempting to acquire the lock simultaneously, however, we identified cases in which the test-and-set lock (with backoff) was more than six times slower than the CLH-NB try lock, while failing (timing out) more than 22 times as often (82% of the time, v. 3.7% for the CLH-NB try lock). While one of course attempts in any parallel program to avoid high lock contention, conversations with colleagues in industry indicate that pathological cases do indeed arise in practice, particularly in transaction processing systems, and graceful performance degregation in these cases is of significant concern to customers.

For small-scale multiprocessors we continue to recommend a simple test-and-set lock with backoff. Queue-based locks, however, 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 than the MCS lock on cache-coherent machines. The CLH-NB try lock is also substantially simpler than the MCS-NB try lock. We would expect the relative performance of the queue-based locks to be reversed on a non-cache-coherent machine, even if the CLH-NB try lock were modified to ensure local-only spinning, using an extra level of indirection [5].

The provision for timeout in scalable queue-based locks makes spin locks a viable mechanism for user-level synchronization on large multiprogrammed systems. In future work we hope to evaluate our algorithms in the context of commercial OLTP codes. We also plan to develop variants that block in the scheduler on timeout [9, 16], cooperate with the scheduler to avoid preemption while in a critical section [6, 10], or adapt dynamically between `test_and_set` and queue-based locking in response to observed contention [11]. In a related vein, we are developing a tool to help verify the correctness of locking algorithms by transforming source code automatically into input for a model checker.

5 Acknowledgments

Our thanks to Paul Martin and Mark Moir of Sun Labs, Boston, for their help in obtaining results on the 64-processor Enterprise machine.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the Twenty-Second International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[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, January 1990.

[3] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[4] D. L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *Computer*, 23(5):35–43, May 1990.

[5] T. S. Craig. Building FIFO and Priority-Queueing Spin Locks from Atomic Swap. TR 93-02-02, Department of Computer Science, University of Washington, February 1993.

[6] J. Edler, J. Lipkis, and E. Schonberg. Process Management for Highly Parallel UNIX Systems. In *Proceedings of the USENIX Workshop on Unix and Supercomputers*, Pittsburgh, PA, September 1988.

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

[8] M. Herlihy. *Personal communication*. Brown University, October 2001.

[9] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 41–55, Pacific Grove, CA, October 1991.

[10] L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott. Scheduler-Conscious Synchronization. *ACM Transactions on Computer Systems*, 15(1):3–40, February 1997.

[11] B.-H. Lim and A. Agarwal. Reactive Synchronization Algorithms for Multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–35, San Jose, CA, October 1994.

[12] V. Luchangco. *Personal communication*. Sun Microsystems Laboratories, Boston, MA, January, 2002.

[13] P. Magnussen, A. Landin, and E. Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 165–171, Cancun, Mexico, April 1994. Expanded version available as “Efficient Software Synchronization on Large Cache Coherent Multiprocessors”, SICS Research Report T94:07, Swedish Institute of Computer Science, February 1994.

[14] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-Class User-Level Threads. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 110–121, Pacific Grove, CA, October 1991.

[15] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[16] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the Third International Conference on Distributed Computing Systems*, pages 22–30, Miami/Ft. Lauderdale, FL, October 1982.

[17] M. L. Scott and W. N. Scherer III. Scalable Queue-Based Spin Locks with Timeout. In *Proceedings of the Eighth ACM Symposium on Principles and Practice of Parallel Programming*, Snowbird, UT, June 2001.

[18] R. C. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Hierarchical Clustering: A Structure for Scalable Multiprocessor Operating System Design. *The Journal of Supercomputing*, 9(1/2):105–134, 1995.

A Code for algorithms

The following three pages contain slightly stylized C code for the CLH-NB try lock, the MCS-NB try lock, and the queue node space management routines. Compilable source code for Sparc V9/Solaris can be found at ftp.cs.rochester.edu/pub/packages/scalable_synch/nb_timeout.tar.gz.

```
// Code to manage a local but shared pool of qnodes.
// All nodes are allocated by, and used by, a given thread, and may
// reside in local memory on an ncc-numa machine.  The nodes belonging
// to a given thread form a circular singly linked list.  The head
// pointer points to the node most recently successfully allocated.
// A thread allocates from its own pool by searching forward from the
// pointer for a node that's marked "unused".  A thread (any thread)
// deallocates a node by marking it "unused".

typedef struct local_qnode {
    union {
        clh_nb_qnode cnq;           // members of this union are
        mcs_nb_qnode mnq;           // never accessed by name
    } real_qnode;
    volatile bool allocated;
    struct local_qnode *next_in_pool;
} local_qnode;

typedef struct {
    local_qnode *try_this_one;     // pointer into circular list
    local_qnode initial_qnode;
} local_head_node;

inline local_qnode *alloc_local_qnode(local_head_node *hp)
{
    local_qnode *p = hp->try_this_one;
    if (!p->allocated) {
        p->allocated = true;
        return p;
    } else {
        local_qnode *q = p->next_in_pool;
        while (q != p) {
            if (!q->allocated) {
                hp->try_this_one = q;
                q->allocated = true;
                return q;
            } else {
                q = q->next_in_pool;
            }
        }
        // All qnodes are in use.  Allocate new one and link into list.
        special_events[mallocs]++;
        q = (local_qnode *) malloc(sizeof(local_qnode));
        q->next_in_pool = p->next_in_pool;
        p->next_in_pool = q;
        hp->try_this_one = q;
        q->allocated = true;
        return q;
    }
}

#define free_local_qnode(p) ((p)->allocated = false)

typedef struct clh_nb_qnode {
    struct clh_nb_qnode *volatile prev;
} clh_nb_qnode;
typedef struct {
    clh_nb_qnode *volatile tail;
    clh_nb_qnode *lock_holder;   // node allocated by lock holder
} clh_nb_lock;

#define AVAILABLE ((clh_nb_qnode *) 1)
```

```
#define cqn_swap(p,v) (clh_nb_qnode *) \
    swap((volatile unsigned long*) (p), (unsigned long) (v))
#define compare_and_store(p,o,n) \
    (cas((volatile unsigned long *) (p), \
         (unsigned long) (o), (unsigned long) (n)) \
            == (unsigned long) (o))

#define alloc_qnode() (clh_nb_qnode *) alloc_local_qnode(my_head_node_ptr())
#define free_qnode(p) free_local_qnode((local_qnode *) p)

bool clh_nb_try_acquire(clh_nb_lock *L, hrtime_t T)
{
    clh_nb_qnode *I = alloc_qnode();

    I->prev = 0;
    clh_nb_qnode pred = cqn_swap(&L->tail, I);
    if (!pred) {
        // lock was free and uncontested; just return
        L->lock_holder = I;
        return true;
    }

    if (pred->prev == AVAILABLE) {
        // lock was free; just return
        L->lock_holder = I;
        free_qnode(pred);
        return true;
    }
    hrtime_t start = START_TIME;

    while (CUR_TIME - start < T) {
        clh_nb_qnode *pred_pred = pred->prev;
        if (pred_pred == AVAILABLE) {
            L->lock_holder = I;
            free_qnode(pred);
            return true;
        } else if (pred_pred) {
            free_qnode(pred);
            pred = pred_pred;
        }
    }

    // timed out; reclaim or abandon own node

    if (compare_and_store(&L->tail, I, pred)) {
        // last process in line
        free_qnode(I);
    } else {
        I->prev = pred;
    }
    return false;
}

void clh_nb_try_release(clh_nb_lock *L)
{
    clh_nb_qnode *I = L->lock_holder;
    if (compare_and_store(&L->tail, I, 0)) {
        // no competition for lock; reclaim qnode
        free_qnode(I);
    } else {
        I->prev = AVAILABLE;
    }
}
```

```
typedef enum {available, leaving, transient, waiting, recycled} qnode_status;
typedef struct mcs_nb_qnode {
    struct mcs_nb_qnode *volatile prev;
    struct mcs_nb_qnode *volatile next;
    volatile qnode_status status;
} mcs_nb_qnode;
typedef struct {
    mcs_nb_qnode *volatile tail;
    mcs_nb_qnode *lock_holder;     // node allocated by lock holder
} mcs_nb_lock;

#define AVAILABLE ((mcs_nb_qnode *) 1)
#define LEAVING   ((mcs_nb_qnode *) 2)
#define TRANSIENT ((mcs_nb_qnode *) 3)
    // Thread puts TRANSIENT in its next pointer and, if possible, transient
    // in its successor's status word, when it wants to leave but was unable
    // to break the link to it from its precedessor.

#define mqn_swap(p,v) (mcs_nb_qnode *) \
    swap((volatile unsigned long *) (p), (unsigned long) (v))
#define s_swap(p,v) (qnode_status)
    swap((volatile unsigned long *) (p), (unsigned long) (v))

#define alloc_qnode() (mcs_nb_qnode *) alloc_local_qnode(my_head_node_ptr())

bool mcs_nb_try_acquire(mcs_nb_lock *L, hrtime_t T)
{
    mcs_nb_qnode *I = alloc_qnode();
    mcs_nb_qnode *pred_next;

    I->next = 0;
    mcs_nb_qnode *pred = mqn_swap(&L->tail, I);
    if (!pred) {
        L->lock_holder = I;
        return true;
    }
    hrtime_t start = START_TIME;

    I->status = waiting;
    while (1) {
        pred_next = mqn_swap(&pred->next, I);
        /* If pred_next is not nil then my predecessor tried to leave or
            grant the lock before I was able to tell it who I am.  Since it
            doesn't know who I am, it won't be trying to change my status
            word, and since its CAS on the tail pointer, if any, will have
            failed, it won't have reclaimed its own qnode, so I'll have to. */
        if (pred_next == AVAILABLE) {
            L->lock_holder = I;
            free_qnode(pred);
            return true;
        } else if (!pred_next) {
            while (1) {
                if (CUR_TIME - start > T) {
                    goto timeout1;
                }
                qnode_status stat = I->status;
                if (stat == available) {
                    L->lock_holder = I;
                    free_qnode(pred);
                    return true;
                } else if (stat == leaving) {
                    mcs_nb_qnode *new_pred = pred->prev;
                    free_qnode(pred);
                    pred = new_pred;
                    I->status = waiting;
                    break;  // exit inner loop; continue outer loop
                } else if (stat == transient) {
                    mcs_nb_qnode *new_pred = pred->prev;
                    if (s_swap(&pred->status, recycled) != waiting) {
                        free_qnode(pred);
                    } /* else when new predecessor changes this to available,
                        leaving, or transient it will find recycled, and will
                        reclaim old predecessor's node. */
                    pred = new_pred;
                    I->status = waiting;
                    break;  // exit inner loop; continue outer loop
                } // else stat == waiting; continue inner loop
            }
        } else if (CUR_TIME - start > T) {
            goto timeout2;
        } else if (pred_next == LEAVING) {
            mcs_nb_qnode *new_pred = pred->prev;
            free_qnode(pred);
            pred = new_pred;
        } else if (pred_next == TRANSIENT) {
            mcs_nb_qnode *new_pred = pred->prev;
            if (s_swap(&pred->status, recycled) != waiting) {
                free_qnode(pred);
            } /* else when new predecessor changes this to available,
                 leaving, or transient it will find recycled, and will
                 reclaim old predecessor's node. */
            pred = new_pred;
        }
    }

    timeout1: {
        I->prev = pred;
        if (compare_and_store(&pred->next, I, 0)) {
            // predecessor doesn't even know I've been here
            mcs_nb_qnode *succ = mqn_swap(&I->next, LEAVING);
            if (succ) {
                if (s_swap(&succ->status, leaving) == recycled) {
                    /* Timing window: successor already saw my modified
                        next pointer and declined to modify it.  Nobody is
                        going to look at my successor node, but they will
                        see my next pointer and know what happened. */
                    free_qnode(succ);
                } /* else successor will reclaim me when it sees my
                    change to its status word. */
            } else {
                // I don't seem to have a successor.
                if (compare_and_store(&L->tail, I, pred)) {
                    free_qnode(I);
                } /* else a newcomer hit the timing window or my successor
                    is in the process of leaving.  Somebody will discover
                    I'm trying to leave, and will free my qnode for me. */
            }
        } else {
            /* Predecessor is trying to leave or to give me (or somebody)
                the lock.  It has a pointer to my qnode, and is planning
                to use it.  I can't wait for it to do so, so I can't free
                my own qnode. */
            mcs_nb_qnode *succ = mqn_swap(&I->next, TRANSIENT);
            if (succ) {
                if (s_swap(&succ->status, transient) == recycled) {
                    /* Timing window: successor already saw my modified
                        next pointer and declined to modify it.  Nobody is
                        going to look at my successor node, but they will see
                    my next pointer and know what happened. */
                free_qnode(succ);
            } /* else successor will reclaim me when it sees my
                change to its status word. */
        } /* else I don't seem to have a successor, and nobody can
            wait for my status word to change.  This is the
            pathological case where we can temporarily require
            non-linear storage. */
    }
    return false;
}

timeout2: {
    /* pred_next is LEAVING or TRANSIENT; pred->next is I.
        Put myself in transient state, so some successor can
        eventually clean up. */
    mcs_nb_qnode *succ;
    /* put predecessor's next field back, as it would be if I
        had timed out in the inner while loop and been unable to
        update predecessor's next pointer: */
    pred->next = pred_next;
    I->status = (pred_next == LEAVING ? leaving : transient);
    I->prev = pred;
    succ = mqn_swap(&I->next, TRANSIENT);
    if (succ) {
        if (s_swap(&succ->status, transient) == recycled) {
            /* Timing window: successor already saw my modified next
                pointer and declined to modify it.  Nobody is going
                to look at my successor node, but they will see my
                next pointer and know what happened. */
            free_qnode(succ);
        } /* else successor will reclaim me when it sees my change
            to its status word. */
    } /* else I don't seem to have a successor, and nobody can wait
        for my status word to change.  This is the pathological case
        where we can temporarily require non-linear storage. */
    return false;
}
}
```

```
void mcs_nb_try_release(mcs_nb_lock *L)
{
    mcs_nb_qnode *I = L->lock_holder;
    mcs_nb_qnode *succ = mqn_swap(&I->next, AVAILABLE);
    /* As a general rule, I can't reclaim my own qnode on release because
        my successor may be leaving, in which case somebody is going to
        have to look at my next pointer to realize that the lock is
        available.  The one exception (when I *can* reclaim my own node)
        is when I'm able to change the lock tail pointer back to nil. */
    if (succ) {
        if (s_swap(&succ->status, available) == recycled) {
            /* Timing window: successor already saw my modified next
                pointer and declined to modify it.  Nobody is going to
                look at my successor node, but they will see my next
                pointer and know what happened. */
            free_qnode(succ);
        } /* else successor (old or new) will reclaim me when it sees my
            change to the old successor's status word. */
    } else {
        // I don't seem to have a successor.
        if (compare_and_store(&L->tail, I, 0)) {
            free_qnode(I);
        } /* else a newcomer hit the timing window or my successor is in
            the process of leaving.  Somebody will discover I'm giving
            the lock away, and will free my qnode for me. */
    }
}
```