Preemption Adaptivity in Time-Published Queue-Based Spin Locks*

Bijun He, William N. Scherer III, and Michael L. Scott

Technical Report #867

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

May, 2005

Abstract

The proliferation of multiprocessor servers and multithreaded applications has increased the demand for high-performance synchronization. Traditional scheduler-based locks incur the overhead of a full context switch between threads and are thus unacceptably slow for many applications. Spin locks offer low overhead, but they either scale poorly on large-scale SMPs (test-and-set style locks) or behave poorly in the presence of preemption (queue-based locks).

Previous work has shown how to build preemption-tolerant locks using an extended kernel interface, but such locks are neither portable to nor even compatible with most operating systems.

In this work, we propose a *time-publishing* heuristic in which each thread periodically records its current timestamp to a shared memory location. Given the high resolution, roughly synchronized clocks of modern processors, this convention allows threads to guess accurately which peers are active based on the currency of their timestamps. We implement two queue-based locks, MCS-TP and CLH-TP, and evaluate their performance relative to both traditional spin locks and preemption-safe locks on a 32-processor IBM p690 multiprocessor. Experimental results indicate that time-published locks make it feasible, for the first time, to use queue-based spin locks on multiprogrammed systems with a standard kernel interface.

1 Introduction

Historically, spin locks have found most of their use in operating systems and dedicated servers, where the entire machine is dedicated to whatever task the locks are protecting. This is fortunate, because spin locks typically don't handle preemption very well: if the thread that holds a lock is suspended before releasing it, any processor time given to waiting threads will be wasted on fruitless spinning.

Recent years, however, have seen a marked trend toward multithreaded user-level programs, such as databases and on-line servers. Further, large multiprocessors are increasingly shared among

*This work was supported in part by NSF grants numbers CCR-9988361, EIA-0080124, CCR-0204344, and CNS-0411127, by financial and equipment grants from Sun Microsystems Laboratories, and by an IBM Shared University Research grant.

multiple multithreaded programs. As a result, modern applications cannot in general count on any specific number of processors; spawning one thread per processor does not suffice to avoid preemption.

For multithreaded servers, the high cost of context switches makes scheduler-based locking unattractive, so implementors are increasingly turning to spin locks to gain performance. Unfortunately, this solution comes with hidden drawbacks: queue-based locks are highly vulnerable to preemption, but test-and-set locks do not scale beyond a modest number of processors. Although several heuristic strategies can reduce wasted spinning time [13, 16], multiprogrammed systems usually rely on non-queue-based locks [20]. Our goal is to combine the efficiency and scalability of queue-based spin locks with the preemption tolerance of the scheduler-based approach.

1.1 Related Work

One approach to avoiding excessive wait times can be found in *abortable locks* (sometimes called *try locks*), in which a thread "times out" if it fails to acquire the lock within a specified *patience* interval [12, 27, 28]. Although timeout prevents a thread from being blocked behind a preempted peer, it does nothing to improve system-wide throughput if the lock is squarely in the application's critical path. Further, any timeout sequence that requires cooperation with neighboring threads in a queue opens yet another window of preemption vulnerability. Known approaches to avoiding this window result in unbounded worst-case space overhead [27] or very high base time overhead [12].

An alternative approach is to adopt *nonblocking* synchronization, eliminating the use of locks [8]. Unfortunately, while excellent nonblocking implementations exist for many important data structures (only a few of which we have room to cite here [21, 23, 24, 29, 30]), general-purpose mechanisms remain elusive. Several groups (including our own) are working on this topic [6, 10, 18, 26], but it still seems unlikely that nonblocking synchronization will displace locks entirely soon.

Finally, several researchers have suggested operating system mechanisms that provide user applications with a limited degree of control over scheduling, allowing them to avoid [4, 5, 14, 19, 25] or recover from [1, 2, 31, 33] inopportune preemption. Commercial support for such mechanisms, however, is neither universal nor consistent.

Assuming, then, that locks will remain important, and that many systems will not provide an OS-level solution, how can we hope to leverage the fairness and scalability of queue-based spin locks in multithreaded user-level programs?

In this work, we answer this question with two new abortable queue-based spin locks that combine fair and scalable performance with good preemption tolerance: the MCS time-published lock (MCS-TP) and the CLH time-published (CLH-TP) lock. In this context, we use the term *time-published* to mean that contending threads periodically write their wall clock timestamp to shared memory in order to be able to estimate each other's runtime states. In particular, given a low-overhead hardware timer with bounded skew across processors and a memory bus that handles requests in bounded time (both of which are typically available in modern multiprocessors), we can guess with high accuracy that another thread is preempted if the current system time exceeds the thread's latest timestamp by some appropriate threshold. We now have the ability to selectively pass a lock only between active threads. Although this doesn't solve the preemption problem completely (threads can be preempted while holding the lock, and our heuristic suffers from a race condition in which we read a value that has just been written by a thread immediately before it was preempted), experimental results (Sections 4 and 5) confirm that our approach suffices to make the

Lock	MCS-TP	CLH-TP
Link Structure	Queue linked head to tail	Queue linked tail to head
Lock handoff	Lock holder explicitly grants the lock to a waiter	Lock holder marks lock available and leaves; next-in-queue claims lock
Timeout precision	Strict adherence to patience	Bounded delay from removing timed-out and preempted predecessors
Queue management	Only the lock holder removes timed-out or preempted nodes (at handoff)	Concurrent removal by all waiting threads
Space management	Semi-dynamic allocation: waiters may reinhabit abandoned nodes until removed from the queue	Dynamic allocation: separate node per acquisition attempt

Figure 1: Comparison between MCS and CLH time-published locks.

locks *preemption adaptive*: free, in practice, from virtually all preemption-induced performance loss.

2 Algorithms

We begin this section by presenting common features of our two time-published (TP) locks; Sections 2.1 and 2.2 cover algorithm-specific details.

Our TP locks are abortable variants of the well-known MCS [20] and CLH [3, 17] queue-based spin locks. Their `acquire` functions return `success` or `failure` to indicate whether the thread acquired the lock within a *patience* interval specified as a parameter. Both locks maintain a linked-list queue in which the head node corresponds to the thread that holds the lock.

With abortable queue-based locks, there are three ways in which preemption can interfere with throughput. First, as with any lock, a thread that is preempted in its critical section will block all competitors. Second, a thread preempted while waiting in the queue will block others once it reaches the head; strict FIFO ordering is generally a disadvantage in the face of preemption. Third, any timeout protocol that requires explicit handshaking among neighboring threads will block a timed-out thread if its neighbors are not active.

The third case can be avoided with *nonblocking* timeout protocols, which guarantee a waiting thread can abandon an acquisition attempt in a bounded number of its own time steps [27]. To address the remaining cases, we use a timestamp-based heuristic. Each waiting thread periodically writes the current system time to a shared location. If a thread A finds a stale timestamp for another thread B, A assumes that B has been preempted and removes B's node from the queue. Further, any time A fails to acquire the lock, it checks the critical section entry time recorded by the current lock holder. If this time is sufficiently far in the past (farther than longest plausible critical section time—the exact value is not critical), A yields the processor in the hope that a suspended lock holder might resume.

There is a wide design space for time-published locks, which we have only begun to explore. Our initial algorithms, described in the two subsections below, are designed to be fast in the common case, where timeout is uncommon. They reflect our attempt to adopt straightforward strategies consistent with the head-to-tail and tail-to-head linking of the MCS and CLH locks, respectively. These strategies are summarized in Figure 1. Time and space bounds are considered in Appendix A.

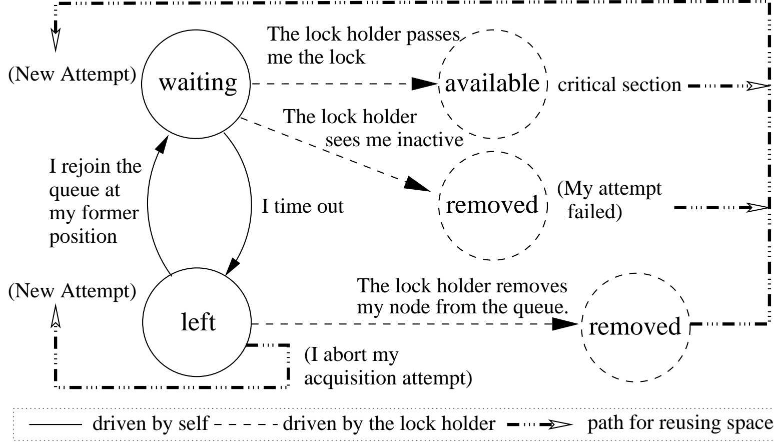


Figure 2: State transitions for MCS-TP queue nodes.

2.1 MCS Time-Published Lock

Our first algorithm is adapted from Mellor-Crummey and Scott's MCS lock [20]. In the original MCS algorithm, a contending thread A atomically swaps a pointer to its queue node α into the queue's tail. If the swap returns `nil`, A has acquired the lock; otherwise the return value is A's predecessor B. A then updates B's `next` pointer to α and spins until B explicitly changes the A's state from `waiting` to `available`. To release the lock, B reads its `next` pointer to find a successor node. If it has no successor, B atomically updates the queue's tail pointer to `nil`.

The MCS-TP lock uses the same head-to-tail linking as MCS, but adds two additional states: `left` and `removed`. When a waiting thread times out before acquiring the lock, it marks its node `left` and returns, leaving the node in the queue. When a node reaches the head of the queue but is either marked `left` or appears to be owned by a preempted thread (i.e., has a stale timestamp), the lock holder marks it `removed`, and follows its `next` pointer to find a new candidate lock recipient, repeating as necessary. Figure 2 shows the state transitions for MCS-TP queue nodes. Source code can be found in Appendix B.

The MCS-TP algorithm allows each thread at most one node per lock. If a thread that calls `acquire` finds its node marked `left`, it reverts the state to `waiting`, resuming its former place in line. Otherwise, it begins a fresh attempt from the tail of the queue. To all other threads, timeout and retry are indistinguishable from an execution in which the thread was waiting all along.

To guarantee bounded-time lock handoff, we must avoid a pathological case in which waiting threads might repeatedly time out, have their nodes removed, rejoin the queue, and then time out again before obtaining the lock. In this scenario, a lock holder might see an endless treadmill of abandoned nodes, and never be able to release the lock. We therefore arrange for the lock holder to remove only the first T abandoned nodes it encounters; thereafter, it scans the list until it either reaches the end or finds a viable successor. Only then does it mark the scanned nodes `removed`. (If a scanned node's owner comes back to waiting before being marked `removed`, it will eventually see the `removed` state and quit as a failed attempt). Because skipped nodes' owners reclaim their existing (bypassed) spots in line, the length of the queue is bounded by the total number of threads T and this process is guaranteed to terminate in at most $2T$ steps. In practice, we have never observed the worst case; lock holders typically find a viable successor within the first few nodes.

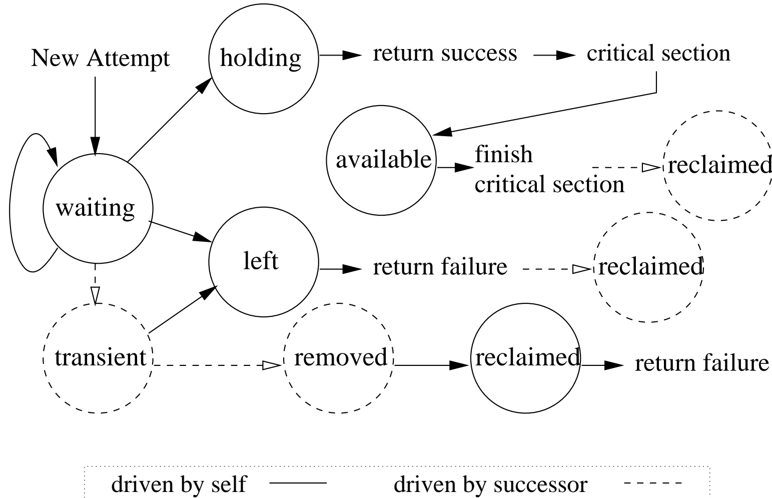


Figure 3: State transitions for CLH-TP queue nodes.

2.2 CLH Time-Published Lock

Our second time-published lock is based on the CLH lock of Craig [3] and Landin and Hagersten [17]. In the original CLH lock, as in MCS, a contending thread A atomically swaps a pointer to its queue node α into the queue's tail. This swap always returns a pointer to the node β inserted by A's predecessor B (or, the very first time, to a dummy node, marked `available`, created at initialization time). A updates α's `prev` pointer to β and spins until β's state is `available`. Note that, in contrast to MCS, links point from the tail of the queue toward the head, and a thread spins on the node inserted by its predecessor. To release the lock, a thread marks the node it inserted `available`; it then takes the node inserted by its predecessor for use in its next acquisition attempt. Because a thread cannot choose the location on which it is going to spin, the CLH lock requires cache-coherent hardware in order to bound contention-inducing remote memory operations.

CLH-TP retains the link structure of the CLH lock, but adds both non-blocking timeout and removal of nodes inserted by preempted threads. Unlike MCS-TP, CLH-TP allows any thread to remove the node inserted by a preempted predecessor; removal is not reserved to the lock holder. Middle-of-the-queue removal adds significant complexity to CLH-TP; experience with earlier abortable locks [27, 28] suggests that it would be very difficult to add to MCS-TP. Source code for the CLH-TP lock can be found in Appendix C, together with a control flow graph for lock acquisition.

We use low-order bits in a CLH-TP node's `prev` pointer to store the node's state, allowing us to modify the state and the pointer together, atomically. If `prev` is a valid pointer, its two lowest-order bits specify one of three states: `waiting`, `transient`, and `left`. Alternatively, `prev` can be a `nil` pointer with low-order bits set to indicate three more states: `available`, `holding`, and `removed`. Figure 3 shows the state transition diagram for CLH-TP queue nodes.

In each lock acquisition attempt, thread B dynamically allocates a new node β and links it to predecessor α as before. While waiting, B handles three events. The simplest occurs when α's state changes to `available`; B atomically updates β's state to `holding` to claim the lock.

```
int cas_w_waiting(node_t * volatile *addr,
                  unsigned long     oldv,
                  unsigned long     newv,
                  node_t * volatile *me) {
 do {
    unsigned long tmp = LL(addr);  // if needed, add a member barrier here to
                                   // ensure the read order of addr and me.
    if (tmp != oldv || !is_waiting(me))
      return 0;
  } while(!SC(addr, newv));
  return 1;
}
```

Figure 4: Conditional updates in CLH-TP

The second event occurs when B believes A to be preempted or timed out. Here, B performs a three-step *removal sequence* to unlink A's node from the queue. First, B atomically changes α's state from `waiting` to `transient`, to prevent A from acquiring the lock or from reclaiming and reusing α if it is removed from the queue by some successor of B (more on this below). Second, B removes α from the queue, simultaneously verifying that B's own state is still `waiting` (since β's `prev` pointer and state share a word, this is a single *compare-and-swap*). Hereafter, α is no longer visible to other threads in the queue, and B spins on A's predecessor's node. Finally, B marks α as safe for reclamation by changing its state from `transient` to `removed`.

The third event occurs when B times out or when it notices that β is `transient`. In either case, it attempts to atomically change β's state from `transient` or `waiting` to `left`. If the attempt (a *compare-and-swap*) succeeds, B has delegated responsibility for reclamation of β to a successor. Otherwise, B has been removed from the queue and must reclaim its own node. In both cases, whichever of B and its successor is the last to notice that β has been removed from the queue handles the memory reclamation; this simplifies memory management.

A corner case occurs when, after B marks α `transient`, β is marked `transient` by some successor thread C before B removes α from the queue. In this case, B leaves α for C to clean up; C recognizes this case by finding α already `transient`.

The need for the `transient` state derives from a race condition in which B decides to remove α from the queue but is preempted before actually doing so. While B is not running, successor C may remove both β and α from the queue, and A may reuse its node in this or another queue. When B resumes running, we must ensure that it does not modify (the new instance of) A. The `transient` state allows us to so, if we can update α's state and verify that β is still `waiting` as a single atomic operation. The custom atomic construction shown in Figure 4 implements this operation, assuming the availability of *load-linked/store-conditional*. Since C must have removed β before modifying α, if B reads α's state before C changes β, then the value read must be α's state from before C changed α. Thereafter, if α is changed, the *store-conditional* will force B to recheck β. Alternative solutions might rely on a tracing garbage collector (which would decline to recycle α as long as B has a reference) or on manual reference-tracking methodologies [9, 22].

3 Scheduling and Preemption

TP locks publish timestamps to enable a heuristic that guesses whether the lock holder or a waiting thread is preempted. This heuristic admits a race condition wherein a thread's timestamp is polled

just before it is descheduled. In this case, the poller will mistakenly assume the thread to be active. In practice (see Section 4), the timing window is too narrow to have a noticeable impact on performance. Nevertheless it is instructive to consider modified TP locks that use a stronger scheduler interface to completely eliminate preemption vulnerabilities.

Extending previous work [14], we distinguish three levels of APIs for user-level feedback to the kernel scheduler implementation:

(I) Critical section protection (CSP): these APIs can bracket a block of code to request that a thread not be preempted while executing it.

(II) Runtime state check: these APIs allow inquiries as to whether a thread is currently preempted.

(III) Directed preemption avoidance: threads can ask the scheduler not to preempt others.

Several commercial operating systems, including AIX 5L, provide Level I APIs [11]. Level II and III APIs are generally confined to research systems [1, 4, 19]. The Mach scheduler [2] provides a variant of the Level III API that includes a directed yield of the processor to a specified thread.

For Test-and-Set (TAS) locks, preemption-safe variants need only a Level I API [14] to avoid preemption during the critical section of a lock holder. By comparison, a thread contending for a (non-abortable) queue-based lock is sensitive to preemption in two additional timing windows—windows not addressed by the preemption adaptivity of the MCS-TP and CLH-TP locks. The first window occurs between swapping a node into the queue's tail and connecting links with the remainder of the queue. The second occurs between when a thread is granted the lock and when it starts actually using the lock. We say that a lock is *preemption-safe* only if it provably prevents all such timing windows.

Previous work proposed two algorithms for preemption-safe MCS variants: the Handshaking and SmartQ locks [14]. Both require a Level I API to prevent preemption in the critical section and in the first (linking-in) window described above. For the second (lock-passing) window, the lock holder in the Handshaking lock exchanges messages with its successor to confirm that it has invoked the Level I API. In practice, this transaction has undesirably high overhead (two additional remote coherence misses on the critical path), so SmartQ employs Level II and III APIs to replace it. We characterize the preemption safety of the Handshaking lock as *heuristic*, in the sense that a thread guesses the status of a successor based on the speed of its response, and may err on the side of withholding the lock if, for example, the successor's processor is momentarily busy handling an interrupt. By contrast, the preemption safety of the SmartQ lock is *precise*.

Our MCS-TP lock uses a one-way handoff transaction similar to, but simpler and faster than, that of the Handshaking lock. However, because of the reduced communication, the lock cannot be made preemption safe with a Level I API. By contrast, a preemption-safe CLH variant can be built efficiently from the CLH-TP lock. The tail-to-head direction of linking eliminates the first preemption window. The second is readily addressed if a thread invokes the Level I API when it sees the lock is available, but before updating its state to grab the lock. If the lock holder grants the lock to a preempted thread, the first active waiter to remove all inactive nodes between itself and the queue's head will get the lock. We call this clock CLH-CSP (critical section protection). Like the Handshaking lock, it is heuristically preemption safe. For precise preemption safety, one can use a Level II API for preemption monitoring (CLH-PM).

Note that TAS locks require nothing more than a Level I API for (precise) preemption safety. The properties of the various lock variants are summarized in Figure 5. The differences among

This table lists the minimum requirements of NB/PA/PS capabilities and their implementations in this paper. The Handshaking and SmartQ locks are from Kontothanassis et al. [14]. “CSP” indicates use of a Level I API for critical section protection; “PM” indicates preemption monitoring with a Level II API; “try” indicates an abortable (timeout-capable) lock. **NB**: Non-Blocking; **PA**: Preemption-Adaptive; **PS**: Preemption-Safe; **/**: unnecessary.

Support	TAS	MCS	CLH
Atomic instructions	PA NB-try (TAS-yield)	standard lock (MCS)	standard lock (CLH)
NB timeout algorithms	/	NB-try (MCS-NB)	NB-try (CLH-NB)
TP algorithms	/	PA NB-try (MCS-TP)	PA NB-try (CLH-TP)
Level I API	precise PS (TAS-CSP)	heuristic PS (Handshaking)	heuristic PS (CLH-CSP)
Level II API	/	/	precise PS (CLH-PM)
Level III API	/	precise PS (SmartQ)	/

Figure 5: Families of locks.

families (TAS, MCS, CLH) stem mainly from the style of lock transfer. In TAS locks, the opportunity to acquire an available lock is extended to all comers. In the MCS locks, only the current lock holder can determine the next lock holder. In the CLH locks, waiting threads can pass preempted peers to grab an available lock, though they cannot bypass active peers.

4 Microbenchmark Results

We test our TP locks on an IBM pSeries 690 (Regatta) multiprocessor. For comparison purposes, we include a range of user-level spin locks: TAS, MCS, CLH, MCS-NB, and CLH-NB. TAS is a test-and-`test-and-set` lock with (well tuned) randomized exponential backoff. MCS-NB and CLH-NB are abortable queue-based locks with non-blocking timeout [27]. We also test spin-then-yield variants [13] of each lock in which threads yield the processors after exceeding a wait threshold.

Finally, we test preemption-safe locks dependent on scheduling control APIs: CLH-CSP, TAS-CSP, Handshaking, CLH-PM, and SmartQ. TAS-CSP and CLH-CSP are TAS and CLH locks augmented with critical section protection (the Level I API discussed in Section 3). The Handshaking lock [14] also uses CSP. CLH-PM adds the Level II preemption monitoring API to assess the preemptive state of threads. The SmartQ lock [14] uses all three API Levels.

Our p690 has 32 1.3GHz Power4 processors, running AIX 5.2. Since AIX 5.2 provides no scheduling control APIs, we have also implemented a synthetic scheduler similar to that used by Kontothanassis et al. [14]. This scheduler runs on one dedicated processor, and sends Unix signals to threads on other processors to mark the end of each $20\,ms$ quantum. If the receiving thread is preemptable, that thread’s signal handler spins in an idle loop to simulate execution of a compute-bound thread in some other application; otherwise, preemption is deferred until the thread is preemptable. Our synthetic scheduler implements all three Levels of scheduling control APIs.

Our microbenchmark application has each thread repeatedly attempt to acquire a lock. We simulate critical sections (CS) by updating a variable number of cache lines; we simulate non-critical sections (NCS) by varying the time spent spinning in an idle loop between acquisitions. We measure the total throughput of lock acquisitions and we count successful and unsuccessful acquisition attempts, across all threads for one second, averaging results of 6 runs. For abortable locks, we retry unsuccessful acquisitions immediately, without executing a non-critical section. We use a fixed patience of $50\,\mu s$.

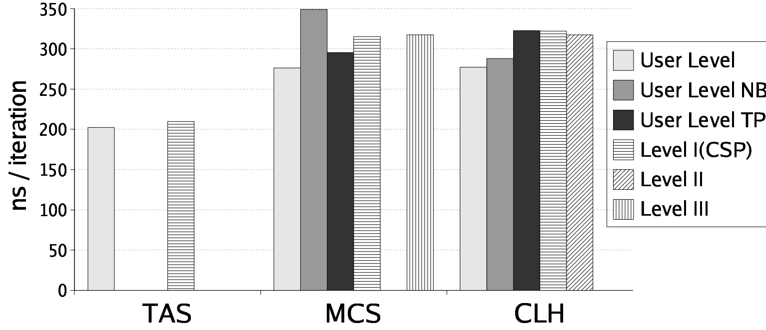


Figure 6: Single-thread performance results.

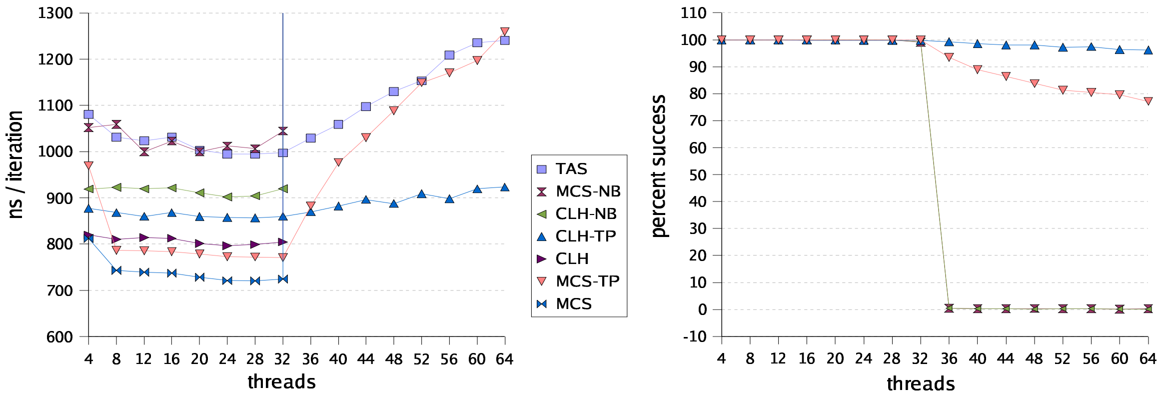


Figure 7: User-level locks; 2 cache line-update critical section (CS); 1 μs non-critical section (NCS).
Critical section service time (left) and success rate (right)

4.1 Single Thread Performance

Because low overhead is crucial for locks in real systems, we assess it by measuring throughput absent contention with one thread and empty critical and non-critical sections. We organize the results by lock family in Figure 6.

As expected, the TAS variants are the most efficient for one thread, absent contention. MCS-NB has one *compare-and-swap* more than the base MCS lock; this appears in its single-thread overhead. Similarly, other differences between locks trace back to the operations in their `acquire` and `release` methods. We note that time-publishing functionality adds little overhead to locks.

A single-thread atomic update on our p690 takes about $60\,ns$. Adding additional threads introduces delays from memory and processor interconnect bus traffic contention and from cache coherence overhead when transferring a cache line between processors. We have measured overheads for an atomic update at 120 and $420\,ns$ with 2 and 32 threads.

4.2 Comparison to User-Level Locks

Under high contention, serialization of critical sections causes application performance to depend primarily on the overhead of handing a lock from one thread to the next; other overheads are typically subsumed by waiting. We present two typical configurations for critical and non-critical section lengths.

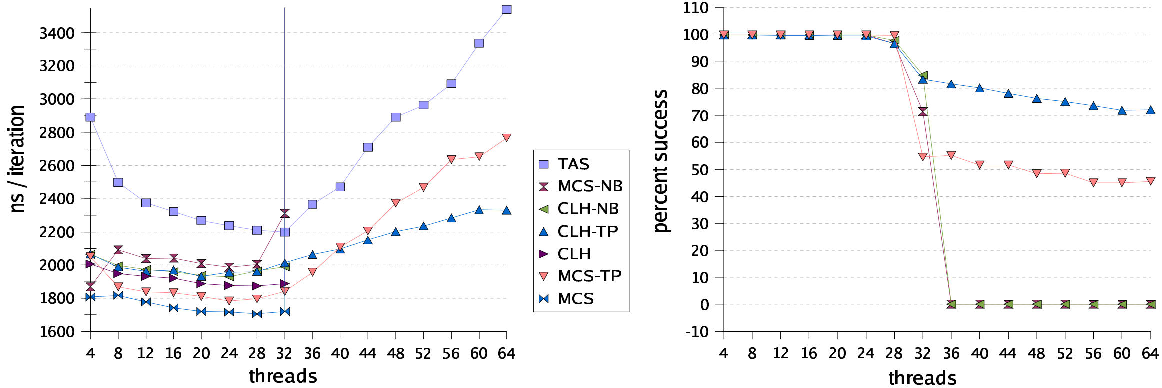


Figure 8: User-level locks; 40 cache line CS; $4\,\mu s$ NCS.
Critical section service time (left) and success rate (right)

Our first configuration simulates contention for a small critical section with a 2-cache-line-update critical section and a $1\,\mu s$ non-critical section. Figure 7 plots the performance of the user-level locks with a generic kernel interface (no scheduler control API). Up through 32 threads (our machine size), queue-based locks outperform TAS; however, only the TP and TAS locks maintain throughput in the presence of preemption. MCS-TP's overhead increases with the number of preempted threads because it relies on the lock holder to remove nodes. By contrast, CLH-TP distributes cleanup work across active threads and keeps throughput more steady. The right-hand graph in Figure 7 shows the percentage of successful lock acquisition attempts for the abortable locks. MCS-TP's increasing handoff time forces its success rate below that of CLH-TP as the thread count increases. CLH-NB and MCS-NB drop to nearly zero due to preemption while waiting in the queue.

Our second configuration uses 40-cache-line-update critical sections $4\,\mu s$ non-critical sections. It models larger longer operations in which preemption of the lock holder is more likely. Figure 8 shows the behavior of user-level locks with this configuration. That the TP locks outperform TAS demonstrates the utility of cooperative yielding for preemption recovery. Moreover, the CLH-TP–MCS-TP performance gap is smaller here than in our first configuration since the relative importance of removing inactive queue nodes goes down compared to that of recovering from preemption in the critical section.

In Figure 8, the success rates for abortable locks drop off beyond 24 threads. Since each critical section takes about $2\,\mu s$, our $50\,\mu s$ patience is just enough for a thread to sit through 25 predecessors. TP locks adapt better to insufficient patience.

One might expect a spin-then-yield policy [13] to allow other locks to match TP locks in preemption adaptivity. In Figure 9 we test this hypothesis with a $50\,\mu s$ spinning time threshold and a 2 cache line critical section. (Other settings produce similar results.) Although yielding improves the throughput of non-TP queue-based locks, they still run off the top of the graph. TAS benefits enough to become competitive with MCS-TP, but CLH-TP still outperforms it. These results confirm that targeted removal of inactive queue nodes is much more valuable than simple yielding of the processor.

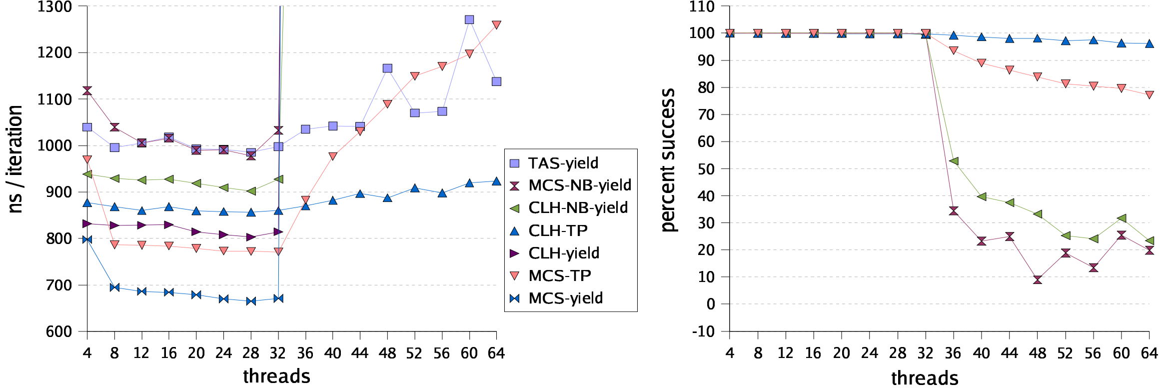
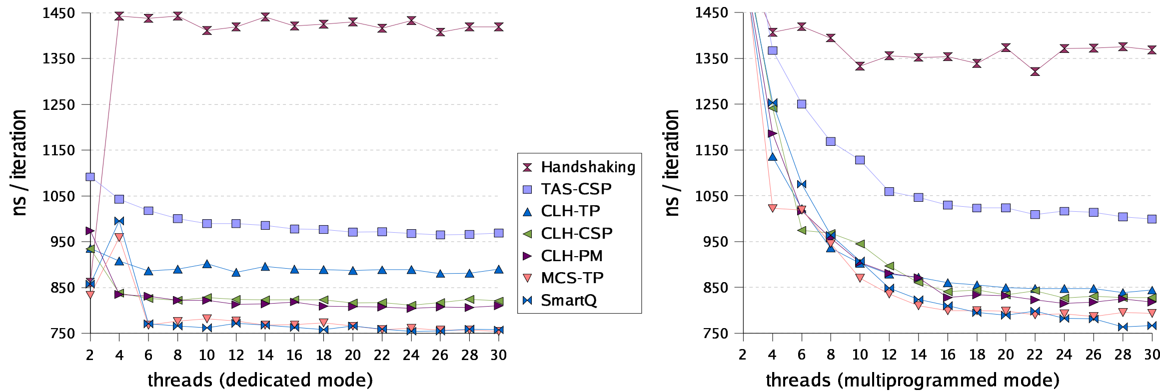


Figure 9: Spin-then-yield variants; 2 cache line CS; 1 μs NCS.



Critical section service time with (right) and without (left) 50% load on processors.
Left, the curves of SmartQ and MCS-TP overlap with each other and those of CLH-CSP and CLH-PM overlap.
Right, five curves are very close to each other, which suggests that they have similar adaptivity to preemption.

Figure 10: Preemption-safe locks: 2 cache line CS; 1 μs NCS.

4.3 Comparison to Preemption-Safe Locks

For this section, we used our synthetic scheduler to compare TP and preemption-safe locks. Results for short critical sections are shown in Figure 10, both with (multiprogrammed mode) and without (dedicated mode) an external 50% load.

Overall, TP locks are competitive with preemption-safe locks. The modest increase in performance gained by locks that use high levels of scheduling control is comparatively minor. In dedicated mode, CLH-TP is 8–9% slower than the preemption-safe CLH variants, but it performs comparably in multiprogrammed mode. MCS-TP closely matches SmartQ (based on MCS) in both modes. Both TP locks clearly outperform the Handshaking lock.

In dedicated mode, CLH-TP incurs additional overhead from reading and publishing timestamps. In multiprogrammed mode, however, overhead from the preemption recovery mechanisms dominates. Since all three CLH variants handle preemption by removing inactive predecessor nodes from the queue, their performance is very similar.

Among preemption-safe locks, CLH-PM slightly outperforms CLH-CSP because it can more accurately assess whether threads are preempted. SmartQ significantly outperforms the Hand-

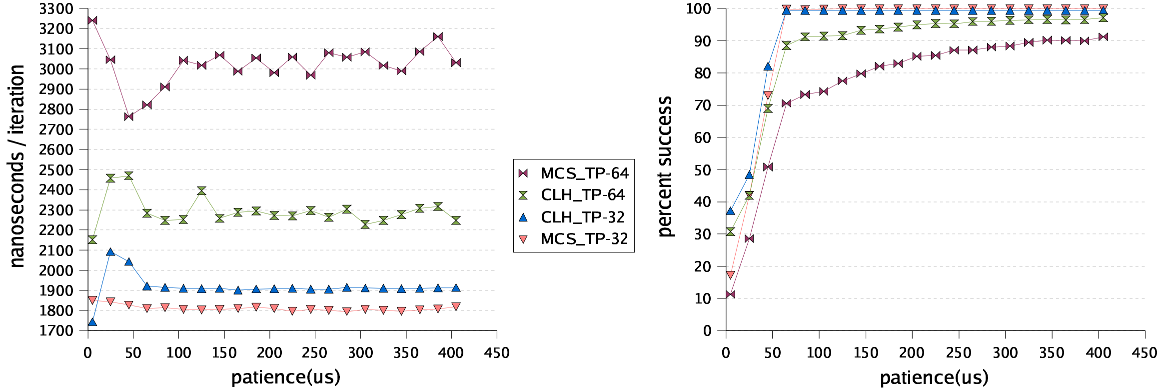


Figure 11: Varying patience for TP locks; 40 cache line CS; $4\,\mu s$ NCS.

shaking lock due to the latter's costly roundtrip handshakes and its use of timeouts to confirm preemption.

4.4 Sensitivity to Patience

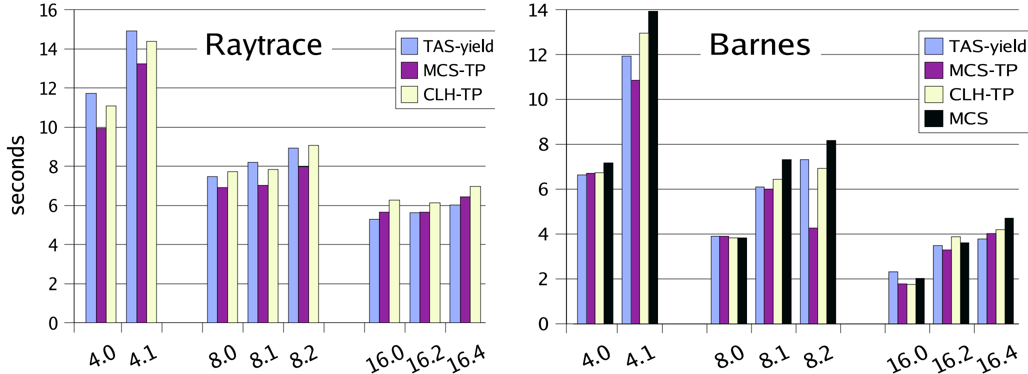
Timeout patience is an important parameter for abortable locks. Insufficient patience yields low success rates and long average critical section service times [27, 28]. Conversely, excessive patience can delay a lock's response to bad scenarios. Our experiments show TP locks to be highly robust to changes in patience. Figure 11 shows the case with a large critical section; for smaller critical sections, the performance is even better. Overall, TP locks are far less sensitive to tuning of patience than other locks; with very low patience, the self-timeout and removal behaviors of the locks help to maintain critical section service time even as the acquisition rate plummets.

4.5 Time and Space Bounds

As a final experiment, we measure the time overhead for removing an inactive node. On our Power4 p690, we calculate that the MCS-TP lock holder needs about 200–350 ns to delete each node. Similarly, a waiting thread in CLH-TP needs about 250–350 ns to delete a predecessor node. By combining these values with our worst-case analysis for the number of inactive nodes in the lock queues (Appendix A), one can estimate an upper bound on delay for lock handoff when the holder is not preempted.

In our analysis of the space bounds for the CLH-TP lock (Appendix A) we show a worst-case bound quadratic in the number of threads, but claim an expected linear value. Two final tests maximize space consumption to gather empirical evidence for the expected case. One test maximizes contention via empty critical and non-critical sections. The other attacks concurrent timeouts and removals by presetting the lock to held, so that every contending thread times out.

We ran both tests 6 times, for 5 and 10 second runs. We find space consumption to be very stable over time, getting equivalent results with both test lengths. With patience as short as $15\,\mu s$, the first test consumed at most 77 queue nodes with 32 threads, and at most 173 nodes with 64 threads. The second test never used more than 64 or 134 nodes with 32 or 64 threads. Since our allocator always creates a minimum of $2T$ nodes, 64 and 128 are optimal. The results are far closer to the expected linear than the worst-case quadratic space bound.



Configuration *M.N* means M application threads and $(32 - M) + N$ external threads on the 32-way SMP.

Figure 12: Parallel execution times for Raytrace and Barnes on a 32-processor machine

5 Application Results

In a final set of tests we measure the performance of the TP locks on the Raytrace and Barnes benchmarks from the SPLASH-2 suite [32]. Other applications are a subject for future work.

Application Features: Raytrace and Barnes spend much time in synchronization [15, 32]. Raytrace uses no barriers but features high contention on a small number of locks. Barnes uses limited barriers (17 for our test input) but numerous locks. Both offer reasonable parallel speedup.

Experimental Setup: We test each of the locks in Section 4 plus the native `pthread_mutex` on our p690, averaging results over 6 runs. We choose inputs large enough to execute for several seconds: 800×800 for Raytrace and 60K particles for Barnes. We limit testing to 16 threads due to the applications' limited scalability. External threads running idle loops generate load and force preemption.

Raytrace: The left side of Figure 12 shows results for three preemption adaptive locks: TAS-yield, MCS-TP and CLH-TP. Other spin locks give similar performance absent preemption; when preemption is present, non-TP queue-based locks yield horrible performance (Figures 7, 8, and 9). The `pthread_mutex` lock also scales very badly; with high lock contention, it can spend 80% of its time in kernel mode. Running Raytrace with our input size took several hours for 4 threads.

Barnes: Preemption adaptivity is less important here than in Raytrace because Barnes distributes synchronization over a very large number of locks, greatly reducing the impact of preemption. We demonstrate this by including a highly preemption-sensitive lock, MCS, with our preemption adaptive locks in the right side of Figure 12; MCS "only" doubles its execution time with heavy preemption.

With both benchmarks, we find that our TP locks maintain good throughput and adapt well to preemption. With Raytrace, MCS-TP in particular yields 8-18% improvement over a yielding TAS lock with 4 or 8 threads. Barnes is less dependent on lock performance in that different locks have similar performance.

6 Conclusions and Future Work

In this work we have demonstrated that published timestamps provide an effective heuristic by which a thread can accurately guess the running state of its peers, without support from a non-standard scheduler API. We have used this published-time heuristic to implement preemption

adaptive versions of standard MCS and CLH queue-based locks. Empirical tests confirm that these locks combine scalability, strong tolerance for preemption, and low observed space overhead with throughput as high as that of the best previously known solutions. Given the existence of a low-overhead time-of-day register with low system-wide skew, our results make it feasible, for the first time, to use queue-based locks on multiprogrammed systems with a standard kernel interface.

For cache-coherent machines, we recommend CLH-TP when preemption is frequent and strong worst-case performance is needed. MCS-TP gives better performance in the common case. With unbounded clock skew, slow system clock access, or a small number of processors, we recommend a TAS-style lock with exponential backoff combined with a spin-then-yield mechanism. Finally, for non-cache-coherent (e.g. Cray) machines, we recommend MCS-TP if clock registers support it; otherwise the best choice is the abortable MCS-NB try lock.

As future work, we conjecture that time can be used to improve thread interaction in other areas, such as preemption-tolerant barriers, priority-based lock queueing, dynamic adjustment of the worker pool for bag-of-task applications, and contention management for nonblocking concurrent algorithms. Further, we note that we have examined only two points in the design space of TP locks; other variations may merit consideration.

References

[1] 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 Trans. on Computer Systems*, 10(1):53–79, Feb. 1992.

[2] D. L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer*, 23(5):35–43, 1990.

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

[4] J. Edler, J. Lipkis, and E. Schonberg. Process management for highly parallel UNIX systems. In *Proc. of the USENIX Workshop on Unix and Supercomputers*, pages 1–17, Pittsburgh, PA, Sept. 1988.

[5] H. Franke, R. R, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Ottawa Linux Symp.*, June 2002.

[6] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proc. of the 18th Annual ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, Oct. 2003.

[7] B. He, W. N. Scherer III, and M. L. Scott. Preemption adaptivity in time-published queue-based spin locks. Technical Report URCS-867, University of Rochester, May 2005.

[8] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, Jan. 1991.

[9] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proc. of the 16th Conf. on Distributed Computing*, Toulouse, France, Oct. 2002.

[10] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proc. of the 22nd Annual Symp. on Principles of Distributed Computing*, pages 92–101, 2003.

[11] IBM. *AIX 5L Differences Guide, Version 5.0*, 2001.

[12] P. Jayanti. Adaptive and efficient abortable mutual exclusion. In *Proc. of the Annual ACM Symp. on Principles of Distributed Computing*, pages 295–304, Boston, MA, 2003.

[13] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proc. of the 13th ACM Symp. on Operating Systems Principles*, pages 41–55, Pacific Grove, CA, 1991.

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

[15] S. Kumar, D. Jiang, R. Chandra, and J. P. Singh. Evaluating synchronization on shared address space multiprocessors: Methodology and performance. In *Measurement and Modeling of Computer Systems*, pages 23–34, 1999.

[16] B.-H. Lim and A. Agarwal. Reactive synchronization algorithms for multiprocessors. In *Proc. of the 6th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 25–35, Oct. 1994.

[17] P. S. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proc. of the 8th Intl. Symp. on Parallel Processing*, pages 165–171. IEEE Computer Society, 1994.

[18] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Design Tradeoffs in Modern Software Transactional Memory Systems. In *Proc. of the 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, Oct. 2004.

[19] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *Proc. of the 13th ACM Symp. on Operating Systems Principles*, pages 110–121, Pacific Grove, CA, 1991.

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

[21] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proc. of the 14th Symp. on Parallel Algorithms and Architectures*, pages 73–82, Winnipeg, MB, Canada, Aug. 2002.

[22] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. on Parallel and Distributed Systems*, 15(8), Aug. 2004.

[23] M. M. Michael. Scalable lock-free dynamic memory allocation. In *Proc. of the SIGPLAN 2004 Symp. on Programming Language Design and Implementation*, pages 35–46, Washington, DC, June 2004.

[24] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51:1–26, 1998.

[25] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *Proc. of the 3rd Intl. Conf. on Distributed Computing Systems*, 1982.

[26] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proc. of the 24th ACM Symp. on Principles of Distributed Computing*, Jul. 2005.

[27] M. L. Scott. Non-blocking timeout in scalable queue-based spin locks. In *Proc. of the 21st Annual Symp. on Principles of Distributed Computing*, pages 31–40, 2002.

[28] M. L. Scott and W. N. Scherer III. Scalable queue-based spin locks with timeout. In *Proc. of the 8th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 44–52, 2001.

[29] H. Sundell and P. Tsigas. NOBLE: A non-blocking inter-process communication library. In *Proc. of the 6th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, Washington, DC, Mar. 2002.

[30] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Proc. of the 2003 Intl. Parallel and Distributed Processing Symp.*, Nice, France, Apr. 2003.

[31] H. Takada and K. Sakamura. A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels. In *18th IEEE Real-Time Systems Symp.*, pages 134–143, 1997.

[32] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *Proc. of the 22nd Annual Intl. Symp. on Computer Architecture*, pages 24–36. ACM Press, 1995.

[33] H. Zheng and J. Nieh. SWAP: A scheduler with automatic process dependency detection. In *Networked Systems Design and Implementation*, 2004.

A Time and Space Bounds

This appendix provides an informal analysis of time and space requirements for the MCS-TP and CLH-TP locks. Figure 13 provides an overview summary of worst- and expected-case processor time steps for timing out and lock handoff, as well as per-lock queue length and total memory requirements.

A.1 MCS-TP Bounds

We first consider space management in MCS-TP. Because no thread can ever have more than one node in the queue, the queue length is trivially linear in the number of threads T. A thread cannot reuse a node for another lock until that node is first removed from the previous lock's queue. This gives a worst-case space consumption for L locks of $O(T \times L)$. However, since lock holders clean up timed-out nodes during lock handoff, a thread will rarely have more than a small constant number of allocated nodes; this gives an expected space requirement of $O(T + L)$.

To time out, a waiting thread must update its node's state from `waiting` to `left`. It must also reclaim its node if removed from the queue by the lock holder. Both operations require a constant number of steps, so the overall time requirement for leaving is $O(1)$.

As discussed in Section 2.1, the MCS-TP lock holder removes at most T nodes from the queue before switching to a scan. Since each removal and each step of the scan can be done in $O(1)$ time, the worst case is that the lock holder removes T nodes and then scans through T more timed-out nodes before reaching the end of the queue. It then marks the queue empty and re-traverses the (former) queue to remove each node, for a total of $O(T)$ steps. In the expected case a thread's immediate successor is not preempted, allowing handoff in $O(1)$ steps.

A.2 CLH-TP Bounds

In our implementation, the CLH-TP lock uses a timeout protocol in which it stops publishing updated timestamps $k\mu s$[1] before its patience has elapsed, where k is the staleness bound for deciding that a thread has been preempted. Further, so long as a thread's node remains `waiting`, the thread continues to remove timed-out and preempted predecessors. In particular, a thread only checks to see if it has timed out if its predecessor is active.

A consequence of this approach is that thread A cannot time out before its successor B has had a chance to remove it for inactivity. If B is itself preempted, then any successor active before it is rescheduled will remove B's and then A's node; otherwise, B will remove A's node once rescheduled. This in turn implies that any pair of nodes in the queue abandoned by the same thread have at least one node between them that belongs to a thread that has not timed out. In the worst case, $T - 1$ nodes precede the first, suspended, "live" waiter, $T - 2$ precede the next, and so on, for a total of $O(T^2)$ total nodes in the queue.

As in MCS-TP, removing a single predecessor can be performed in $O(1)$ steps. As the queue length is bounded, so, too, is the timeout sequence. Unlike MCS-TP, successors are responsible for actively claiming the lock; a lock holder simply updates its state to show that it is no longer using the lock, clearly an $O(1)$ operation.

Since all waiting threads concurrently remove inactive nodes, it is unlikely that an inactive node will remain in the queue for long. In the common case, then, the queue length is close to the total number of threads currently contending for the lock. Since a thread can only contend for one

[1]For best performance, $k\mu s$ should be greater than the round-trip time for a memory bus or interconnect transaction on the target machine, plus the maximal pairwise clock skew observable between processors.

	Worst case		Common case
	MCS-TP	CLH-TP	(both locks)
Timeout	$O(1)$	$O(T)$	$O(1)$
Lock handoff	$O(T)$	$O(1)$	$O(1)$
Queue length	$O(T)$	$O(T^2)$	$O(T)$
Total space	$O(T \times L)$	$O(T^2 \times L)$	$O(T + L)$

Figure 13: Time/space bounds for T contending threads and L locks.

lock at a time, we can expect average-case space $O(T + L)$. Similarly, the average timeout delay is $O(1)$ if most nodes in the queue are actively waiting.

B MCS-TP Lock Algorithm

B.1 Data structures

```
typedef struct mcstp_qnode {
  mcstp_lock *last_lock;      /* lock ptr from last timeout attempt. */
  volatile hrtime_t time;     /* node owner's published timestamp */
  volatile qnode_status status;
  struct mcstp_qnode *volatile next;
} mcstp_qnode;

typedef struct mcstp_lock {
  mcstp_qnode *volatile tail;
  volatile hrtime_t cs_start_time;
} mcstp_lock;
```

B.2 Lock Routines

B.2.1 Global constants and subroutines

- **MAXIMUM_CS_TIME**: An estimated maximum length for a critical section. Any thread holding the lock for a longer period is assumed to have been preempted. To help the lock holder be rescheduled, threads which time out whilst waiting yield their processors.

  If this constant is under-estimated, waiting threads may yield when the lock holder is not preempted. If over-estimated, waiting threads will be less reactive to preemption in the critical section.

- **gethrtime()** : returns the latest processor time

- **swap(ptr, new)**:

  ```
  atomic { old = (*ptr); (*ptr) = new; return old; }
  ```

- **is_TIMEDOUT(start_time, T)**: returns true if the current time has passed the deadline (start_time + T)

- **CAS_BOOL(ptr, oldval, newval)**: compare-and-swap, return true if successful

  ```
  atomic {
    if ((*ptr) != oldval)
      return false;
    (*ptr) = newval;
    return true;
  }
  ```

B.2.2 Main acquire routine

```
int mcstp_acquire(mcstp_lock * L, mcstp_qnode * I, hrtime_t T)
Input:
   L - the lock pointer
   I - my queue node for the lock
   T - my patience to wait in the queue, in CPU ticks
Output:
   1:  the attempt is successful
   -1: the attempt failed (possibly because I was preempted) and my
   queue node is safe to reclaim
   -2: the attempt timed out and my queue node remains enqueued
begin
  mcstp_qnode *pred;
  hrtime_t start_time = gethrtime();

  /* if my status is "timeout", CAS it to waiting; otherwise,
```

```
     just start a new try   */
  if (I->status == timeout && I->last_lock == L &&
    CAS_BOOL(&I->status, timeout, waiting))
    /* I have rejoined the queue. */
  else
    /* initialize and enqueue */
    I->status = waiting;
    I->next = 0;
    pred = swap(&L->tail, I);

    if (!pred)   /* lock was free */
      L->cs_start_time = gethrtime();
      return 1;
    else
      pred->next = I;
    endif

  loop
    if (I->status == available)
      L->cs_start_time = gethrtime();
      return 1;

    elif (I->status == failed)
      if (is_TIMEDOUT(L->cs_start_time, MAXIMUM_CS_TIME)
         sched_yield();
      endif
      I->last_lock = L;
      return -1;
    endif

    while (I->status == waiting)  /* the lock holder hasn't changed me. */
      I->time = gethrtime();
      if (!is_TIMEDOUT(start_time, T))
         continue;
      endif
      if (!CAS_BOOL(&I->status, waiting, timeout))
         break;
      endif
      if (is_TIMEDOUT(L->cs_start_time, MAXIMUM_CS_TIME))
         sched_yield();
      endif
      return -2;
    endwhile
  endloop
end
```

B.3 Unlock Routine

B.3.1 Global constants and subroutines

- **MAX_THREAD_NUM**: The approximate maximum number of threads in the system.

  This is used when handling the pathological (treadmill) case. An under-estimate will introduce some delay when aborting inactive acquisition attempts. An over-estimate will force the lock holder to scan more nodes in the pathological case.

- **UPDATE_DELAY**: The approximate length of time it takes a process to see a timestamp published on another thread, including any potential skew between the two system clocks.

B.3.2 Main routine

```
void mcstp_release (mcstp_lock *L, mcstp_qnode *I)
Input:
   L - the lock pointer
   I - my queue node for the lock
begin
  mcstp_qnode *succ, *currentI;
  mcstp_qnode *scanned_queue_head=NULL;
  int scanned_nodes = 0;
  currentI = I;    /* ptr to the currently scanned node */
  /* Case 1: if no successor, fix the global tail ptr and return;
     Case 2: if a successor inactive, grab its next ptr and fail its attempt;
     Case 3: if a successor active, set it to available. */
  loop
    succ = currentI->next;
    if (!succ)
      if (CAS_BOOL(&L->tail, currentI, 0))
        currentI->status = failed;
        return;                 /* I was last in line. */
      endif
      while (!succ)
        succ = currentI->next;
      endwhile
    endif
    if (++scanned_nodes < MAX_THREAD_NUM)
      currentI->status = failed;
    elif (!scanned_queue_head)
      scanned_queue_head = currentI; /* handle the treadmill case */
    endif
    if (succ->status == waiting)
      hrtime_t succ_time = succ->time;
      if (!is_TIMEDOUT(succ_time, UPDATE_DELAY) &&
           CAS_BOOL(&succ->status, waiting, available))
        if (scann_queue_head)
          set status to "failed" for nodes from scan_queue_head to currentI;
        endif
        return;
      endif
    endif
    currentI = succ;
  endloop
end
```

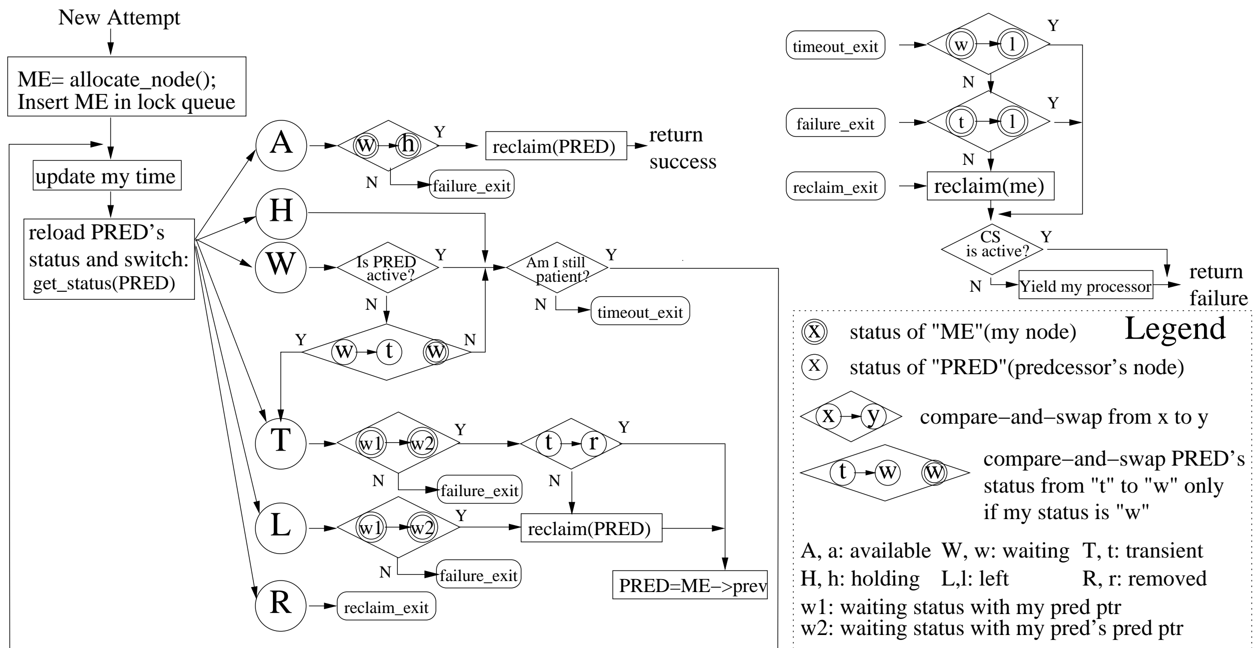


Figure 14: Control Flow Graph for the CLH-TP Locking

C CLH-TP Lock Algorithm

Figure 14 provides an overview of control flow for the CLH-TP algorithm.

C.1 Data structures

```
typedef struct clhtp_qnode {
  volatile hrtime_t time;
  char padding[CACHELINE];
  struct clhtp_qnode *volatile prev;
} clhtp_qnode;

typedef clhtp_qnode *volatile clhtp_qnode_ptr;

typedef struct clhtp_lock {
  clhtp_qnode_ptr tail;
  clhtp_qnode_ptr lock_holder;
  volatile hrtime_t cs_start_time;
} clhtp_lock;
```

C.2 Lock Routine

C.2.1 Global constants and subroutines

```
#define UPDATE_DELAY 200
// bit tags
#define WAITING  0
#define PTIMEOUT 2              /* transient */
#define SUCRC    1              /* left */
#define TAGMASK  0x3
#define PTRMASK  (~ TAGMASK)
```

```
// value tags
#define INITIAL   (void*) 0x7  /*   111B */
#define HOLDING   (void*) 0xB  /*  1011B */
#define AVAILABLE (void*)0x13  /* 10011B */
#define SELFRC    (void*)0x23  /*100011B -- removed */

#define get_tagbits(_ptr)   ((unsigned long)(_ptr) & TAGMASK)
#define get_ptr(_taggedptr) ((clhtp_qnode *)((unsigned long)(_taggedptr) & PTRMASK))
#define replace_tag(_ptr, _tag) (((unsigned long)(_ptr) & PTRMASK) | _tag )
#define tagged_wptr(_ptr, _tag) ((unsigned long)(_ptr) | _tag)

bool clhtp_swap_and_set(clhtp_qnode *volatile *swap_addr,
        clhtp_qnode *new_ptr, clhtp_qnode *volatile *set_addr)
/* atomic operation which saves the old value of swap_addr in
   set_addr, and swaps the new_ptr into the swap_addr. */
begin
  unsigned long pred;
  repeat
    pred = LL(swap_addr);
    (*set_addr) = (clhtp_qnode *) pred;
  while (0 == SC(swap_addr, new_ptr));
  return (clhtp_qnode *)pred;
end

bool clhtp_tag_cas_bool(clhtp_qnode * volatile * p,
        unsigned long oldtag, unsigned long newtag)
/* atomic compare and swap the tag in the pointer. */
begin
  unsigned long oldv, newv;
  repeat
    oldv = LL(p);
    if (get_tagbits(oldv) != oldtag)
      return false;
    endif
    newv = replace_tag(oldv, newtag);
  while (0 == SC(p, newv));
  return true;
end

bool clhtp_rcas_bool(clhtp_qnode *volatile *stateptr, clhtp_qnode *volatile *ptr,
        clhtp_qnode *oldp, unsigned long newv)
begin
  unsigned long oldv = (unsigned long)oldp;
  repeat
    unsigned long tmp = LL(ptr);
    if (get_tagbits(*stateptr) != WAITING)
      return false;
    endif
    if (tmp != oldv)
      return true;
    endif
  while (0 == SC(ptr, newv));
  return true;
end
```

```
void clhtp_failure_epilogue(clhtp_lock *L, clhtp_qnode *I)
begin
  if (I->prev==SELFRC || !clhtp_tag_cas_bool(&I->prev, PTIMEOUT, SUCRC))
    free_clhtp_qnode(I, 5);
  endif
end

void clhtp_success_epilogue(clhtp_lock *L, clhtp_qnode *I, clhtp_qnode *pred)
begin
  L->lock_holder = I;
  L->cs_start_time = gethrtime();
  free_clhtp_qnode(pred, 8);
end
```

C.2.2 Main routines

The main locking routine is **clhtp_acquire()**.

```
bool clhtp_acquire(clhtp_lock *L, hrtime_t T)
begin
  clhtp_qnode *I = alloc_qnode();
  clhtp_qnode *pred;

  I->time = gethrtime();
  pred = swap_and_set(&L->tail, I, &I->prev);
  if (pred->prev == AVAILABLE)
    if (CAS_BOOL(&I->prev, pred, HOLDING))
      clhtp_success_epilogue(L, I, pred);
      return true;
    else
      clhtp_failure_epilogue(L, I);
      if (gethrtime() - L->cs_start_time > MAXIMUM_CSTICKS)
         sched_yield();
      endif
      return false;
    endif
  endif

  /* the lock is occupied, set local time variables and go waiting. */
  bool result = clhtp_acquire_slow_path(L, T, I, pred);
  if (!result)
     if (gethrtime() - L->cs_start_time > MAXIMUM_CSTICKS)
         sched_yield();
     endif
  endif
  return result;
end

bool clhtp_acquire_slow_path(clhtp_lock *L, hrtime_t T,
        clhtp_qnode * I,clhtp_qnode * pred)
begin
  hrtime_t my_start_time, current, pred_time;

  my_start_time = I->time;
```

```
pred_time = pred->time;

while (true)
  clhtp_qnode *pred_pred;
  current = gethrtime();
  I->time = current;
  pred_pred  = pred->prev;

  if (pred_pred == AVAILABLE)
    if (CAS_BOOL(&I->prev, pred, HOLDING))     goto label_success;
    else                                       goto label_failure;
    endif
  elif (pred_pred == SELFRC)                   goto label_self_rc;
  elif (pred_pred == HOLDING or INITIAL)       goto check_self;

  //  INITIAL: I->prev's write may haven't propagated to other processors
  //  though the tail has been propagate.
  else
    clhtp_qnode *pp_ptr;
    unsigned int pred_tag;

    pred_tag = get_tagbits(pred_pred);
    pp_ptr = get_ptr(pred_pred);

    if (pred_tag == SUCRC)
      if (!CAS_BOOL(&I->prev, pred, pp_ptr))   goto label_failure;
      endif
      free_clhtp_qnode(pred,1);
      pred = pp_ptr;
      pred_time = pred->time;
      continue;

    elif (pred_tag == PTIMEOUT)
      if (!CAS_BOOL(&I->prev, pred, pp_ptr))   goto label_failure;
      endif
      if (!CAS_BOOL(&pred->prev, pred_pred, SELFRC))
        free_clhtp_qnode(pred, 2);
      endif
      pred = pp_ptr;
      pred_time = pred->time;
      continue;

    elif (pred_tag == WAITING)
      if (is_TIMEDOUT(pred_time, current, UPDATE_DELAY))
        if (pred->time != pred_time)
          pred_time = pred->time;
          continue;
        elif (clhtp_rcas_bool(&I->prev, &pred->prev, pred_pred,
                                 tagged_wptr(pred_pred, PTIMEOUT)))
          continue;
      endif
    endif
  endif
```

```
  check_self:
    unsigned int my_tag;
    pred = I->prev;
    if (pred == SELFRC)         goto label_self_rc;
    endif
    my_tag = get_tagbits(pred);
    if (my_tag == PTIMEOUT)     goto label_failure;
    elif (my_tag == WAITING)
      if (is_TIMEDOUT(my_start_time, current, T))
        goto label_self_timeout;
      endif
    endif
  endwhile

 label_success:
  clhtp_success_epilogue(L, I, pred);
  return true;

 label_failure:
 label_self_rc:
  clhtp_failure_epilogue(L, I);
  return false;

 label_self_timeout:
  if (!CAS_BOOL(&I->prev, pred, tagged_wptr(pred, SUCRC)))
    clhtp_failure_epilogue(L, I);
  return false;
end
```

C.3 Unlock Routine

```
void clhtp_try_release(clhtp_lock *L)
begin
  clhtp_qnode *I = L->lock_holder;
  I->prev = AVAILABLE;
end
```