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

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

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

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 (test-and-set style locks) or handle preemption badly (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 traditional spin locks on a 32-processor IBM p690 multiprocessor. These results show 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 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.

* This work was supported in part by NSF grants numbers CCR-9988361, EIA-0080124, CCR-0204344, and CNS-0411127, by DARPA/ITO under AFRL contract F29601-00-K-0182, by financial and equipment grants from Sun Microsystems Laboratories, and by an IBM Shared University Research grant.

D.A. Bader et al. (Eds.): HiPC 2005, LNCS 3769, pp. 7–18, 2005.

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 [12, 15], multiprogrammed systems usually rely on non-queue-based locks [19]. 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 [11, 26, 27]. 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 [26] or very high base time overhead [11].

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 [20, 22, 23, 28, 29]), general-purpose mechanisms remain elusive. Several groups (including our own) are working on this topic [6, 10, 17, 25], 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, 13, 18, 24] or recover from [1, 2, 30, 32] 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 run-time states. In particular, given a low-overhead hardware timer with bounded skew across processors and a memory bus that handles requests in bounded time[1] 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 can then selectively pass a lock only between active threads. Although this doesn't solve

[1] Our requirements are modest: While it must be possible to read the clock within, say, $100ns$, clock skew or remote access times of tens of microseconds would be tolerable. Most modern machines do much better than that.

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 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 [19] and CLH [3, 16] queue-based spin locks. Their `acquire` functions return `success` if the thread acquired the lock within a supplied *patience* interval parameter, and `failure` otherwise. In both locks, the thread owning the head node of a linked-list queue 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 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 [26]. 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, 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 Table 1. Time and space bounds are considered in the technical report version of this paper [7].

2.1 MCS Time-Published Lock

Our first algorithm is adapted from Mellor-Crummey and Scott's MCS lock [19]. 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`.

Table 1. Comparison between MCS and CLH time-published locks

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

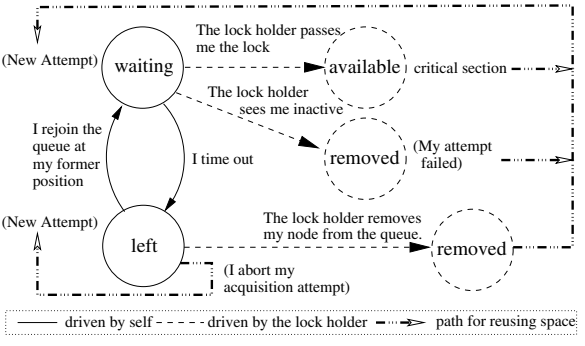

Fig. 1. State transitions for MCS-TP queue nodes

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 1 shows the state transitions for MCS-TP queue nodes. Source code can be found in the technical report version of this paper [7]. It runs to about 3 pages.

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.

2.2 CLH Time-Published Lock

Our second time-published lock is based on the CLH lock of Craig [3] and Landin and Hagersten [16]. In CLH, 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 time-out 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 [26, 27] suggests that it would be very difficult to add to MCS-TP. Source code for the CLH-TP lock can be found in the technical report version of this paper [7]. It runs to about 5 pages.

We use low-order bits in a CLH-TP node's `prev` pointer to store the node's state, allowing us to atomically modify the state and the pointer together. 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 2 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.

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,

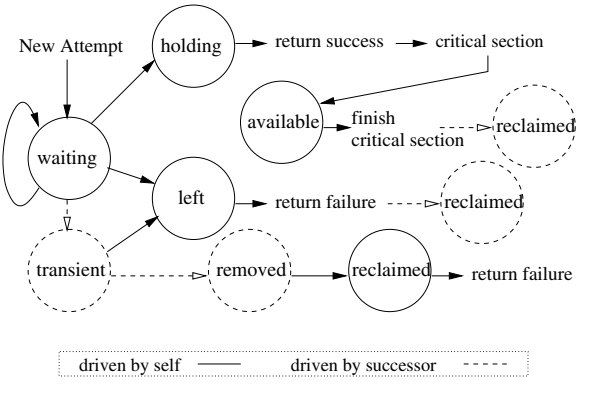

Fig. 2. State transitions for CLH-TP queue nodes

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 change β's state atomically from `transient` or `waiting` to `left`. If the attempt 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. Either way, whichever of B and its successor notices *last* that β has been removed from the queue handles the memory reclamation.

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. A custom atomic construction (ommitted here but shown in the TR version [7]) implements this operation using *load-linked/store-conditional*. 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, 21].

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, the timing window is too narrow to have a noticeable impact on performance. Although space limitations preclude further discussion of scheduler-conscious locks, a full analysis and an empirical study of the matter may be found in the TR version of this paper [7].

4 Microbenchmark Results

We test our TP locks on an IBM pSeries 690 (Regatta) multiprocessor with 32 1.3 GHz Power4 processors, running AIX 5.2. 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 [26]. We also test *spin-then-yield* variants [12] in which threads yield after exceeding a wait threshold.

In our microbenchmark, each thread repeatedly attempts 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 $50\,\mu s$.

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 present results in Figure 3.

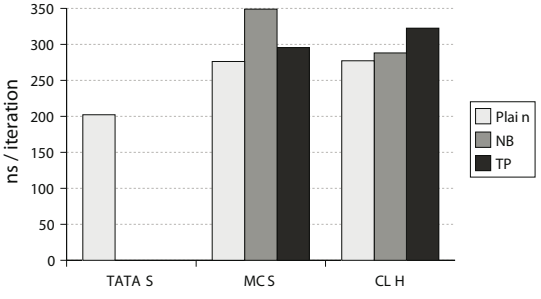

Fig. 3. Single-thread performance results

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 contention from memory and processor interconnect bus traffic and adds cache coherence overhead when transferring a cache line between processors. We have measured atomic update overheads of 120 and $420\,ns$ with 2 and 32 threads.

4.2 Multi-thread Performance

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 examine two typical configurations.

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 4 plots lock performance for this configuration. Up through 32 threads (our machine size), queue-based locks outperform TAS; however, only the TP and TAS locks maintain throughput

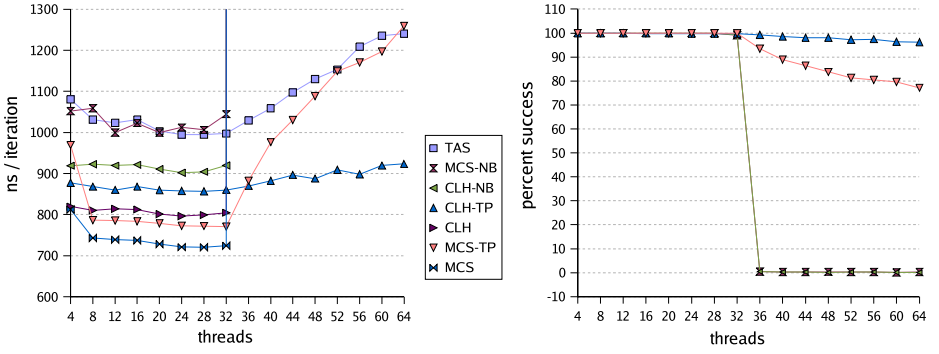

Fig. 4. 2 cache line-update critical section (CS). 1 μs non-critical section (NCS). Critical section service time (left) and success rate (right) on a 32-processor machine.

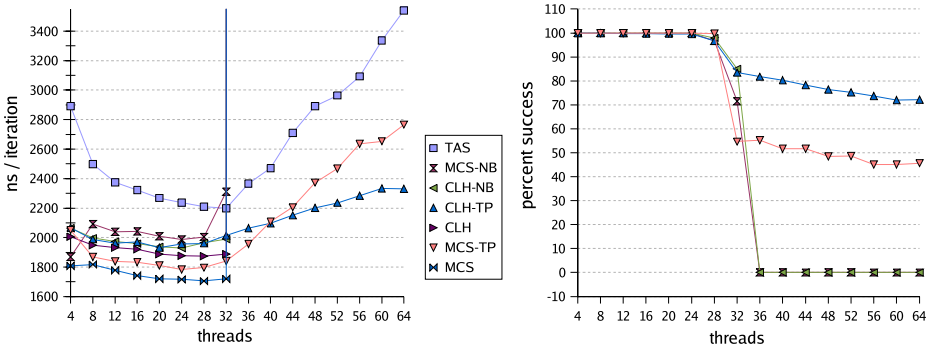

Fig. 5. 40 cache line CS; 4 μs NCS. CS service time (left) and success rate (right).

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 4 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 μs non-critical sections. It models larger longer operations in which preemption of the lock holder is more likely. Figure 5 shows lock performance for 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 5, the success rates for abortable locks drop off beyond 24 threads. Since each critical section takes about 2 μs, our 50 μs patience is just enough for a thread to sit through 25 predecessors. We note that TP locks adapt better to insufficient patience.

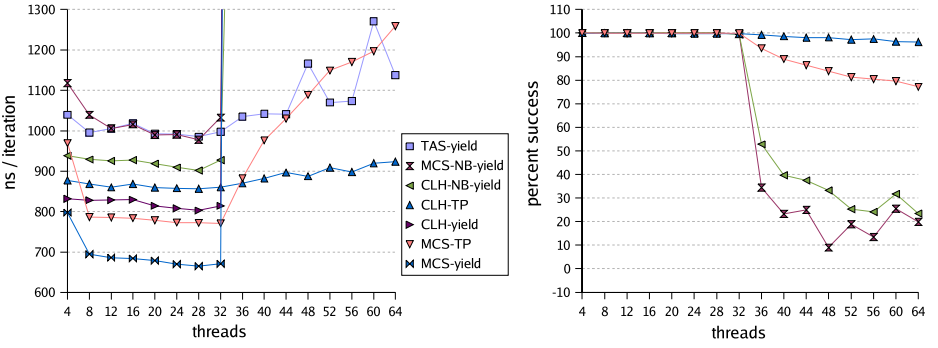

Fig. 6. Spin-then-yield variants; 2 cache line CS; 1 μs NCS

One might expect a spin-then-yield policy [12] to allow other locks to match TP locks in preemption adaptivity. In Figure 6 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.

4.3 Time and Space Bounds

Finally, 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 [7], one can estimate an upper bound on delay for lock handoff when the holder is not preempted.

In our analysis of the CLH-TP lock's space bounds [7] we show a worst-case bound quadratic in the number of threads, but claim an expected linear value. In tests designed to maximize space contention (full details available in the TR version [7]), we find space consumption to be very stable over time. Even with very short patience, we obtain results far closer to the expected linear than the worst-case quadratic space bound.

5 Application Results

In this section we measure the performance of our TP locks on the Raytrace and Barnes benchmarks from the SPLASH-2 suite [31].

Application Features: Raytrace and Barnes are heavily synchronized [14, 31]. Raytrace uses no barriers but features high contention on a small number of locks. Barnes uses few barriers but numerous locks. Both offer reasonable parallel speedup.

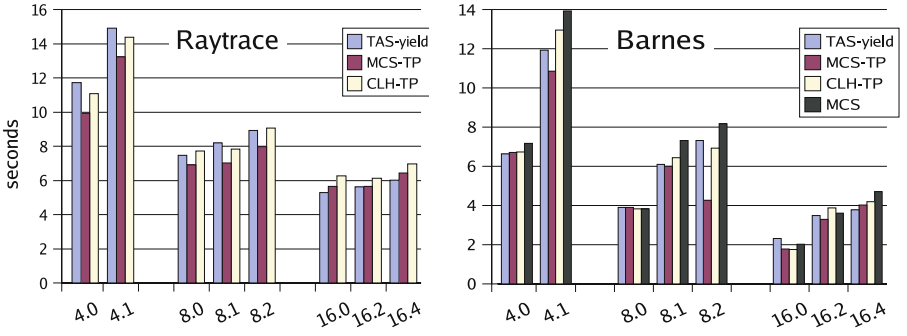

Fig. 7. Parallel execution times for Raytrace and Barnes on a 32-processor IBM p690 Regatta. Test *M.N* uses M application threads and $(32 - M) + N$ external threads.

Experimental Setup: We test the locks from Section 4 and 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 7 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 4, 5, and 6). 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. This can be confirmed by noting that a highly preemption-sensitive lock, MCS, "only" doubles its execution time given heavy preemption. We therefore attribute Barnes' relatively severe preemption-induced performance degradation to the barriers it uses.

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. Overall, MCS-TP outperforms CLH-TP; this is consistent with our microbenchmark results. We speculate that this disparity is due to lower base-case overhead in the MCS-TP algorithm combined with short-lived lock acquisitions in these applications.

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 nonstandard 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. *USENIX Workshop on Unix and Supercomputers*, Sep. 1988.

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

[6] T. Harris and K. Fraser. Language support for lightweight transactions. *18th Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 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. *16th Conf. on Distributed Computing*, Oct. 2002.

[10] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. *22nd ACM Symp. on Principles of Distributed Computing*, July 2003.

[11] P. Jayanti. Adaptive and efficient abortable mutual exclusion. *22nd ACM Symp. on Principles of Distributed Computing*, July 2003.

[12] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. *13th SIGOPS Symp. on Operating Systems Principles*, Oct. 1991.
[13] L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott. Scheduler-conscious synchronization. *ACM Trans. on Computer Systems*, 15(1):3–40, Feb. 1997.
[14] S. Kumar, D. Jiang, R. Chandra, and J. P. Singh. Evaluating synchronization on shared address space multiprocessors: Methodology and performance. *SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, May 1999.
[15] B.-H. Lim and A. Agarwal. Reactive synchronization algorithms for multiprocessors. *6th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994.
[16] P. S. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. *8th Intl. Parallel Processing Symp.*, Apr. 1994.
[17] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Design Tradeoffs in Modern Software Transactional Memory Systems. *7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, Oct. 2004.
[18] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. *13th SIGPOPS Symp. on Operating Systems Principles*, Oct. 1991.
[19] 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.
[20] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. *14th Symp. on Parallel Algorithms and Architectures*, Aug. 2002.
[21] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. on Parallel and Distributed Systems*, 15(8)491–504, 2004.
[22] M. M. Michael. Scalable lock-free dynamic memory allocation. *SIGPLAN Symp. on Programming Language Design and Implementation*, June 2004.
[23] 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.
[24] J. K. Ousterhout. Scheduling techniques for concurrent systems. *3rd Intl. Conf. on Distributed Computing Systems*, Oct. 1982.
[25] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. *24th ACM Symp. on Principles of Distributed Computing*, July 2005.
[26] M. L. Scott. Non-blocking timeout in scalable queue-based spin locks. *21st ACM Symp. on Principles of Distributed Computing*, July 2002.
[27] M. L. Scott and W. N. Scherer III. Scalable queue-based spin locks with timeout. *8th ACM Symp. on Principles and Practice of Parallel Programming*, June 2001.
[28] H. Sundell and P. Tsigas. NOBLE: A non-blocking inter-process communication library. *6th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, Mar. 2002.
[29] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Intl. Parallel and Distributed Processing Symp.*, Apr. 2003.
[30] H. Takada and K. Sakamura. A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels. *18th Real-Time Systems Symp.*, Dec. 1997.
[31] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The Splash-2 programs: Characterization and methodological considerations. *22nd Intl. Symp. on Computer Architecture*, Jun. 1995.
[32] H. Zheng and J. Nieh. SWAP: A scheduler with automatic process dependency detection. *Networked Systems Design and Implementation*, Mar. 2004.