Tech. Rep. 614

Concurrent Update on Multiprogrammed Shared Memory Multiprocessors*

Maged M. Michael   Michael L. Scott

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

April 1996

Abstract

Most multiprocessors are multiprogrammed in order to achieve acceptable response time and to increase utilization. Unfortunately, inopportune preemption may significantly degrade the performance of synchronized parallel applications. To address this problem, researchers have developed two principal strategies for concurrent, atomic update of shared data structures: *preemption-safe locking* and *non-blocking* (lock-free) *algorithms*. Preemption-safe locking requires kernel support. Non-blocking algorithms generally require a universal atomic primitive such as `compare_and_swap` or `load_linked/store_conditional`.

We focus in our study on four simple but important concurrent data structures—stacks, FIFO queues, priority queues and counters—in synthetic kernels and real applications on a 12-processor SGI Challenge multiprocessor. Our results indicate that efficient, data-structure-specific non-blocking algorithms, which exist for stacks, FIFO queues and counters, outperform both preemption-safe and ordinary locks by 20–40% in real applications and 40–55% in synthetic kernels on both multiprogrammed and dedicated systems (general-purpose non-blocking techniques do not yet appear to be practical). At the same time, preemption-safe locks outperform conventional locks by significant margins on multiprogrammed systems. The clear conclusion is that data-structure-specific non-blocking algorithms should be used whenever possible. For data structures for which such algorithms are not known, operating systems for multiprogrammed parallel machines should provide preemption-safe locks.

Keywords: non-blocking, lock-free, mutual exclusion, locks, multiprogramming, concurrent queues, concurrent stacks, concurrent heaps, `compare_and_swap`, `load_linked`, `store_conditional`.

*This work was supported in part by NSF grants nos. CDA–94–01142 and CCR–93–19445, and by ONR research grant no. N00014–92–J–1801 (in conjunction with the DARPA Research in Information Science and Technology—High Performance Computing, Software Science and Technology program, ARPA Order no. 8930).

1 Introduction

On shared memory multiprocessors, processes communicate using shared data structures. To ensure the consistency of these data structures, processes perform synchronized concurrent update operations, mostly using critical sections protected by mutual exclusion locks. In order to achieve acceptable response time and high utilization, most multiprocessors are multiprogrammed by time-slicing processors among processes. The performance of mutual exclusion locks in parallel applications degrades on time-slicing multiprogrammed systems [29] due to preemption of processes holding locks. Any other processes busy-waiting on the lock are then unable to perform useful work until the preempted process is rescheduled and subsequently releases the lock.

Alternative multiprogramming schemes to time-slicing have been proposed in order to avoid the adverse effect of time-slicing on the performance of synchronization operations. However, each has limited applicability and/or reduces the utilization of the multiprocessor. Coscheduling, proposed by Ousterhout [18], ensures that all processes of an application run together. It has the disadvantage of reducing the utilization of the multiprocessor if applications have a variable amount of parallelism, or if processes cannot be evenly assigned to time-slices of the multiprocessor. Another alternative is hardware partitioning, under which no two applications share a processor. This scheme has the disadvantage of requiring applications to be able to adjust their number of processes as new applications join the system. Otherwise, processes from the same application might have to share the same processor, allowing one to be preempted while holding a mutual exclusion lock. Traditional time-slicing remains the most widely used scheme of multiprogramming on multiprocessor systems.

For time-sliced systems, researchers have proposed two principal strategies to avoid inopportune preemption: *preemption safe locking* and *non-blocking algorithms*. Most preemption-safe locking techniques require a widening of the kernel interface, to facilitate cooperation between the application and the kernel. Generally, these techniques try either to recover from the preemption of lock-holding processes (or processes waiting on queued locks), or to avoid preempting processes while holding locks. Further discussion of this direction is presented in section 2.

An implementation of a data structure is *non-blocking* (also known as *lock-free*) if it guarantees that at least one process of those trying to concurrently update the data structure will succeed in completing its operation within a bounded amount of time, assuming that at least one process is active, regardless of the state of other processes. Non-blocking algorithms do not require any communication with the kernel and by definition they cannot use mutual exclusion. Rather, they generally rely on hardware support for a universal[1] atomic primitives such as `compare_and_swap`[2] or the pair `load_linked` and `store_conditional`,[3] while mutual exclusion locks can be implemented using weaker atomic primitives such as `test_and_set`, `fetch_and_increment`, or `fetch_and_store`. Further discussion of non-blocking implementations is presented in section 3.

Few of the above mentioned techniques have been evaluated experimentally, and then only in comparison to ordinary mutual exclusion locks. Our contribution is to evaluate the relative performance of preemption-safe and non-blocking atomic update on multiprogrammed (time-sliced) systems. We focus on four simple but important data structures: counters, queues, stacks, and priority queues. Our experiments employ both synthetic kernels and real applications, on a 12-processor Silicon Graphics Challenge multiprocessor. We describe our methodology and results in section 4. We find that efficient (data-structure-specific) non-blocking algorithms clearly outperform both ordinary and preemption-safe lock-based alternatives, not only on time-sliced systems, but on dedicated machines as well. At the same time, preemption-safe algorithms outperform ordinary locks on time-sliced systems, and should therefore be supported by multiprocessor operating systems. We do not examine general-purpose non-blocking techniques in detail; previous work indicates that they are highly inefficient (though they provide a level of fault tolerance unavailable with locks). We summarize our conclusions and recommendations in section 5.

[1] Herlihy [9] presented a hierarchy of non-blocking objects that also applies to atomic primitives. A primitive is in level n of the hierarchy if it can provide a non-blocking solution to a consensus problem for up to n processors. Primitives in higher levels of the hierarchy can provide non-blocking implementations of those in lower levels, while the reverse is impossible. `Compare_and_swap` and the pair `load_linked` and `store_conditional` are *universal* primitives as they are in level ∞ of the hierarchy. Widely supported primitives such as `test_and_set`, `fetch_and_add`, and `fetch_and_store`are in level 2.

[2] `Compare_and_swap`, introduced on the IBM System 370, takes as arguments the address of a shared memory location, an expected value, and a new value. If the shared location currently holds the expected value, it is assigned the new value atomically. A Boolean return value indicates whether the replacement occurred.

[3] `Load_linked` and `store_conditional`, proposed by Jensen et al. [11], must be used together to read, modify, and write a shared location. `Load_linked` returns the value stored at the shared location. `Store_conditional` checks if any other processor has since written to that location. If not then the location is updated and the operation returns success, otherwise it returns failure. `Load_linked/store_conditional` is supported (on bus-based multiprocessors) by the MIPS II, PowerPC, and Alpha architectures.

2 Preemption-Safe Locking

For simple mutual exclusion locks (e.g. `test_and_set`), preemption-safe locking techniques allow the system either to avoid or to recover from the preemption of processes holding locks. Edler et al.'s Symunix system [7] provides an avoidance technique: a process may set a flag requesting that the kernel not preempt it because it is holding a lock. The kernel will honor the request up to a pre-defined time limit, setting a second flag to indicate that it did so, and deducting any extra execution time from the beginning of the process's next quantum. A process should yield the processor if it finds, upon leaving a critical seciton, that it was granted an extension.

The *first-class threads* of Marsh et al.'s Psyche system [14] provide a different avoidance technique: they require the kernel to warn an application process a fixed amount of time in advance of preemption, by setting a flag that is visible in user space. If a process verifies that the flag if unset before entering a critical section (and if critical sections are short), then it is guaranteed to be able to complete its operation in the current quantum. If it finds the flag is set, it can voluntarily yield the processor.

Black's work on Mach [6] includes a recovery technique: a process may suggest to the kernel that it be descheduled in favor of some specific other process (presumably the one that is holding a desired lock). The *scheduler activations* of Anderson et al. [4] also support recovery: when a processor is taken from an application, another processor belonging to the same application is informed via software interrupt. If the preempted thread was holding a lock, the interrupted processor can perform a context switch to the preempted thread and push it through the critical section.

Simple preemption-avoidance techniques rely on the fact that processes acquire a `test_and_set` lock in non-deterministic order. Unfortunately, `test_and_set` locks do not scale well to large machines. Queue-based locks scale well, but impose a deterministic order on lock acquisitions, forcing a preemption-avoidance technique to deal with preemption not only of the process holding a lock, but of processes waiting in the lock's queue as well. Preempting and scheduling processes in an order inconsistent with their order in the lock's queue can degrade performance dramatically. Kontothanassis et al. [12, 27, 28] present scheduler-conscious versions of the ticket lock, the MCS lock [15], and Krieger et al.'s reader-writer lock [13]. These algorithms detect the descheduling of critical processes using handshaking and/or a widened kernel-user interface.

The proposals of Black and of Anderson et al. require the application to recognize the preemption of lock-holding processes and to deal with the problem. By performing recovery on a processor other than the one on which the preempted process last ran, they also sacrifice cache footprint. The proposal of Marsh et al. requires the application to estimate the maximum duration of a critical section, which is not always possible. To represent the preemption-safe approach in our experiments, we employ test-and-`test_and_set` locks with exponential backoff, based on the kernel interface of Edler et al. For machines the size of ours (12 processors), the results of Kontothanassis et al. indicate that these will out-perform queue-based locks.

3 Non-Blocking Implementations

Several non-blocking concurrent implementations of widely used data structures as well as general methodologies for developing such implementations systematically have been proposed in the literature. These implementations and methodologies were motivated in large part by the performance degradation of mutual exclusion locks as a result of arbitrary process delays, particularly those due to preemption on a multiprogrammed system.

Herlihy [8, 10] presented a general methodology for transforming sequential implementations of data structures to concurrent non-blocking implementations using `compare_and_swap` or `load_linked/store_conditional`. The basic methodology requires copying the whole data structure on every update. Herlihy also proposed an optimization by which the programmer can avoid some fraction of the copying for certain data structures; he illustrated this optimization in a non-blocking implementation of a skew-heap-based priority queue. Alemany and Felten [1] proposed several techniques to reduce unnecessary copying and useless parallelism associated with Herlihy's methodologies using extra communication between the operating system kernel and application processes. Barnes [5] presented a similar general methodology in which processes record and timestamp their modifications to the shared object, and cooperate whenever conflicts arise. Turek et al. [23] and Prakash et al. [20] presented methodologies for transforming multiple lock concurrent objects into lock-free concurrent objects.

In comparison to data-structure-specific non-blocking algorithms, these general methodologies tend to be inefficient, with significant overheads for copying and/or logging of operational steps. Many of the papers present their algorithms without experimental results. In general, the performance of non-blocking algorithms resulting from general methodologies is acknowledged to be significantly inferior to that of the corresponding lock-based implementations.

Prakash et al. [19, 21], Valois [24, 25], and Michael and Scott [17] proposed non-blocking implementations of concurrent link-based queues. Treiber [22] proposed a non-blocking implementation of concurrent link-based stacks. Valois [26] proposed a non-blocking implementation of linked lists. Anderson and Woll [3] proposed a non-blocking solution to the union-find problem. Simple non-blocking centralized counters can be implemented using a `fetch_and_add` atomic primitive (if supported by hardware), or a read-modify-check-write cycle using `compare_and_swap` or `load_linked/store_conditional`.

Performance results were reported for only a few of these algorithms [17, 21, 24]. The results of Michael and Scott indicate that their non-blocking implementation of link-based queues outperforms all other non-blocking and lock-based implementations, on both multiprogrammed and dedicated multiprocessors. The queue of Prakash et al. outperforms lock-based implementations in the case of multiprogramming.

No performance results were reported for non-blocking stacks. However, Treiber's stack is very simple and can be expected to be quite efficient. We also observe that a stack derived from Herlihy's general methodology, with unnecessary copying removed, seems to be simple enough to compete with lock-based implementations.

No practical non-blocking implementations for array-based stacks or circular queues have been proposed. The general methodologies can be used, but the resulting implementations would be very inefficient. For these data structures lock-based implementations seem to be the only option. However, many applications use link-based queues and stacks; if the data size is more than few bytes, then the time overhead of copying data in an array-based implementation is likely to overshadow the space overhead of links.

As representatives of the best available non-blocking algorithms on simple data structures, we use the following in our experiments: the non-blocking link-based queues of Michael and Scott [17] and Prakash et al. [21], the non-blocking link-based stack of Treiber [22], an optimized version of a stack resulting from applying Herlihy's methodology [10], a skew heap implementation due to Herlihy using his general methodology with optimized copying [10], and a `load_linked/store_conditional` implementation of counters.

4 Experimental Results

We use a 12-processor Silicon Graphics Challenge multiprocessor to compare the performance of the best non-blocking, ordinary lock-based, and preemption-safe lock-based implementations of counters and of link-based queues, stacks, and skew heaps. We use synthetic kernels to compare the performance of the alternative implementations under various levels of contention. We also use two versions of a parallel quicksort application, together with a parallel solution to the traveling salesman problem, to compare the performance of the implementations when used in a real application. C code for all the synthetic kernels and the real application can be obtained from `ftp://ftp.cs.rochester.edu/pub/packages/s ched_conscious_synch/multiprogramming`.

To ensure the accuracy of our results, we prevented other users from accessing the multiprocessor during the experiments. To evaluate the performance of the algorithms under different levels of multiprogramming, we used a feature of the Challenge's Irix operating system that allows programmers to pin processes to processors. We then used one of the processors to serve as a pseudo-scheduler. Whenever a process is due for preemption, the pseudo-scheduler interrupts it, forcing it into a signal handler. The handler spins on a flag which the pseudo-scheduler sets when the process can continue computation. The time spent executing the handler represents the time during which the processor is taken from the process and handed over to a process that belongs to another application.

All ordinary and preemption-safe locks used in the experiments are test-and-`test_and_set` locks with bounded exponential backoff. All non-blocking implementations also use bounded exponential backoff. The effectiveness of backoff in reducing contention on locks and synchronization data is demonstrated in the literature [2, 15]. The backoff was chosen to yield good overall performance for all implementations, and not to exceed 30 μs. Higher backoff delays sometimes appear to improve the performance of the synthetic kernels in the case of high contention, but this is only because they allow one process to perform a large number of operations without suffering any cache misses or interference from other processors, something that rarely happens in real applications.

In the figures, multiprogramming level represents the number of applications sharing the machine, with one process per processor per application. A multiprogramming level of 1 (the top graph in each figure) therefore represents a dedicated machine; a multiprogramming level of 3 (the bottom graph in each figure) represents a system with a process from each of three different applications on each processor.

4.1 Queues

Figure 1 shows performance results for six queue implementations on a dedicated system (no multiprogramming), and on multiprogrammed systems with 2 and 3 processes per processor. The six implementations are: the usual single-lock implementation using both ordinary and preemption-safe locks; an implementation due to Michael and Scott [17] that uses a pair of locks to protect the head and the tail of the queue, again using both ordinary and preemption-safe locks; and non-blocking implementations due to Michael and Scott [17] and Prakash et al. [21].

The horizontal axes of the graphs represent the number of processors. The vertical axes represent execution time normalized to that of the preemption-safe single lock implementation. This implementation was chosen as the basis of normalization because it yields the median performance among the set of implementation. We use normalized time in order to show the difference in performance between the implementations uniformly across different number of processors. If we were to use absolute time, the vertical axes would have to be extended to cover the high absolute execution time on a single processor, making the graph too small to read for larger numbers of processors. The absolute times in seconds for the preemption-safe single-lock implementation on one and 11 processors, with 1, 2, and 3 processes per processor, are 18.2 and 15.6, 38.8 and 15.4, and 57.6 and 16.3, respectively.

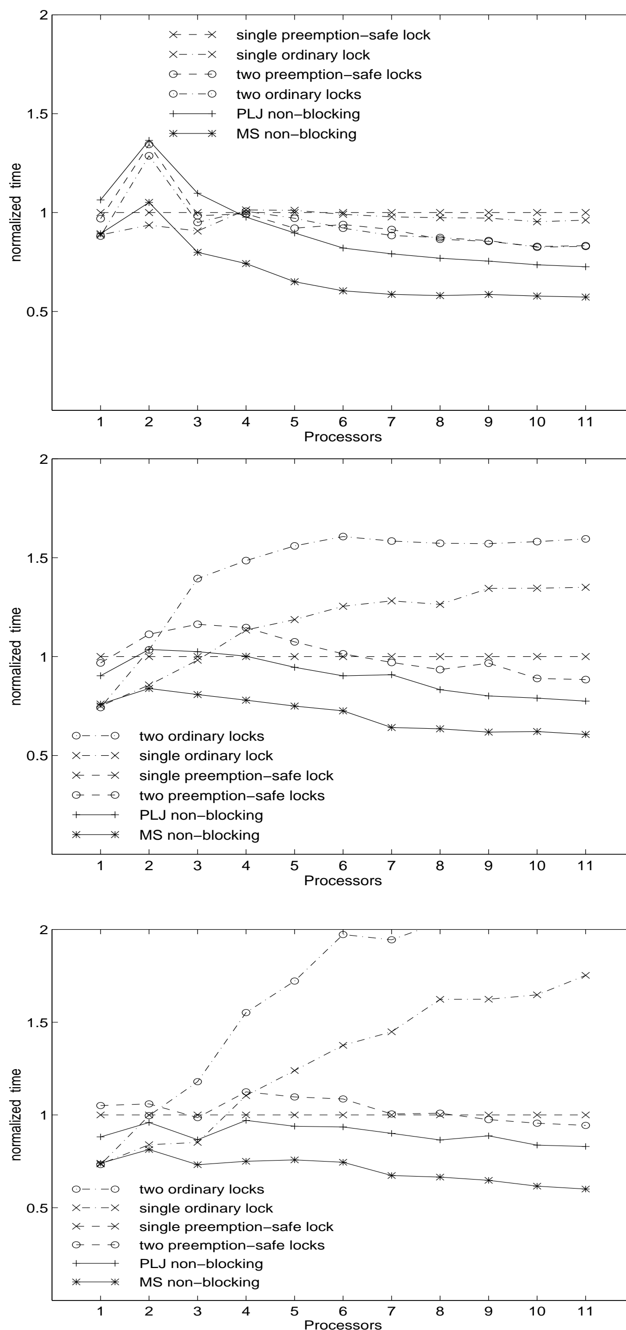

Figure 1: Normalized execution time for one million enqueue/dequeue pairs on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

The execution time is the time taken by all processors to perform one million pairs of enqueues and dequeues to an initially empty queue (each process performs 1,000,000/p enqueue/dequeue pairs, where p is the number of processors). Every process spends 6 μs spinning in an empty loop after performing every enqueue or dequeue operation (total 12 μs per iteration). This time is meant to represent "real" computation. It prevents one process from dominating the data structure and finishing all its operations while other processes are starved by caching effects and backoff.

The results show that as the level of multiprogramming increases the performance of ordinary locks degrades significantly, while the performance of preemption-safe locks and non-blocking algorithms remains relatively unchanged. Absolute time increases roughly linearly with increasing levels of multiprogramming. The "bump" at two processors is primarily due to cache misses, which do not occur on one processor.

The two-lock implementation outperforms the single-lock in the case of high contention, but it suffers more with multiprogramming when using ordinary locks, as the chances are larger that a process will be preempted while holding a lock needed by other processes. The non-blocking implementations provide added concurrency without being vulnerable to the effect of multiprogramming.

The non-blocking implementation of Michael and Scott yields the best overall performance even on dedicated systems. It outperforms the single-lock preemption-safe implementation by more than 40% on 11 processors with various levels of multiprogramming.

4.2 Stacks

Figure 2 shows performance results for four stack implementations on a dedicated system, and on multiprogrammed systems with 2 and 3 processes per processor. The four implementations are: the usual single lock implementation using ordinary and preemption-safe locks, a non-blocking implementations due to Treiber [22], and an optimized non-blocking implementation based on Herlihy's general methodology [10].

The axes in the graphs have the same semantics as those for the queue graphs. Execution time is normalized to that of the preemption-safe single lock implementation. The absolute times in seconds for the preemption-safe lock-based implementation on one and 11 processors, with 1, 2, and 3 processes per processor, are 18.9 and 20.3, 40.8 and 20.7, and 60.2 and 21.6, respectively. As in the synthetic application for queues, each process executes 1,000,000/p push/pop pairs on an initially empty stack, with a 6 μs delay between operations.

As the level of multiprogramming increases, the performance of ordinary locks degrades, while the performance of the preemption-safe and non-blocking implementations remains relatively unchanged. Treiber's implementation outperforms all the others even on dedicated systems. It outperforms the preemption-safe implementation by over 45% on 11 processors with various levels of multiprogramming.

4.3 Heaps

Figure 3 shows performance results for three heap implementations on a dedicated system, and on multiprogrammed systems with 2 and 3 processes per processor. The three implementations are: the usual single-lock implementation using ordinary and preemption-safe locks, and an optimized non-blocking implementation due to Herlihy [10].

The axes in the graphs have the same semantics as those for the queue and stack graphs. Execution time is normalized to that of the preemption-safe single lock implementation. The absolute times in seconds for the preemption-safe lock-based implementation on one and 11 processors, with 1, 2, and 3 processes per processor, are 21.0 and 27.7, 43.1 and 27.4, and 65.0 and 27.6, respectively. Each process executes 1,000,000/p insert/delete_min pairs on an initially empty heap with a 6 μs delay between operations. Experiments with non-empty heaps resulted in relative performance similar to that reported in the graphs.

As the level of multiprogramming increases the performance of ordinary locks degrades, while the performance of the preemption-safe and non-blocking implementations remains relatively unchanged. The degradation of the ordinary locks is larger than that suffered by the locks in the queue and stack implementations, because the heap operations are more complex and result in higher levels of contention. Unlike the case for queues and stacks, the non-blocking implementation of heaps is quite complex. It cannot match the performance of the preemption-safe lock implementation on either dedicated or multiprogrammed systems. Heap implementations resulting from general non-blocking methodologies (without data-structure-specific elimination of copying) are even more complex, and could be expected to perform worse.

4.4 Counters

Figure 4 shows performance results for three implementations of counters on a dedicated system, and on multiprogrammed systems with 2 and 3 processes per processor. The three implementations are: the usual single lock implementation using ordinary and preemption-safe locks, and the standard implementations using `load_linked/store_conditional` (`compare_and_swap` could be used instead).

The axes in the graphs have the same semantics to those for the previous graphs. Execution time is normalized to that of the preemption-safe single lock implementation. The absolute times in seconds for the preemption-safe lock-based implementation on one and 11 processors, with 1, 2, and 3 processes per processor, are 17.7 and 10.8, 35.0 and 11.3, and 50.6 and 10.9, respectively. Each process executes 1,000,000/p increments on a shared counter with a 6 μs delay between successive operations.

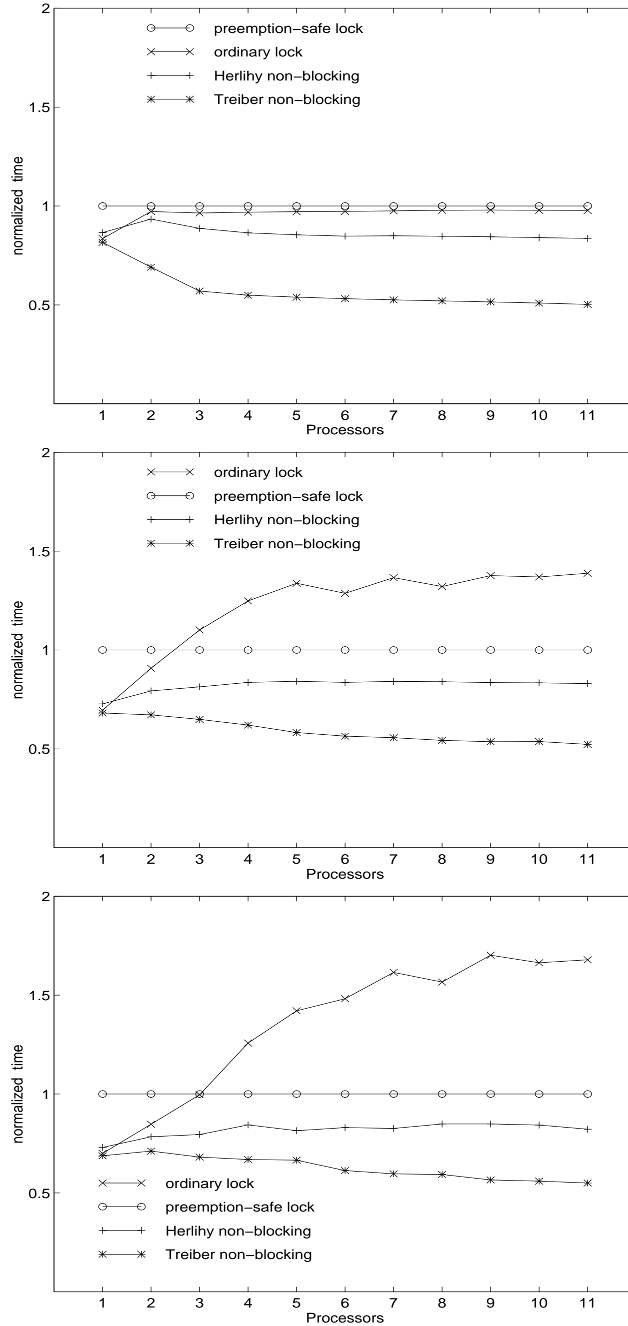

Figure 2: Normalized execution time for one million push/pop pairs on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

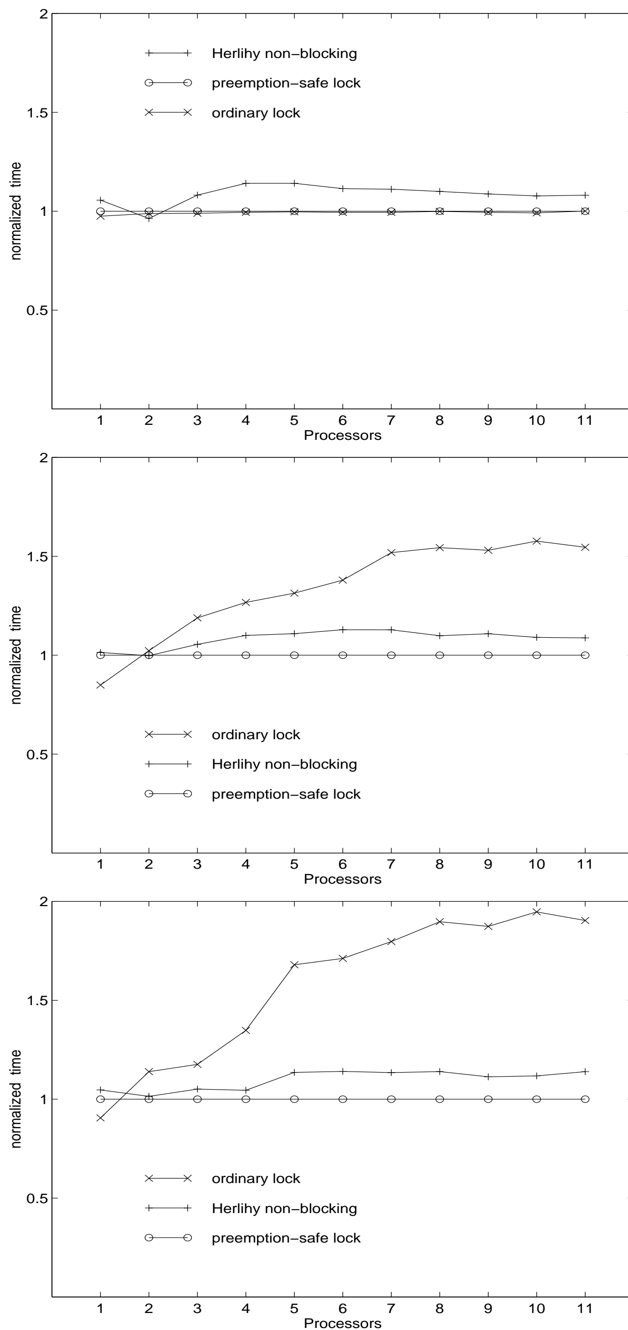

Figure 3: Normalized execution time for one million insert/delete_min pairs on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

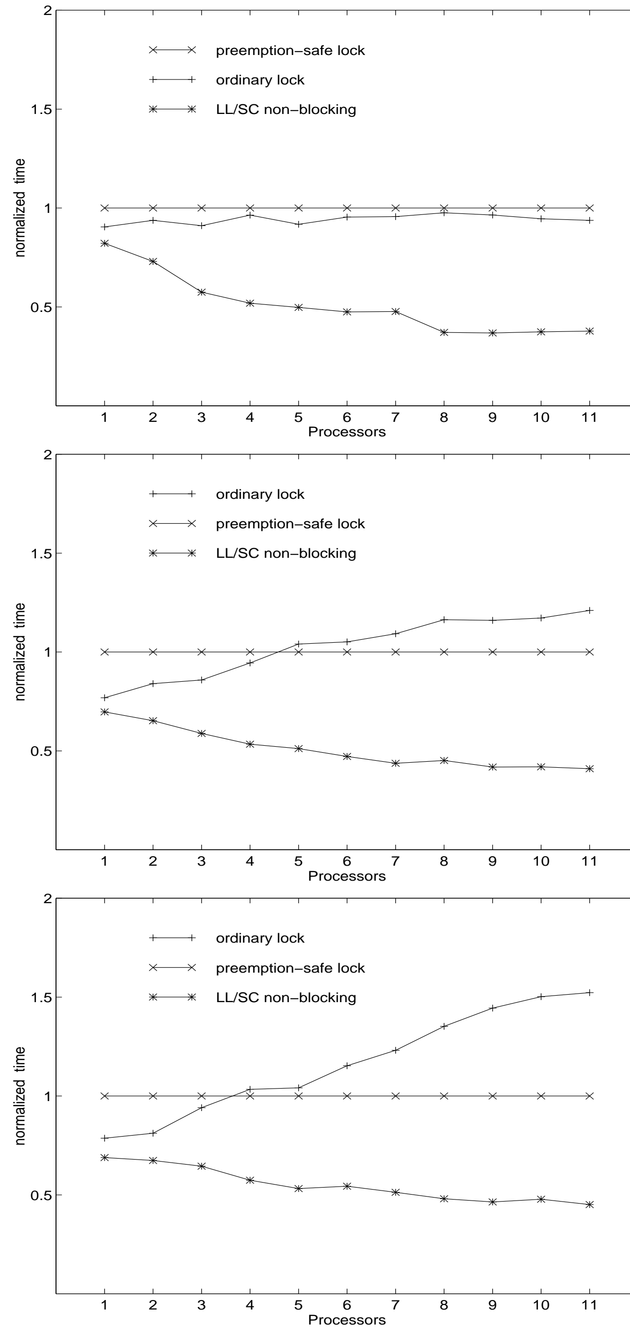

Figure 4: Normalized execution time for one million atomic increments on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

The results are similar to those observed for queues and stacks, but are even more pronounced. The non-blocking implementation outperforms the preemption-safe lock-based counter by more than 55% on 11 processors with levels of multiprogramming. The performance of a `fetch_and_add` atomic primitive would be even better [16].

4.5 Quicksort Application

We performed experiments on two versions of a parallel quicksort application, one that uses a link-based queue, and another that uses a link-based stack for distributing items to be sorted among the cooperating processes. We used three implementations for each of the queue and the stack: the usual single lock implementation using ordinary and preemption-safe locks, and the non-blocking implementations of Michael and Scott, and Treiber, respectively. In each execution, the processes cooperate in sorting an array of 500 pseudo-random numbers using quicksort for intervals of more than 20 elements, and insertion sort for smaller intervals.

Figure 5 shows the performance results for the three queue-based versions; figure 6 shows the performance results for the three stack-based versions. Execution times are normalized to those of the preemption-safe lock-based implementations. The absolute times in seconds for the preemption-safe lock-based implementation on one and 11 processors, with 1, 2, and 3 processes per processor, are 4.0 and 1.6, 7.9 and 2.3, and 11.6 and 3.3, respectively for a shared queue, and 3.4 and 1.5, 7.0 and 2.3, and 10.2 and 3.1, respectively for a shared stack.

The results confirm our observations from experiments on synthetic applications. Performance with ordinary locks degrades under multiprogramming, though not as severely as before, since more work is being done between atomic operations. Simple non-blocking implementations yield superior performance even on dedicated systems, making them the implementation of choice under any level of contention or multiprogramming.

4.6 Traveling Salesman Application

We performed experiments on a parallel implementation of a solution to the traveling salesman problem. The program uses a shared heap, stack, and counters. We used three implementations for each of the heap, stack, and counters: the usual single lock implementation using ordinary and preemption-safe locks, and the best respective non-blocking implementations (Herlihy–optimized, Treiber, and `load_linked/store_conditional`). In each execution, the processes cooperate to find the shortest tour in a 17-city graph. The processes use the priority queue heap to share information about the most promising tours, and the stack to keep track of the tours that are yet to be computed. We ran experiments with each of the three implementations of the data structures. In addition, we ran experiments with a "hybrid" program that uses the version of each data structure that ran the fastest for the synthetic applications: non-blocking stacks and counters, and a preemption-safe priority queue.

Figure 7 shows the performance results for the four different implementations. Execution times are normalized to those of the preemption-safe lock-based implementation. The absolute times in seconds for the preemption-safe lock-based implementation on one and 11 processors, with 1, 2, and 3 processes per processor, are 34.9 and 14.3, 71.7 and 15.7, and 108.0 and 18.5, respectively. As expected, the implementation based on ordinary locks suffer under multiprogramming. The hybrid implementation yields the best performance.

5 Conclusions

For atomic update of a shared data structure, the programmer may ensure consistency using a (1) single lock, (2) multiple locks, (3) a general-purpose non-blocking technique, or (4) a special-purpose (data-structure-specific) non-blocking algorithm. The locks in (1) and (2) may or may not be preemption-safe.

Options (1) and (3) are easy to generate, given code for a sequential version of the data structure, but options (2) and (4) must be developed individually for each different data structure. Good data-structure-specific multi-lock and non-blocking algorithms are sufficiently tricky to devise that each has tended to constitute an individual publishable result.

Our experiments indicate that for simple data structures, special-purpose non-blocking atomic update algorithms will outperform all alternatives, not only on multiprogrammed systems, but on dedicated machines as well. Given the availability of a universal atomic hardware primitive (`compare_and_swap` or `load_linked/store_conditional`), there seems to be no reason to use any other version of a stack, a queue, or a small, fixed-sized object like a counter.

For less trivial data structures, however, or for machines without appropriate primitives, preemption-safe algorithms are clearly important. Preemption-safe locks impose a modest performance penalty on dedicated systems (up to 20% in our traveling salesman results; less than 5% in the other experiments), but provide dramatic savings on time-sliced systems. Further research in general-purpose non-blocking techniques is clearly warranted, though we doubt that the results will ever match the performance of preemption-safe locks.

For the designers of future systems, we recommend (1) that hardware always include a universal atomic primitive, and (2) that kernel interfaces provide a mechanism for preemption-safe locking. For small-scale machines, the Synunix interface appears to work well [7]. For larger machines, a more elaborate interface may be appropriate [12].

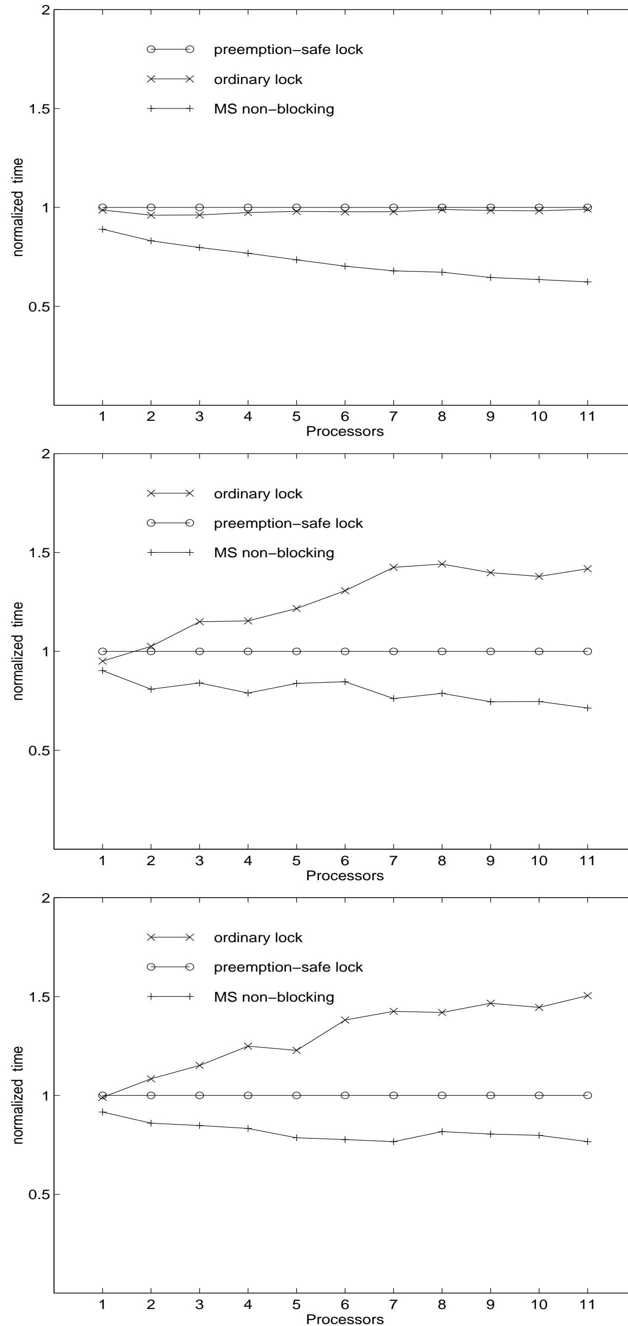

Figure 5: Normalized execution time for quicksort of 500 items using a shared queue on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

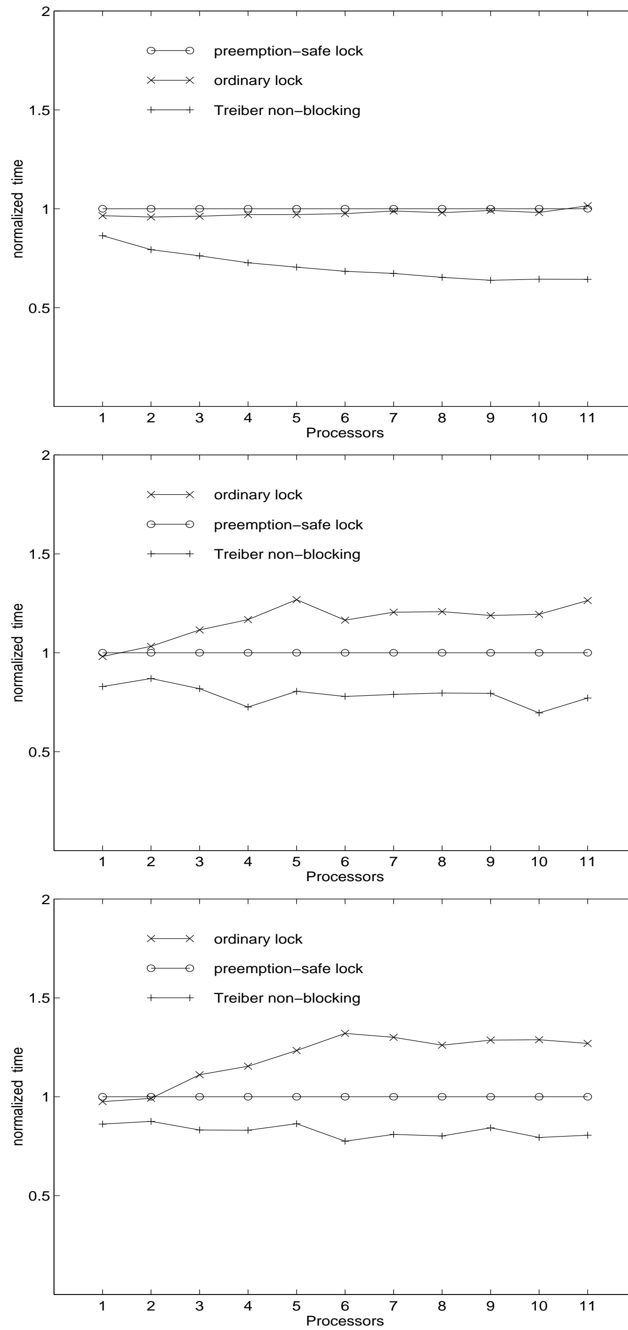

Figure 6: Normalized execution time for quicksort of 500 items using a shared stack on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

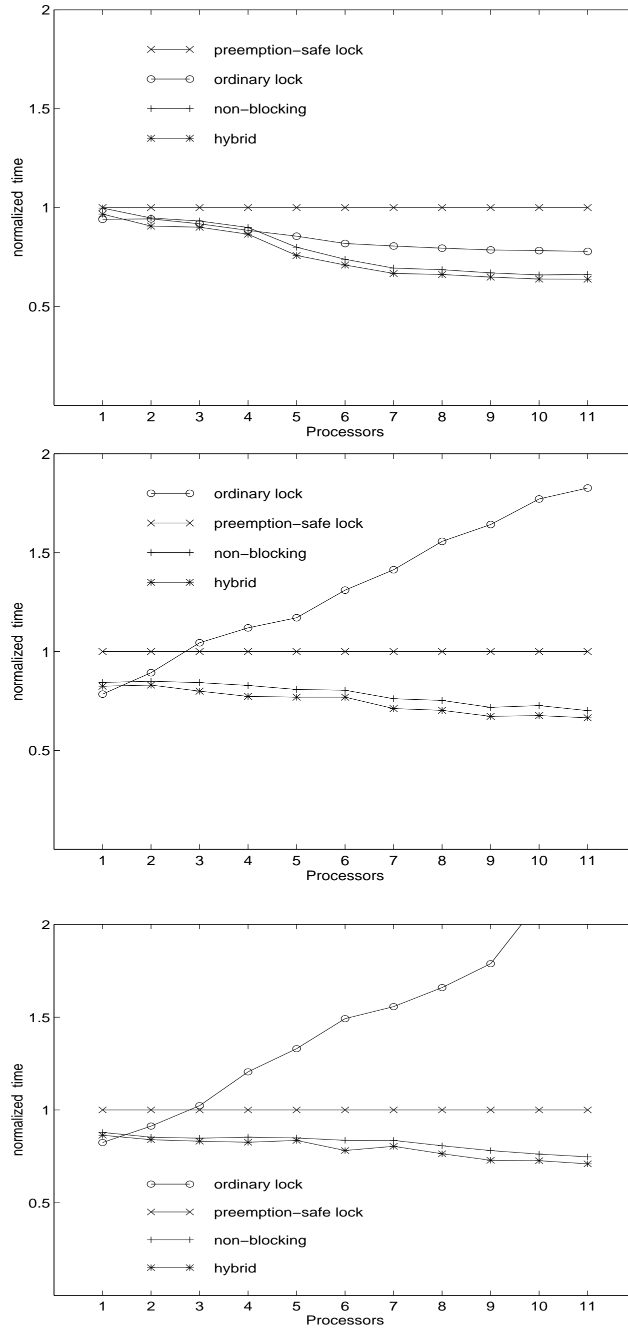

Figure 7: Normalized execution time for a 17-city traveling salesman problem using a shared priority queue, stack and counters on a multiprogrammed system, with multiprogramming levels of 1 (top), 2 (middle), and 3 (bottom).

References

[1] J. Alemany and E. W. Felten. Performance Issues in Non-blocking Synchronization on Shared-Memory Multiprocessors. In *Proceedings of the Eleventh ACM Symposium on Principles of Distributed Computing*, Vancouver, BC, Canada, August 1992.

[2] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[3] R. J. Anderson and H. Woll. Wait-Free Parallel Algorithms for the Union-Find Problem. In *Proceedings of the Twenty-Third ACM Symposium on Theory of Computing*, pages 370–380, May 1991.

[4] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992. Originally presented at the *Thirteenth ACM Symposium on Operating Systems Principles*, October 1991.

[5] G. Barnes. A Method for Implementing Lock-Free Data Structures. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, Velen, Germany, June – July 1993.

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

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

[8] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Structures. In *Proceedings of the Second ACM Symposium on Principles and Practice of Parallel Programming*, pages 197–206, Seattle, WA, March 1990.

[9] M. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[10] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[11] E. H. Jensen, G. W. Hagensen, and J. M. Broughton. A New Approach to Exclusive Data Access in Shared Memory Multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, November 1987.

[12] L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott. Scheduler-Conscious Synchronization. TR 550, Computer Science Department, University of Rochester, December 1994. Submitted for publication.

[13] O. Krieger, M. Stumm, and R. Unrau. A Fair Fast Scalable Reader-Writer Lock. In *Proceedings of the 1993 International Conference on Parallel Processing*, St. Charles, IL, August 1993.

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

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

[16] M. M. Michael and M. L. Scott. Implementation of General-Purpose Atomic Primitives for Distributed Shared-Memory Multiprocessors. In *Proceedings of the First International Symposium on High Performance Computer Architecture*, pages 222–231, Raleigh, NC, January 1995. Also available as TR 528, Computer Science Department, University of Rochester, July 1994.

[17] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing*, Philadelphia, PA, May 1996. Earlier version available as TR 600, Computer Science Department, University of Rochester, December 1995.

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

[19] S. Prakash, Y.-H. Lee, and T. Johnson. A Non-Blocking Algorithm for Shared Queues using Compare-and-Swap. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages II:68–75, St. Charles, IL, August 1991.

[20] S. Prakash, Y. H. Lee, and T. Johnson. Non-Blocking Algorithms for Concurrent Data Structures. Technical Report 91-002, University of Florida, 1991.

[21] S. Prakash, Y. H. Lee, and T. Johnson. A Nonblocking Algorithm for Shared Queues Using Compare-and-Swap. *IEEE Transactions on Computers*, 43(5):548–559, May 1994.

[22] R. K. Treiber. Systems Programming: Coping with Parallelism. RJ 5118, IBM Almaden Research Center, April 1986.

[23] J. Turek, D. Shasha, and S. Prakash. Locking without Blocking: Making Lock Based Concurrent Data Structure Algorithms Nonblocking. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 212–222, 1992.

[24] J. D. Valois. Implementing Lock-Free Queues. In *Seventh International Conference on Parallel and Distributed Computing Systems*, Las Vegas, NV, October 1994.

[25] J. D. Valois. Lock-Free Data Structures. Ph. D. dissertation, Rensselaer Polytechnic Institute, May 1995.

[26] J. D. Valois. Lock-free Linked Lists using Compare-and-swap. In *Proceedings of the Fourteenth ACM Symposium on Principles of Distributed Computing*, Ottawa, Ontario, Canada, August 1995.

[27] R. W. Wisniewski, L. Kontothanassis, and M. L. Scott. Scalable Spin Locks for Multiprogrammed Systems. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 583–589, Cancun, Mexico, April 1994. Earlier but expanded version available as TR 454, Computer Science Department, University of Rochester, April 1993.

[28] R. W. Wisniewski, L. I. Kontothanassis, and M. L. Scott. High Performance Synchronization Algorithms for Multiprogrammed Multiprocessors. In *Proceedings of the Fifth ACM Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[29] J. Zahorjan, E. D. Lazowska, and D. L. Eager. The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):180–198, April 1991.