Relative Performance of Preemption-Safe Locking and Non-Blocking Synchronization on Multiprogrammed Shared Memory Multiprocessors

Maged M. Michael   Michael L. Scott

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

Abstract

Most multiprocessors are multiprogrammed to achieve acceptable response time. 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: (1) preemption-safe locking *and (2)* non-blocking *(lock-free)* algorithms*. Preemption-safe locking requires kernel support. Non-blocking algorithms generally require a universal atomic primitive, and are widely regarded as inefficient.*

We present a comparison of the two alternative strategies, focusing on four simple but important concurrent data structures—stacks, FIFO queues, priority queues and counters—in micro-benchmarks and real applications on a 12-processor SGI Challenge multiprocessor. Our results indicate that data-structure-specific non-blocking algorithms, which exist for stacks, FIFO queues and counters, can work extremely well: not only do they outperform preemption-safe lock-based algorithms on multiprogrammed machines, they also outperform ordinary locks on dedicated machines. At the same time, since general-purpose non-blocking techniques do not yet appear to be practical, preemption-safe locks remain the preferred alternative for complex data structures: they outperform conventional locks by significant margins on multiprogrammed systems.

1 Introduction

On shared memory multiprocessors, processes communicate using shared data structures, which they typically update atomically under the protection of mutual exclusion locks. At the same time, most multiprocessors are preemptively multiprogrammed, for the sake of response time and processor utilization. Unfortunately, preemption of a process holding a lock can degrade application performance dramatically [25]; any other process busy-waiting on the lock is unable to perform useful work until the preempted process is rescheduled and subsequently releases the lock.

*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).

Alternatives to time-slicing can avoid inopportune preemption, but have limited applicability and/or lower processor utilization. Coscheduling reduces utilization if applications have a variable amount of parallelism, or if processes cannot be evenly assigned to time-slices of the multiprocessor. Fixed-size hardware partitions lead to poor response time when the number of processes is larger than the number of processors. Variable-size hardware partitions require that applications be able to adjust their number of processes as new applications join the system.

For time-sliced systems, researchers have proposed two principal strategies to avoid inopportune preemption: *preemption safe locking* and *non-blocking algorithms*. Most preemption-safe 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.

An implementation of a data structure is *non-blocking* (also known as *lock-free*) if it guarantees that at least one of the processes trying to update a data structure concurrently will succeed within a bounded amount of time, regardless of the state of other processes. Non-blocking algorithms do not require any communication with the kernel and by definition they have no critical sections in which preemption can occur. While mutual exclusion locks can be implemented using comparatively weak atomic primitives (e.g. `test_and_set`, `fetch_and_increment`, or `fetch_and_store`), non-blocking algorithms generally require a universal [8] primitive such as `compare_and_swap` (CAS) or the pair `load_linked` and `store_conditional` (LL/SC).

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 micro-benchmarks and real applications, on a 12-processor SGI Challenge multiprocessor. We discuss related work on preemption-safe and non-blocking techniques in sections 2 and 3, respectively. We then 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 summarize our conclusions and recommendations in section 5.

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 [6] employs an avoidance technique: a process may request that the kernel not preempt it because it is holding a lock, and the kernel will honor the request up to a pre-defined time limit. The *first-class threads* of Marsh et al.'s Psyche system [12] require the kernel to warn an application process a fixed amount of time in advance of preemption. If a process verifies that it has not been warned of preemption 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. Otherwise, it can yield the processor voluntarily.

Black's work on Mach [5] employs 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. [3] 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. [10] present scheduler-conscious versions of the MCS queue lock [14] and other scalable locks. 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

Motivated in part by the problem of preemption in critical sections, researchers have developed non-blocking implementations of several widely-used data structures, together with general methodologies for developing such implementations systematically for arbitrary data structures.

Herlihy [9] presented a general methodology for transforming sequential implementations of data structures to concurrent non-blocking implementations using CAS or LL/SC. 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. Alemany and Felten [1] and LaMarca [11] proposed 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 [4] presented a similar general methodology in which processes record and timestamp their modifications to the shared object, and cooperate whenever conflicts arise. Shavit and Touitou [20] presented *software transactional memory*, which implements a k-word CAS using LL/SC. Turek et al. [22] and Prakash et al. [18] presented methodologies for transforming multiple lock concurrent objects into lock-free concurrent objects. Unfortunately, the performance of non-blocking algorithms resulting from general methodologies is acknowledged to be significantly inferior to that of the corresponding lock-based implementations [9, 11, 20].

Prakash et al. [19], Valois [30], and Michael and Scott [17] proposed non-blocking implementations of concurrent link-based queues. Treiber [21] proposed a non-blocking implementation of concurrent link-based stacks. Valois [24] proposed a non-blocking implementation of linked lists. Anderson and Woll [2] 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 CAS or LL/SC.

Performance results were reported for only a few of these algorithms [17, 19, 30]. 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.

Massalin and Pu [13] presented non-blocking algorithms for array-based stacks, FIFO queues, and linked lists, using a `double_compare_and_swap` (DCAS) primitive that operates on two arbitrary memory locations simultaneously. Unfortunately, this primitive is available only on the Motorola 68020 processor and its direct successors. Herlihy and Wing [7] proposed a non-blocking array-based queue algorithm that requires infinite arrays. Valois [23] presented a non-blocking array-based queue algorithm that requires either a non-aligned CAS (not supported on any architecture) or a Motorola-like DCAS. 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.

As representatives of the best available non-blocking algo-

rithms 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. [19], the non-blocking link-based stack of Treiber [21], an optimized version of a stack resulting from applying Herlihy's methodology [9], a skew heap implementation due to Herlihy using his general methodology with optimized copying [9], and a LL/SC implementation of counters.

4 Experimental Results

We use an SGI Challenge multiprocessor with twelve 100 MHz MIPS R4400 processors 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 micro-benchmarks 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.[1] More detailed results are available in a technical report version of this paper [16].

Our results were obtained on a dedicated machine with application processes "pinned" to 11 processors and with one processor dedicated to running 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. The time quantum is 10 ms.

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 backoff was chosen to yield good overall performance for all implementations, and not to exceed 30 μs.

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 left graph in each figure) therefore represents a dedicated machine; a multiprogramming level of 3 (the right 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 (**single ordinary lock** and **single preemption-safe lock**); a two-lock implementation due to Michael and Scott [17], again using both ordinary and preemption-safe locks (**two ordinary locks** and **two preemption-safe locks**); and non-blocking implementations due to Michael and Scott [17] (**MS non-blocking**) and Prakash et al. [19] (**PLJ non-blocking**).

The two-lock implementation of Michael and Scott implements the queue as a singly-linked list with *Head* and *Tail* pointers and uses two locks, one lock to protect each of *Head* and *Tail*. An extra dummy node always pointed to by *Head* is used to ensure that the linked list is never empty. Enqueue operations never have to access the head pointer, and dequeue operations never have to access the tail pointer, so enqueues and dequeues can always proceed concurrently, with no possibility of deadlock. The non-blocking algorithm of Michael and Scott uses a similar structure except for the locks. Instead, the algorithm uses CAS to add and remove nodes at the tail and the head of the linked list, respectively.

The non-blocking algorithm of Prakash et al. uses a singly-linked list to represent the queue with *Head* and *Tail* pointers. It uses CAS to enqueue and dequeue nodes at the tail and the head of the list, respectively. A process performing an enqueue or a dequeue operation first takes a snapshot of the data structure and determines if there is another operation in progress. If so it tries to complete the ongoing operation and then takes another snapshot of the data structure. Otherwise it tries to complete its own operation. The process keeps trying until it completes its operation.

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 implementations. 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.

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 ($\pm$ 10% randomization) spinning in an empty loop after performing every enqueue or dequeue operation (for a total of 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. The two-lock implementation outperforms the single-lock in the case of high contention because it allows more concurrency, 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 better performance; they provide added concurrency without being vulnerable to interference from multiprogramming. Overall, the non-blocking implementation of Michael and Scott yields the best performance. It outperforms the single-lock preemption-safe implementation by more than 40% on 11 processors with various levels of multiprogramming, since it needs to access a fewer memory locations and allows more concurrency than the other implementations. In the case of no contention, it is essentially tied with the single ordinary lock.

4.2 Stacks

Figure 2 shows performance results for four stack implementations: the usual single lock implementation using ordinary and

[1] C code for all the micro-benchmarks and the real applications are available from `ftp://ftp.cs.rochester.edu/pub/packages/sched_conscious_synch/multiprogramming`.

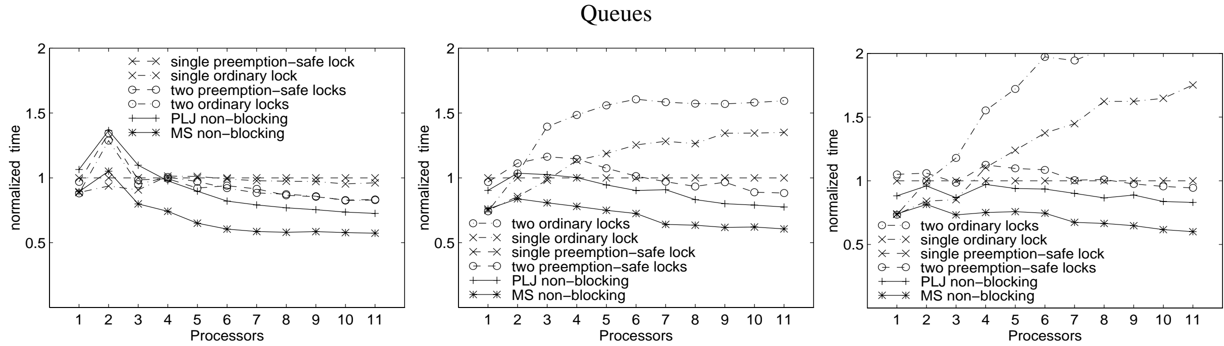

Figure 1. Normalized execution time for one million enqueue/dequeue pairs on a multiprogrammed system, with multiprogramming levels of 1 (left), 2 (middle), and 3 (right).

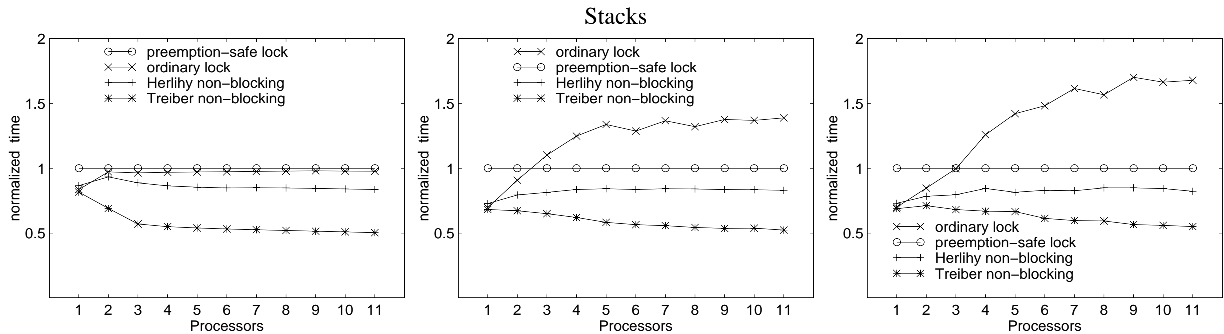

Figure 2. Normalized execution time for one million push/pop pairs on a multiprogrammed system, with multiprogramming levels of 1 (left), 2 (middle), and 3 (right).

preemption-safe locks, a non-blocking implementation due to Treiber [21], and an optimized non-blocking implementation based on Herlihy's general methodology [9].

Treiber's non-blocking algorithm represents the stack as a singly-linked list with a *Top* pointer. It uses CAS to modify the value of *Top* atomically. The optimized implementation based on Herlihy's methodology also uses a singly-linked list to represent the stack with a *Top* pointer. However, every process has its own copy of *Top* and an operation is successfully completed only when the process uses LL/SC to swing a shared pointer to its copy of *Top*. The shared pointer can be considered as pointing to the latest version of the stack.

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 19.0 and 20.3, 40.8 and 20.7, and 60.2 and 21.6, respectively. As in the micro-benchmark for queues, each process executes 1,000,000/p push/pop pairs on an initially empty stack, with a 6 μs average 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. This is mainly due to the fact that a push or a pop in Treiber's algorithm typically needs to access only two cache lines in the data structure, while a lock-based implementation has the overhead of accessing lock variables as well. Accordingly, Treiber's implementation yields the best performance even with no contention.

4.3 Heaps

Figure 3 shows performance results for three skew heap implementations: the usual single-lock implementation using ordinary and preemption-safe locks, and an optimized non-blocking implementation due to Herlihy [9]. The latter uses a binary tree to represent the heap with a *Root* pointer. Every process has its own copy of *Root*. A process performing a heap operation copies the nodes it intends to modify to local free nodes and finally tries to swing a global shared pointer to its copy of *Root* using LL/SC. If it succeeds, the local copies of the copied nodes become part of the global structure and the copied nodes can be recycled for use in future operations.

Execution time in the graphs 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 average 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,

Heaps

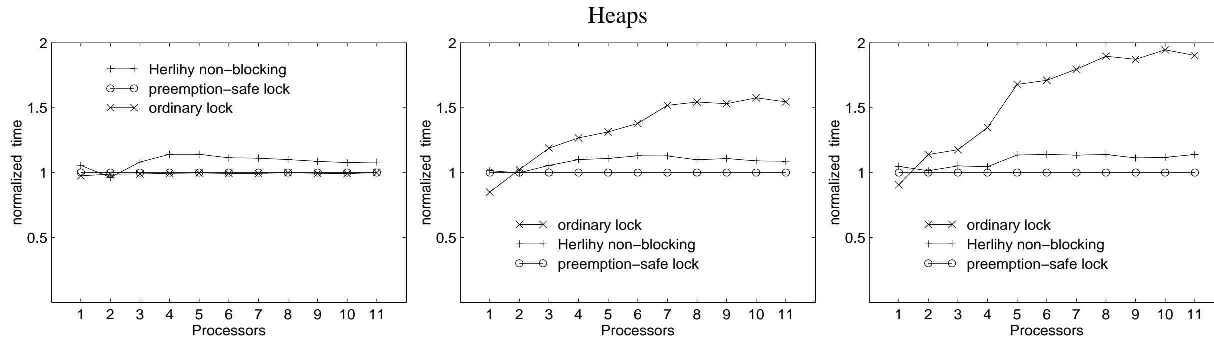

Figure 3. Normalized execution time for one million insert/delete_min pairs on a multiprogrammed system, with multiprogramming levels of 1 (left), 2 (middle), and 3 (right).

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, with and without contention. 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 much worse.

4.4 Counters

Figure 4 shows performance results for three implementations of counters: the usual single lock implementation using ordinary and preemption-safe locks, and the standard implementations using LL/SC. Execution time in the graphs 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 average delay between successive operations.

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 [15].

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 500,000 pseudo-random numbers using quicksort for intervals of more than 20 elements, and insertion sort for smaller intervals.

Figures 5 and 6 show the performance results for the three queue-based versions and three stack-based versions, respectively. 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 micro-benchmarks. 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 LL/SC). 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 micro-benchmarks: 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. Confirming our results with micro-benchmarks, the implementation based on ordinary locks suffers under multiprogramming. The hybrid implementation yields the best performance, since it uses the best implementation of each of the data structures.

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.

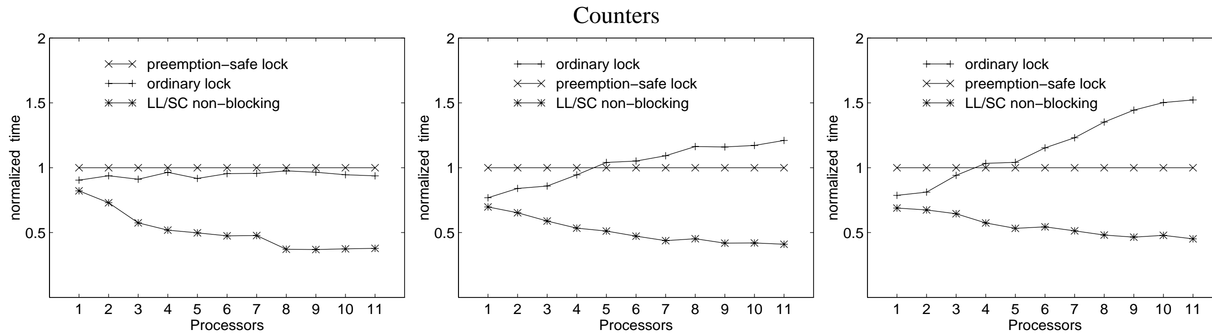

Figure 4. Normalized execution time for one million atomic increments on a multiprogrammed system, with multiprogramming levels of 1 (left), 2 (middle), and 3 (right).

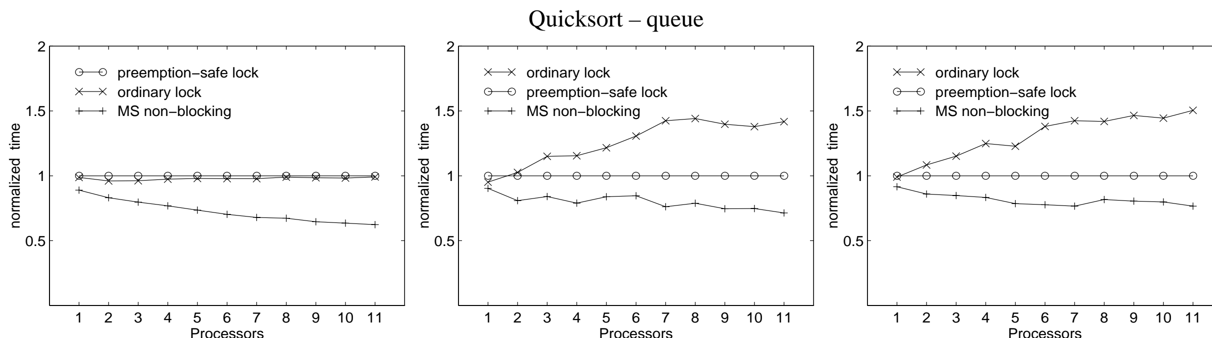

Figure 5. Normalized execution time for quicksort of 500,000 items using a shared queue on a multiprogrammed system, with multiprogramming levels of 1 (left), 2 (middle), and 3 (right).

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 (CAS or LL/SC), there seems to be no reason to use any other version of a link-based stack, a link-based queue, or a small, fixed-sized object like a counter.

For more complex data structures, however, or for machines without universal atomic primitives, preemption-safe locks are clearly important. Preemption-safe locks impose a modest performance penalty on dedicated systems, 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 [10].

References

[1] J. Alemany and E. W. Felten. Performance Issues in Non-blocking Synchronization on Shared-Memory Multiprocessors. In *Proc. of the 11th ACM Symposium on Principles of Distributed Computing*, pp. 125–134, Aug. 1992.

[2] R. J. Anderson and H. Woll. Wait-Free Parallel Algorithms for the Union-Find Problem. In *Proc. of the 23rd ACM Symposium on Theory of Computing*, pp. 370–380, May 1991.

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

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

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

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

[7] M. P. Herlihy and J. M. Wing. Axions for Concurrent Objects. In *Proc. of the 14th ACM Symposium on Principles of Programming Languages*, pp. 13–26, Jan. 1987.

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

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

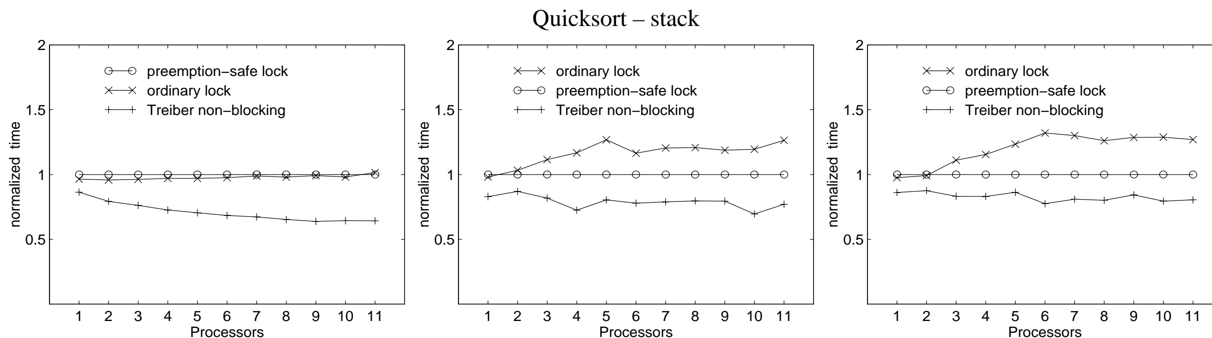

Figure 6. Normalized execution time for quicksort of 500,000 items using a shared stack on a multiprogrammed system, with multiprogramming levels of 1 (left), 2 (middle), and 3 (right).

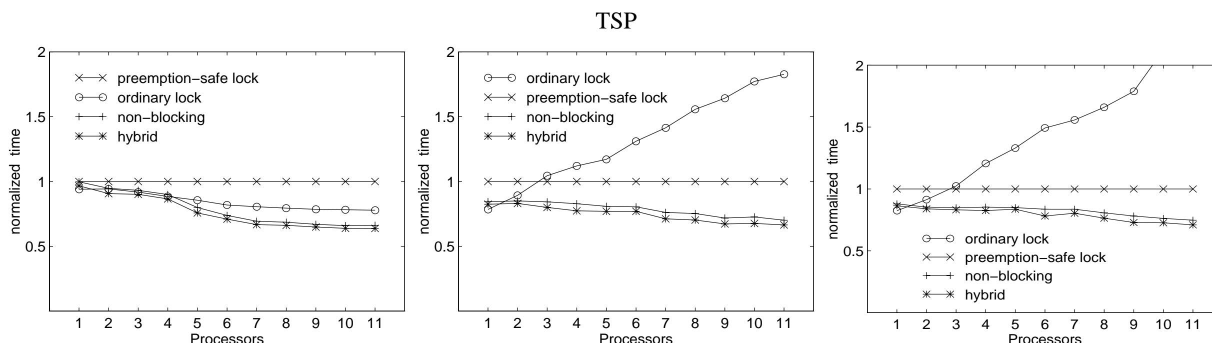

Figure 7. Normalized execution time for a 17-city traveling salesman problem using a shared priority queue, stack and counters on a multiprogrammed system, with multiprogramming levels of 1 (left), 2 (middle), and 3 (right).

[10] L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott. Scheduler-Conscious Synchronization. *ACM Trans. on Computer Systems*, 15(1), Feb. 1997.

[11] A. LaMarca. A Performance Evaluation of Lock-free Synchronization Protocols. In *Proc. of the 13th ACM Symposium on Principles of Distributed Computing*, Aug. 1994.

[12] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-Class User-Level Threads. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, Oct. 1991.

[13] H. Massalin and C. Pu. A Lock-Free Multiprocessor OS Kernel. Tech. Report CUCS-005-91, Computer Science Department, Columbia University, 1991.

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

[15] M. M. Michael and M. L. Scott. Implementation of Atomic Primitives on Distributed Shared-Memory Multiprocessors. In *Proc. of the First International Symposium on High Performance Computer Architecture*, pp. 222–231, Jan. 1995.

[16] M. M. Michael and M. L. Scott. Concurrent Update on Multiprogrammed Shared Memory Multiprocessors. Tech. Report 614, Computer Science Dept., Univ. of Rochester, April 1996.

[17] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proc. of the Fifteenth ACM Symposium on Principles of Distributed Computing*, pp. 267–275, May 1996.

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

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

[20] N. Shavit and D. Touitou. Software Transactional Memory. In *Proc. of the 14th ACM Symposium on Principles of Distributed Computing*, Aug. 1995.

[21] R. K. Treiber. Systems Programming: Coping with Parallelism. In *RJ 5118, IBM Almaden Res. Center*, April 1986.

[22] J. Turek, D. Shasha, and S. Prakash. Locking without Blocking: Making Lock Based Concurrent Data Structure Algorithms Nonblocking. In *Proc. of the 11th ACM Symposium on Principles of Database Systems*, pp. 212–222, 1992.

[23] J. D. Valois. Implementing Lock-Free Queues. In *Proc. of the Seventh International Conference on Parallel and Distributed Computing Systems*, Oct. 1994.

[24] J. D. Valois. Lock-free Linked Lists using Compare-and-swap. In *Proc. of the 14th ACM Symposium on Principles of Distributed Computing*, Aug. 1995.

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