IPPS '94

Scalable Spin Locks for Multiprogrammed Systems

Robert W. Wisniewski, Leonidas Kontothanassis, and Michael L. Scott
bob, kthanasi, and scott@cs.rochester.edu
Department of Computer Science
University of Rochester
Rochester, NY 14627-0226

Abstract

Synchronization primitives for large shared-memory multiprocessors need to minimize latency and contention. Software queue-based locks address these goals, but suffer if a process near the end of the queue waits for a preempted process ahead of it. To solve this problem, we present two queue-based locks that recover from in-queue preemption. The simpler, faster lock employs an extended kernel interface that shares information in both directions across the user-kernel boundary. Results from experiments with real and synthetic applications on SGI and KSR multiprocessors demonstrate that high-performance software spin locks are compatible with multiprogramming on both large-scale and bus-based machines.

1 Introduction

Many parallel applications are written using mutual exclusion locks. When processors are uniprogrammed, or when the expected waiting time for a lock is less than twice the context switch time, spinning in anticipation of acquiring a lock is more efficient than rescheduling. As a result, busy-wait (spinning) mutual exclusion locks are widely used.

Unfortunately, spin locks suffer from two serious problems: 1. Both the common `test_and_set` lock and the read-polling `test_and_test_and_set` lock degrade badly as the number of competing processors increases; 2. In multiprogrammed systems, a process that is preempted during its critical section can delay every other process that needs to acquire the lock.

This work was supported in part by National Science Foundation Institutional Infrastructure grant no. CDA-8822724, DARPA grant no. MDA972-92-J-1012, and 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 Technical program, ARPA Order no. 8930).

To address the first of these problems, several researchers have devised queue-based locks in which every process spins on a different, local location, essentially eliminating contention [1, 6, 12]. To address the second problem, several operating systems have incorporated schemes in which applications communicate with the kernel scheduler to prevent [5] or recover from [2] preemption in a critical section, or to avoid entering a critical section when preemption is imminent [11]. What has not been clear from previous work is how to solve both problems at once.

The various mechanisms for dealing with preemption can all be applied in a straightforward manner to programs using `(test_and_)test_and_set` locks, resulting in good performance, at least on small machines. Their application to programs using queue-based locks is much less straightforward. None of [2], [5], or [11] discusses queue-based locks, and [12] explicitly recommends non-queue-based locks for multiprogrammed environments. Our contribution in this paper is to demonstrate that simple extensions to the interface of a preemptive scheduler can be combined with an appropriately-designed queue-based lock to provide excellent performance on systems that are *both* very large (with a high degree of competition for locks) *and* multiprogrammed.

2 Related Work

When two processes spin on the same location, coherence operations or remote memory references (depending on machine type) create substantial amounts of contention for memory and for the processor-memory interconnect. The key to good performance is therefore to minimize active sharing.

The queue-based spin locks of Anderson [1] and of Graunke and Thakkar [6] minimize active sharing on coherently-cached machines by arranging for every waiting processor to spin on a different element of an array. Each element of the array lies in a separate cache line, which migrates to the spinning pro-

cessor. The queue-based spin lock of Mellor-Crummey and Scott [12] represents its queue with a distributed linked list instead of an array. Each waiting processor uses a `fetch_and_store` operation to obtain the address of the list element (if any) associated with the previous holder of the lock. It then modifies that list element to contain a pointer to its *own* element, on which it then spins. Because it spins on a location of its own choosing, a process can arrange for that location to lie in local memory even on machines without coherent caches.

The efficiency of synchronization primitives is also dependent on the scheduling discipline used by the operating system. A growing body of evidence [4, 10, 15, 17] suggests that throughput is maximized by a processor-partitioned environment in which each application runs exactly one process per processor. In such an environment, the possibility arises that the lock owning process will be preempted and other running processes will busy-wait for the release of the lock held by the preempted process.

To address the possibility of preemption, several researchers have invented forms of synchronization-sensitive scheduling. The Scheduler Activation proposal of Anderson et al. [2] allows a parallel application to recover from untimely preemption. When a processor is taken away from an application, another processor in the same application is given a software interrupt, informing it of the preemption. The second processor can then perform a context switch to the preempted process if desired, e.g. to push it through its critical section. In a similar vein, Black's work on Mach [3] allows a process to suggest to the scheduler that it be de-scheduled in favor of some specific other process, eg. the holder of a desired lock.

Rather than recover from untimely preemption, the Symunix system of Edler et al. [5] and the Psyche system of Marsh et al. [11] provide mechanisms to avoid or prevent it. The Symunix scheduler allows a process to request that it not be preempted during a critical section, and will honor that request, within reason. The Psyche scheduler provides a "two-minute warning" that allows a process to estimate whether it has enough time remaining in its quantum to complete a critical section. If time is insufficient, the process can yield its processor voluntarily, rather than start something that it may not be able to finish.

Applications that use locks to protect both long and short critical sections can use on-line adaptive algorithms to guess whether it is better to spin or reschedule in a given situation [8]. The possibility of preemption, however, introduces very high variance in the apparent length of critical sections, and makes past behavior an unreliable predictor of whether to spin or block.

The approaches described above are adequate for `test_and_set` and `test_and_test_and_set` locks, where only preemption inside the critical section needs to be addressed. For queue-based locks, however, it is also crucial to consider the possibility of preemption while waiting to acquire the lock. FIFO ordering prevents processes down the queue from acquiring the lock if any of their predecessors are preempted even though they may not be in the critical section. DASH [9] uses a timeout mechanism to detect preemption and then chooses a different successor. In this paper, we present a software version of the DASH approach using the Symunix kernel interface and the spinlock of Mellor-Crummey and Scott. We also extend the kernel-user interface and design a simpler more efficient queue-based lock that also guarantees no process waits for a process that is not running.

3 Algorithms

In this section we briefly describe one lock and present the code for another. Both are extensions of the list-based queuing lock of Mellor-Crummey and Scott. Both employ the Symunix mechanism to prevent preemption in a critical region. The first uses a handshaking technique to avoid giving a lock to a process that is not running. The second obtains simpler and faster code by using an extended kernel interface. The interface consists of user-readable flags specifying whether a process is running and whether it desires the no-preemption mode.

As originally proposed, the Symunix interface includes a pair of flags for every process. One flag is set by the user and inspected by the kernel. It indicates that the process desires not to be preempted. The kernel honors this request whenever possible, deducting the time it adds to the end of the current quantum from the beginning of the next. The second flag is set by the kernel and inspected by the user. It indicates that the kernel wanted to preempt the user, but honored a request not to do so. Upon exiting a critical section, the user should clear the first flag and voluntarily yield the processor if the second flag is set. These conventions suffice to avoid preemption during a critical section.

To avoid giving the lock to a preempted process in the queue without extending the Symunix kernel interface, we can employ a handshaking protocol. A process releases a lock by notifying its successor process in the queue. If the successor does not promptly

acknowledge the notification by setting a flag in the releaser's queue node, the releaser assumes that the successor is blocked. It rescinds the successor's notification and proceeds to the following process. To avoid a timing window, both the releaser and the successor access the successor's notification flag with atomic `fetch_and_store` instructions. If the successor sees its notification just before the releaser attempts to rescind it, the releaser can tell what happened by inspecting the return value from its `fetch_and_store`. In either case, the successor waits for a final `ACK` or `NACK` from the releaser before proceeding. Further details on this lock can be found in the technical report version of this paper [16].

To handle preemption without handshaking, we can extend the kernel interface. By accessing a multi-value state variable with a `compare_and_swap` instruction, the kernel can atomically change a process from `preemptable` to `preempted`. Similarly, the releaser of a lock can atomically change a process from `preemptable` to `unpreemptable`.

A lock that uses this mechanism appears in figure 1. To close a timing window, we actually need four values of the process state variable. Two of these distinguish between a process that has made itself unpreemptable, and a process that has been made unpreemptable by its predecessor in the lock queue.

As before, the kernel maintains ultimate control of the processor by refusing to honor a request for non-preemption more than once per quantum. This implies that critical sections need to be shorter than the extension given to the process by the kernel.

```
type context_block = record
    state : (preempted, preemptable,
             unpreemptable_self, unpreemptable_other)
    warning : Boolean
type qnode = record
    self : ^context_block
    next : ^qnode
    next_done : Boolean
    status : (waiting, success, failure)
type lock = ^qnode
 private cb : ^context_block;
 procedure acquire_lock(L : ^lock, I : ^qnode)
    repeat
        I->next := nil
        I->self := cb
        cb->state := unpreemptable_self
        pred : ^qnode := fetch_and_store (L, I)
        if pred = nil
            return
        I->status := waiting
        pred->next := I
        (void) compare_and_swap (&cb->state,
                    unpreemptable_self, preemptable)
        repeat while I->status = waiting    // spin
    until I->status = success
```

```
procedure release_lock (L : ^lock, I : ^qnode)
    shadow : ^qnode := I
    candidate : ^qnode := shadow->next
    if candidate = nil
        if compare_and_swap (L, shadow, nil)
            return          // no one waiting for lock
    candidate := shadow->next
    loop
        while candidate = nil // spin; prob non-local
            candidate := shadow->next
        // order of following checks is important
        if compare_&_swap (&candidate->self->state,
               unpreemptable_self, unpreemptable_other)
            or compare_&_swap (&candidate->self->state,
                   preemptable, unpreemptable_other)
            candidate->status := success
            exit                            // leave loop
        // else candidate seems to be blocked
        shadow := candidate
        candidate := shadow->next
        shadow->status := failure
        if candidate = nil
            if compare_and_swap (L, shadow, nil)
                exit                        // leave loop
    cb->state := preemptable
    if cb->warning
        yield
```

Figure 1: Code for the Smart-Q Lock

A caveat with both of the locks just described is that they give up the FIFO ordering of the original list-based queuing lock. It is thus possible (though unlikely) that a series of adverse scheduling decisions could cause a process to starve. If this became a problem it would be possible to modify our algorithm to leave preempted processes in the queue, rather than removing them. The drawback of this approach is that preempted processes might be checked many times before actually acquiring the lock. Alternatively, taking our lead from Black's work, we could have the releaserer of a lock give its processor to its successor in the lock queue, if that successor were currently blocked. The drawback of this approach is that it entails additional context switches, and violates processor affinity [14].

4 Experiments and Results

We studied each lock implementation on three different programs. The first was a synthetic program that allowed us to explore the parameter space extensively. To verify results from this program, we also ran two real applications: the Cholesky program from the SPLASH suite [13] and a multiprocessor version of Quicksort. These applications were good candidates for testing because they synchronize only with locks. The rest of this section describes the experimental environment, the different types of locks we

implemented, and the performance results for different parameter values.

4.1 Methodology

We implemented eight different locks:

TAS_B – A standard `test_and_test_and_set` lock with exponential backoff that polls a lock's value and attempts to acquire it when it is free (on the SGI this is the native lock with backoff).

TAS_B-no_preempt – A `test_and_test_and_set` lock with exponential backoff. The critical section is marked non-preemptable using the Symunix approach.

Queued – A queued lock with local-only spinning.

Queued-no_preempt – An extension to the queued lock that prevents preemption while in the critical section.

Queued-Handshaking – Our extension to the queued lock that uses the Symunix kernel interface, and employs handshaking to ensure the lock is not transferred to a preempted process.

Smart-Q – Our better queued lock, with two-way sharing of information between the kernel for simpler code and lower overhead than the queued-handshaking lock.

Native – A lock employing machine-specific hardware (extra cache states on the KSR). This is the standard lock that would be used by a programmer familiar with the machine's capabilities.

Native-no_preempt – An extension to the native lock that prevents preemption while in the critical section.

We would expect the locks developed using the **no-preempt** capability to out-perform all other options on these two machines, and our experiments bear this out.

The machines we used were a Silicon Graphics Challenge, with 12 processors, and a Kendall Square Research KSR1, with 64 processors. The **Native** lock on the SGI is a `test_and_test_and_set` lock implemented using the `load_linked` and `store_conditional` instructions of the R4400 microprocessor. The KSR1 incorporates a cache line locking mechanism that provides the equivalent of queued locks in hardware. The queuing is based on physical proximity in a ring-based interconnection network, rather than on the chronological order of requests. While we would not expect software queuing to outperform (scheduler-sensitive use of) the native locks on the KSR, our goal was to come close enough to argue that special-purpose hardware is not crucial for scalable locks in multiprogrammed systems.

The queued locks require both `fetch_and_store` and `compare_and_swap`, primitives not available on the SGI or KSR. We implemented a software version of these atomic instructions using the native spinlocks. We also used the native spinlocks to implement `test_and_set`. This approach is acceptable as long as the time spent in the critical section protected by the higher-level lock is longer than the time spent simulating the execution of the atomic instruction. This was true for all the experiments we ran, so the results should be comparable to what would happen with hardware `fetch_and_`Φ instructions.

Each process in our synthetic program executes a simple loop containing a critical section. The total number of loop iterations is proportional to the number of executing processes. When using the synthetic program we were able to control four dimensions of the parameter space: multiprogramming level, number of processors, relative size of critical and non-critical sections, and quantum size. The first two parameters were found to have the greatest impact on performance; they are the focus of the next two sections.

In all the experiments an additional processor (beyond the reported number) is dedicated to running a user-level scheduler. The scheduler preempts a process by sending it a Unix signal. Each worker process catches this signal; the handler spins on a per-process flag, which the scheduler clears at the end of the "de-scheduled" interval. Implementation of our ideas in a kernel-level scheduler would be straightforward, but was not necessary for our experiments. (We also lacked the authorization to make kernel changes on the KSR.) To reduce the possibility of lock-step effects, we introduced a small amount of random variation in quantum lengths and the lengths of the synthetic program's critical and non-critical code sections.

The *multiprogramming level* reported in the experiments indicates the number of processes per processor. A multiprogramming level of 1.0 indicates one worker process for each available processor. A multiprogramming level of 2.0 indicates one additional (simulated) process on each available processor. Fractional multiprogramming levels indicate additional processes on some, but not all, of the processors.

4.2 Varying the Multiprogramming Level

Figures 2 and 3 show the running time for a fixed number of processors (11 on the SGI and 63 on the KSR) while varying the multiprogramming level.

On the SGI, the scheduling quantum is fixed at 20 ms, the critical section length at approximately 15 μs, and the non-critical section length at approximately 210 μs. Because the ratio of critical to non-critical

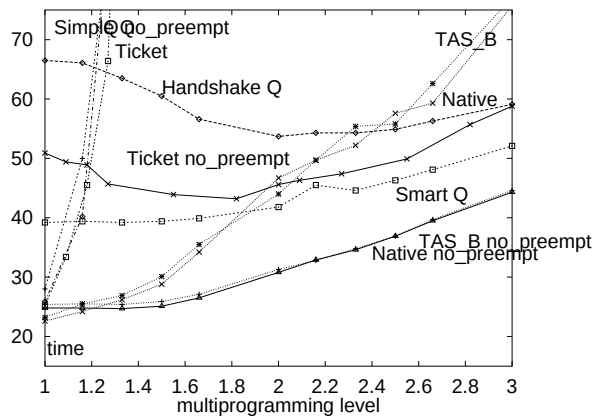

Figure 2: Varying Multiprogramming Levels on a 11-processor SGI Challenge.

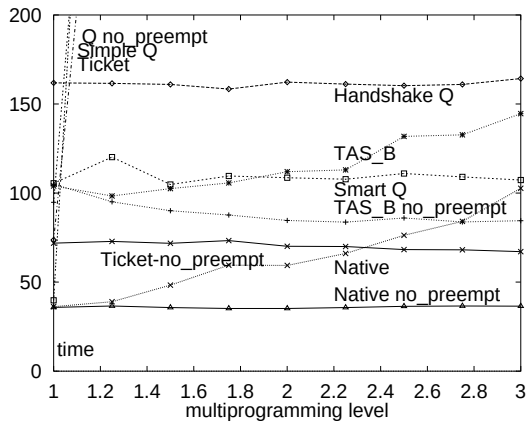

Figure 3: Varying Multiprogramming Levels on a 63-processor KSR1.

work is 1:14, while only 11 processors are running, the critical section does not constitute a bottleneck. Processes are able to use almost all of their cycles for "useful" work, rather than waiting for the lock, and completion time has a tendency to increase linearly with an increase in the multiprogramming level, as processes receive smaller and smaller fractions of their processors. The **Queued** and **Queued-no_preempt** locks, however, show much greater degradation, as processes begin to queue up behind a de-scheduled peer. Preventing preemption in the critical section helps a little, but not much: preemption of processes waiting in the queue is clearly the dominant problem. Much better behavior is obtained by preventing critical section preemption *and* ensuring that the lock is not given to a blocked process waiting in the queue: the **Queued-Handshaking** and **Smart-Q** locks perform far better than the other Queued locks, and also outperform the **Native** and **TAS_B** locks at multiprogramming levels of 2 and above. The **Native** and **TAS_B-no_preempt** locks display the best results, though they presumably generate more bus traffic than the **Smart-Q** lock, and might be expected to interfere more with "legitimate" memory traffic. (The synthetic program does not capture this effect; it operates entirely out of registers during its critical and non-critical sections.)

On the KSR, 63 processors were available, so a 1:14 ratio of critical to non-critical wok would lead to an inherently serial program. We therefore dropped the ratio far enough to eliminate serialization. Quantum length remained the same. The results show a somewhat different behavior from that on the SGI. The **Queued** and **Queued-no_preempt** locks suffer an enormous performance hit as the multiprogramming level increases. The **Queued-Handshaking** lock improves performance considerably since it eliminates both the critical section and queue preemption problems. Unfortunately, it requires a significant number of high-latency remote references, resulting in a high steady level of overhead. The **Smart-Q** lock lowers this level by a third, but is still somewhat slower than the **TAS_B-no_preempt** lock, and substantially slower than the native lock, from which all the others are constructed. The **TAS_B** and **Native** locks perform well when the multiprogramming level is low, but deteriorate as that level increases.

4.3 Varying the Number of Processors

Increasing the number of processor working in parallel can result in a significant amount of contention especially if the program needs to synchronize frequently. Previous work has shown that queue locks improve performance in such an environment, but as indicated by the graphs in figures 2 and 3 they can experience difficulties under multiprogramming. The graphs in figures 4 and 5 show the effect of increasing the number of processors on the different locks at a multiprogramming level of 2.0.

The synthetic program runs a total number of loop iterations proportional to the number of processors, so running time does not decrease as processors are added. Ideally, it would remain constant, but contention and scheduler interference can cause it to increase. With quantum size and critical to non-critical ratio fixed as before, results on the SGI again show the **Queued** and **Queued-no_preempt** locks performing very badly, as a result of untimely preemption. The performance of the **TAS_B** and **Native** locks also degrades with additional processors, either because of increased contention, or because of the increased likelihood of preemption in the critical sec-

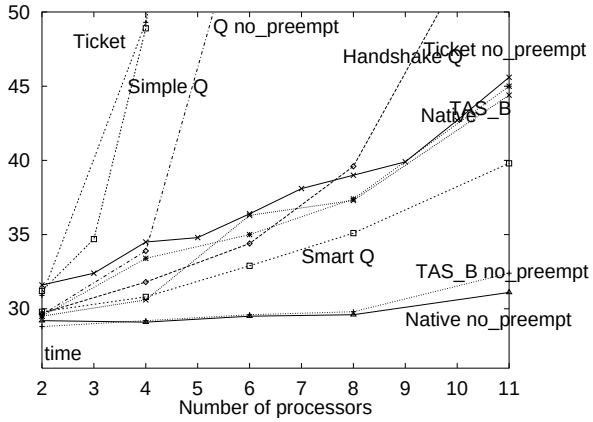

Figure 4: Varying the Number of Processors on the SGI Challenge with a multiprogramming level of 2 (one unrelated process per processor).

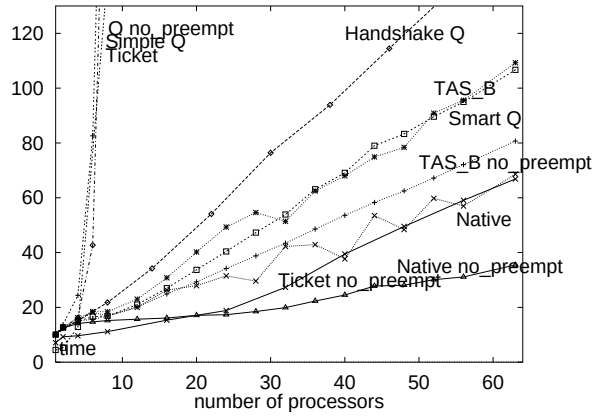

Figure 5: Varying the Number of Processors on the KSR1 with a multiprogramming level of 2 (one unrelated process per processor).

tion. The **Smart-Q** degrades more slowly, but also appears to experience contention. Contention does not seem to affect the **TAS_B-no_preempt** and **Native-no_preempt** locks until there are more than about 8 processors active.

The results on the KSR resemble those on the SGI. Contention continues to rise with increases in the number of processors, and the locks that avoid preemption out-perform their naive counterparts. The native lock, with our modification to avoid critical section preemption, is roughly twice as fast as the nearest competition, presumably because of the hardware queuing effect. Among the all-software locks, **TAS_B-no_preempt** performs best, but **TAS_B** and **Smart-Q** are close.

It should be noted that backoff constants for the **TAS_B** locks must be determined via trial and error. The best values differ from machine to machine, and even from program to program. The queued locks are in some sense more portable. As noted above, we would expect contention on both machines to become a serious problem sooner if the code in the critical and non-critical sections generated memory traffic. Because their traffic is deterministic, the queued locks should suffer less from this effect.

4.4 Results for Real Applications

To verify the results obtained from the synthetic program, and to investigate the effect of "legitimate" memory traffic, we measured the performance of a pair of real lock-based applications. Figure 6 shows the completion times of these applications, in seconds, when run with a multiprogramming level of 2.0 using 11 processors on the SGI and 63 processors on the KSR. As with the synthetic program, naive queuing of preemptable processes is disastrous. This time, however, with real computation going on, the **Smart-Q** lock is able to tie the performance of the **TAS_B-no_preempt** and **Native-no_preempt** locks on the SGI, and to out-perform the former in the Quicksort program on the KSR.

5 Conclusions

The ability to implement our locks testifies to the flexibility of `fetch_and_Φ` instructions. The ease with which such instructions can be implemented, and their utility in other areas (e.g. wait-free data structures [7]) makes them a very attractive alternative to special-purpose synchronization hardware. The na-

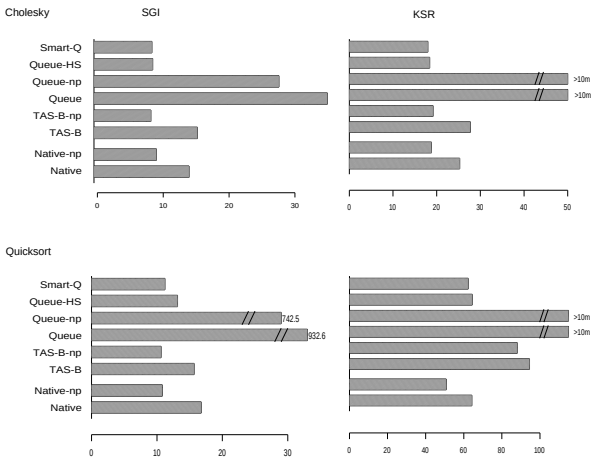

Figure 6: Completion time (in seconds) of real applications on a 11-processor SGI Challenge and a 63-processor KSR1 (multiprogramming level = 2).

tive locks of the KSR1, for example, are faster than the **Smart-Q** lock, but not by very much. Whether they are worth the effort of implementation probably depends on the extent to which they complicate the construction of the machine. For machines without queued locks in hardware (e.g., the BBN TC2000), our locks are likely to outperform all previous alternatives in the presence of multiprogramming.

This paper makes three primary contributions. First, it demonstrates the need for queue-based locks to be extended to environments with both high levels of contention and preemption due to multiprogramming. Second, it describes an algorithm based on the Symunix model that accomplishes this by preventing critical section preemption and by ensuring that a lock is not given to a preempted process in the queue (one of the referees points out that this same technique could be used to skip over processes that have chosen to block instead of spin, as described by Karlin et al.) Third, it shows that by sharing appropriate information between the scheduler and application processes, we can make the lock simpler and faster.

Our work suggests the possibility of using kernel-user sharing for additional purposes. We are interested, for example, in using it to help manage resources such as memory. We are also interested in studying the effect of scheduler information in systems where priorities are important, i.e., real-time applications.

Acknowledgements

We would like to thank Donna Bergmark and the Cornell Theory Center for the use of their KSR1.

References

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

[2] 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, Feb. 1992.

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

[4] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos. Multiprogramming on Multiprocessors. In *Proc. of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 590-597, Dec. 1991.

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

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

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

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

[9] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *Computer*, 25(3):63-79, Mar. 1992.

[10] S. T. Leutenegger and M. K. Vernon. Performance of Multiprogrammed Multiprocessor Scheduling Algorithms. In *Proc. of the 1990 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Boulder, CO, May 1990.

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

[12] 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, Feb. 1991.

[13] J. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *ACM SIGARCH Computer Architecture News*, 20(1):5-44, Mar. 1992.

[14] M. S. Squillante. Issues in Shared-Memory Multiprocessor Scheduling: A Performance Evaluation. Ph. D. dissertation, TR 90-10-04, Department of Computer Science and Engineering, University of Washington, Oct. 1990.

[15] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. In *Proc. of the Twelfth ACM Symposium on Operating Systems Principles*, pages 159-166, Litchfield Park, AZ, Dec. 1989.

[16] R. W. Wisniewski, L. Kontothanassis, and M. L. Scott. Scalable Spin Locks for Multiprogrammed Systems. TR 454, Computer Science Department, University of Rochester, Apr. 1993.

[17] J. Zahorjan and C. McCann. Processor Scheduling in Shared Memory Multiprocessors. In *Proc. of the 1990 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 214-225, Boulder, CO, May 1990.