Tech. Rep. 454

Scalable Spin Locks for Multiprogrammed Systems

Robert W. Wisniewski, Leonidas Kontothanassis, and Michael L. Scott*

Computer Science Department
University of Rochester
Rochester, NY 14627-0226

The University of Rochester
Computer Science Department
Rochester, New York 14627

September 1993

Abstract

Synchronization primitives for large scale multiprocessors need to provide low latency and low contention to achieve good performance. Queue-based locks (implemented in software with `fetch_and_Φ` instructions) can greatly reduce contention and improve overall performance by arranging for processors to spin only on local locations. Unfortunately, queued locks exhibit poor behavior in the presence of multiprogramming: a process near the end of the queue, in addition to waiting for any process that is preempted during its critical section, must also wait for any preempted processes ahead of it in the queue.

To solve this problem, we present two queue-based locks that recover from in-queue preemption. The first lock employs the kernel interface of the NYU Symunix project. The second employs an extended interface that shares information in both directions across the user-kernel boundary, resulting in simpler code and better performance. We describe experiments with these locks in both real and synthetic applications on Silicon Graphics and Kendall Square multiprocessors. Results demonstrate the feasibility of high performance software locks with multiprogramming on scalable systems, and show competitive behavior on smaller, bus based machines.

---

*This work was supported in part by NSF Institutional Infrastructure award number CDA-8822724, NSF grant number CCR-9005633, and ONR research contract number N00014-92-J-1801 (in conjunction with the ARPA Research in Information Science and Technology—High Performance Computing, Software Science and Technology program, ARPA Order No. 8930).

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 very widely used.

Unfortunately, spin locks suffer from two serious problems:

1. Both the common `test_and_set` lock, and its variant, the read-polling `test_and_test_and_set` lock, suffer from severe performance degradation as the number of processors competing for the lock increases.

2. In multiprogrammed systems, a process that is preempted during its critical section can delay the progress of every other process that needs to acquire the lock.

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, 7, 12]. To address the second problem, several operating systems have incorporated schemes in which applications communicate with the kernel scheduler to prevent [6] or recover from [3] 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. Their application to programs using queue-based locks is much less straightforward. None of [3], [6], and [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. We discuss related work in more detail in section 2, touching on scalable spin locks, multiprogramming of multiprocessors, and synchronization-sensitive scheduling. We present our algorithms in section 3, with a focus on scheduler interfaces resembling that of Symunix [6], and locks resembling those of Mellor-Crummey and Scott [12]. We present empirical results in section 4, comparing the performance of a variety of locks on a variety of workloads, applications, and machines. Our conclusions appear in section 5.

2 Related Work

2.1 Scalable Spin Locks

Active sharing of memory locations is the principal obstacle to scalability in spin locks. 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 [7] 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 processor. In Anderson's lock, each waiting processor uses a `fetch_and_add` operation to obtain the index of the array element on which it is to spin. To release the lock, it toggles the state of the next (circularly) higher element of the array. In Graunke and Thakkar's lock, each waiting processor uses a `fetch_and_store` operation to obtain the index of an array element permanently associated with the previous holder of the lock. To release the lock, it toggles the state of its own (permanently associated) element, which will in turn be polled by the next processor to acquire the lock.

The queue-based spin lock of Mellor-Crummey and Scott [12] represents its queue with a distributed linked list instead of an array. Much as in Graunke and Thakkar's lock, 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. In this case, however, it 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.

2.2 Multiprogramming Multiprocessors

A multiprocessor that is to run more than one parallel application can be scheduled in any of several ways, including pure time-slicing (in which all processes of all applications compete equally and chaotically for physical processors), coscheduling (in which the scheduler attempts to ensure that all processes of a given application run simultaneously), and processor partitioning (in which each application is given exclusive use of a subset of the processors of the machine for an extended period of time). A growing body of evidence [2, 5, 10, 15, 16] suggests that throughput is maximized by a processor-partitioned environment in which each application runs exactly one process per processor. In such an environment, or in one that employs coscheduling, all the processes that share a given lock will always run simultaneously. Unfortunately, this ideal situation (from the point of view of synchronization) is not always easy to arrange.

Accommodating the computing needs of applications amounts to a two-dimensional bin-packing problem (processors $\times$ time). It is complicated by the fact that neither the set of applications nor the number of processors a given application can fruitfully employ is static. If applications are not able to adjust their number of processes to match the number of available processors, processes may sometimes be preempted while others with which they share a lock continue to run. In such an environment, the possibility arises that running processes will busy-wait for action on the part of a process that is not running.

2.3 Synchronization-Sensitive Scheduling

When locks are used to protect lengthy critical sections, rescheduling may be preferable to busy-waiting. 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 [9]. 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. Also, existing on-line algorithms are designed for scenarios in which any waiting process can acquire a lock, and are not directly applicable to queue-based locks.

To address the possibility of preemption, several researchers have invented forms of synchronization-sensitive scheduling. The Scheduler Activation proposal of Anderson et al. [3] 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 [4] allows a process to suggest to the scheduler that it be descheduled in favor of some specific other process. If lock data structures indicate which process holds the lock, a process that spins "too long" may guess that the lock holder is not running, and can offer to give it its processor.

Rather than recover from untimely preemption, the Symunix system of Edler et al. [6] 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.

For `test_and_set` and `test_and_test_and_set` locks, synchronization-sensitive scheduling need only address the possibility of preemption during a critical section. For queue-based locks, however, it is also crucial to consider the possibility of preemption while waiting to acquire the lock, since no process later in the queue will be able to acquire the lock until all earlier processes have done so. In the following section we present queue-based spin locks that ensure scalability by spinning only on local locations, while also interacting with the scheduler in such a way that no process waits for a process that isn't running. Our code is based on the Symunix scheduler interface, and on the spin lock of Mellor-Crummey and Scott, though it should also be possible to devise solutions based on the scheduler interfaces of Scheduler Activations, Mach, or Psyche, or the spin locks of Anderson or Graunke and Thakkar.

3 Algorithms

In this section we present two locks. 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.

The first, handshaking lock appears in figures 1 and 2. As in the original list-based queuing lock, parameter `I` of `acquire_lock` and `release_lock` points to a `qnode` record allocated (in an enclosing scope) in shared memory locally-accessible to the invoking processor. The `compare_and_swap` instruction returns true if it successfully replaced the value specified by its second argument with the value specified by its third argument.

```
type context_block = record
    preemptable,            // set by user; read by kernel
    warning : Boolean       // set by kernel; read by user
type multi_flag = (not_yet, can_go, got_it, lost_it, ack, nack)
type qnode = record
    pred, next : ^qnode
    next_done : Boolean
    status : multi_flag
type lock = ^qnode

private cb : ^context_block;
```

Figure 1: Declarations for the Queued-Handshake lock.

The Symunix interface makes it easy to prevent preemption in the critical section. However, to avoid giving the lock to a preempted process in the queue, we designed a handshaking algorithm between the releasing and acquiring processes. 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 qnode, the releaser assumes 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. In either case, the successor waits for a final `ACK` or `NACK` from the releaser before proceeding, to avoid changing `qnode` fields that the releaser is still reading.

To handle preemption in a queue based lock, two types of information are needed. The kernel needs to be aware when a process is about to enter a critical section, and the library code implementing the lock needs to know the status of processes in the application. To avoid the complexity and cost of handshaking, we propose that the preemptive scheduler for a multiprocessor explicitly indicate which processes are currently running. In place of the `preemptable` Boolean flag in a process's context block, we propose a state variable with several possible values. By accessing this 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 3. 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 in Symunix, 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. When giving a time extension, the kernel sets a warning flag, which it clears at the beginning of each quantum. When exiting a critical section, a process needs to yield if the warning flag is set.

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

```
procedure acquire_lock (L : ^lock, I : ^qnode)
    loop
        I->next := nil
        cb->preemptable := false
        I->pred := fetch_and_store (L, I)
        if I->pred = nil
            return
        I->status := not_yet
        I->pred->next := I
        repeat
            cb->preemptable := true
            if cb->warning                  // kernel wanted to preempt me
                yield
            cb->preemptable := false
        while I->status = not_yet           // spin
        val : multi_flag := fetch_and_store (I->status, got_it)
        if val = can_go
            I->pred->next_done := true      // tell pred I'm done with its qnode
            repeat until I->status = ack    // let pred finish using my qnode
            return
        while val != nack
            val := I->status

procedure release_lock (L : ^lock, I: ^qnode)
    if I->next = nil                        // no known successor
        if compare_and_swap (L, I, nil)
            goto rtn
        repeat while I->next = nil          // spin
    I->next_done := false
    loop
        I->next->status := can_go
        for i in 1..TIMEOUT                 // spin
            if I->next_done
                I->next->status := ack
                goto rtn
        if fetch_and_store (I->next->status, lost_it) = got_it
                // oh! successor was awake after all
            repeat until I->next_done
            I->next->status := ack
            goto rtn
        succ : ^qnode := I->next->next
        if succ = nil
            if compare_and_swap (L, I->next, nil)
                I->next->status := nack
                goto rtn
            repeat while (succ := I->next->next) = nil  // spin; probably non-local
        I->next->status := nack
        I->next := succ
        succ->pred := I
    cb->preemptable := true
rtn:
    if cb->warning      // kernel wanted to preempt me
        yield
```

Figure 2: Queued-Handshake lock for the Symunix kernel interface.

```
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; probably non-local
            candidate := shadow->next
        // order of following checks is important
        if compare_and_swap (&candidate->self->state,
                unpreemptable_self, unpreemptable_other)
            or compare_and_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 3: Code for the Smart-Q lock.

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 extensively explore the parameter space. To verify the results we obtained from the synthetic program, two real programs were run—Cholesky from the SPLASH suite [13] and a multiprocessor version of Quicksort. The two programs were good candidates for examining the effectiveness of the locks since this is the only form of synchronization they use. 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 - A standard `test_and_test_and_set` lock that polls a lock's value and attempts to acquire it when it is free.

TAS-no_preempt - A `test_and_test_and_set` lock in which the critical section is marked as non-preemptable, using the approach described by the Symunix developers.

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 (the synchronization bus on the SGI; extra cache states on the KSR). No consideration for preemption is taken. 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.

The machines we used were a Silicon Graphics Iris 4D/480, with 8 processors, and a Kendal Square Research KSR1, with 32 processors. Both of these machines provide special hardware for

high-performance locks. The SGI Iris incorporates a separate synchronization bus for test_and_set operations. The bus has sufficient bandwidth to allow all 8 processors to spin without serious contention. The KSR 1 incorporates a cache line locking mechanism that provides the equivalent of queued locks in hardware. The queueing is based on physical proximity in a ring-based interconnection network, rather than on the chronological order of requests, and is not vulnerable to the preemption of waiting processes that have not yet acquired their lock.

The native operating systems do not provide the mechanisms necessary to avoid waiting for a preempted process, but both could be modified to do so. We would expect the result to out-perform all other options on these two machines, and our experiments bear this out. Our goal in implementing scheduler-sensitive queued locks in software was to approach the performance of (scheduler-sensitive use of) the native locks, demonstrating that special-purpose hardware is not crucial for scalable locks in multiprogrammed systems.

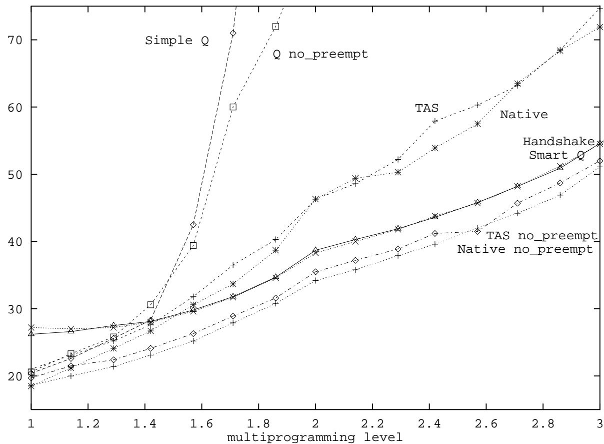
The queued locks require both `fetch_and_store` and `compare_and_swap`, primitives not available on the SGI or KSR. We implemented a software version of these atomic instructions using the native spinlocks. We also used the native spinlocks to implement `test_and_set`. This approach is acceptable as 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. Within the critical section, each process decrements a shared counter that indicates the total number of iterations remaining in the experiment; it exits when this counter reaches zero. 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. Data to be shared between the "kernel" and the user is created with `shmget`. Worker processes are created with `fork`. 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



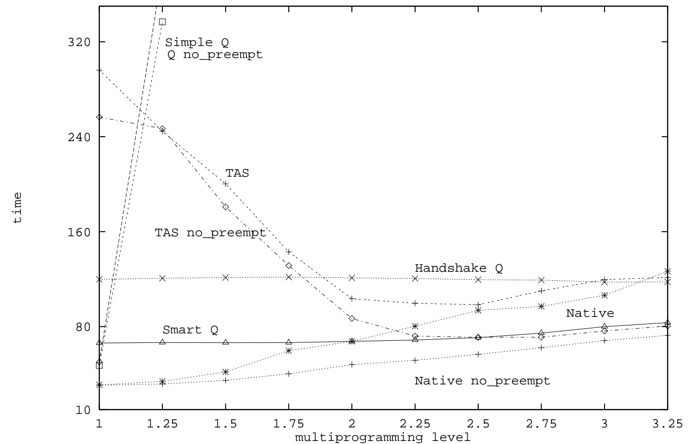


Figure 4: Varying Multiprogramming Levels on a 7-processor SGI Iris.

Figure 5: Varying Multiprogramming Levels on a 31-processor KSR1.

Figures 4 and 5 show the running time for a fixed number of processors (7 on the SGI and 31 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 150 μs. Because the ratio of critical to non-critical work is 1:10, while only 7 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 standard **TAS** lock. They do not outperform the **TAS-no_preempt** lock on the SGI, but only because the small number of processors available does not give the contention-reducing property of queuing an opportunity to overcome the lower constant overhead of TAS. As one would expect, the native lock performs best of all (the other locks are built on top of it, after all), though only when modified (**Native-no_preempt**) to prevent preemption in the critical section.

On the KSR, 31 processors were available, so a 1:10 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 when compared to the SGI results. The **Queued** and **Queued-no_preempt** locks suffer a high 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. The handshaking algorithm however requires several remote references and the penalty paid

Work:Crit. Sec.	Multiprog.	Work	Acquire	Crit. Sect.	Release	Preempted	Time(sec)
16.7	1.0	5.7%	90.9%	1.2%	2.0%	0%	49.3
16.7	1.4	6.1%	67.7%	1.3%	5.5%	19.3%	46.0
16.7	2.0	7.7%	34.6%	1.8%	6.5%	49.3%	36.3
66.7	1.0	23.8%	72.5%	1.3%	2.2%	0%	44.7
66.7	1.4	27.7%	45.5%	1.6%	5.5%	19.7%	38.2
66.7	2.0	35.1%	8.0%	2.0%	5.7%	49.1%	30.2
120	1.0	56.0%	39.4%	1.8%	2.8%	0%	34.5
120	1.4	63.7%	8.8%	2.1%	5.6%	19.6%	30.4
120	2.0	43.0%	2.7%	1.3%	3.0%	49.8%	44.0

Table 1: Breakdown of execution time for the **TAS-no_preempt** lock

on the KSR is high enough to produce a high steady level of overhead. The Smart-Q lock performs better than all other queued or test_and_set locks since it avoids critical section preemption, ensures that the lock is not given to a preempted process, and incurs little overhead in doing so. Again as expected, the **Native-no_preempt** lock performs the best of all.

The lock with the most surprising behavior is the **TAS-no_preempt**. While all other locks become worse as the multiprogramming level increases, the **TAS-no_preempt** lock improves in performance. In order to explain this we profiled the execution of the synthetic program to determine the breakdown of execution time. As can be seen in table 1 most of the time is spent in the *acquire* routine. The reason for this is that the high contention experienced by the lock effectively increases the portion of time spent in serial work. Introducing multiprogramming reduces contention and consequently the amount of serial work in the program. This effect results in an actual performance improvement, even though there are fewer total cycles available to the application. As the multiprogramming level increases even more the reduction in contention cannot offset the reduction of available cycles and the performance starts to degrade again. At high multiprogramming levels the reduction in contention can be high enough to actually make the **TAS-no_preempt** lock perform better than the Smart-Q lock, but not by much. It is also unclear that anyone would want to run parallel applications in such a severely time-sliced environment. With modest levels of multiprogramming, the Smart-Q lock outperforms the TAS-no_preempt lock by a considerable margin.

4.3 Varying the Number of Processors

The intent of increasing the number of processors working in parallel is to increase the amount of simultaneous work occurring. However, if the programs need to synchronize, considerable contention can develop for the lock. Previous work has shown that queue locks improve performance in such an environment, but as indicated by the graphs in figures 4 and 5 they can experience difficulties under multiprogramming. The graphs in figures 6 and 7 show the effect of increasing the number of processors on the different locks at a multiprogramming level of 2.0.

Because the synthetic program runs a total number of loop iterations proportional to the number of processors, 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

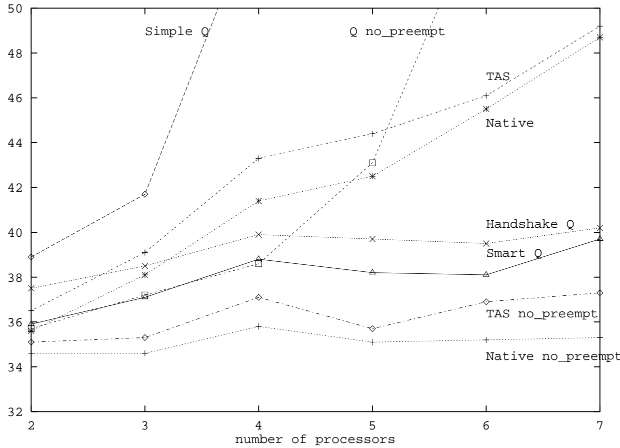


Figure 6: Varying the Number of Processors on the SGI Iris with a multiprogramming level of 2 (one unrelated process per processor).

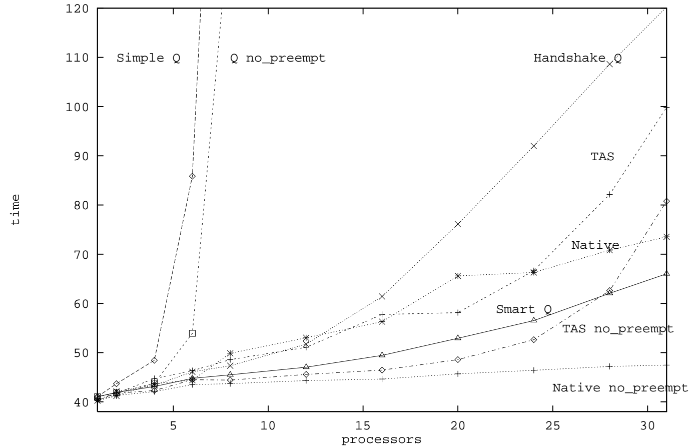


Figure 7: Varying the Number of Processors on the KSR1 with a multiprogramming level of 2 (one unrelated process per processor).

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** and **Native** locks also degrades with additional processors, in part this is due to the increased contention, but primarily because of the increased likelihood of preemption while in the critical section. The **Smart-Q**, **TAS-no_preempt**, and **Native-no_preempt** locks display more-or-less even performance, with overhead highest for **Handshake** and **Smart-Q**.

The results on the KSR resemble those on the SGI, except that the larger number of processors and the higher cost of remote operations allow the **Smart-Q** lock to overtake the **TAS-no_preempt** lock. The native locks, with our modification to avoid critical section preemption, perform the best of all. One likely reason why the KSR **Native-no_preempt** lock performs so well is that the ring-based interconnection topology of the KSR provides the native lock with queue based behavior allowing the lock to scale to larger numbers of processors.

Since the queued locks are targeted for scalable multiprocessors, it important to focus on the right sides of the graphs. While figure 6 demonstrates that dealing with preemption is necessary for achieving good performance, the effectiveness of avoiding preemption and achieving scalability can be seen more clearly in figure 7. The performance of the **TAS no_preempt** lock starts to degrade quickly after 25 processors. On the other hand, the **Smart-Q** and **Native no_preempt** locks handle the increased contention of the larger number of processors much better yielding the best performance for large numbers of processors.

4.4 Results for Real Applications

To verify the results obtained from the synthetic program, we also measured the performance of a pair of real lock-based applications. Figure 8 shows the completion times of these applications, in seconds, when run with a multiprogramming level of 2.0 using 7 processors on the SGI and 31

processors on the KSR. Both sets of graphs confirm the conclusions reached from the synthetic program. The **Smart-Q** lock outperforms all but the **Native-no_preempt** lock in all cases.

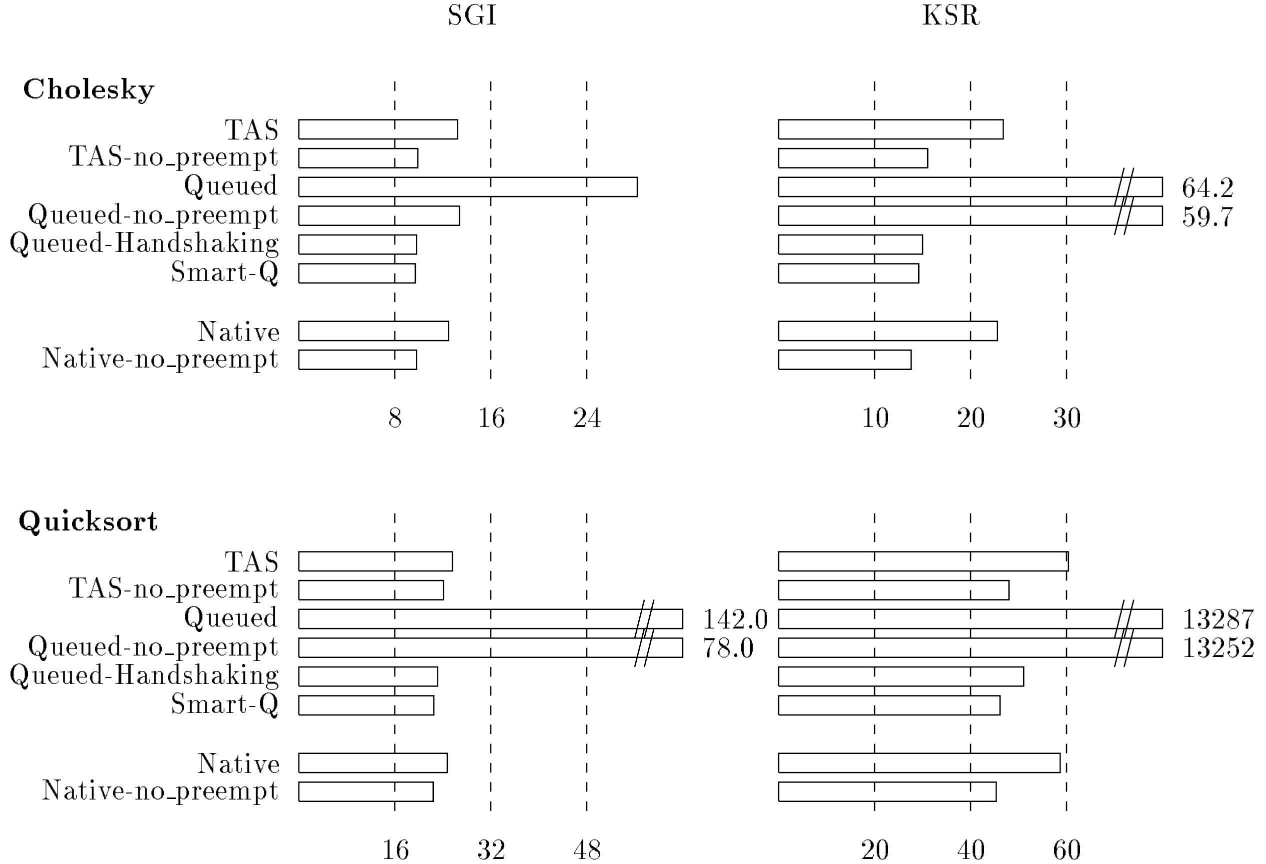


Figure 8: Completion times (in seconds) of real applications on a 7-processor SGI Iris and a 31-processor KSR1 (multiprogramming level = 2).

5 Conclusions

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.

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 [8]) makes them a very attractive alternative to special-purpose synchronization hardware. The native 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 or the Stanford Dash[?]), our locks are likely to outperform all previous alternatives in the presence of multiprogramming.

There are three primary contributions of this paper. 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 presents 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 blocked process in the queue. Third, it shows that by sharing appropriate information between the scheduler and application processes, we can make the lock both simpler and faster.

Acknowledgements

We would like to thank Donna Bergmark and the Cornell Theory Center for the use of their KSR 1.

References

[1] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEETPDS*, 1(1):6-16, January 1990.

[2] T. E. Anderson. Operating System Support for High Performance Multiprocessing. Ph.D. dissertation, TR 91-08-10, UWASHCSED, August 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. *TOCS*, 10(1):53-79, February 1992. Originally presented at the *Thirteenth SOSP*, 13-16 October 1991.

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

[5] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos. Multiprogramming on Multiprocessors. In *PROC of the Third SPDP*, pages 590-597, December 1991.

[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*, Pittsburgh, PA, 26-27 September 1988.

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

[8] M. Herlihy. Wait-Free Synchronization. *TOPLAS*, 13(1):124-149, January 1991.

[9] 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 SOSP*, pages 41-55, Pacific Grove, CA, 13-16 October 1991.

[10] S. T. Leutenegger and M. K. Vernon. Performance of Multiprogrammed Multiprocessor Scheduling Algorithms. In *PROC of the 1990 MMCS*, Boulder, CO, 22-25 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 SOSP*, pages 110-121, Pacific Grove, CA, 14-16 October 1991.

[12] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *TOCS*, 9(1):21-65, February 1991.

[13] J. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *CAN*, 20(1):5-44, March 1992.

[14] M. S. Squillante. Issues in Shared-Memory Multiprocessor Scheduling: A Performance Evaluation. Ph.D. dissertation, TR 90-10-04, UWASHCSED, October 1990.

[15] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. In *PROC of the Twelfth SOSP*, pages 159-166, Litchfield Park, AZ, 3-6 December 1989.

[16] J. Zahorjan and C. McCann. Processor Scheduling in Shared Memory Multiprocessors. In *PROC of the 1990 MMCS*, pages 214-225, Boulder, CO, 22-25 May 1990.