Scheduler-Conscious Synchronization

LEONIDAS I. KONTOTHANASSIS
DEC Cambridge Research Laboratory
ROBERT W. WISNIEWSKI
Silicon Graphics, Inc.
and
MICHAEL L. SCOTT
University of Rochester

Efficient synchronization is important for achieving good performance in parallel programs, especially on large-scale multiprocessors. Most synchronization algorithms have been designed to run on a dedicated machine, with one application process per processor, and can suffer serious performance degradation in the presence of multiprogramming. Problems arise when running processes block or, worse, busy-wait for action on the part of a process that the scheduler has chosen not to run. We show that these problems are particularly severe for scalable synchronization algorithms based on distributed data structures. We then describe and evaluate a set of algorithms that perform well in the presence of multiprogramming while maintaining good performance on dedicated machines. We consider both large and small machines, with a particular focus on scalability, and examine mutual-exclusion locks, reader-writer locks, and barriers. Our algorithms vary in the degree of support required from the kernel scheduler. We find that while it is possible to avoid pathological performance problems using previously proposed kernel mechanisms, a modest additional widening of the kernel/user interface can make scheduler-conscious synchronization algorithms significantly simpler and faster, with performance on dedicated machines comparable to that of scheduler-oblivious algorithms.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Pro-

---

This work was supported in part by National Science Foundation grants CCR-9319445 and CDA-8822724, by ONR contract N00014-92-J-1801 (in conjunction with the ARPA Research in Information Science and Technology–High Performance Computing, Software Science and Technical program, ARPA order no 8930), and by ARPA research grant MDA972-92-J-1012. R. Wisniewski was supported in part by an ARPA Fellowship in High Performance Computing administered by the Institute for Advanced Computer Studies, University of Maryland. Experimental results were obtained in part through use of resources at the Cornell Theory Center, which receives major funding from NSF and New York State; additional funding comes from ARPA, the NIH, IBM Corp., and other members of the Center's Corporate Research Institute. The U.S. Government has certain rights in this material.
Authors' addresses: L. I. Kontothanassis, DEC Cambridge Research Laboratory, One Kendall Square, Cambridge, MA 02139; email: kthanasi@crl.dec.com; R. W. Wisniewski, Silicon Graphics, Inc., 2011 North Shoreline Boulevard, Mailstop 8U 500, Mountain View, CA 94043; email: bobw@engr.sgi.com; M. L. Scott, Computer Science Department, University of Rochester, Rochester, NY 14627-0226; email: scott@cs.rochester.edu.
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 1997 ACM 0734-2071/97/0200–0003 $03.50

gramming—*parallel programming;* D.4.1 [**Operating Systems**]: Process Management—*multiprocessing/multiprogramming*

General Terms: Algorithms, Performance, Reliability

Additional Key Words and Phrases: Barriers, busy-waiting, kernel-user interaction, locks, mutual exclusion, preemption, scalability, scheduling, synchronization

1. INTRODUCTION

One of the most basic questions for any synchronization mechanism is whether a process that is unable to continue should *spin*—repeatedly testing the desired condition—or *block*—yielding the processor to another, runnable process. Spinning makes sense when the expected wait time of a synchronization operation is less than twice the context switch time, or when the spinning processor has nothing else useful to do. Researchers have developed a wealth of busy-wait (spinning) mechanisms, including mutual-exclusion locks, reader-writer locks (which allow concurrent access among readers, but guarantee exclusive access by writers), and barriers (which guarantee that no process continues past a given point in a computation until all other processes have reached that point). Of particular interest in recent years have been *scalable* synchronization algorithms, which employ backoff or distributed data structures to minimize contention [Anderson 1990; Graunke and Thakkar 1990; Hensgen et al. 1988; Krieger et al. 1993; Lee 1990; Lubachevsky 1989; Magnussen et al. 1994; Mellor-Crummey and Scott 1991a; 1991b; Scott and Mellor-Crummey 1994; Yang and Anderson 1993; Yew et al. 1987].

Unfortunately, busy-waiting in user-level code tends to work well only if each process runs on a separate physical processor. If the total number of processes in the system exceeds the number of processors, them some processors will have to be multiprogrammed. The processes on a given processor may be from different applications or, if the scheduler partitions the machine, from a single application. In either case, serious performance problems can arise when the order in which the scheduler chooses to run processes is different from the order required by the application's synchronization operations.

Several research groups have addressed one or more aspects of the interaction between scheduling and synchronization. Some have shown how to avoid preempting a process that holds a test_and_set lock [Edler et al. 1988; Marsh et al. 1991], or to recover from this preemption if it occurs [Anderson et al. 1992; Black 1990]. Others have developed heuristics that allow a process to guess whether it would be better to relinquish the processor, rather than spin, while waiting for a lock or barrier [Karlin et al. 1991; Lim and Agarwal 1993; Ousterhout 1982].

Our work builds on these previous efforts in three specific ways:

(1) We demonstrate that interactions between scheduling and synchronization are a much more serious problem for scalable synchronization

algorithms than they are for small-scale, centralized algorithms. Moreover, existing proposals to address the small-scale problem do not generalize to the large-scale case. In our experiments, algorithms that provide excellent performance in the absence of multiprogramming sometimes perform orders of magnitude worse when multiprogramming is introduced.

(2) We propose minor extensions to the user-kernel interface that allow an application to interact with the scheduler more effectively. We distinguish between *preemption-safe* algorithms, in which a process communicates with the scheduler regarding its own state only, and *scheduler-conscious* algorithms, in which a process can also determine and modify the state of its peers. We find that scheduler-conscious synchronization algorithms can often be substantially simpler and faster than those that are (merely) preemption safe.

(3) We present six new synchronization algorithms: a preemption-safe ticket lock, a preemption-safe queue lock, a scheduler-conscious queue lock, a scheduler-conscious queued reader-writer lock, a scheduler-conscious small-scale barrier, and a scheduler-conscious scalable barrier. We evaluate these algorithms via experiments on Silicon Graphics and Kendall Square multiprocessors, and we identify the circumstances under which each algorithm is beneficial.

In Section 2 we explain why scalable synchronization algorithms make it particularly difficult, from a conceptual point of view, to deal with untimely preemption. We then present our kernel extensions in Section 3 and our algorithms in Section 4. Section 5 contains empirical results. These quantify the impact of preemption on scheduler-oblivious algorithms and demonstrate the value of preemption-safe and scheduler-conscious alternatives.

For barrier-based applications, our centralized and scalable scheduler-conscious barriers are clearly the mechanisms of choice. For lock-based applications the conclusion is less clear. Multiprogramming effectively reduces contention by reducing the number of processes from a given application that actually run in parallel, thereby weakening the argument for queue-based locks. As a result, while our algorithms allow queue locks to perform acceptably on a multiprogrammed system, we find that simpler, preemption-safe centralized locks (with backoff) often run a little faster. On larger machines (beyond what was available for our experiments) we would expect the balance to tip back toward queue-based locks.

2. SCHEDULING AND SCALABLE SYNCHRONIZATION

The basic problem with scheduling and scalable synchronization is that scalable synchronization algorithms tend to perform operations in different processes in deterministic order, and this order may differ from the order in which the scheduler chooses to run the processes. Scheduling is less of a problem for simple synchronization algorithms because they are less deterministic.

2.1 Locks

The simplest busy-wait locks are variants on test_and_set: every process that wants to acquire a lock attempts repeatedly to change the lock's state from free to busy. While the lock is busy, the attempts accomplish nothing. The moment the lock is freed by its current owner, one of the next few attempts succeeds; which one depends on details of machine architecture and timing. Unfortunately, unsuccessful attempts to acquire the lock cause contention for communication and memory resources. This contention can be enormous on large machines [Anderson 1990; Mellor-Crummey and Scott 1991a]. There are two possible ways to deal with the problem: reduce contention via backoff, or redesign the algorithm to place waiting processes in a queue. Backoff works well in practice, though it still admits a significant amount of unnecessary network traffic [Mellor-Crummey and Scott 1991a], and thus it may be problematic on very large machines. Queuing eliminates contention by allowing each process to spin on a separate location [Anderson 1990; Graunke and Thakkar 1990; Mellor-Crummey and Scott 1991a].

With a nondeterministic lock, performance can suffer if a process is preempted while in the critical section. Several solutions to this problem are known (see Section 6). The solutions fall into two rough camps: those that bound the time that may be spent in a critical section, by temporarily disabling preemption or recovering if it occurs, and those in which waiting processes "give up" and block if they think that the process holding the lock has been preempted.

Unfortunately, neither type of solution works for scalable, deterministic locks. The problem is that we must worry not only about preemption of a process in the critical section, but also about preemption of processes that are currently "waiting in line." Otherwise when a lock is released we may give it to a process that has already been preempted, while processes later in line are actively spinning. We cannot bound the time that a process spends waiting in line, since this depends on the number of processes, and we cannot simply "give up" if we wait too long, because we are linked into a deterministic queue. Our experiments show clearly (Section 5) that it is not sufficient with deterministic locks to cope with preemption only in the critical section: the resulting performance can be orders of magnitude worse in the presence of multiprogramming than it is with one process per processor. We present solutions in Sections 4.1 and 4.2 that link a process out of the queue whenever it is preempted.

2.2 Barriers

The simplest busy-wait barriers employ a central counter, incremented via an atomic hardware primitive or protected by a lock. Each process as it arrives at the barrier increments the counter and then spins on a single, shared completion flag. The order in which processes arrive depends on timing details of the application and machine. The last-arriving process (which realizes it is last from the value of the counter) flips the flag,

allowing all of the processes to proceed. As with test_and_set locks, competing attempts to increment the counter can incur significant contention, particularly on large machines. In addition, even in the absence of contention, serial counter updates imply an asymptotic running time of $O(p)$, which becomes unacceptable as the number of processes p grows large. Several researchers have shown how to increase scalability by building barriers based on log-depth tree or FFT-like patterns of point-to-point notifications among processes (see Section 6).

On a multiprogrammed machine, performance can suffer at a barrier if processes spin uselessly, waiting for preempted peers that have not yet reached the barrier. Avoiding or recovering from this preemption is not an option, because there is no a priori bound on the time it will take a process to reach the next barrier. Solutions are therefore limited to determining when a waiting process should spin and when it should block. For small, nondeterministic barriers, we present an algorithm in Section 4.3 that makes an optimal decision, blocking when and only when some other process could make better use of the processor.

Unfortunately, the deterministic notification patterns of scalable barriers may require that processes run in a different order from the one chosen by the scheduler. If blocked processes simply yield the processor, remaining in the ready list, then the scheduler may cycle through the entire list a logarithmic number of times, one for each level of the barrier's tree or network [Crovella et al. 1991]. On the assumption that processes will seldom migrate among processors on a large machine, we present a solution in Section 4.3 that makes an optimal spin-versus-block decision within processors and employs a scalable busy-wait barrier among processors. We assume that the scheduler partitions the machine among the current applications; the barrier adapts dynamically to changes in the number of available processors.

3. SOLUTION STRUCTURE

Our work is based on the assumption that deterministic ordering is essential for scalability and that synchronization algorithms must therefore cooperate with the scheduler to decide what the order will be.

Because it is ultimately responsible for the fair allocation of resources among competing applications, a kernel-level scheduler cannot in general afford to accept arbitrary directives from user-level code. Our algorithms assume that a process can influence the behavior of the scheduler enough to prevent itself from being preempted in the middle of a (short) critical section. This capability suffices for small-scale (nondeterministic) locks and need not compromise fairness or security (see below). Over longer periods of time, we assume that it is the application's responsibility to cope with the decisions of the scheduler. For barriers and for scalable locks, a process must be able to obtain information about the status of *other* processes and about the set of processors available to the application. This information may be (1) guessed via past experience using heuristics, (2) deduced

through interaction with other processes, e.g., via "handshaking," or (3) provided by the kernel itself. In order, these options provide information of increasing accuracy and thus result in simpler algorithms and better performance.

Experience-based heuristics can be successful to the extent that the present and future resemble the past. They form the basis of the competitive lock algorithms of Karlin et al. [1991] and of the heuristic barriers we describe near the beginning of Section 4.3. For locks, the goal is to block if the wait time will be longer than twice the context switch time and to spin if it will be shorter. For barriers, the goal is to block if some preempted process could use the current processor to make progress toward the barrier, and to spin otherwise. In both cases, the algorithm can determine by reading the clock whether blocking or spinning would have been a better policy at the most recent synchronization operation. If it finds it made the wrong decision, it biases its decision in favor of the other alternative the next time around. Whether this sort of adaptation works better than a simple static choice appears to depend on the relative costs of bookkeeping and context switching; Karlin et al. [1991] and Lim and Agarwal [1993] reach different conclusions.

Interaction with peer processes can provide better information about the peers' status, provided that they respond promptly to enquiries (when running). In Section 4.1 we use a "handshaking" technique in two of our mutual exclusion algorithms. To hand a peer a lock, a process sets a flag on which the peer is expected to be spinning and then waits for the peer to set an acknowledgment flag. If the acknowledgment does not appear within a certain amount of time, the signaling process assumes that the peer is currently preempted. There is an inherent inefficiency to this approach: if the signaling process does not wait long enough, it will too often skip over a running peer by mistake. Any time it waits, however, is lost to computation.

We have found heuristics and handshaking to be expensive both in implementation and execution cost. Algorithms based on kernel-provided information are simpler and easier to design. They generally provide superior performance, in part because the information from the kernel is more accurate than user-level estimates and in part because the kernel can collect the information more efficiently than user-level code can guess it.

3.1 Kernel Extensions

Our kernel extensions appear in Figure 1. They build upon ideas proposed by the Symunix project at NYU [Edler et al. 1988]. Similar extensions could be based on the kernel interfaces of Psyche [Marsh et al. 1991] or Scheduler Activations [Anderson et al. 1992] (see Section 6). The state field of a context_block is readable and writable in user space; the remaining fields of both the context_block and partition_block records are readable in user space, but writable only by the kernel.

To control its own preemption, a process uses the preemptable and unpreemptable_self values of context_block.state and the context_block.

```
type context_block = record
    state : (preempted, preemptable, unpreemptable_self, unpreemptable_other)
    warning : Boolean
    ...

type partition_block = record
    num_processors, generation : integer
    processes_on_processor : array [MAX_PROCESSORS] of integer
    processor_ids : array [MAX_PROCESSES] of integer
    ...
```

Fig. 1. Pseudocode declarations for the kernel-application interface.

warning flag. These fields are borrowed directly from Symunix. Before entering a critical section, a process sets its state to unpreemptable_self to request to the kernel scheduler that it not be preempted. The kernel will honor this request precisely once per quantum and for a fixed amount of time (the length of this "grace period" is known to applications, which must ensure that critical sections can normally be finished in a smaller amount of time).[1] When it passes over a process, the kernel sets the warning flag. When it leaves a critical section, a process should check this flag and voluntarily yield the processor if the flag is set. If the process does not yield before the end of the grace period, the kernel will preempt it anyway. In either case (yield or preempt), the kernel resets the warning flag and subtracts any extra time the process received from the beginning of its next quantum.

We call an algorithm *preemption-safe* if (1) it never spins for more than a constant amount of time when some other process could profitably be using the processor and (2) it employs no kernel extensions other than those required to avoid its own preemption in critical sections (in our work, context_block.warning and the preemptable and unpreemptable_self values of context_block.state). We refer to an algorithm as *scheduler-conscious* if it interacts with the scheduler to determine or alter the states of other processes or to determine the set of processors available to the application. In our work, scheduler-conscious algorithms use the preempted and unpreemptable_other values of context_block.state and/or the various fields of partition_block. These fields are a generalization of the interface described in our work on small-scale scheduler-conscious barriers [Kontothanassis and Wisniewski 1993]. They also resemble the "magic page" of information provided by the Psyche kernel [Scott et al. 1990]. The preempted and unpreemptable_other states allow one process to pass a lock to another and to make the other unpreemptable, atomically. The information in a

[1]The possibility of page faults means that we cannot in general provide a guarantee against inopportune preemption. The best we can hope to do in any of our algorithms is to minimize the chance that such preemption will occur. To provide real guarantees (e.g., for a real-time system), the kernel would need to ensure that a process that sets its state variable to unpreemptable_self will always be able to execute some minimum number of instructions within a small bounded period of time.

partition_block allows a barrier to determine which processes share which processors and to track changes in this information over time.

None of the extensions requires the kernel to maintain any new information, to access any user-level code or data structures, to understand the particular synchronization algorithm(s) being used by the application, or to do anything more often than once per quantum. Overall system correctness never depends on correct use of fields by applications, though the performance of a particular application may suffer if it uses the fields incorrectly.

3.2 Hardware Support

We assume the availability of special instructions that allow a process to read, modify, and write a shared variable as a single atomic operation. In several cases, for example, we must change a state field to unpreemptable_self or unpreemptable_other, if and only if it was preemptable before. All of our atomic instructions can be emulated efficiently by load-linked and store-conditional.

Some multiprocessors, especially the larger ones, provide more sophisticated hardware support for synchronization. Examples include the queue-based locks of the Stanford Dash machine [Lenoski et al. 1992], the QOLB (queue-on-lock-bit) operation of the IEEE Scalable Coherent Interface [Aboulenein et al. 1994], and the near-constant-time barriers of the Thinking Machines CM-5 and the Cray Research T3D/E. Hardware queued locks and barriers are faster than software alternatives, but cannot be customized, e.g., to avoid the scheduling problems described in Section 2. Hardware barriers provide an asymptotic performance improvement that almost certainly makes them worthwhile on large machines. For hardware queued locks, the tradeoffs among cost, performance, and flexibility are less clear.

Our scalable barrier and queue locks arrange for processors to spin only on local locations, on which no other processor spins. In most cases, we ensure that those locations will be local not only on cache-coherent machines (on which they migrate to the spinning processor), but also on machines that lack hardware cache coherence. On these latter, NCC-NUMA machines (non-cache-coherent, nonuniform memory access), variables on which processes spin must be allocated statically in the local memory of the spinning processor; spins are terminated by a single uncached remote write by another processor.

4. ALGORITHMS

In this section we present several new preemption-safe and scheduler-conscious synchronization algorithms. We consider mutual exclusion, reader-writer locks, and barriers. Space constraints preclude the inclusion of actual code. Readers are encouraged to access pseudocode at http://www.cs.rochester.edu/u/scott/synch_pseudocode/ps_and_sc.html. Earlier versions appear in the technical report version of this article [Kontothanassis et al. 1994]. Both pseudocode and C source code are available from ftp://ftp.cs.rochester.edu/pub/packages/sched_conscious_synch.

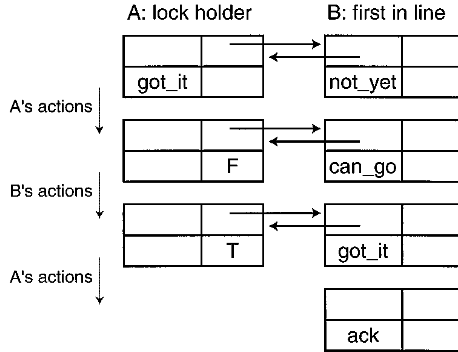

Fig. 2. The simple case in the Queue-HS lock. Process A sets its next_done flag (lower right box) to false and changes B's status field (lower left box) to can_go. B responds by changing its status to got_it and flipping A's next_done flag. A acknowledges by changing B's status to ack, and then both can proceed.

All of our algorithms work well in a dynamic hardware-partitioned environment—an environment widely believed to provide the best combination of throughput and response time for large-scale multiprocessors [Crovella et al. 1991; Leutenegger and Vernon 1990; Tucker and Gupta 1989; Zahorjan and McCann 1990]. (Note that multiprogramming is still an issue on a partitioned machine, since an application with P processes may be forced to run in a partition with $N < P$ processors.) Except for the barriers, which require partition information, all of the algorithms will also work well under ordinary time sharing. For a coscheduled environment the additional complexity of preemption-safe and scheduler-conscious algorithms is not necessary, but does not introduce much overhead.

4.1 Mutual Exclusion

In this section we present two variants of the MCS queue-based lock [Mellor-Crummey and Scott 1991a]: one preemption-safe, the other scheduler-conscious. We also present a preemption-safe variant of the ticket lock. Because they awaken processes in a deterministic order, all three locks must address preemption not only within critical sections, but also while waiting in line.

Our preemption-safe queue lock (**Queue-HS** in the figures of Section 5) uses handshaking to determine the status of other processes (Figure 2). When releasing a lock, a process notifies its successor in the queue that it (the successor) is now the holder of the lock. The successor must then acknowledge receipt of the lock by setting another flag. If this acknowledgment is not received within a fixed amount of time, the releasing process assumes that its successor is preempted, rescinds its notification, and proceeds to the following process. Throughout this period the releasing process is unpreemptable. We use atomic fetch_and_store instructions to avoid a timing window: without them a releaser might conclude that its successor was preempted and proceed to give the lock to another process,

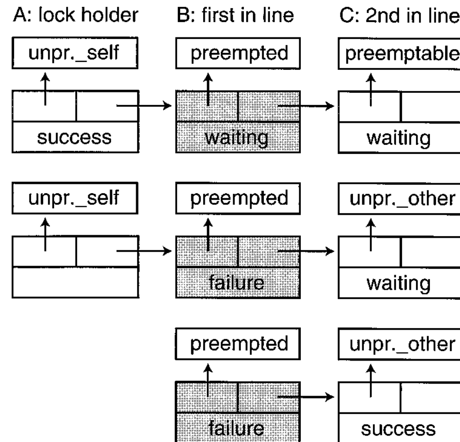

Fig. 3. Passing over a preempted process in the Smart-Q lock. When it sees that B's state is preempted, A changes B's status field (lower box) to failure. It then changes C's state to unpreemptable_other and changes C's status to success, allowing C to proceed.

while the successor thinks that it has acquired the lock and proceeds to the critical section.

The Queue-HS lock solves the preemption problem but unfortunately adds significant overhead to the common case. Processes need to interact several times when a lock is released. To address this limitation, we have designed a scheduler-conscious algorithm (**Smart-Q** in the figures) in which the releasing process examines its successor's state variable, which is kept up-to-date by the kernel. If the successor is preempted, the releaser proceeds to other candidates later in the queue (Figure 3). If the successor is running, the releaser uses an atomic instruction to change the successor's state to other_unpreemptable. If the change is successful the lock is passed to the successor. The atomic instruction resolves a potential race between the releaser and the kernel: after determining that the successor is not preempted, we must make it unpreemptable without giving the kernel an opportunity to preempt it.

One of the problems with queue-based locks is high overhead in the absence of contention. On small-scale machines and for low-contention locks a test_and_set or ticket lock with backoff may be preferable [Mellor-Crummey and Scott 1991a]. The ticket lock guarantees FIFO service; the test_and_set lock is nondeterministic. The two locks also use different atomic instructions, making them appropriate on different machines.

When a process wishes to acquire a ticket lock, it performs an atomic fetch_and_increment on a "next available number" variable. It then spins until a "now serving" variable matches the value returned by the atomic instruction. To avoid contention on large-scale machines, a process should wait between reads of the "now serving" variable for a period of time equal to the difference between the last read value and the value returned by the fetch_and_increment of the "next available number" variable, times the minimum length of a critical section. (We call this technique "proportional

backoff"; it works equally well on dedicated and multiprogrammed machines.) To release the lock, a process increments the "now serving" variable.

Our preemption-safe, handshaking version of the ticket lock (**Ticket-PS** in the figures) uses one additional "acknowledgment" variable, which contains the number of the last granted but unacknowledged ticket. A releasing process increments both the additional variable and the "now serving" variable. If these are different from the "next available number" variable, then at least one process is waiting, in which case the releaser waits (spinning) for the next acquiring process to update the acknowledgment variable. If the update does not occur within a fixed amount of time, the releaser attempts to withdraw its grant of the lock by performing the update itself. Both the acquirer and the releaser use compare_and_swap, so there is never any doubt as to which of them has succeeded. If it successfully withdraws its grant of the lock, the releaser sets the acknowledgment variable and the "now serving" variable to the next higher value and repeats. We assume that the "now serving" variable does not have the opportunity to wrap all the way around and reach a value it previously had, while a process remains preempted. For 32-bit integers, a 1GHz processor, and an empty critical section, a process would have to be preempted for more than three minutes before correctness would be lost. Going to 64-bit integers would extend this time to over 16,000 years.

There is no obvious way to develop a scheduler-conscious version of the ticket lock without exporting lock code into the kernel. The problem is that the lock does not keep track of the identities of waiting processes. The releaser of a lock is therefore unable to determine the status of its successor: it does not know who the successor is.

A caveat with all three of our modified locks is that they give up the FIFO ordering of the scheduler-oblivious version. It is thus possible (though highly unlikely, and with probability lower than the probability of starvation with a test_and_set lock) that a series of adverse scheduling decisions could cause a process to starve. We have considered algorithms that leave preempted processes in the queue so that they only lose their turn while they are preempted. Markatos [1991] adopted a similar approach in his real-time queue lock, where the emphasis was on passing access to the highest-priority waiting process. For simple unprioritized mutual exclusion, leaving preempted processes in the queue makes the common case more expensive: processes releasing a lock have to skip over their preempted peers repeatedly. We consider the (unlikely) possibility of starvation insignificant in comparison to this overhead.

The algorithms described in this section work only for singly nested locks. For multiply nested locks, a process should make itself preemptable only after releasing the outermost lock. It can accomplish this by incrementing a local "nesting level" variable on acquires and decrementing it on releases. The state flag should be set to preemptable only when the level reaches zero.

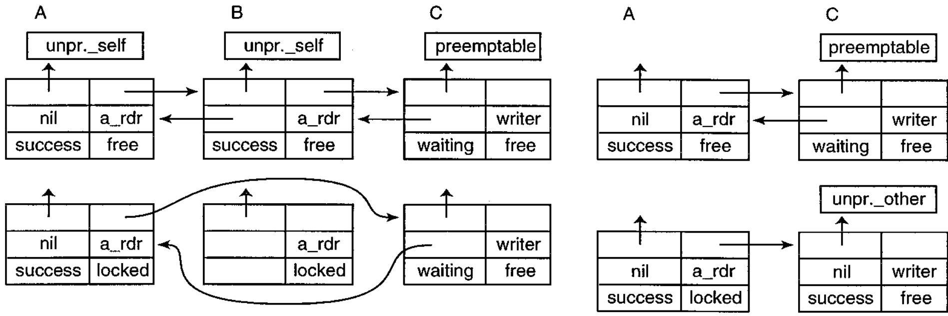

Fig. 4. Two of many possible scenarios in the RW-Smart-Q lock. On the left side of the figure, processes A and B are active readers, and process B finishes its critical section first. When it discovers that it still has a predecessor, it locks A's queue node and its own, links itself out of the queue, and releases the locks. On the right side of the figure, A finishes its critical section. When A discovers that it has no predecessor, it changes C's state to unpreemptable_other and changes C's status (lower left box) to success, allowing C to proceed.

4.2 Reader-Writer Locks

Reader-writer locks are a refinement of mutual exclusion locks. They provide exclusive access to a shared data structure on the part of writers (processes making changes to the data), but allow concurrent access by any number of readers. There are several versions of reader-writer locks, distinguished by the policy they use to arbitrate among competing requests from both readers and writers. Our scheduler-conscious reader-writer lock (**RW-Smart-Q** in the figures) is based on a fair, scalable reader-writer lock devised by Krieger et al. [1993]. When a writer releases a lock for which both readers and writers are waiting, and the longest-waiting unpreempted process is a reader, the code grants access to all readers that have been waiting longer than any writers. (An alternative interpretation of fairness would grant access in the same situation to *all* currently waiting unpreempted readers.) Like the Smart-Q lock, the RW-Smart-Q lock uses all four values of context_block.state.

Requests for the lock are inserted in a doubly linked list. A reader arriving at the lock checks the status of the previous request. If the previous request is an active reader or if there is no previous request, then the newly arriving reader marks itself as an active reader and proceeds. In all other cases the newly arriving process spins, waiting to be released by its predecessor. A process releasing a lock must first remove itself from the queue. If the process is a writer this is an easy task, since it has no predecessor in the queue and since the procedure is similar to the one followed in the Smart-Q lock. If it is a reader however, then the process may have to remove itself from the middle of the queue (Figure 4). To ensure correct manipulation of the linked-list data structure, a reader process locks both its own list node and that of its predecessor. It then updates the link pointers to reflect the new state of the list. The locks protecting individual list elements use test_and_set. We have opted for this type of lock because (1) the critical sections are short and (2) the maximum

number of contending processes is three. When an unlocking reader acquires the test_and_set lock on its predecessor's list element, it double-checks the identity of the predecessor and backs out if it has changed. Change is possible if the predecessor removed itself from the list after the reader established the predecessor's identity, but before the reader managed to acquire the lock.

After a process has linked itself out of the queue, it must wake up its successor if there is one. The procedure is similar to the one followed in the Smart-Q lock. The releasing process checks the state of its successor and attempts to set that state to unpreemptable_other. If the attempt is successful, the releasing process proceeds to notify its successor that it has been granted the lock. If the attempt fails, it notifies its successor of failure by setting a flag in the successor's node and proceeds to the next process in the queue. When notified that it has been granted the lock, a reader uses this same procedure to release its own successor, if that successor is also a reader.

4.3 Barriers

In this section we present two scheduler-conscious barriers. The first is designed for bus-based multiprocessors, or for small partitions on larger machines, in which migration is assumed to be relatively inexpensive. It uses partition_block.num_processors to make an optimal spin-versus-block decision in each individual process. The second barrier is designed for large-scale multiprocessors, on which migration is assumed to be an expensive, uncommon event. This barrier makes optimal spin-versus-block decisions within each processor (or within each cluster of a machine in which migration is inexpensive among small sets of processors), uses a logarithmic-time scalable barrier across processors/clusters, and adapts dynamically to changes in the allocation of processes to processors or processors to applications.

Inspired by the work of Karlin et al. [1991] for locks, we have also experimented with small-scale (nondeterministic) barriers in which a process attempts to guess whether it is running in a multiprogrammed environment based on how long it had to wait during previous barrier episodes [Kontothanassis and Wisniewski 1993]. Our results for these barriers are mainly negative and are consistent with the findings of Lim and Agarwal [1993]: the performance of various spin-then-block heuristics is virtually indistinguishable from that of "always block." We include results for the simplest heuristic (**B-sp-blk**—spin for a while, then block) in the figures in Section 5.

The problem with heuristics is that they lead to a uniform policy for all processes: either all will spin, or all will block. On a small machine, our *Scheduler Information* barrier (**B-sched** in the figures) makes an optimal spin-versus-block decision in each individual process: on a system with N processes and P processors (and inexpensive migration) the first $N - P$ processes to reach the barrier will block while the remaining P will spin. P

is simply context_block.num_processors. N is easily built into the code. And since centralized barriers already keep a count of how many processes have arrived, each process can tell whether it should spin or block.

For large machines, our scalable scheduler-conscious barrier (**Scal-SC** in the figures) assumes that a process should block only if its processor could be used by another *local* process. Within each processor or bus-based cluster we use the Scheduler Information barrier. The last-arriving process in each processor/cluster then participates in a global instance of Mellor-Crummey and Scott's scalable tree barrier [Mellor-Crummey and Scott 1991a]. The trickiest part of the code copes with the possibility that the kernel may change the number of processors available to the application, or the mapping of processes to processors, during execution. If it does so, it will update the contents of the partition_block and in particular the generation count.

The barrier algorithm keeps a shadow copy of this generation count. The process at the root of the interprocessor barrier checks the shadow against the value in the partition_block. If the two are different, processes within each new processor/cluster elect a representative, and the representatives then go through a barrier reorganization phase, initializing tree pointers appropriately.[2] This approach has the property that barrier data structures can be reorganized only at a barrier departure point. As a result, processes may go through one episode of the barrier using outdated information. While this does not affect correctness it could have an impact on performance. If repartitioning were a frequent event, then processes would use old information too often, and performance would suffer. However, we consider it unlikely that repartitioning would occur more than a few times per second on a large-scale, high-performance machine, in which case the impact of using out-of-date barrier data structures would be negligible.

5. RESULTS

This section presents a performance evaluation of different preemption-safe and scheduler-conscious synchronization algorithms, including a comparison to the best-known scheduler-oblivious algorithms. We begin by describing our experimental methodology. We then consider mutual exclusion, reader-writer locks, and barriers in turn.

5.1 Methodology

We have tested our algorithms on two different architectures. For an example of a small-scale bus-based machine we use a 12-processor, 100MHz Silicon Graphics Challenge. For an example of a large-scale distributed-memory machine we use a 64-processor partition of a 20MHz Kendall Square KSR 1. We have used both synthetic and real applications.

[2]Note that the representative for the reorganization phase is not necessarily the process that will participate in the interprocessor phase of subsequent barriers; this latter role is played by the last process to arrive at each intraprocessor barrier.

The synthetic applications allow us to thoroughly explore the parameters that may affect synchronization performance, including the ratio between the lengths of critical and noncritical sections, the degree of multiprogramming, the quantum size, and others. The real applications allow us to validate our findings in the context of a larger computation, potentially capturing effects that are missing in the synthetic applications, and providing a measure of the impact of the synchronization algorithms on overall system performance. We have chosen applications that make heavy use of synchronization constructs to ensure that synchronization time is a significant portion of program runtime. For applications that make little use of synchronization constructs, we expect that the choice of the synchronization algorithm will have little effect on performance.

Our synchronization algorithms employ atomic operations not available on either of the two target architectures. We have implemented software versions of these instructions using load_linked and store_conditional on the Challenge and using small critical sections bracketed by the native synchronization primitive (get_subpage and free_subpage) on the KSR. Our approach for the KSR adds overhead to the algorithms, but this overhead is small. Moreover, because we are running scalable algorithms, in which processes use backoff or spin only on local locations, competition is essentially nonexistent for the critical sections that implement the "atomic" operations and does not result in any significant increase in overall levels of network and memory contention. Our results for the nonnative locks are therefore slightly higher in absolute time, but qualitatively very close in character, to what would be achieved with hardware-supported fetch_and_Φ instructions.

The *multiprogramming level* reported in the experiments indicates the average number of processes per processor. For the lock-based experiments, one of these processes belongs to the application program; the others are assumed to belong to other applications. For the barrier-based experiments, multiple application processes reside on each processor and participate in all the barriers. The reason for the difference in methodology is that for lock-based applications we are principally concerned about processes being preempted while holding a critical resource, while for barrier-based applications we are principally concerned about processes wasting processor resources while their peers could be doing useful work. Our lock algorithms are designed to work in any multiprogrammed environment; the barriers assume that processors are partitioned among applications.

For the sake of simplicity, we employ a user-level scheduler in our experiments. One processor is dedicated to running the scheduler. While the kernel interface described in Section 3 would not be hard to implement, it was not needed for our experiments, and we lacked the authorization to make kernel changes on the KSR. For the lock experiments, the scheduler simulates preemption by sending a signal to an application process. The handler for this signal spins on a flag that the scheduler process sets at the beginning of the process' next quantum. The time spent spinning is meant to represent execution by one or more processes belonging to other, unre-

lated applications. We would expect a kernel-level implementation of our mechanisms to provide performance indistinguishable from that of the experimental environment.

For the barriers experiments, we simulate multiprogramming by multiplexing one or more application processes on the same processor. Both the SGI and KSR operating systems allow us to do this by binding processes to processors. The centralized barrier experiments require process migration. On the SGI we can restrict processors (prevent processes from executing on them). Restricting a processor increases the multiprogramming level on the remaining processors. Processes are allowed to migrate among the unrestricted processors. The KSR operating system does not provide an analogue of the SGI restrict operation, so we were unable to control the number of processors available to migrating processes. For this reason we do not report results for the centralized barriers on the KSR.

In most respects we believe that performance results on a real implementation of our kernel extensions would be indistinguishable from those reported here; the scheduler itself does very little work and does it only once per quantum. The one exception arises in the lock experiments, where simulation of preemption via spinning in a signal handler fails to capture any delays due to loss of cache, TLB, or memory footprint during preemption. Since these effects are inherently dependent on the memory reference characteristics of whatever unrelated processes happen to be running on the machine, they would be difficult to model in any experimental setting.

5.2 Mutual Exclusion

We implemented ten different mutual-exclusion algorithms:

(1) *TAS-B:* A standard test-and-test_and_set lock with bounded exponential backoff. This algorithm repeatedly reads a central flag until it appears to be unset, then attempts to set it atomically in order to acquire the lock. On the SGI Challenge, this is the native lock, augmented with backoff. (The backoff bound is not critical. In our experiments, it is about 1000 cycles.)

(2) *TAS-B-PS:* The same as TAS-B, but avoids preemption in critical sections by using the Symunix kernel interface.

(3) *Queue:* The MCS list-based queue lock [Mellor-Crummey and Scott 1991a].

(4) *Queue-NP:* An extension to the MCS lock that avoids preemption in critical sections, also using the Symunix kernel interface. This algorithm does *not* avoid passing the lock to a process that has been preempted while waiting in line.

(5) *Queue-HS:* An extension to the Queue-NP lock that uses handshaking to ensure that the lock is not transferred to a preempted process. This is the first algorithm described in Section 4.1.

(6) *Smart-Q:* An alternative extension to the Queue-NP lock that uses all four values of context_block.state to obtain simpler code and lower

overhead than in the Queue-HS lock. This is the second algorithm described in Section 4.1.

(7) *Ticket:* The standard ticket lock with proportional backoff, but with no special handling of preemption in the critical section or the queue.

(8) *Ticket-PS:* A preemption-safe ticket lock with backoff, using the Symunix interface to avoid preemption in the critical section and handshaking to pass over preempted processes in line. This is the third algorithm described in Section 4.1.

(9) *Native:* A lock employing machine-specific hardware. This is the standard lock that would be used by a programmer familiar with the machine's capabilities. It does not incorporate backoff.

(10) *Native-PS:* An extension to the native lock that uses the Symunix interface to avoid preemption while in the critical section.

The Native lock on the SGI Challenge is a test-and-test_and_set lock implemented using the load_linked and store_conditional instructions of the R4400 microprocessor. The Native lock on the KSR 1 employs a cache line locking mechanism that provides the equivalent of queue locks in hardware. The queuing is based on physical proximity in a ring-based interconnection network, rather than on the chronological order of requests.[3] We would expect the Native-PS locks to outperform all other options on these two machines, not only because they make use of special hardware, but because the atomic operations in all the other locks are built on top of them. Our experiments confirm this expectation.

Our synthetic application executes a simple loop consisting of a critical section and a noncritical section. To prevent the critical section from becoming a bottleneck, we set the ratio of the lengths of the critical and noncritical sections on both machines to slightly less than the inverse of the maximum number of processors. Absolute quantum length (in cycles or microseconds) had no significant effect on performance. We therefore concentrate here on the remaining variables in the synthetic application: multiprogramming level and number of processors.

Figures 5 and 6 plot execution time of the synthetic application against multiprogramming level for a fixed number of processors (11 on the SGI and 63 on the KSR). On the SGI, the scheduling quantum is fixed at 20ms. To avoid serialization on the critical section, we set the execution time ratio of the critical and noncritical sections at 1:14. We used a random-number generator to vary the length of the critical section within a narrow range, to more closely approximate the behavior of real applications and to avoid possible lock-stepping of the different processes. The Queue, Queue-NP, and Ticket locks show the worst performance degradation, because processes queue up behind preempted peers. Preventing preemption in the critical section helps a little, but not much: preemption of processes waiting in the queue is the dominant problem.

[3]We use the KSR's gspnwt instruction in a loop, rather than gspwt. Counterintuitively, the latter does not perform well when there are more than a handful of contending processors.

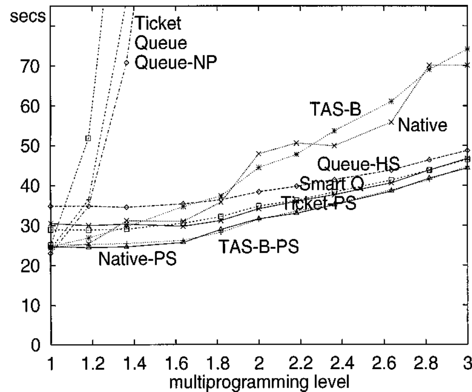

Fig. 5. Varying multiprogramming level on an 11-processor SGI Challenge.

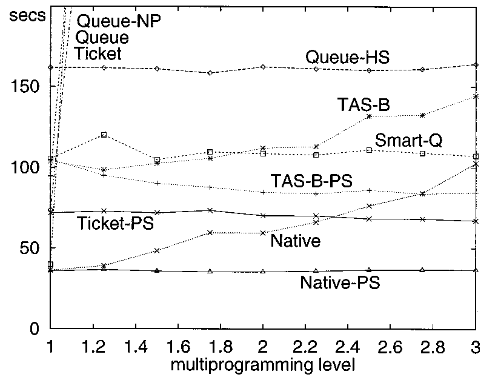

Fig. 6. Varying multiprogramming level on a 63-processor KSR 1.

Considerably 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 Queue-HS, Smart-Q, and Ticket-PS locks perform far better than the other scalable locks and outperform the Native and TAS-B locks at multiprogramming levels greater than 2. The Native-PS and TAS-B-PS locks display the best results, though past work [Mellor-Crummey and Scott 1991a] suggests that they will generate more bus traffic than the scalable locks and would interfere more with ordinary memory accesses in other processes or applications.[4]

On the KSR, the scheduling quantum is fixed at 50ms, and the ratio of critical to noncritical section lengths is 1:65. The results show slightly different behavior from that on the SGI. The Queue, Queue-NP, and Ticket locks suffer an even greater performance loss as the multiprogramming level increases. The Queue-HS lock improves performance considerably,

[4]The synthetic application does not capture this effect; it operates almost entirely out of registers during its critical and noncritical sections. The impact on data access traffic can be seen in our experiments with real applications.

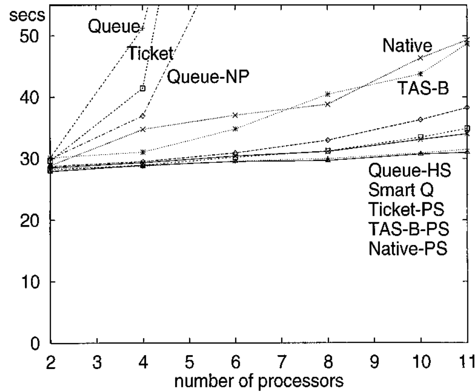

Fig. 7. Varying the number of processors on the SGI Challenge with a multiprogramming level of 2.

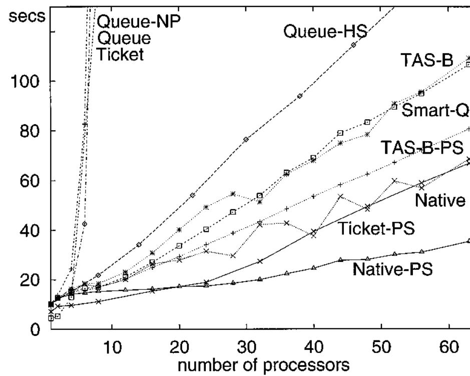

Fig. 8. Varying the number of processors on the KSR 1 with a multiprogramming level of 2.

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 it is still a little slower than the TAS-B-PS lock. The best nonnative lock is Ticket-PS.

The Native-PS lock provides the best overall performance. Since all the nonnative locks use native locks internally to implement atomic operations, this is expected behavior. The TAS-B and Native locks perform well when the multiprogramming level is low, but deteriorate as it increases. If the necessary atomic operations (fetch_and_add, swap, etc.) were available on the KSR 1, we would expect the queue and ticket locks to perform better than they do by a small constant factor. The closeness with which those locks follow the performance of KSR's relatively complex built-in primitive suggests that that primitive is probably not cost effective.

Increasing the number of processors 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

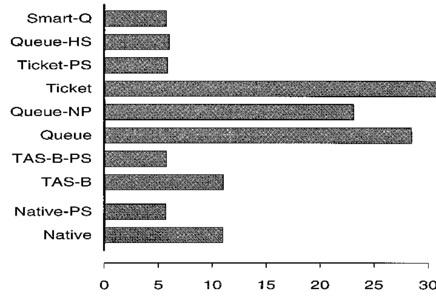

Fig. 9. Completion time (in seconds) for Cholesky on the SGI using 11 processors (multiprogramming level = 2).

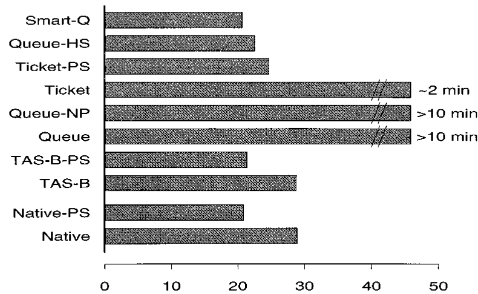

Fig. 10. Completion time (in seconds) for Cholesky on the KSR using 63 processors (multiprogramming level = 2).

performance in such an environment, but as indicated by the graphs in Figures 5 and 6 they experience difficulties under multiprogramming. The graphs in Figures 7 and 8 show the effect of increasing the number of processors on the different locks at a multiprogramming level of 2.

The synthetic program runs a total number of loop iterations proportional to the number of processors, so execution time should not decrease as processors are added. Ideally, it would remain constant, but contention and scheduler interference will cause it to increase. With quantum size and critical-to-noncritical ratio fixed as before, results on the SGI again show the Queue, Queue-NP, and Ticket locks performing poorly, as a result of untimely preemption. The performance of the TAS-B and Native locks also degrades with additional processors, because of increased contention. The Smart-Q and Ticket-PS locks degrade more slowly, but also appear to experience higher overheads. Increasing the number of processors does not affect the TAS-B-PS and Native-PS locks until there are more than about eight processors active (the point at which bus contention becomes an issue).

The results on the KSR indicate that contention effects are important for larger numbers of processors. The native lock, with our modification to avoid critical section preemption, is roughly twice as fast as the nearest competition, because of the hardware queuing effect. Among the all-software locks, Ticket-PS performs the best, but TAS-B-PS and Smart-Q are still reasonably close.

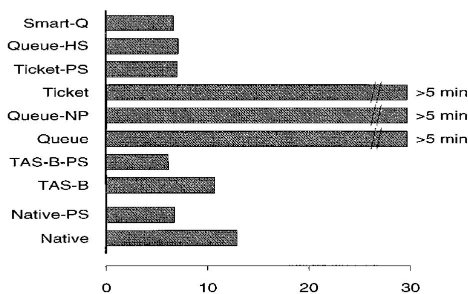

Fig. 11. Completion time (in seconds) for Quicksort on the SGI using 11 processors (multiprogramming level = 2).

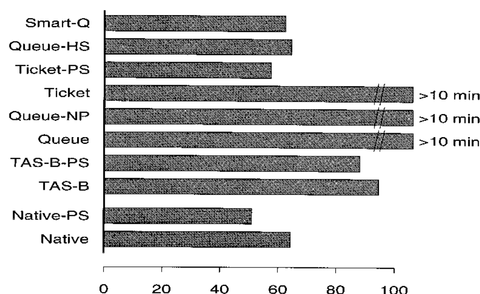

Fig. 12. Completion time (in seconds) for Quicksort on the KSR using 63 processors (multiprogramming level = 2).

Backoff constants for the TAS-B and Ticket locks were determined by trial and error. The best values differ from machine to machine, and even from program to program. The queue locks are more portable. As noted above, contention on both machines becomes a serious problem sooner if the code in the critical and noncritical sections generates memory traffic. Results from real applications indicate that the queue locks suffer less from this effect.

To verify the results obtained from the synthetic program, and to investigate the effect of memory traffic generated by data accesses, we measured the performance of three real applications: the Cholesky program from the Stanford SPLASH suite [Singh et al. 1992] running on matrix bccstk15, a multiprocessor version of Quicksort on 2 million integers, and a program that solves the traveling salesperson (TSP) problem for a 17-city fully connected graph. These programs contain no barriers; they synchronize only with locks. Due to lack of space we have omitted the graphs for TSP; its behavior is qualitatively similar to that of Quicksort. Figures 9 through 12 show the completion times for the remaining two applications, in seconds, when run with a multiprogramming level of 2 using 11 processors on the SGI and 63 processors on the KSR. As with the synthetic program, multiprogramming was simulated by spinning in a signal handler when other applications were supposed to be running. Again we see that scheduler-oblivious queuing of preemptable processes is disastrous. This time, however, with real computation going on, the Ticket-PS and Smart-Q

Table I. Latency (acquire + release) for Mutual Exclusion Locks on the SGI

Lock	Latency (μs)	
TAS-B	2.10	
TAS-B-PS	2.47	0.37 = 18%
Ticket	2.10	
Ticket-PS	2.87	0.77 = 36%
Queue	2.26	
Queue-HS	2.46	0.20 = 9%
Smart-Q	2.44	0.18 = 8%
Native	2.04	
Native-PS	2.39	0.35 = 17%

The extra numbers in the right-hand column indicate the absolute and percentage increase in latency of the preemption-safe and scheduler-conscious locks with respect to their scheduler-oblivious counterparts.

locks match the performance of the TAS-B-PS and Native-PS locks on the SGI and outperform TAS-B-PS in Quicksort on the KSR. We also ran experiments with a multiprogramming level of 1. Results (not shown) indicate that Quicksort and TSP run about 10% slower when using the Queue-HS lock than they do with the regular Queue lock. Otherwise, performance differences between applications with preemption-safe or scheduler-conscious locks and applications with the corresponding scheduler-oblivious locks were negligible.[5]

We have also collected single-process latency numbers (i.e., the time to acquire and release a lock in the absence of competition for the lock) to establish the performance overhead of the preemption-safe and scheduler-conscious algorithms with respect to their scheduler-oblivious counterparts.[6] Results appear in Table I. They were collected by having a single process acquire the lock repeatedly in a loop. As a result, they do not count the time required to bring the lock into the cache if it was most recently accessed by a different processor.

In their original scheduler-oblivious form, queue locks have roughly an additional 8% overhead over centralized (ticket or test-and-test_and_set) locks. Adding code to avoid inopportune preemption adds no more than 9% to the cost of the queue locks, but significantly raises the cost of the centralized locks, not only because they are faster to begin with, but also because they must pay the overhead of scheduler consciousness in all cases,

[5]In addition to a lock-protected work queue, TSP uses five atomic counters, which we implemented with fetch_and_add. Implementing them with critical sections instead dramatically increases the impact of synchronization on program runtime. In this case, TSP runs an additional 10% slower when using the Queue-HS or Smart-Q locks than it does with the regular Queue lock.

[6]At the time we collected the latency numbers, the KSR had been decommissioned, so we were able to collect them only on the SGI. However, since these are single-processor experiments we do not believe that the KSR numbers would have added anything significant to the understanding of the algorithms.

while the queue locks are able to skip most of the special-purpose code when the queue of waiting processes is empty.

5.3 Reader-Writer Locks

We implemented six different reader-writer locks:

(1) *RW-TAS-B:* A centralized reader-writer lock based on a standard test-and-test_and_set lock with exponential backoff.
(2) *RW-TAS-B-PS:* The same as RW-TAS-B, but with avoidance of preemption in critical sections, using the Symunix kernel interface.
(3) *RW-Queue:* A scalable reader-writer lock based on the lock by Krieger et al. [1993].
(4) *RW-Smart-Q:* An extension to the RW-Queue lock that uses all four values of context_block.state to avoid preemption in the critical section and to avoid passing the lock to a preempted process. This is the algorithm described in Section 4.2.
(5) *RW-Native:* A reader-writer lock based on the native synchronization primitive. On the SGI this is identical to the RW-TAS-B lock.
(6) *RW-Native-PS:* The same as RW-Native, but with avoidance of preemption in critical sections, using the Symunix kernel interface. On the SGI this is identical to the RW-TAS-B-PS lock.

Figures 13 and 14 show the performance of the various reader-writer locks under varying levels of multiprogramming on the SGI (11 processors) and KSR (63 processors), respectively. Figures 15 and 16 show performance on varying numbers of processors, at a multiprogramming level of 2.

Reader-writer locks display behavior similar to that of mutual exclusion locks. The RW-Native-PS lock outperforms all the others in a multiprogrammed environment. The RW-Smart-Q lock is a close second. The algorithms that do not cope with preemption behave increasingly worse as the multiprogramming level increases, though this effect is less pronounced than it was in the case of mutual exclusion. Five percent of the critical sections in the results reported here acquire a writer lock; the rest acquire a reader lock and can proceed in parallel with other readers. Preempting a process that holds a lock usually means preempting a reader, not a writer, so other readers can still proceed (so long as a writer is not yet in line). For completeness, we ran experiments with 1, 5, and 50% writers. Larger numbers of writers cause a higher degree of contention—expected since there is less concurrency available—and degrade the performance of the RW-TAS-B and RW-TAS-B-PS locks.

As in the case of mutual exclusion, the centralized RW-TAS-B and RW-TAS-B-PS locks still suffer from contention on large numbers of processors. Contention effects are more pronounced than they were for mutual exclusion. With the ratio of critical to noncritical work the same as in the mutual exclusion experiments, we expected that the additional

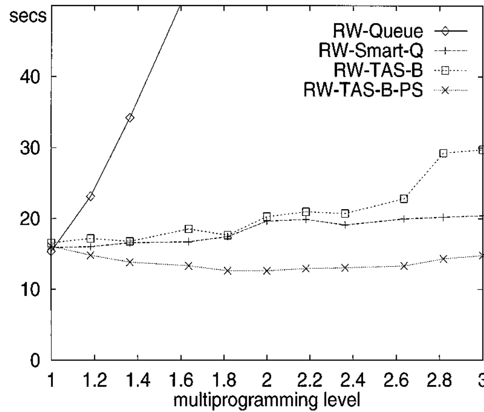

Fig. 13. Varying the multiprogramming level for the reader-writer lock on the SGI (11 processors, 5% writers).

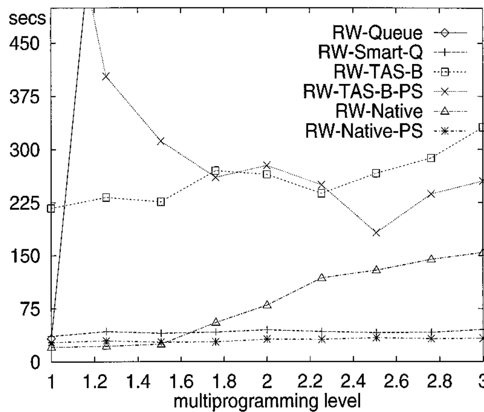

Fig. 14. Varying the multiprogramming level for the reader-writer lock on the KSR (63 processors, 5% writers).

parallelism available due to the concurrency of readers would reduce the observed contention, but this turned out not to be the case. Figure 14 shows that the centralized RW-TAS-B-PS lock actually improves in performance as multiprogramming increases. With fewer processes running in parallel, reductions in contention allow lock operations to complete faster, even though there are fewer total cycles available to the application per unit of time.

We have also collected single-process latency numbers for both the reader and writer parts of the locks. Results appear in Table II. They were collected by having a single process acquire and release the lock repeatedly in a loop. As a result, they do not count the time required to bring the lock into the cache if it was most recently accessed by a different processor. Most of the additional complexity of the preemption-safe and scheduler-conscious versions of the reader locks appears in the code for writers, rather than readers. Moreover, since the scheduler-oblivious overhead for

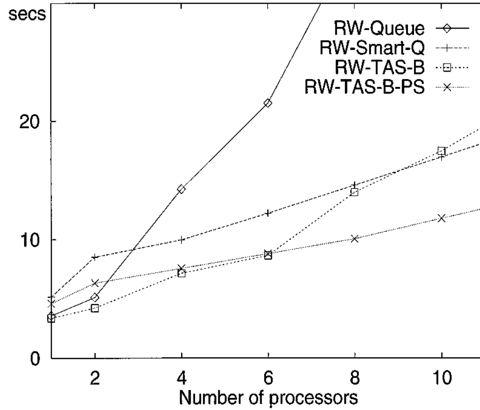

Fig. 15. Varying the number of processors for the reader-writer lock on the SGI (multiprogramming level = 2, 5% writers).

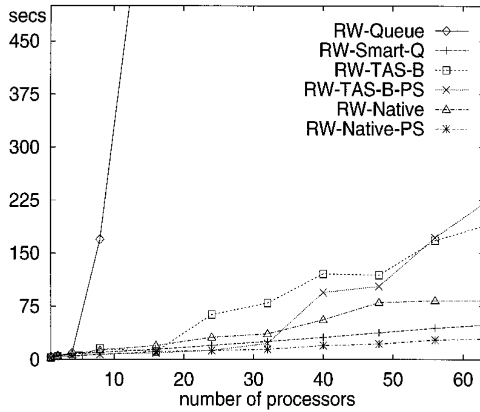

Fig. 16. Varying the number of processors for the reader-writer lock on the KSR (multiprogramming level = 2, 5% writers).

Table II. Latency (acquire + release) for Reader-Writer Locks on the SGI

Lock	R-Latency (μs)	W-Latency (μs)
RW-TAS-B	3.13	2.10
RW-TAS-B-PS	3.15 0.02 = 0%	2.52 0.42 = 20%
RW-Queue	5.28	2.21
RW-Smart-Q	5.48 0.20 = 4%	2.67 0.46 = 21%

The extra numbers in the right-hand columns indicate the absolute and percentage increase in latency of the preemption-safe and scheduler-conscious locks with respect to their scheduler-oblivious counterparts.

readers is already nearly 50% higher than the cost of a mutual exclusion lock, the percentage increase in latency for readers when moving to a preemption-safe or scheduler-conscious lock is insignificant. The increase in latency for writers is on the order of 20%.

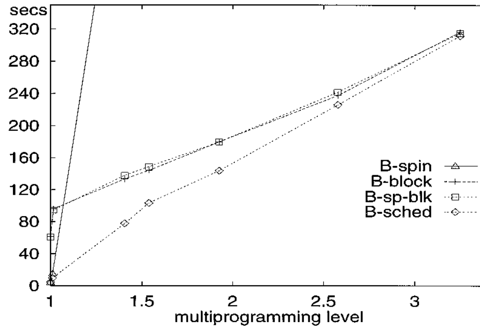

Fig. 17. Performance of the small-scale barriers for the synthetic program on the SGI (11 processors).

5.4 Barriers

We present results on barrier synchronization in two sections: one for small-scale machines such as the SGI Challenge (these results also apply to small partitions of a larger machine) and one for large-scale machines such as the KSR 1.

5.4.1 *Small-Scale Barriers.* For small-scale, centralized barriers, we report results for four different algorithms:

(1) *B-spin:* All processes spin while waiting for their peers to reach the barrier.
(2) *B-block:* Processes never spin; if they need to wait, they place themselves on a semaphore queue. The last process to arrive at the barrier wakes up its peers by performing V operations on the semaphore.
(3) *B-sp-blk:* Processes spin for a bounded amount of time equal to the cost of a context switch. If the bound expires before the barrier is achieved, then the process yields the processor by performing a P operation on a semaphore. The last process to arrive at the barrier checks the semaphore queue and wakes up any processes that are blocked.
(4) *B-sched:* The Scheduler Information barrier of Section 4.3, which makes an optimal spin-versus-block decision based on the number of available processors (partition_block.num_processors) and the number of processes that have yet to reach the barrier.

We also experimented with three heuristics that attempt to guess whether the machine/partition is multiprogrammed [Kontothanassis and Wisniewski 1993], but found their performance to be virtually indistinguishable from that of B-sp-blk.

Our synthetic barrier application is just a simple loop: all it does between barriers is increment a counter. Figure 17 shows the performance of this application on the SGI Challenge when using different barrier implementations, as the multiprogramming level increases. Our experiments assume a dynamic hardware-partitioned environment, where the number of proces-

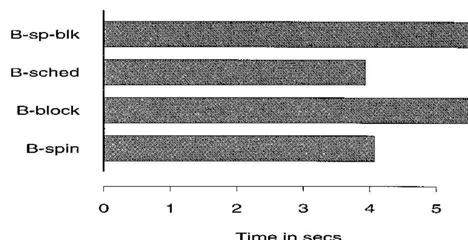

Fig. 18. Gaussian Elimination runtime for different barrier implementations (multiprogramming level = 1).

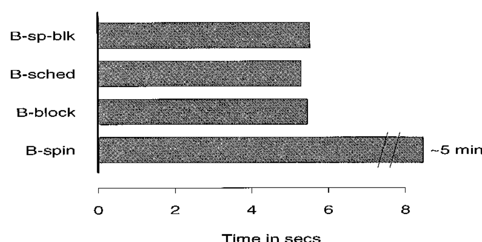

Fig. 19. Gaussian Elimination runtime for different barrier implementations (multiprogramming level = 2).

sors available to the application varies between 5 and 11, with an average of 7.9. The multiprogramming level is calculated based on the average number of processors and the number of processes used by the application. The kernel scheduler moves processes among processors in order to balance load. Processes running on the same processor are multiprogrammed with a quantum length of 30ms, the default value used by the IRIX kernel scheduler.

In the absence of multiprogramming, the B-sched barrier performs as well as the B-spin barrier—its overhead is low—and significantly better than B-sp-blk. As the multiprogramming level increases, the spinning barrier's performance degrades sharply, while the B-sched barrier retains its good performance. It never spins when other processes could make use of the current processor, and it avoids the overhead of blocking in the last P processes to arrive at the barrier. At very high multiprogramming levels, B-sched is only slightly faster than B-sp-blk and B-block: as the number of processes per processor increases it becomes less important to avoid blocking in the final process on each processor.

To validate the results obtained with the synthetic application, we experimented with two real applications: Gaussian Elimination and Successive Over-Relaxation (SOR). Gauss solves a 640×640 problem without pivoting; SOR works on an 800×800 matrix for 20 iterations. Both applications use 11 processes on 5–11 processors, with a quantum length of 30ms and a repartition operation (a random change in the number of processors) every 80ms.

The main difference we observed with respect to the synthetic results is a decrease in the impact of synchronization on overall performance, since it is

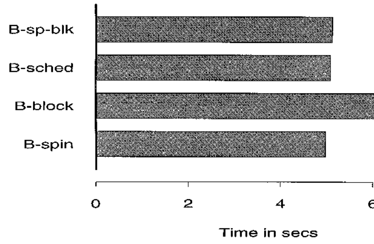

Fig. 20. Successive Over-Relaxation runtime for different barrier implementations (multiprogramming level = 1).

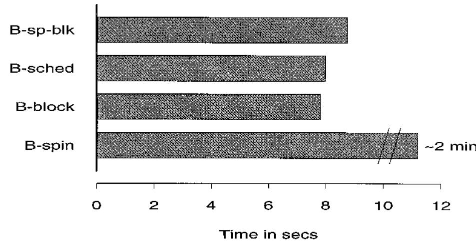

Fig. 21. Successive Over-Relaxation runtime for different barrier implementations (multiprogramming level = 2).

combined with the time spent in real computation. Figures 18 and 19 show the completion time of Gaussian Elimination at multiprogramming levels of 1 and 2 respectively; the corresponding results for SOR appear in Figures 20 and 21. Overall the B-sched barrier provides the best performance.

As with mutual exclusion and reader-writer locks, we experimented with a variety of other values for each of the experimental parameters. To save space, results are not shown here. The only parameter (other than multiprogramming level and number of processors) to display a noticeable impact on performance was the frequency of repartitioning decisions. As the time between repartitions increases, the performance of the heuristic barriers improves to some extent, since they need time to adapt to a change in partition size, and an increase in the time between repartitions allows them to amortize their adaptation cost over a larger number of episodes. The performance of the blocking and spinning barriers is essentially independent of the time between repartitions.

We were initially surprised to see a small but steady improvement in the performance of the B-sched barrier as the time between repartitions increased. The explanation is that it is possible for the algorithm to err when a repartitioning decision occurs at the same time that the application is going through a barrier. In this case, some threads will use old information to guide their decision and thus may decide suboptimally. When the time between scheduling decisions is large, suboptimal decisions happen less frequently, resulting in a small performance improvement.

5.4.2 *Scalable Barriers.* For large-scale machines we implemented and tested three barriers on the KSR:

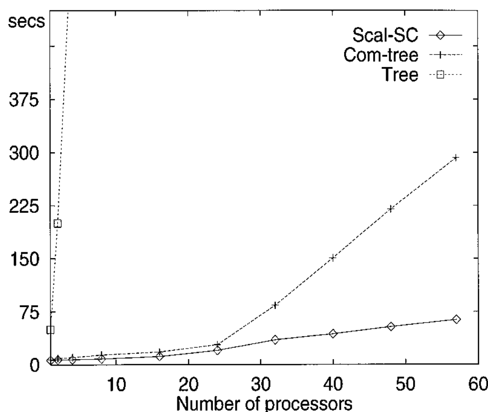

Fig. 22. Varying the number of processors for the barriers on the KSR (multiprogramming level = 2).

(1) *Tree:* Mellor-Crummey and Scott's tree barrier with flag wakeup [Mellor-Crummey and Scott 1991a] associates processes with nodes (both internal and leaves) in a static 4-ary fan-in tree. After waiting for their children (if any) and signaling their parent (if any) in the arrival tree, processes spin on locally cached copies of a single, global wakeup flag. The last arriving process sets this flag. On KSR's ring-based topology, the resulting invalidations and reloads approximate hardware broadcast.
(2) *Com-tree:* A heuristic variant of the Tree barrier, in which processes spin for only a bounded amount of time, as in the B-sp-blk algorithm of the previous section.
(3) *Scal-SC:* A scalable scheduler-conscious barrier that uses the B-sched barrier among the processes on a given processor and a scalable tree barrier across processors. This is the final algorithm presented in Section 4.3.

Barriers based on a centralized counter do not scale well to larger machines for two reasons. First, their critical path length is linear in the number of processors; second, the centralized counter can become a significant source of contention. Given that processes do not migrate among processors/clusters, the Scal-SC algorithm avoids these problems while making optimal spin-versus-block decisions.

Figure 22 compares the performance of the various barriers in our synthetic application on the KSR 1, with a multiprogramming level of two and with varying numbers of processors. Repartitioning decisions were made at one-second intervals. As can be seen from the graph, the Tree barrier is rendered useless with the introduction of multiprogramming. Its performance degrades due to the large number of context switches required in order to go through a barrier episode and the amount of time wasted before each context switch—equal to the scheduling quantum. It is surprising to see that even the "spin then block" heuristic of the Com-tree barrier

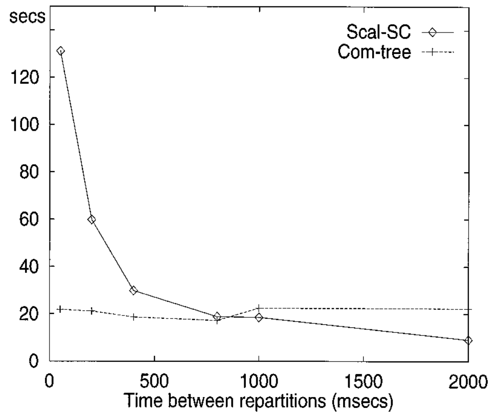

Fig. 23. Varying the frequency of repartitioning decisions for the barriers on the KSR (57 processors).

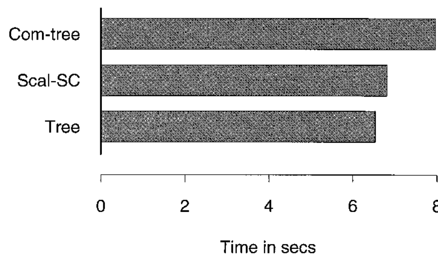

Fig. 24. Gaussian Elimination runtime on the KSR using 57 processors (multiprogramming level = 1).

performs quite badly in the presence of multiprogramming. While processes do not have to waste a quantum before yielding their processors they still have to suffer the large number of context switches that degrade performance. The Scal-SC barrier improves performance by an order of magnitude compared to the Com-tree barrier. It requires the minimum possible number of context switches, while still maintaining the logarithmic path length and low-contention properties of the Tree barrier.

As we mentioned in Section 4, the Scal-SC barrier can be sensitive to the frequency of scheduling decisions. We ran experiments to determine the level of sensitivity. Figure 23 shows that if the time between repartition decisions is very small, performance degrades quite sharply. We believe, however, that repartitioning will be a rare event on large machines—as rare as the arrival and departure of jobs from the system. For repartition intervals greater than 500ms, the Scal-SC barrier performs well.

To validate the synthetic results, we ran a barrier-based version of Gaussian elimination on 57 KSR processors.[7] The results appear in Figures

[7]We used pthreads to express parallelism in our barrier experiments. Due to limitations in the pthreads environment on the KSR, only 57 of the 64 processors in the partition could be utilized.

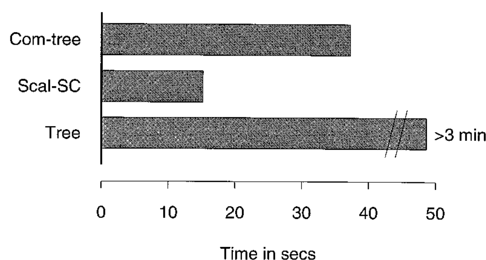

Fig. 25. Gaussian Elimination runtime on the KSR using 57 processors (multiprogramming level = 2).

24 and 25. In the absence of multiprogramming the Scal-SC barrier is only slightly worse than the Tree barrier and significantly better than the Com-tree barrier. The introduction of multiprogramming renders the Tree barrier useless; its performance degrades by at least an order of magnitude. At the same time, the Scal-SC barrier outperforms Com-tree by more than 50%.

6. RELATED WORK

6.1 Preemption-Safe Small-Scale Locks

It has long been recognized that performance can suffer greatly when a process is preempted while holding a lock. Ousterhout [1982] introduced *spin-then-block* locks that attempt to minimize the impact of preemption (or other sources of delay) in critical sections by having a waiting process spin for a small amount of time and then, if unsuccessful, block. Karlin et al. [1991] present and evaluate a richer set of spin-then-block alternatives, including *competitive* techniques that adjust the spin time based on past experience.[8] Their goal is to adapt to variability in the length of critical sections, rather than to cope with preemption. Competitive spinning works best when the behavior of a lock does not change rapidly with time, so that past behavior is an appropriate indicator of future behavior. Lim and Agarwal [1993] present a detailed analytical and experimental study of spin-then-block alternatives on a large-scale machine in which context switch times are relatively low compared to communication costs. In this environment, they find that blocking without spinning works reasonably well. Better results can be obtained by choosing a spin duration statically, based on analysis of the application.

Zahorjan et al. [1988; 1991] present a formal model of spin-wait times. They also describe the problem that occurs when a process is preempted while waiting in line for a queue-based lock. For lock-based applications in which all processes on a given processor belong to the same application, they show that performance problems can be avoided if the operating

[8]A competitive algorithm is one whose worst-case performance can be proven to be within a constant factor of optimal worst-case performance.

system limits its role to partitioning processes among the processors, allowing the application to make its own intraprocessor scheduling decisions (and never preempting a process with a lock).

Several groups have proposed extensions to the kernel-user interface that allow a system to avoid adverse scheduler/lock interactions while still doing scheduling in the kernel. The Scheduler Activation proposal of Anderson et al. [1992] 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 [Black 1990] allows a process to suggest to the scheduler that it be descheduled in favor of some specific other process, e.g., the holder of a desired lock. Both of these proposals assume that process migration is relatively cheap.

Rather than recover from untimely preemption, the Symunix system of Edler et al. [1988] and the Psyche system of Marsh et al. [1991] 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"[9] 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.

If we were building our algorithms on top of Psyche, we would replace code that sets the context_block.state variable to unpreemptable_self with code that blocks if the "two-minute warning" is in effect; we would delete code that resets context_block.state and blocks when context_block.warning is set. Rather than change another process' state variable to unpreemptable_other, we would inspect its warning flag and treat it as preempted if set. If we were building on top of Scheduler Activations, we would have the user-level scheduler that receives notice of kernel-initiated preemptions treat the state and warning variables just as the kernel does in our Symunix-based proposal; it would need to resume execution of any "unpreemptable" process. Alternatively, we could have the user-level scheduler perform some synchronization algorithm-specific recovery, such as linking a process out of a queue, but this introduces additional complexity, because with Scheduler Activations the user-level scheduler itself can be preempted.

6.2 Scalable Locks

The queue-based spin locks of Anderson [1990] and of Graunke and Thakkar [1990] minimize active sharing on coherently cached machines by

[9]In American football, a "two-minute warning" is a special game timeout called by the officials during the first and second halves of a game. It is designed to give either team fair notice that game time is about to expire.

arranging for every waiting processor to spin on a different element of an array. Each element of the array lies in a separate, dynamically chosen cache line, which migrates to the spinning processor. The MCS lock [Mellor-Crummey and Scott 1991a] represents its queue with a distributed linked list in which each process tracks the identity of its successor, rather than its predecessor. Because the list is "linked backward," each process spins on a location of its own choosing and can arrange for that location to lie in local memory even on machines without coherent caches. Magnussen et al. [1994] have shown how to modify the MCS lock to minimize interprocessor communication on a coherently cached machine. Lim has shown how to build a *reactive* lock that switches between test_and_set and MCS, depending on observed contention [Lim and Agarwal 1994]. Markatos [1991] uses a variant of the MCS queue to pass a lock to the highest-priority waiting process in real-time systems. Our Smart-Q lock first appeared at *IPPS* in 1994 [Wisniewski et al. 1994]. Craig [1993] and Takada and Sakamura [1994] have adapted this lock for use in real-time systems.

6.3 Alternative Approaches to Atomic Update

Alternatives to the use of preemption-safe or scheduler-conscious locks include *nonblocking* and *wait-free* data structures and remote invocation of object methods.

Herlihy [1991; 1993] has led the development of *nonblocking* and *wait-free* data structures. The algorithms are designed in such a way as to guarantee both atomicity and forward progress, despite arbitrary delays on the part of individual processes. The key idea in most cases is to modify a copy of (a portion of) the data structure and then swap it for the original in one atomic step (assuming the original has not been modified since the copy was created). Nonblocking algorithms guarantee that some process will make forward progress in a bounded number of time steps. Wait-free algorithms guarantee that *every* process will do so. Tolerance of arbitrary delays means that nonblocking and wait-free data structures are immune to the performance effects of inopportune preemption. It also means that they can tolerate some page faults and even certain kinds of hardware failure. Unfortunately, the current state of the art in general-purpose nonblocking and wait-free synchronization incurs substantial performance overhead, even when there is no competition for access to the data structure.

A second way to avoid the use of locks is to create a manager process that is responsible for all operations on the "shared" data structure and to require other processes to send messages to the manager. This sort of organization is common in distributed systems. It can be cast as a natural interpretation of monitors or as *function shipping* [Lindsay et al. 1984; Stamos and Gifford 1990] to a common destination. Several recent machines provide hardware support for very fast invocation of functions on remote processors [Kranz et al. 1993; Noakes et al. 1993]. Even on more conventional hardware, programming techniques such as *active messages*

[von Eicken et al. 1992] can make remote invocation very fast. Because computation is centralized and requests are processed serially, remote invocation provides implicit synchronization. On the other hand, it does not permit concurrency and can only be used when the manager is not a bottleneck.

6.4 Barriers

Small-scale barriers are generally based on centralized counters. As with centralized locks, contention for counters creates serious performance problems on large machines. Scalable barriers, generally based on log-depth tree or FFT-like patterns of point-to-point notifications among processes, have received significant attention [Arenstorf and Jordan 1989; Hensgen et al. 1988; Lee 1990; Lubachevsky 1989; Mellor-Crummey and Scott 1991a; Scott and Mellor-Crummey 1994; Yew et al. 1987].

Comparatively little work has addressed the interaction of scheduling and barriers, possibly because barrier-based applications tend to be used with a one-process-per-processor system model. Much of the work on spin-then-block techniques (e.g., that of Ousterhout [1982] and of Lim and Agarwal [1993]) applies to barriers as well as to locks. Lim and Agarwal note that waiting time at barriers could be reduced if the application knew how many of its processes had already reached the barrier (and presumably how many processors it was using), but they dismiss this possibility as being unreasonably communication intensive on a large-scale machine. Our scalable combination barrier tracks the number of processes at the barrier only within each processor or cluster.

With a scalable busy-wait barrier, Markatos et al. [1991] have shown that the scheduler may need to cycle through the entire ready list a logarithmic number of times (with a full quantum's worth of spinning between context switches) in order to achieve the barrier. To avoid this problem, they suggest (without an implementation) that blocking synchronization be used among the processes on a given processor, with a scalable busy-wait barrier among processors. Such *combination* barriers were originally suggested by Axelrod [1986] to minimize resource needs in barriers constructed from OS-provided locks. Our scalable combination barrier builds on these earlier proposals by using scheduler information to make an optimal spin-versus-block decision within processors and to adapt to partitioning changes at runtime.

7. CONCLUSIONS

In this article we presented solutions to the problem of synchronization on multiprogrammed multiprocessors, for both small and large-scale machines. We identified the main sources of performance loss for the two most common types of synchronization algorithms: locks and barriers. We also demonstrated that the scalable versions of synchronization algorithms based on distributed data structures are particularly sensitive to multiprogramming. Using a slightly extended kernel interface, in which processes

are able to defer preemption briefly, we examined several heuristic techniques—preemption-safe test_and_set locks (developed by previous researchers), "handshaking" queue and ticket locks, and various spin-then-block barriers—that avoid the worst performance anomalies in multiprogrammed systems.

To provide improved performance, we defined an extended kernel interface that allows a process to determine the state of its peers, inspect the mapping of processes to processors, and defer the preemption of peers. We used this interface to construct *scheduler-conscious* algorithms: the **Smart-Q** mutual-exclusion and reader-writer locks, the Scheduler Information small-scale barrier, and the scheduler-conscious scalable barrier. We demonstrated that these algorithms perform well on dedicated machines and provide significant performance advantages over their scheduler-oblivious counterparts in a multiprogrammed environment. The performance gains come from three sources: avoiding preemption when holding a lock, never passing a lock to a process that is currently preempted, and giving up the processor at a barrier when and only when some other process could use that processor productively.

For barrier-based applications, the Scheduler Information barrier and the scheduler-conscious scalable barrier clearly outperform both the heuristic spin-then-block barriers and the scheduler-oblivious alternatives. For lock-based applications the choice between preemption-safe centralized (test_and_set or ticket) locks and scheduler-conscious queue locks is less clear. Increasing the multiprogramming level decreases the contention observed by the application, since the number of processes accessing a synchronization variable concurrently is reduced. As a result, scheduler-conscious queue locks were often (though not always) inferior to the centralized alternatives in our experiments. As future increases in machine size increase the number of contending processors in multiprogrammed environments, the balance should tip back toward queue-based algorithms. Moreover, it is likely that coherence protocols on future machines will lack the ability to efficiently keep track of a large number of processors sharing a common variable. As a result, the cost of coherence management for the data structures of centralized synchronization algorithms is likely to be unacceptably high. This will again argue in favor of queue-based algorithms, in which no two processes spin on the same location.

As in previous papers [Mellor-Crummey and Scott 1991a; Scott and Mellor-Crummey 1994], we find that good performance can be achieved without exotic synchronization hardware. All of our algorithms can be constructed using a universal atomic primitive such as compare_and_store or load_linked/store_conditional. The closeness with which the various software queued locks follow the performance of the native lock on the KSR 1 suggests that the native lock may not be cost effective. For future machines, we suspect that barrier hardware will continue to prove worthwhile, but that hardware locks will not, unless they are very inexpensive. Moreover, any hardware synchronization mechanisms will need to be

carefully designed to maximize their flexibility and generality, e.g., for use on multiprogrammed systems.

Further extensions to the kernel-user interface might improve the performance of scheduler-conscious algorithms. Such extensions might include allowing the kernel to run user-supplied functions in response to particular kernel events or choosing the partition size based on the application's characteristics. For example, it does not make sense to remove a single processor from a 64-processor barrier-based application: the application would run almost as fast on 32 processors. We believe that as large-scale multiprocessors become more common they will inevitably be multiprogrammed, and the importance of exchanging information across the kernel-user boundary will increase.

ACKNOWLEDGMENTS

Our thanks to Hiroaki Takada and to Injong Rhee and Chi-Yung Lee for discovering timing windows in the Smart-Q algorithm. Thanks also to Donna Bergmark and the Cornell Theory Center for their help with the KSR 1, to Maged Michael and the anonymous referees for their careful reading and helpful suggestions, and to Ken Birman for pushing us a little when we needed it.

REFERENCES

ABOULENEIN, N. M., GOODMAN, J. R., GJESSING, S., AND WOEST, P. J. 1994. Hardware support for synchronization in the Scalable Coherent Interface (SCI). In *Proceedings of the 8th International Parallel Processing Symposium.* IEEE, New York, 141–150.

ANDERSON, T. E. 1990. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst. 1,* 1 (Jan.), 6–16.

ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. 1992. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst. 10,* 1 (Feb.), 53–79.

ARENSTORF, N. S. AND JORDAN, H. 1989. Comparing barrier algorithms. *Parallel Comput. 12,* 157–170.

AXELROD, T. S. 1986. Effects of synchronization barriers on multiprocessor performance. *Parallel Comput. 3,* 129–140.

BLACK, D. L. 1990. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Comput. 23,* 5 (May), 35–43.

CRAIG, T. S. 1993. Queuing spin lock algorithms to support timing predictability. In *Proceedings of the 14th IEEE Real-Time Systems Symposium.* IEEE, New York, 148–157.

CROVELLA, M., DAS, P., DUBNICKI, C., LEBLANC, T., AND MARKATOS, E. 1991. Multiprogramming on multiprocessors. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing.* IEEE, New York, 590–597.

EDLER, J., LIPKIS, J., AND SCHONBERG, E. 1988. Process management for highly parallel UNIX systems. In *Proceedings of the USENIX Workshop on Unix and Supercomputers.* USENIX Assoc., Berkeley, Calif.

GRAUNKE, G. AND THAKKAR, S. 1990. Synchronization algorithms for shared-memory multiprocessors. *IEEE Comput. 23,* 6 (June), 60–69.

HENSGEN, D., FINKEL, R., AND MANBER, U. 1988. Two algorithms for barrier synchronization. *Int. J. Parallel Program. 17,* 1, 1–17.

HERLIHY, M. 1993. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst. 15,* 5 (Nov.), 745–770.

KARLIN, A. R., LI, K., MANASSE, M. S., AND OWICKI, S. 1991. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles.* ACM, New York, 41–55.

KONTOTHANASSIS, L. AND WISNIEWSKI, R. 1993. Using scheduler information to achieve optimal barrier synchronization performance. In *Proceedings of the 4th ACM Symposium on Principles and Practice of Parallel Programming.* ACM, New York, 64–72.

KONTOTHANASSIS, L. I., WISNIEWSKI, R. W., AND SCOTT, M. L. 1994. Scheduler-conscious synchronization. Tech. Rep. 550, Computer Science Dept., Univ. of Rochester, Rochester, N.Y.

KRANZ, D., JOHNSON, K., AGARWAL, A., KUBIATOWICZ, J., AND LIM, B.-H. 1993. Integrating message-passing and shared-memory: Early experience. In *Proceedings of the 4th ACM Symposium on Principles and Practice of Parallel Programming.* ACM, New York, 54–63.

KRIEGER, O., STUMM, M., AND UNRAU, R. 1993. A fair fast scalable reader-writer lock. In *Proceedings of the 1993 International Conference on Parallel Processing.* CRC Press, Boca Raton, Fla., II: 201–204.

LEE, C. A. 1990. Barrier synchronization over multistage interconnection networks. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing.* IEEE, New York, 130–133.

LENOSKI, D., LAUDON, J., GHARACHORLOO, K., WEBER, W.-D., GUPTA, A., HENNESSY, J., HORWITZ, M., AND LAM, M. S. 1992. The Stanford Dash multiprocessor. *IEEE Comput. 25,* 3 (Mar.), 63–79.

LEUTENEGGER, S. T. AND VERNON, M. K. 1990. Performance of multiprogrammed multiprocessor scheduling algorithms. In *Proceedings of the 1990 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems.* ACM, New York, 226–236.

LIM, B.-H. AND AGARWAL, A. 1993. Waiting algorithms for synchronization in large-scale multiprocessors. *ACM Trans. Comput. Syst. 11,* 3 (Aug.), 253–294.

LIM, B.-H. AND AGARWAL, A. 1994. Reactive synchronization algorithms for multiprocessors. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, New York, 25–35.

LINDSAY, B. G., HAAS, L. M., MOHAN, C., WILMS, P. F., AND YOST, R. A. 1984. Computation and communication in R*: A distributed database manager. *ACM Trans. Comput. Syst. 2,* 1 (Feb.), 24–28.

LUBACHEVSKY, B. 1989. Synchronization barrier and related tools for shared memory parallel programming. In *Proceedings of the 1989 International Conference on Parallel Processing.* Penn State University Press, University Park, Pa., II:175–179.

MAGNUSSEN, P., LANDIN, A., AND HAGERSTEN, E. 1994. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Parallel Processing Symposium.* IEEE, New York, 165–171.

MARKATOS, E., CROVELLA, M., DAS, P., DUBNICKI, C., AND LEBLANC, T. 1991. The effects of multiprogramming on barrier synchronization. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing.* IEEE, New York, 662–669.

MARKATOS, E. P. 1991. Multiprocessor synchronization primitives with priorities. In *Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software.* IEEE, New York, 1–7.

MARSH, B. D., SCOTT, M. L., LEBLANC, T. J., AND MARKATOS, E. P. 1991. First-class user-level threads. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles.* ACM, New York, 110–121.

MELLOR-CRUMMEY, J. M. AND SCOTT, M. L. 1991a. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst. 9,* 1 (Feb.), 21–65.

MELLOR-CRUMMEY, J. M. AND SCOTT, M. L. 1991b. Scalable reader-writer synchronization on shared-memory multiprocessors. In *Proceedings of the 3rd ACM Symposium on Principles and Practice of Parallel Programming.* ACM, New York, 106–113.

NOAKES, M., WALLACH, D., AND DALLY, W. 1993. The J-machine multicomputer: An architecture evaluation. In *Proceedings of the 20th International Symposium on Computer Architecture.* ACM, New York, 224–235.

Ousterhout, J. K. 1982. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems.* IEEE, New York, 22–30.

Scott, M. L. and Mellor-Crummey, J. M. 1994. Fast, contention-free combining tree barriers. *Int. J. Parallel Program, 22,* 4 (Aug.), 449–481.

Scott, M. L., LeBlanc, T. J., and Marsh, B. D. 1990. Multi-model parallel programming in Psyche. In *Proceedings of the 2nd ACM Symposium on Principles and Practice of Parallel Programming.* ACM, New York, 70–78.

Singh, J. P., Weber, W.-D., and Gupta, A. 1992. SPLASH: Stanford parallel applications for shared-memory. *ACM SIGARCH Comput. Arch. News 20,* 1 (Mar.), 5–44.

Stamos, J. W. and Gifford, D. K. 1990. Remote evaluation. *ACM Trans. Program. Lang. Syst. 12,* 4 (Oct.), 537–565.

Takada, H. and Sakamura, K. 1994. Predictable spin lock algorithms with preemption. In *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software.* IEEE, New York, 2–6.

Tucker, A. and Gupta, A. 1989. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles.* ACM, New York, 159–166.

von Eicken, T., Culler, D. E., Goldstein, S. C., and Schauser, K. E. 1992. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture.* ACM, New York, 256–266.

Wisniewski, R. W., Kontothanassis, L., and Scott, M. L. 1994. Scalable spin locks for multiprogrammed systems. In *Proceedings of the 8th International Parallel Processing Symposium.* IEEE, New York, 583–589.

Yang, H.-H. and Anderson, J. H. 1993. Fast, scalable synchronization with minimal hardware support (extended abstract). In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing.* ACM, New York, 171–182.

Yew, P.-C., Tzeng, H.-F., and Lawrie, D. H. 1987. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Trans. Comput. C-36,* 4 (Apr.), 388–395.

Zahorjan, J. and McCann, C. 1990. Processor scheduling in shared memory multiprocessors. In *Proceedings of the 1990 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems.* ACM, New York, 214–225.

Zahorjan, J., Lazowska, E. D., and Eager, D. L. 1991. The effect of scheduling discipline on spin overhead in shared memory parallel systems. *IEEE Trans. Parallel Distrib. Syst. 2,* 2 (Apr.), 180–198.

Received January 1995; revised October 1995; accepted October 1996