PPoPP '95

High Performance Synchronization Algorithms for Multiprogrammed Multiprocessors

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

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

{bob,kthanasi,scott}@cs.rochester.edu

Abstract

Scalable busy-wait synchronization algorithms are essential for achieving good parallel program performance on large scale multiprocessors. Such algorithms include mutual exclusion locks, reader-writer locks, and barrier synchronization. Unfortunately, scalable synchronization algorithms are particularly sensitive to the effects of multiprogramming: their performance degrades sharply when processors are shared among different applications, or even among processes of the same application. In this paper we describe the design and evaluation of scalable scheduler-conscious mutual exclusion locks, reader-writer locks, and barriers, and show that by sharing information across the kernel/application interface we can improve the performance of scheduler-oblivious implementations by more than an order of magnitude.

1 Introduction

Most busy-wait synchronization algorithms assume a dedicated machine with one process per processor. The introduction of multiprogramming can cause severe performance degradation for these algorithms, especially for the variants designed to scale well on dedicated machines. Performance degrades in the presence of multiprogramming under the following circumstances:

- A process is preempted while holding a lock. This situation arises in both mutual exclusion and reader-writer locks when a process is preempted in its critical section. It has been addressed by several researchers [2, 4, 7, 15].

- A process is preempted while waiting for a lock and then is handed the lock while still preempted. This situation can arise in locks that enforce a predetermined ordering, either for the sake of fairness or to minimize contention on large-scale machines [21].

- A process spins waiting for its peers when some of them are preempted. This situation arises with locks, but is satisfactorily addressed by techniques that choose dynamically between spinning and yielding, based on observed lengths of critical sections [9]. A more severe version of the problem occurs with barriers, where the decision between spinning and yielding needs to be based not on critical section lengths, but on whether there are preempted processes that have not yet reached the barrier [10]. Scalable barriers exacerbate the problem by requiring portions of the barrier code in different processes to be interleaved in a deterministic order—an order that may conflict with the scheduling policy on a multiprogrammed processor, leading to an unreasonable number of context switches [14].

We have developed a simple set of mechanisms to handle synchronization difficulties arising from multiprogramming. The key idea is to share information

This work was supported in part by National Science Foundation grants numbers CCR-9319445 and CDA-8822724, by ONR contract number 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 no. MDA972-92-J-1012. Robert 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 Corporation, and other members of the Center's Corporate Research Institute. The Government has certain rights in this material.

across the application-kernel interface in order to eliminate the sources of overhead mentioned above. Specifically, the kernel exports the number of processors in the partition,[1] the state of each application process, and the identities of the processes on each processor. Each process indicates when it is performing critical operations and should not be preempted. Critical operations include portions of the synchronization algorithms themselves, and the critical sections protected by locks. Placing limits on when the kernel honors a request for disabling preemption allows the kernel to maintain control of the processor.

In previous work we used the concept of kernel/application information sharing to design small-scale scheduler-conscious barriers [10] and scalable scheduler-conscious locks [21] for multiprogrammed machines. In this paper we extend this work to encompass three important busy-wait synchronization algorithms that have not previously been addressed. Specifically we provide algorithms for:

1. A mutual-exclusion ticket lock. The ticket lock has lower overhead than a queued lock in the absence of contention, and scales almost as well when equipped with proportional backoff. It has constant space requirements and is fairer than a test-and-set lock. For multiprogrammed, large scale multiprocessors our results indicate that the ticket lock is the best-performing software mutual exclusion algorithm.

2. A queue-based reader-writer lock. The version presented in this paper provides fair service to both readers and writers. Versions that give preference to readers or to writers can also be devised using the same kernel interface.

3. A scalable tree-based barrier. This algorithm employs a centralized barrier within a processor and a tree barrier across processors. Our code uses kernel-provided information to identify the border between the centralized and tree portions of the barrier, to decide between spinning and blocking for the centralized portion, and to re-organize the barrier data structures when processors are repartitioned among applications.

The rest of the paper is organized as follows. Section 2 discusses related work. We describe our scheduler-conscious algorithms in section 3. Section 4 discusses performance results. Conclusions appear in section 5.

[1] Our barrier algorithm assumes that processors are partitioned among applications (i.e. that each processor is dedicated to a particular application), as suggested by several recent studies [6, 13, 22, 20]. Our lock algorithms work in a more general, time-shared setting; for them, the current "partition" is simply the set of processors on which the application's processes are running (or ran last if not currently running).

2 Related Work

Mutual exclusion algorithms in which many processes spin on the same location can incur large amounts of contention on scalable multiprocessors, degrading parallel program performance. Several researchers [1, 8, 16] have observed that the key to good performance is to minimize active sharing by spinning on local locations. Similar approaches have been adopted for more complex synchronization primitives, including barriers [16] and reader-writer locks [12, 17].

The efficiency of synchronization primitives depends in large part on the scheduling discipline used by the operating system. A growing body of evidence [6, 13, 20, 22] suggests that throughput is maximized by partitioning processors among applications. Unfortunately, if an application receives fewer processors than it has processes, the resulting multiprogramming can degrade performance by allowing processes to spin while their peers could be doing useful work. Several researchers have shown how to avoid preempting a process that holds a lock [7, 15], or to recover from such preemption if it occurs [2, 4]. Others have shown how to guess whether a lock is going to be held long enough that it makes sense to yield the processor, rather than busy-wait for access [9, 18, 23]. In previous work we have shown how to make an optimal decision between spinning and yielding in a small-scale centralized barrier [10]. We have also shown how to maintain good performance in a multiprogrammed environment for queue-based mutual-exclusion locks [21]. Other researchers [5, 19] have shown how to extend our work to real-time environments.

As noted in the introduction, scalable synchronization algorithms are particularly susceptible to scheduler-induced performance problems on multiprogrammed machines. They may give locks to preempted processes, spin at a barrier that has not yet been reached by a preempted peer, or force the scheduler through unnecessary context switches. Techniques that avoid or recover from preemption in critical sections, or that make spin versus yield decisions based on estimated critical section lengths do not address these problems.

3 Algorithms

In this section we present three scalable synchronization algorithms that perform well in the presence of multiprogramming. The first is a mutual-exclusion ticket lock that uses handshaking to detect preempted waiting processes and avoid giving them the lock. The second is a scheduler-conscious fair reader-writer lock based on the scheduler-oblivious code of Krieger, Stumm, and Unrau [12]. The third is a scheduler-conscious tree barrier. It incorporates our counter-based barrier [10] into

a two-level barrier scheme, and adjusts dynamically to the available number of processors in an application's partition. Two-level barriers employ a single counter on each processor, and a scalable barrier between processors. They were originally proposed by Axelrod [3] to minimize requirements for locks; Markatos et al. [14] first suggested their use to minimize overhead on multiprogrammed systems.

The ability to perform well in the presence of multiprogramming is a combination of intelligent algorithms and extensions to the kernel interface. We have extended the kernel interface in three ways:

1. The kernel and application cooperate to maintain a state variable for each process. This variable can have one of four values: `preemptable`, `preempted`, `self_unpreemptable`, and `other_unpreemptable`. `Preemptable` indicates that the process is running, but that the kernel is free to preempt it. `Preempted` indicates that the kernel has preempted the process. `Self_unpreemptable` and `other_unpreemptable` indicate that the process is running and should not be preempted. The kernel honors this request whenever possible, deducting any time it adds to the end of the current quantum from the beginning of the next. A process changes its state to `self_unpreemptable` before it attempts to execute critical-section code. Its peers can change its state to `other_unpreemptable` when handing it a lock. Synchronization algorithms can inspect the state variable to avoid such things as passing a lock to a preempted process. Most changes to the state variable are valid only for a particular previous value (e.g. user-level code cannot change a state variable from `preempted` to anything else). To enforce this requirement, changes are made with an atomic `compare_and_store`[2] instruction.

2. The kernel and application also cooperate to maintain a per-process Boolean flag. The kernel sets the flag whenever it wants to preempt the process, but honors a request not to do so. Upon exiting a critical section, the application should change its state variable to `preemptable` and then voluntarily yield the processor if the Boolean flag is set. (The kernel clears the flag whenever the process stops running.) These conventions suffice to avoid preemption during a critical section, provided that critical sections take less than one quantum to execute.

3. The kernel maintains a data structure that indicates the number of processors available to the application, the number of processes being scheduled on each processor, and the processor on which each process is running or is scheduled to run. It also maintains a *generation count* for the application's partition. The kernel increments this count each time it changes the allocation of processes to processors.

[2] `Compare_and_store(location,expected_value,new_value)` inspects the contents of the specified location and, if they match the expected value, overwrites them with the new value. It returns a status code indicating whether the overwrite occurred.

Extensions (1) and (2) are based in part on ideas introduced in Symunix [7], and described in our work on queued locks [21]. Extension (3) is a generalization of the interface described in our work on small-scale scheduler-conscious barriers [10]. None of these extensions requires the kernel to maintain information that it does not already have available in its internal data structures. Furthermore, the kernel requires no knowledge of the particular synchronization algorithm(s) being used by the application, and does not need to access any user-level code or data structures. Although we have run our experiments in user space as described in section 4, a kernel-level implementation of our ideas would not be hard to build.

The rest of this section describes the scheduler-conscious synchronization algorithms in more detail. Pseudocode for these algorithms can be found in a technical report [11] (it is omitted here to save space). C code for all algorithms is available via anonymous ftp from `cs.rochester.edu` in directory `/pub/packages/sched_conscious_synch` file `multiprogramming-sync-code.tar.Z`.

3.1 Ticket Lock

The basic idea of the ticket lock is reminiscent of the "please take a number" and "now serving" signs found at customer service counters. When a process wishes to acquire the 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 proportional to the difference between the last read value and the value returned by the `fetch_and_increment` of the "next available number" variable. To release the lock, a process increments the "now serving" variable.

The scheduler-conscious version of the ticket lock is implemented using a handshaking technique to ensure that the releaser and acquirer agree that the lock has passed from one to the other. After incrementing the "now serving" flag, the releaser waits for confirmation from the acquirer. If that confirmation does not arrive within a certain amount of time, it withdraws its grant of the lock, and re-increments the "now serving" flag in an attempt to find another acquirer.

For nested locks we need to make sure that releasing the inner lock does not reset the state variable to `preemptable`. We can achieve this by keeping track of the nest depth and clearing the state variable only when we release the outermost lock.

3.2 Reader-Writer Lock

Our scheduler-conscious reader-writer lock is fair in that it gives equal preference to both readers and writers. The algorithm uses the first two kernel extensions.

When a process attempts to acquire a lock it inserts itself into a doubly-linked queue. If the process at the tail of the queue is an active reader and the new process is also a reader, then the newcomer may continue; otherwise it will need to spin, on a local location, waiting for its predecessor to indicate that it may go.

When releasing the lock, a process removes itself from the queue. It then checks to see whether its successor is preempted. If a process finds that its successor is preempted, it links that successor out of the queue and sets a flag in the successor's queue record indicating that it needs to retry the lock when it wakes up.

To avoid race conditions when manipulating queue links, processes acquire and release mutual exclusion locks in the individual queue records. Because these are test_and_set locks in our code, they may lead to spinning on non-local locations on a non-cache-coherent machine. We could replace them with locks that spin only on local locations, but performance would suffer: the expected level of contention is low enough that minimum critical path lengths (especially in the expected case in which preemption does not occur) are the overriding concern. Nesting of reader-writer locks is handled in the same way as with the ticket-lock.

3.3 Scalable Barrier

Our scalable scheduler-conscious barrier uses the third kernel extension. Processes running on the same processor use the small-scale scheduler-conscious barrier described in previous work [10] to coordinate among themselves, spinning or yielding as appropriate. The last process arriving at the barrier on a given processor becomes the unique representative of that processor. In steady state (no re-partitioning), representative processes then participate in a scalable tree barrier [16].

In response to re-partioning in a dynamic multiprogrammed environment, the barrier goes through a data structure reorganization phase that allows it to maintain a single representative process per processor. This is accomplished using the partition generation counter provided by the kernel. We shadow this counter with a counter that belongs to the barrier. The process at the root of the inter-processor barrier tree checks the barrier generation counter against the partition generation counter. If the two counters are found to be different, processes go through a representative election phase based on the number of available processors. The elected representatives then go through a barrier reorganization phase, setting up tree pointers appropriately. 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 re-partitioning were an extremely frequent event, then processes would use old information too often and performance would suffer. However, it is unlikely that re-partitioning would occur more than a couple times per second on a large, high-performance machine.

4 Experiments and Results

We have tested our algorithms on a 12 processor Silicon Graphics Challenge and a 64 processor Kendall Square KSR 1 using both synthetic and real applications.[3] We have used the synthetic applications to thoroughly explore the parameter space, and the real applications to verify results and to measure the impact of ordinary (non-synchronization) references. Our results indicate that scheduler-oblivious algorithms suffer severe performance degradation when used in a multiprogrammed environment, and that the scheduler-conscious algorithms eliminate this problem without introducing significant overhead when used on a dedicated machine. Due to lack of space we have omitted graphs with the SGI results. A more thorough description of all the results can be found in a technical report [11]. The SGI graphs resemble the KSR graphs in terms of the comparison between scheduler-conscious and scheduler-oblivious algorithms, but favor centralized algorithms over the scalable alternatives, because the small number of processors reduces the level of contention.

Our synchronization algorithms require atomic operations not available on the SGI or KSR machines. We implemented a software version of these atomic instructions using the native spinlocks. This approach is acceptable (does not induce contention) as long as the time between high-level synchronization operations is significantly longer than the time spent simulating the execution of the atomic instruction. Since this is true for our experiments, our results are very close to what would be achieved with hardware `fetch_and_Φ` instructions.

[3] No application that uses reader-writer locks and exhibits a high degree of contention was available to us, so our reader-writer results all use a synthetic application.

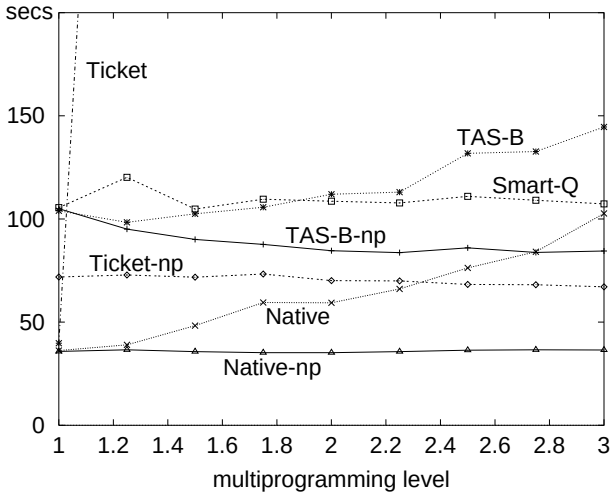

Figure 1: Varying multiprogramming level on a 63-processor KSR 1.

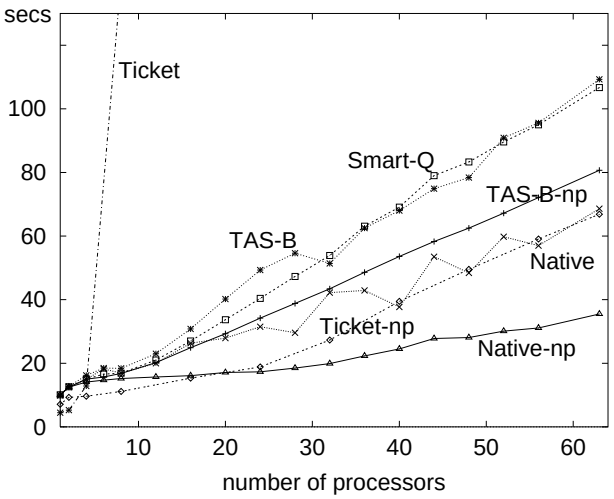

Figure 2: Varying the number of processors on the KSR 1.

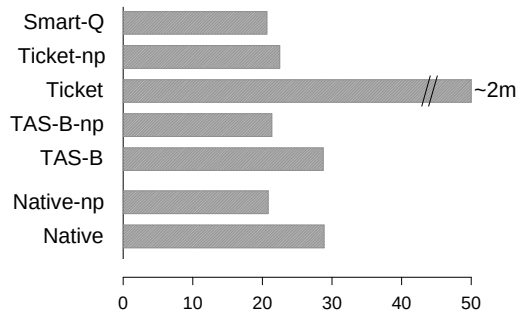

Figure 3: Completion time (in seconds) for `Cholesky` on the KSR 1.

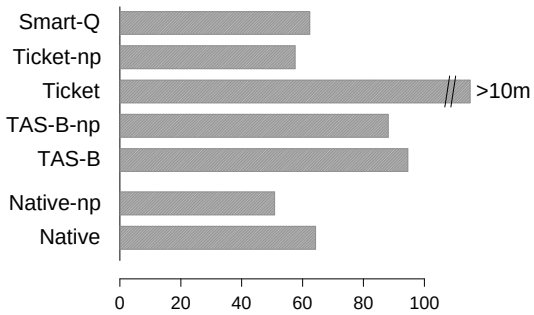

Figure 4: Completion time (in seconds) for `quicksort` on the KSR 1.

The *multiprogramming level* reported in the experiments indicates the average number of processes per processor. A multiprogramming level of 1.0 indicates one process on each processor. Fractional multiprogramming levels indicate additional processes on some, but not all, processors. For the lock-based experiments, one process per processor belongs to the tested application and the others are dummy processes assumed to belong to another application. For the barrier-based experiments, all the processes belong to the application, and participate in all the barriers. The principal reason for the difference in methodology for experiments with the two types of synchronization is that for lock-based applications we were principally concerned about processes being preempted while holding a critical resource, while for barrier-based applications we were 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 barrier assumes that processors are partitioned among applications.

In all the experiments, an additional processor (beyond the reported number) is dedicated to running a user-level scheduler. For the lock experiments, each application process has its own processor. Preemption is simulated by sending a user-defined signal to the process. The process catches this signal and jumps to a handler where it spins waiting for the scheduler process to indicate that it can return to executing application code. In the barrier environment there can be multiple application processes per processor. The scheduler uses primitives provided by the operating system to move processes to specific processors, or to make processors unavailable for process execution.

Processes in the lock-based synthetic programs execute a loop containing critical and non-critical sections. For the experiments presented in this paper, we ran 5000 iterations of the loop for a total of $5000 * 63 = 315000$ lock acquisition and release pairs. Critical to non-critical section work ratio is set to be inversely propor-

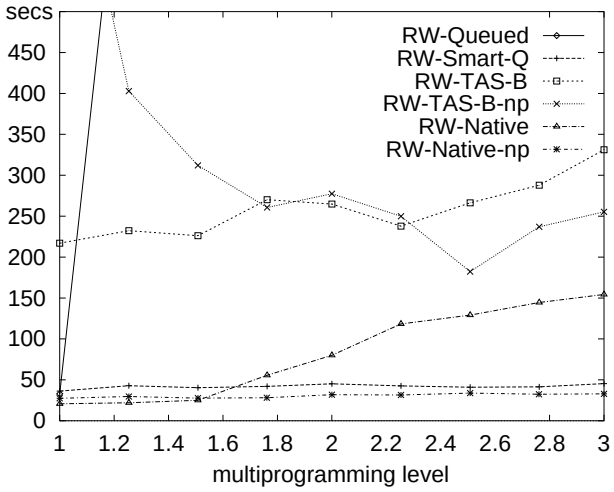

Figure 5: Varying the multiprogramming level on a 63-processor KSR 1.

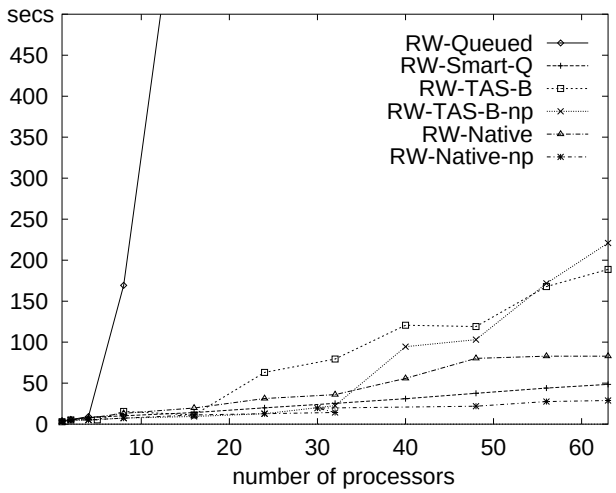

Figure 6: Varying the number of processors on the KSR 1

tional to the number of processors. Scheduling quantum was set to 50 ms on the KSR 1.

Figure 1 compares the performance of the scheduler-conscious ticket lock (**Ticket-np**) to that of several other locks on a 63-processor KSR 1 as the multiprogramming level increases. Figure 2 presents the analogous results for varying numbers of processors. The other locks include a plain (scheduler-oblivious) ticket lock (with proportional backoff) (**Ticket**), our scheduler-conscious queue-based lock (**Smart-Q**) [21], and both scheduler-conscious (**TAS-B-np**) and scheduler-oblivious (**TAS-B**) versions of a test_and_set lock with exponential backoff. We also include results for the native (hardware-implemented) spinlock (**Native**) and a modified version of the native lock that avoids preemption in the critical section (**Native-np**). The scheduler-conscious ticket lock provides performance improvements of more than an order of magnitude over the scheduler-oblivious version. Furthermore it is the best software lock for almost all multiprogramming levels. The ticket lock provides good tolerance for contention with modest overhead, properties that make it an ideal candidate for multiprogrammed environments with a limited amount of contention. Similar results were obtained for the real applications. Figures 3 and 4 show the execution time for Cholesky factorization for the `bcsstk15` matrix and a parallel version of quicksort for 2 million integers on a 63-processor KSR 1 with a multiprogramming level of 2.0.

Similar results to those obtained for mutual exclusion were seen in our experiments with reader-writer locks. We present results for a scheduler-conscious queue-based reader-writer lock (**RW-Smart-Q**), a scheduler-oblivious, queue-based reader-writer lock (**RW-Queued**), a preemption-safe centralized reader-writer lock (**RW-TAS-B-np**), a preemption-unsafe centralized reader-writer lock (**RW-TAS-B**), and preemption-safe and unsafe version of the reader-writer lock based on the native mutual exclusion lock (**RW-Native-np** and **RW-Native** respectively).

We have modified the synthetic program in the reader-writer locks to increase the amount of time spent in the critical section. We expected that due to the increased concurrency of readers, reader/writer locks would experience less contention than mutual exclusion locks. To our surprise we found that this was not case. Contention in reader/writer locks is just as important as in mutual exclusion locks. Our scheduler-conscious queued lock provides more than an order of magnitude improvement over the naive queued lock and is significantly better than the scheduler-conscious centralized (test_and_set based) version. The performance of the different reader/writer lock implementations can be seen in figures 5 and 6.

Processes in the barrier-based synthetic program execute a loop containing a barrier. The amount of work between barriers was chosen randomly in each process in each iteration, and varies uniformly between 240 μs and 290 μs. The scheduler re-partitions processors among applications once per second. With probability 1/2 the application is given all available processors. The other half of the time it receives 1/2, 1/3, or 1/4 of the available processors, with equal probability. For the experiments presented in this paper we timed 4000 iterations of the loop.

Figure 7 shows the performance of a set of alternative barrier implementations as the number of processors increases. The multiprogramming level in this experiment was set to 2.0. The four lines represent: (1) a standard arrival tree barrier in which processes al-

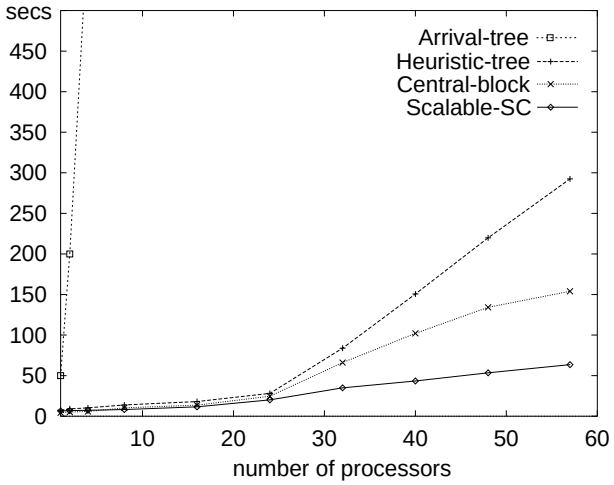

Figure 7: Barrier performance on the KSR 1 for different numbers of processors with a multiprogramming level of 2.

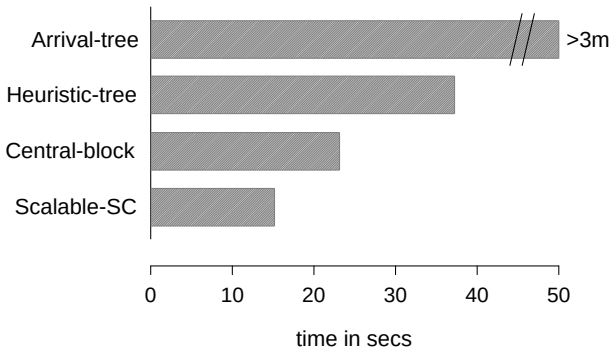

Figure 8: `Gauss` performance on the KSR 1 for different barrier types with a multiprogramming level of 2.

ways spin (until preempted by the kernel scheduler) (**Arrival-tree**); (2) a tree barrier that uses a simple spin_then_block heuristic to avoid wasting a quantum when there are preempted peers (**Heuristic-tree**); (3) a centralized barrier in which processes always yield the processor, rather than spin (**Central-block**); and (4) our scheduler-conscious two-level barrier (**Scalable-SC**). The first of these outperforms the others in the absence of multiprogramming [16], but performs terribly with multiprogramming, because a process can spin away the bulk of a scheduling quantum waiting for a peer to reach the barrier, when that peer has been preempted. The heuristic tree barrier improves performance significantly but still suffers from the large number of context switches that are required at every barrier episode. The centralized blocking algorithm provides even better performance by decreasing the number of context switches required at a barrier episode. Finally, the scheduler-conscious tree barrier provides the best performance: it combines the minimum number of context switches with the low contention and low (logarithmic) cost of the tree barrier. Experiments with a real barrier-based application (Gaussian elimination on a 640X640 matrix) confirm the observations obtained from the synthetic program. Figure 8 shows the relative performance of `gauss` on 64 processors and a multiprogramming level of two, for different barrier implementations.

We have also run experiments to study the impact of scheduling decision frequency on the performance of our barrier algorithms. When re-partitioning decisions are extremely frequent, our scheduler-conscious two-level barrier suffers significant performance degradation, because processes frequently base the structure of the inter-processor tree on out-of-date partition information. Performance is excellent, however, when re-partitioning decisions are as little as 500 ms apart.

5 Conclusions

This paper makes three primary contributions. First, it demonstrates that synchronization performance, especially for scalable algorithms, can degrade rapidly in the presence of multiprogramming. This can occur because a process is preempted in the critical section, is given a lock while preempted, or is spinning when its preempted peers could be doing useful work. Second, it describes mechanisms to solve these problems. Specifically, it proposes that the kernel export information about the processes and processors in each machine partition, and that the application indicate which of its processes are performing critical operations and should not be preempted. Third, it describes how to use this information to implement three scalable synchronization algorithms: a mutual exclusion ticket lock, a fair, queued, reader-writer lock, and a scalable barrier. All three algorithms perform significantly better than their scheduler-oblivious competitors.

Acknowledgement

Our thanks to Donna Bergmark and the Cornell Theory Center for their help with the KSR 1.

References

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

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

[3] T. S. Axelrod. Effects of Synchronization Barriers on Multiprocessor Performance. *Parallel Computing*, 3:129–140, 1986.

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

[5] Travis S. Craig. Queuing Spin Lock Algorithms to Support Timing Predictability. In *Proceedings of the Fourteenth IEEE Real-Time Systems Symposium*, Raleigh-Durham, NC, December 1993.

[6] Mark Crovella, Prakash Das, Cesary Dubnicki, Thomas LeBlanc, and Evangelos Markatos. Multiprogramming on Multiprocessors. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 590–597, December 1991.

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

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

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

[10] L. Kontothanassis and R. Wisniewski. Using Scheduler Information to Achieve Optimal Barrier Synchronization Performance. In *Proceedings of the Fourth ACM Symposium on Principles and Practice of Parallel Programming*, May 1993.

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

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

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

[14] Evangelos Markatos, Mark Crovella, Prakash Das, Cesary Dubnicki, and Thomas LeBlanc. The Effects of Multiprogramming on Barrier Synchronization. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 662–669, December 1991.

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

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

[17] J. M. Mellor-Crummey and M. L. Scott. Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors. In *Proceedings of the Third ACM Symposium on Principles and Practice of Parallel Programming*, pages 106–113, Williamsburg, VA, April 1991.

[18] B. Mukherjee and K. Schwan. Improving Performance by Use of Adaptive Objects: Experimentation with a Configurable Multiprocessor Thread Package. In *Proceedings of the Second International Symposium on High Performance Distributed Computing*, Spokane, WA, July 1993.

[19] Hiroaki Takada and Ken Sakamura. Predictable Spin Lock Algorithms with Preemption. In *Proceedings of the Eleventh IEEE Workshop on Real-Time Operating Systems and Software*, pages 2–6, Seattle, WA, May 1994. Expanded version available as TR 93-2, Department of Information Science, University of Tokyo, July 1993.

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

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

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

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