

Informing Algorithms for Efficient Scheduling of Synchronizing Threads on Multiprogrammed SMPs

Christos D. Antonopoulos^{1,*} Dimitrios S. Nikolopoulos^{2,*} Theodore S. Papatheodorou¹
¹ *High Performance Information Systems Laboratory* ² *Computers and Systems Research Lab*
Computer Engineering and Informatics Department *Coordinated Science Laboratory*
University of Patras *University of Illinois at Urbana-Champaign*
26500 Patras, GREECE *1308 West Main Str., Urbana, IL61801, U.S.A.*
{cda, tsp}@hpclab.ceid.upatras.gr dsn@hpclab.ceid.upatras.gr

Abstract

We present novel algorithms for efficient scheduling of synchronizing threads on multiprogrammed SMPs. The algorithms are based on intra-application priority control of synchronizing threads. We refer to such algorithms with the term "informing algorithms". Prerequisite for informing algorithms is the use of an efficient communication medium between the user- and kernel-level and the existence of in-kernel mechanisms that allow the applications to cooperate with the OS scheduler. The applications are given the opportunity to influence, in a non-intrusive manner, the scheduling decisions concerning their threads. We compare the performance of our informing algorithms with the performance of corresponding scheduler-oblivious algorithms under multiprogramming. We experimented on a small-scale, Intel x86-based SMP, running Linux, using both microbenchmarks and applications from the Splash-2 benchmark suite. The results substantiate the superiority of our approach and indicate that the philosophy of informing algorithms may be applicable in a wide range of algorithms and architectures.

1 Introduction

A typical class of applications suffering severe performance degradation in the presence of multiprogramming are multithreaded applications with frequently synchronizing threads [6]. It has been shown that the problem of poor performance under multiprogramming originates

mainly from the poor scheduling of synchronizing threads on physical processors [4], [6]. Idling time at synchronization points may constitute a significant fraction of execution time if one or more of the following conditions hold: **a)** a thread is preempted while holding a lock, **b)** a preempted lock-waiter thread remains preempted after it has been granted the lock by the previous holder, or **c)** a thread actively waits for other threads to reach a specific point (i.e. a barrier), while some of these threads are preempted.

The performance degradation is worse for synchronization algorithms designed to provide performance scalability on a large number of processors or fair arbitration between contending processors [6]. These algorithms introduce - either implicitly or explicitly - a strict ordering of synchronization operations performed by different threads. However, this ordering generally differs from the order threads are scheduled on physical processors by the OS scheduler.

The problem of scheduling synchronizing threads in a multiprogramming environment has not been adequately addressed, if at all, in contemporary commercial SMP schedulers for small- and medium-scale systems. In this paper, we introduce a kernel-level infrastructure of general applicability. This infrastructure arms multithreaded applications with mechanisms that allow them to control, in a non-intrusive manner, the execution of their own threads. In particular, each application assigns user-level priorities to its threads. These priorities reflect the importance of each thread for the progress of the application. The OS scheduler, on its turn, assigns physical processors to applications based on the scheduling policy imposed by the operating system designers. The priorities provided by the applications are taken into account during the allocation of physical processors to specific threads. The efficient communication between applications and the OS scheduler is facilitated by the use of memory pages, shared between user- and kernel-level [11], [12]. The user-level provided priorities have ap-

*This work has been supported by the Hellenic General Secretariat of Research and Technology (G.S.R.T.) research program 99-EΔ566.

†This work has been carried out while the second author was with the High Performance Information Systems Laboratory, University of Patras, Greece.

plication wide scope, thus they do not interfere with OS scheduling decisions concerning other applications. This guarantees that the fairness imposed by the OS scheduler is not affected. Moreover, this approach makes kernel-level mechanisms independent of and oblivious to the algorithm using them. The mechanisms can thus be used by diverse applications, provided that the latter are capable of quantifying the importance of each thread for the progress of the application. Taking advantage of this property, we have implemented six different informing synchronization algorithms using a common set of kernel mechanisms. Our algorithms can be used for intra-application synchronization only. Extending them to support inter-application synchronization would harm the non-intrusiveness property, given the fact that applications would be allowed to alter the user-level priorities of threads belonging to other applications.

The character of the informing synchronization algorithms presented in this paper is proactive. They guarantee that no lock-waiters or barrier-waiters will be granted a processor while there exist, in the same application, preempted lock-holders or threads that have not reached the barrier yet.

The target operating system in this study is Linux. We have modified the Linux kernel (version 2.2.15) in order to provide the general mechanisms needed by informing algorithms. Both informing synchronization algorithms and the corresponding scheduler-oblivious versions are implemented and provided as a run-time synchronization library.

The rest of this paper is organized as follows. Section 2 discusses related work. In section 3 we present details on the kernel-level design and implementation. In section 4 we describe the novel informing algorithms and their characteristics. In section 5 we present an experimental evaluation of both the novel algorithms and the corresponding scheduler-oblivious counterparts on an Intel x86-based SMP and section 6 concludes the paper.

2 Related Work

The performance degradation suffered by parallel programs with frequent synchronization among their threads, especially in the presence of multiprogramming, has been identified by several researchers in the past. In this section, we outline some of the proposed solutions.

The decision on whether a thread that waits at a synchronization point should actively spin, competitively spin (spin for a certain time-window and then block) or immediately block has been a hot-spot in research. In [5], A. Karlin *et al.* reach the conclusion that adaptive competitive spinning algorithms generally outperform static competitive spinning ones. The latter are considered, in their turn, better than actively spinning and immediate blocking algorithms. The evaluation has been carried out on the Firefly Multiprocessor. However, in [7] B. Lim and A. Agarwal reach a differ-

ent conclusion. Their experiments on the Alewife multiprocessor show that immediately blocking always sustains performance close to the best among all strategies compared. The contradiction of results is attributed to the fact that the context switch on Alewife is significantly faster than on Firefly, which benefits immediate blocking algorithms.

Another approach is the use of non-blocking synchronization algorithms. Such algorithms have by definition no critical sections, during which a thread preemption might cause performance problems. They are easily applicable to simple data structures and have proven to be quite efficient under multiprogramming [10]. Unfortunately, they are based on universal atomic primitives which can atomically change the contents of short (typically up to 64 bits), continuous memory areas. If the data structure is not that short, more complex algorithms must be used. These algorithms involve copying of the contents of the data structure and are generally inefficient.

There has been work in the past to provide synchronization algorithms that cooperate with the OS scheduler. In [6] L. Kontothanassis *et al.* present scheduler-conscious versions for a variety of algorithms. They assume that a thread can influence the scheduler enough to prevent itself from being preempted in a short critical section. It is the application's responsibility to cope with the decisions of the scheduler over longer periods of time. However, the application is given - even for short time periods - the opportunity to influence scheduling decisions, possibly at the expense of the other applications. Moreover, the scheduler-conscious algorithms alter basic properties of their scheduler-oblivious counterparts. For example, they do not preserve the FCFS property of several mutual exclusion algorithms.

In [3] D. Black proposes a priority scheme with user-level provided priorities for threads that contend for mutual exclusive access to atomic regions. These priorities have global scope, among all threads in the system and their role is to *a posteriori* confront with inopportune preemptions of lock-holder processes. The global scope of user-level provided priorities is unacceptable in a real, multiuser operating system, because it provides a means of overriding the standard scheduling policy thus harming fairness. Another similar *a posteriori* approach is presented by T. Anderson *et al.* in [1]. If a thread executing in a critical section is found preempted, it is resumed - via a user-level context switch - and executed until exiting the critical section. Finally, in [8] B. Marsh *et al.* introduce a "two minute" warning mechanism, used to avoid inopportune preemptions of threads executing code in a critical section. The threads are informed that they are going to be preempted and are granted a "grace" time period to exit the critical section. Barrier synchronization is not dealt with in any of the three previous approaches.

The Sun Solaris operating system supports a mechanism

which allows a thread to provide hints to the OS scheduler that it should, if possible, not be preempted [14]. This mechanism can be used for the implementation of weak preemption-safe locks. The Cellular IRIX [13] and Linux operating systems, on the other hand, provide competitive spinning synchronization primitives.

3 Kernel-Level Design and Implementation

The kernel-level implementation has been carried out in the 2.2.15 Linux kernel and originates from our previous work on the same operating system [11]. The applications declare that they are going to use the mechanisms we introduce via a specific system call at the beginning of their execution life. Kernel mechanisms are based on the existence and use of an efficient user- / kernel-level bidirectional communication interface (*Shared Arena*). The Shared Arena is implemented as a shared memory page between each application in the system that uses our mechanisms and the OS kernel. This implementation has the advantage of minimal communication overhead, equal to that of simple memory reads and writes. The applications use the Shared Arena in order to assign priorities to their threads. The kernel, on its turn, uses it to inform the applications on the number of processors allocated to them at any given time and on the exact state of their threads (running, preempted).

At the end of each time quantum or when a processor is idling, the operating system scheduler selects the most appropriate thread to be executed on that processor. This step of the selection process enforces the OS scheduling rules. If the selected thread belongs to an application which uses our mechanisms, a second level scheduler is invoked. That scheduler selects the thread with the highest priority, as specified from user-level, among the threads of the same application and allocates the processor to it. This policy guarantees that each application will be granted exactly the execution time it would be granted without our kernel extensions thus preserving fairness among applications. At the same time, it allows applications to use their cpu-time in a more effective way.

Every decrease of the user-level priority of a thread may result in a situation where higher priority threads are preempted. If this is the case, the thread hands off its processor in favor of the higher priority thread. This is achieved by executing a system call which practically initiates a new second level scheduling for the specific application.

The results of all scheduling decisions that affect applications using our mechanisms are communicated immediately to the user-level via updates of the values of the appropriate fields in the Shared Arena.

The user-level priority control is clearly non-intrusive, in the sense that an application can not affect the scheduling of other applications in the system. Furthermore, the mecha-

nisms are absolutely independent of the algorithms used at user-level. The OS kernel is not provided with any information and does not make any assumptions on the nature of the algorithm. This, makes our kernel mechanisms usable by a multitude of informing algorithms.

4 The Implementation of Informing Synchronization Algorithms

The novel synchronization algorithms we present take advantage of the kernel mechanisms described in section 3. They provide hints to the OS scheduler, via priorities assigned to threads from user-level, in order to improve the scheduling of synchronizing threads. The algorithms are representative of two of the most frequently used synchronization schemes in shared memory systems: mutual exclusion and barrier synchronization. We have implemented a simple test and set lock with local spinning (TTAS Lock), an array based queueing lock (Queue Lock), a variant of the list based queueing lock proposed by Mellor-Crummey and Scott (MCS Lock) [9], a ticket lock (Ticket Lock), a centralized incremental barrier (Incremental Barrier) and a tree barrier based on the algorithm proposed by Mellor-Crummey and Scott (MCS Barrier) [9].

The organization of the priority classes is based on the following rules: **a)** A thread that holds a lock can not be preempted in favor of any other thread. **b)** A thread that waits for a lock which is currently owned by another thread is not executed at the expense of a thread not participating in synchronization operations. The thread that waits for a lock will not execute any useful computation, while a thread that is not synchronizing with others probably will. The only exception are lock-waiters which will be granted the lock next, in algorithms that pose an ordering of lock acquires. In such cases, processor utilization can be sacrificed during short periods, in order to ensure that the thread will proceed to the execution inside the critical region as soon as it is handed the lock. **c)** A thread that has reached a barrier and waits for its peer threads to reach the barrier as well, is assigned lower priority than threads that do not participate in synchronization operations. A run-time system can not assume whether threads that currently execute computation without synchronizing will, at the end of the computation phase, participate in a barrier or not. This means that the priority of the latter can not be increased. The only possible alternative is to decrease the priority of threads that have already reached the barrier.

In our implementation we define and use five priority classes. The basic priority (RUNNING_PRIORITY) is the priority of threads that do not currently participate in any synchronization operation. The highest priority is assigned to threads that hold a lock (LOCK_HOLDER_PRIORITY). Threads that wait for a lock constitute the priority class

below basic priority (`LOCK_WAITER_PRIORITY`). If a mutual exclusion synchronization algorithm hands the lock to waiter threads in deterministic order, the priority of the thread to be granted the lock next is set to `LOCK_IMMEDIATE_WAITER_PRIORITY`. This priority class is lower than `LOCK_HOLDER_PRIORITY` and equal to, or higher than `RUNNING_PRIORITY`. Finally, threads that have reached a barrier and wait for their peers to reach the barrier too, constitute the weakest priority class (`BARRIER_WAITER_PRIORITY`).

Right before participating in a synchronization operation each thread saves the value of its user-level provided priority. The saved value is used to restore the priority right after the synchronization operation is over. This technique allows threads to participate in nested levels of synchronization.

The assignment of user-level provided priorities to threads is achieved, in most cases, with simple write instructions. However, a goal of our design is to eliminate the possibility of allowing, at any time snapshot, the priority assignment to be inconsistent with the actual status of the threads, in the context of synchronization operations. This might happen due to races, if two or more threads attempt, at the same time, to change a user-level priority. Such an inopportune priority assignment could result in performance inferior than the performance sustained by non-informing synchronization algorithms. In order to eliminate this possibility, we have identified the cases in which a race might occur and used atomic instructions for the priority update. In all cases, if there are contending threads they are exactly two and the atomic instruction is executed only once. One of the contenders will succeed and the other will fail and abort the operation. This means that the additional overhead on the memory subsystem is negligible.

Beyond informing algorithms, we have implemented two versions of scheduler-oblivious algorithms: an active spinning and an immediate blocking one. As mentioned in section 2, the decision whether a thread waiting at a synchronization point should actively spin, competitively spin or block is crucial for performance. Our algorithms make an optimal spin vs. block decision: a thread will block only if the application is granted less processors than its threads and there is a preempted thread of higher user-level priority. Competitive spinning strategies have not been evaluated. The context switch overhead for the system we have experimented on ranges from 3 to 22 μ sec, so we expect the performance differentiation among competitive spinning and immediate blocking algorithms to be marginal [7].

It must be noted that informing algorithms preserve - in contradiction with the scheduler-conscious algorithms of L. Kontothanassis *et al.* [6] - the main characteristics of their simple counterparts (time complexity, memory and network overhead, FCFS service of requests etc.). Their memory requirements are generally higher than those of simple algo-

rithms. However, they are - with the exceptions of Ticket Lock and Incremental Barrier - of the same complexity in respect to the number of synchronizing threads.

In the following subsection we describe the characteristics and implementation of the informing MCS Lock algorithm. The presentation of the remaining five informing synchronization algorithms, which has been omitted due to space limitations, can be found in [2].

4.1 MCS Lock

The MCS lock is a list based queueing lock, proposed by J. Mellor-Crummey and M. Scott [9]. It guarantees FIFO ordering of lock acquisitions. All spins are executed on locally-accessible flag variables only. It works equally well - in terms of network overhead - on machines with and without coherent caches. Each lock requires memory space proportional to the number of contending threads.

Each thread that attempts to acquire the lock inserts a node, which represents the thread, to the tail of a queue of waiting threads. The `next` field of the node, which is a pointer to the next node in the queue must be initialized to `NULL` prior to the insertion. The node insertion is achieved with an atomic `fetch_and_store` instruction. If the queue was previously empty, the thread is the lock-holder and proceeds to the critical section. If this was not the case, the thread sets the `locked` field of its node to true, to indicate that it is a lock-waiter and makes the `next` field of its predecessor node point to its node. The predecessor is the node returned by the atomic `fetch_and_store` instruction. Then, the thread waits (either spinning, or blocking) for its `next` field to become false.

In order to release the lock, the lock-holder initially checks if its node's `next` field points to a successor node. If there is no successor set, the lock-holder tries to atomically (`compare_and_swap`) set the tail of the queue to `NULL`. A success completes the lock release. A failure implies the existence of a thread which has inserted its node at the tail of the queue, but has not updated the `next` field of its predecessor yet. In this case, the lock-holder thread waits for the update to complete and then sets the `locked` field of the successor to false. If the initial examination of the `next` field indicates the existence of a successor, its `locked` field is simply set to false.

The pseudocode of an informing MCS lock is depicted in figure 1. The node of each thread is augmented with two additional fields: the thread identifier (`id`) and the field used for saving the initial user-level priority (`previous_priority`). A thread that attempts to acquire the lock initially saves its `id` and user-level priority to the corresponding fields of its node (line 5). It then sets the pointer to the next node equal to `NULL` (line 6) and inserts the node to the tail of the queue executing an atomic

```

1  mcs_lock_init(lock_queue) {
2      Initialize the head of lock_queue to NULL;
3  }

4  mcs_lock(lock_queue, my_node, my_id) {
5      Save both the previous user-level priority of my_id thread and
6      my_id to my_node.previous_priority and my_node.id;
7      Set my_node.next = NULL
8      Execute a fetch_and_store atomic instruction to insert my_node
9      to the tail of the lock_queue and get the previous value of the
10     tail (predecessor);
11     If the predecessor is not NULL {
12         Set my_node.locked = true to indicate that this thread is
13         not the lock owner;
14         If the predecessor is the lock owner {
15             Set the user-level priority of this thread to
16             LOCK_IMMEDIATE_WAITER_PRIORITY;
17             Put my_node in the queue after predecessor
18             (predecessor.next=my_node);
19         }
20         else {
21             Put my_node in the queue after predecessor;
22             If the thread did not in the meantime become the lock
23             owner, try to atomically (compare_and_swap) set its
24             user-level priority to LOCK_WAITER_PRIORITY;
25         }
26     }
27     While this thread is not the lock owner
28     (my_node.locked = true){
29         If the processors granted by the scheduler are less than
30         the threads of the application and if there is a
31         preempted thread with higher user-level priority than
32         this thread, hand off the processor to the higher
33         user-level priority thread;
34     }
35     This thread is now the lock owner: Set its user-level priority
36     to LOCK_HOLDER_PRIORITY;
37     If there is a successor of this thread's node in the lock_queue
38     (my_node->next <> NULL) {
39         Set its user-level priority to LOCK_IMMEDIATE_WAITER_PRIORITY;
40     }
41 }

42 mcs_unlock(lock_queue, my_id) {
43     If there is no successor node in the lock_queue
44     (my_node.next = NULL) {
45         If we are successful at atomically (compare_and_swap) changing
46         the lock_queue tail from pointing to my_node to NULL {
47             Restore the user-level priority of this thread using the
48             value previously saved in my_node;
49             If the processors granted by the scheduler are less than
50             the threads of the application and if there is a preempted
51             thread with higher user-level priority than this thread,
52             hand off the processor to the higher user-level priority
53             thread;
54             Return;
55         }
56         else {
57             Some node is trying to enter the queue as our successor.
58             Wait for it. In the meantime, set the user-level priority
59             of this thread to LOCK_IMMEDIATE_WAITER_PRIORITY (equal to
60             that of the thread to be inserted). If the granted
61             processors are less than the threads of the application
62             and there are preempted threads with higher user-level
63             priority, hand off the processor to them;
64         }
65     }
66     Set the priority of this thread to LOCK_HOLDER_PRIORITY again;
67     Set the priority of this thread's successor (succ_node) in the
68     lock_queue to LOCK_HOLDER_PRIORITY;
69     Set succ_node.locked to false to indicate that that thread is now
70     the lock owner;
71     Restore the user-level priority of this thread using the previously
72     saved value;
73     If the processors granted by the scheduler are less than the threads
74     of the application and if there is a preempted thread with higher
75     user-level priority than this thread, hand off the processor to the
76     higher user-level priority thread;
77 }

```

Figure 1. Informing MCS Lock pseudocode

fetch_and_store instruction. If there was a node in the wait-queue prior to the insertion, it is examined whether that node is the lock-holder or not. If the predecessor is the lock-holder, the user-level priority of the current thread is set equal to LOCK_IMMEDIATE_WAITER_PRIORITY and the node is inserted in the queue after the predecessor (predecessor.next = my_node, lines 10-13). If this is not the case, the thread places its node in the queue after the predecessor and tries to atomically set its user-level priority to LOCK_WAITER_PRIORITY (lines 14-17). In any case, after the priority assignment, the thread polls the

value of the locked field until the latter is found false. In the meantime, the thread may hand off its processor to other threads, should that be necessary (lines 18-20). When the thread is eventually granted the lock, it increases its user-level priority to LOCK_HOLDER_PRIORITY. If there is a successor node in the queue, the user-level priority of the corresponding thread is set equal to LOCK_IMMEDIATE_WAITER_PRIORITY.

During the lock release, the thread examines whether there is a successor node in the wait-queue (line 28). If there is not, the thread tries to atomically (compare_and_swap) set the tail of the queue, which currently points to its node, to NULL (line 29). If the atomic instruction is successful, the user-level priority of the thread is restored using the value saved in previous_priority. The thread then checks whether there are preempted threads of higher user-level priority and if this is the case it hands off the processor to one of them (lines 30-33). A failure of the atomic instruction (line 29) indicates the existence of a thread which has inserted its node at the tail of the wait-queue, but has not yet updated the next field of its predecessor. The lock-holder decreases its user-level priority to LOCK_IMMEDIATE_WAITER_PRIORITY and hands off its processor, in order to give the thread trying to enter the queue the opportunity to execute and update the next field of the lock-holder node (lines 34-36). If (or when) the lock-holder has a successor node, it restores its user-level priority to LOCK_HOLDER_PRIORITY and sets the priority of the successor to the same value (lines 38-39). Consequently, the locked field of the successor node is set to false, in order to grant the lock to the corresponding thread (line 40). Finally, the previous lock-holder restores its user-level priority using the value saved in previous_priority and hands off its processor in favor of another thread, should that be necessary (lines 41-42).

The use of an atomic instruction in line 16 helps eliminate a potential race hazard which would occur if the predecessor thread was granted the lock after the check. Should that happen, the predecessor would attempt to set the user-level priority of this thread to LOCK_IMMEDIATE_WAITER_PRIORITY. The next field of the predecessor must be updated prior to decreasing the priority (line 15). Doing the opposite could result to implications if the thread is preempted before updating the next field of the predecessor. If the predecessor thread is the lock-holder, its next field can be updated after the priority assignment, as its new priority (LOCK_IMMEDIATE_WAITER_PRIORITY) will be high enough to avoid deadlock. This raises the need of an atomic instruction for the priority assignment.

Failing to increase the priority of the lock-holder in the lock-release phase (line 38) could result to undesirable implications (even deadlock, if only one physical processor

has been granted to the application), if the lock-holder is preempted after the increase of the user-level priority of its successor. In such a case, the current lock-holder would not be given the opportunity to set the `locked` field of its successor to false, in order to allow the latter to enter the critical section.

5 Experimental Evaluation

In this section, we present an evaluation of the performance of informing synchronization algorithms. We are particularly concerned with their behaviour under multiprogramming. The comparison involves informing algorithms and their simple counterparts.

We have carried out our evaluation on a Compaq Proliant 5500 system. It is equipped with 4 Pentium Pro processors clocked at 200 MHz with 512 KBytes L2 cache each. The main memory is 512 MBytes. For the purposes of our evaluation we have used both synthetic microbenchmarks and real computational kernels and applications from the Splash-2 benchmark suite [15]. The workloads consist of one or more identical instances of a microbenchmark or an application. Each instance requires 4 processors and the number of concurrently executing instances is equal to the desired degree of multiprogramming. The range of multiprogramming degrees we have experimented with spans from 1 to 8.

The lock microbenchmarks consist of a main loop with alternating synchronizing and working phases. Only one thread can be in the synchronizing phase at any time snapshot. The synchronizing and working phases consist of 1000 and $1500 \pm 10\%$ locally cached memory updates respectively. The length of the main loop is 4096 iterations, which are evenly distributed among the processors. The microbenchmark is executed 32 times and the reported time is the mean value of all executions. The barrier microbenchmarks are identical to the lock microbenchmarks, with the difference that the main loop constitutes of the working phase, described earlier, followed by a barrier. The barrier is implemented using the algorithm under evaluation.

In order to demonstrate that the performance gains achieved by our informing synchronization algorithms can be considerable in real applications as well, we have used two computational kernels (LU, Cholesky) and one application (Radiosity) from the Splash-2 benchmark suite.

The LU kernel factors a dense matrix into the product of a lower and an upper triangular matrix. Blocking techniques have been used in order to exploit temporal locality and reduce communication. Approximately 20 to 50 percent of the execution time is spent on barrier synchronization operations [15]. We have decomposed a 1024×1024 matrix, using 16×16 blocks. The Cholesky kernel implements blocked, sparse Cholesky factorization. It factors

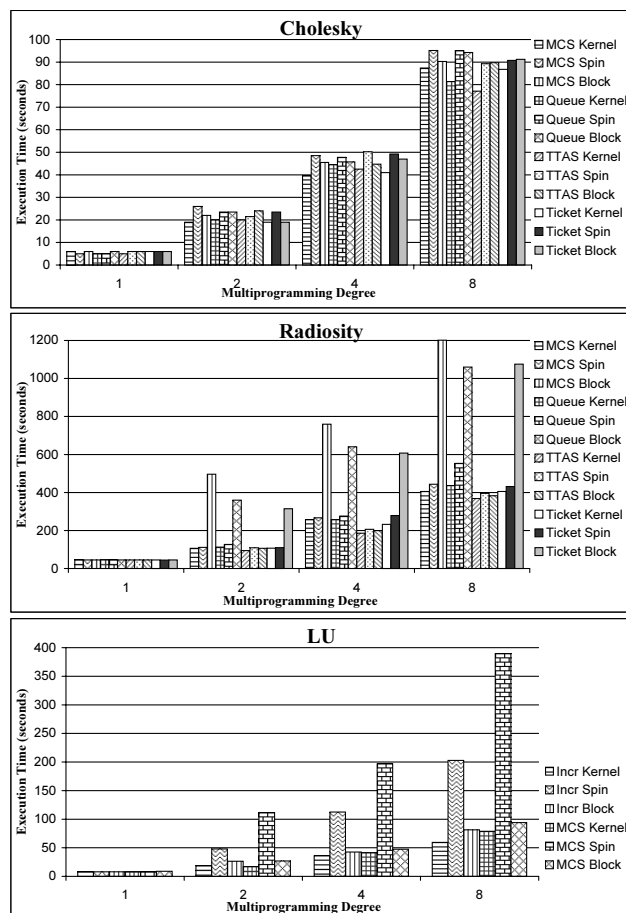


Figure 2. Execution times of Splash-2 applications using lock (Cholesky, Radiosity) and barrier (LU) algorithms

a sparse matrix to the product of a lower triangular matrix and its transpose. It is not globally synchronized between steps. The synchronization is achieved using locks and its overhead ranges between approximately 10 and 70 percent of the total execution time [15]. We have used the standard tk29.O input file, distributed with Cholesky. The Radiosity application computes the equilibrium distribution of light in a scene, using the iterative hierarchical diffuse radiosity method. The structure of the computation is highly irregular. The synchronization operations (locks) contribute approximately 15 to 30 percent to the execution time [15]. We have executed Radiosity in batch mode for the standard room scene. The results attained from the multiprogrammed execution of Splash-2 computational kernels and applications using informing (Kernel), active spinning (Spin) and immediate blocking (Block) synchronization algorithms are depicted in figure 2.

The microbenchmark results (not shown) indicate that,

for the target architecture, simple algorithms (TTAS lock and Incremental Barrier) significantly outperform more complex ones in the presence of multiprogramming. For non-informing synchronization algorithms this can be mainly attributed to the fact that simple algorithms do not impose a strict ordering in the synchronization operations executed by different threads, thus they are more insensitive to inopportune scheduling decisions. In the case of informing algorithms the execution time of the longer instruction sequences required by more complex algorithms contributes a dominant fraction of the total execution time, thus resulting to inferior performance in comparison to simple algorithms. The performance of complex and simple algorithms in a dedicated machine is almost indistinguishable. However, given the fact that complex algorithms are designed for scalability on many processors, we expect them to perform better than their simple counterparts in a larger, dedicated machine. In all cases but TTAS lock with multiprogramming degree 2, informing algorithms outperform scheduler-oblivious ones under multiprogramming. Our informing algorithms for locks execute up to 348.4 times faster (average 177.6 times faster) in comparison to spinning algorithms and up to 8 times faster (average 5.2 times faster) in comparison to immediate blocking ones. The speedup attained for informing barrier algorithms when compared to spinning ones is up to 1356.4 (average 512.4). The comparison with immediate blocking algorithms leads to speedups up to 10.5 (average 4.8). These performance gains are indicative of the correctness of our approach. We believe that, in the presence of multiprogramming degree 2, the always spinning version of the TTAS lock performs better (0.35 sec/16.7%) than the informing TTAS lock because the contention is not high enough for the benefits of our mechanisms to outweigh the extra cost of additional instructions, context switches and cache misses due to thread migrations. Moreover the informing TTAS lock can not fully benefit from our mechanisms, due to its non deterministic nature.

In a dedicated machine, the microbenchmarks that use informing algorithms may need up to 0.07 seconds (6.89%) more to execute than those implemented with simple always spinning algorithms. In an unloaded machine, informing synchronization algorithms degenerate to always spinning algorithms with the cost of some additional instructions in the critical path. These additional instructions might be considered responsible for the - even minimal - slowdown.

It is also worth pointing out that for low or no multiprogramming, the execution times of microbenchmarks using immediate blocking algorithms are worse than those of microbenchmarks using always spinning and informing synchronization algorithms. In the presence of low, or no contention the unnecessary reschedulings and context switches caused by immediate blocking algorithms have a negative effect on performance. The problem is practically elim-

inated with informing algorithms because of the optimal spin vs. block decision these algorithms reach. However, as the multiprogramming degree raises, the highest possible processor utilization is of vital importance and immediate blocking algorithms outperform always spinning (but not informing) ones.

The results from the Splash-2 experiments indicate that the performance benefits attained by informing synchronization algorithms are quite sound in real applications as well. In the absence of multiprogramming, informing synchronization algorithms have practically indistinguishable performance from scheduler-oblivious ones. However, in the presence of even minimal multiprogramming, informing algorithms perform much better. The performance of Cholesky (lock-based synchronization) is up to 36.8% (average 16.7%) higher when informing lock algorithms are used instead of always spinning ones. The comparison with immediate blocking lock algorithms yields maximum and average improvement of 20% and 11% respectively. Radiosity, which also uses lock-based synchronization, executes up to 26.8% faster (average 10.7%) if informing lock algorithms are used instead of always spinning ones and up to 4.7 times faster (average 2.5 times faster) in comparison with immediate blocking algorithms. In LU, threads synchronize using barriers. If informing barriers are used instead of always spinning ones, LU needs up to 6.5 times less execution time (average 4.2 times less). An immediate blocking implementation results to performance up to 58.8% (average 31.9%) worse than that of an informing implementation.

Due to differences in our platform and implementation framework, a direct one-to-one comparison of informing synchronization algorithms with the scheduler-conscious algorithms presented in [6] has not been possible. However, an indirect qualitative comparison of the results shows that informing synchronization algorithms provide improvements of similar or wider margin than those provided by scheduler-conscious synchronization. This can be attributed to the fact that informing algorithms control thread scheduling more effectively. The scheduler-conscious algorithms of Kontothanassis *et al.* try to avoid preemption of the lock-holder at the expense of all threads in the system. In order to eliminate the chance of malicious exploitation of this feature the preemption is avoided within short time-windows only. This makes scheduler-conscious algorithms prone to inefficiencies if the critical regions are not short-enough. Informing algorithms favor the lock holder at the expense of other threads of the same application only. This technique averts the danger of malicious exploitation, so the need for a time-window is raised. Moreover, informing algorithms preserve, as opposed to scheduler-conscious ones, the fairness characteristics of synchronization algorithms, reducing thus the possibility of starvation. Finally, the optimality

of the spin vs. block decision provided by informing algorithms can have significant impact on the performance, especially for barrier algorithms.

6 Conclusions

In this paper we have presented a generally applicable kernel-level infrastructure, which can be used to improve significantly the performance of applications in the presence of multiprogramming. Our mechanisms allow applications to use the processor time they are granted by the OS scheduler more effectively. This is achieved by giving applications the opportunity to inform the scheduler on the relative importance of their threads, by assigning them priorities from user-level. These priorities are taken into account by the OS scheduler in order to decide which specific thread of an application will execute on a processor. The priority control mechanisms are non-intrusive, in the sense that they do not allow applications to improve their performance at the expense of other applications in the system. The kernel infrastructure is general and does not depend on the nature of the application, thus it can be used by many different algorithms. As a proof of concept, we have implemented six different informing synchronization algorithms, i.e. algorithms that use the common kernel mechanisms. We have presented in detail one of them.

In order to evaluate the performance of informing synchronization algorithms under multiprogramming we have experimented with synthetic microbenchmarks and computational kernels and applications from the Splash-2 suite. Informing synchronization algorithms perform significantly better than non-informing ones in multiprogrammed environments and have practically negligible overhead in the absence of multiprogramming. Moreover, the comparative evaluation of non-informing algorithms has driven us to the conclusion that simple, immediate blocking synchronization algorithms, like the TTAS lock or the Incremental Barrier, perform quite well on small-scale machines, even in the presence of multiprogramming. However, even these simple algorithms can benefit from our kernel mechanisms.

We believe that the usage of an infrastructure which allows non-intrusive user-level priority control is not necessarily limited to small-scale SMPs. In any case, the applicability of such mechanisms in larger-scale systems requires thorough investigation, given the fact that such systems generally have different architectural characteristics and may pose significantly higher thread migration and remote memory access costs.

References

[1] 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.
- [2] C. D. Antonopoulos, D. S. Nikolopoulos, and T. S. Papatheodorou. Informing algorithms for efficient scheduling of synchronizing threads on multiprogrammed smps. Technical Report HPCLAB-TR-220101, High Performance Information Systems Laboratory, University of Patras, Patras, Greece, January 2001.
- [3] D. Black. Scheduling support for concurrency and parallelism in the mach operating system. *IEEE Computer*, 23(5), 1990.
- [4] A. Gupta, A. Tucker, and S. Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 120–132, 1991.
- [5] A. Karlin, K. Li, M. Manasse, and S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the 13th ACM Symposium on Operating System Principles (SOSP'91)*, pages 41–55, oct 1991.
- [6] L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott. Scheduler-conscious synchronization. *ACM Transactions on Computer Systems*, 15(1):3–40, February 1997.
- [7] B.-H. Lim and A. Agarwal. Waiting algorithms for synchronization in large-scale multiprocessors. *ACM Transactions on Computer Systems*, 11(3):253–294, August 1993.
- [8] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *13th ACM Symposium on Operating Systems Principles*, October 1991.
- [9] 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, 1991.
- [10] M. M. Michael and M. L. Scott. Relative performance of preemption-safe locking and non-blocking synchronization on multiprogrammed shared memory multiprocessors. In *Proceedings of the 11th International Parallel Processing Symposium*, April 1997.
- [11] D. S. Nikolopoulos, C. D. Antonopoulos, I. E. Venetis, P. E. Hadjidoukas, E. D. Polychronopoulos, and T. S. Papatheodorou. Achieving multiprogramming scalability on parallel programs on intel smp platforms: Nanothreading in the linux kernel. In *Proceedings of the PARCO99 Conference, Delft, Netherlands*, June 1999.
- [12] C. Polychronopoulos, N. Bitar, and S. Kleiman. Nanothreads: A user-level threads architecture. Technical Report CSR-1297, CSR, University of Illinois at Urbana-Champaign, 1993.
- [13] SGI. *Topics In Irix Programming, Chapter 4*. <http://techpubs.sgi.com>.
- [14] Sun Microsystems. *Solaris 8 Reference Manual Collection, Section 3: Threads and Realtime Library Functions*, March 2000. <http://docs.sun.com>.
- [15] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*, pages 24–36, June 1995.