An Efficient Recovery-Based Spin Lock Protocol for Preemptive Shared-Memory Multiprocessors

(Extended Abstract)

Injong Rhee
Department of Math and Computer Science
Emory University
Atlanta, GA 30322, USA
email: rhee@mathcs.emory.edu

Chi-Yung Lee
Department of Computer Science
University of Warwick
Coventry, CV4 7AL, UK

Abstract

Inappropriate preemption handling in shared-memory multiprocessors can severely degrade the performance of spin locks. However, previously proposed prevention-based preemption handling schemes tend to impose overhead on every lock acquisition and release whether or not a preemption occurs. This is because whenever entering an unsafe area (e.g., a critical section), processes need to notify the kernel not to preempt them.

This paper proposes a recovery-based spin lock protocol in which preemptions are allowed to happen at any time, and when an inopportune preemption that causes delay occurs, the preempted process is immediately “recovered” from the preemption. The advantage is that the overhead is incurred only when the inopportune preemption actually happens, but not during the normal operation.

Our experimental study shows that during the normal operation, the recovery-based spin lock imposes much less overhead than other prevention-based spin locks, and under a highly preemptive environment, its performance degrades, but much more slowly than that of the spin locks with no preemption handling facility. Interestingly, the statistical distribution of response times indicates that queue-based spin locks with either a prevention-based or a recovery-based preemption handling scheme are also suitable for real-time computing as they show a predictable performance.

1 Introduction

1.1 Background

Spin locks can provide efficient mutual exclusion in shared memory multiprocessor systems especially when critical sections are small, and processors do not have any other work to do, for the time overhead of blocking a process contending for a critical section can be larger than the time duration that the process spins. Many spin lock algorithms have been proposed [2, 4, 7]. However, most of these spin lock algorithms are designed for non-preemptive systems, in which at most one process is assumed to run on each processor. In more realistic settings, the number of processes exceeds the number of available processors, and preemption is often used to increase CPU utilization.

Unfortunately, some preemptions may cause a significant performance degradation for spin lock algorithms. These “inopportune” preemptions happen when a process is preempted while in a critical section. Then, all the spinning processes which have not been granted the lock also suffer from this preemption because the preempted process still holds the lock. The performance degradation due to preemption is even severer for queue-based spin locks, where processes enter a critical section in the order they join a queue. This is because a preempted process can be granted the lock, so that all the processes behind the preempted process in the queue also have to wait for the preempted process to be rescheduled. These preemptions also cause a severe variation in the times taken to acquire the lock, which could be intolerable, especially in real-time systems where a predictable performance is required.

Wisniewski, Kontothanassis and Scott [8, 9] first proposed several techniques to handle these inopportune preemptions, and applied them to various spin lock algorithms. In their techniques, whenever a process enters a critical section, it communicates with the kernel to prevent the kernel from preempting them in the critical section. Additionally, the preempted processes in the queue are skipped over at the time of lock release to prevent the lock from being granted to the processes. When the preempted processes wake up, they join the queue again. Their protocols are prevention-based because inopportune preemptions are “prevented” before they actually occur.

Although these prevention-based protocols can provide efficient synchronization when preemption is very

*This author is supported in part by NSF grant ASC-9527186.
frequent, they tend to impose overhead during the normal operation. This is because processes need to access (i.e., read from and write to) the kernel interface whenever they are in an unsafe area (e.g., critical section). Every lock acquisition and release incurs this overhead whether or not a preemption occurs. However, in most “well designed” systems, critical sections are small and lock contention is low [5, 1], and therefore inopportune preemptions are infrequent. Furthermore, in a system such as a high-performance user-level thread management where critical sections are considered the main building blocks [3], this overhead could constitute a large portion of the total overhead. It is, therefore, more desirable for such a system to have a preemption handling facility in which overhead is incurred only when inopportune preemptions actually happen, but not during the normal operation.

1.2 Our protocol

We propose a recovery-based spin lock protocol in which preemptions are allowed to happen at any time, but if an inopportune preemption occurs, the preempted process is immediately recovered from the preemption. As it does not prevent preemptions during the normal operation, the overhead is incurred only during the recovery. The protocol is queue-based and requires only a constant number of remote memory accesses. We call the protocol the RLOCK.

The RLOCK is unique in that it handles both (1) preemptions of processes in critical sections and (2) preemptions of processes to which locks are granted to. Anderson et al [3] also proposed a recovery-based technique. However, their protocol deals only with preemptions in critical sections, and overlooks the case where a preempted process can be granted a lock.

The RLOCK has many other merits; (1) it runs with or without a cache coherence support, and (2) ensures strict FIFO ordering on the access to a critical section whereas the prevention-based protocols proposed in [8, 9] don’t guarantee starvation freedom. This makes the RLOCK attractive also for real-time systems since it provides an upper bound on the response time of lock acquisition.

1.3 Experimental results

To validate our intuition about the performance of the prevention and recovery-based spin lock protocols, we implemented our algorithm on the KSR1, a distributed shared-memory multiprocessor, and compared the performance of the RLOCK and several other prevention-based protocols. We tested the protocols on various preemptive environments, and measured the response time, the time taken to acquire a lock, execute a critical section and then release the lock.

Our experiments show that under non-preemptive environments, the RLOCK gives a very close mean response time to that of queue-based algorithms without preemption handling facilities, which indicates that the overhead of the RLOCK is very small during the normal operation. The prevention-based protocols show a significant, but constant overhead over various preemption rates. The experiments also confirm that spin locks without preemption handling facilities suffer from a severe performance degradation under highly preemptive environments. The mean response time of the RLOCK slowly increases as the preemption rate increases, and finally exceeds that of the prevention-based protocols under high preemption rates. But its performance under high preemption rates is still comparable to that of the prevention-based protocols.

We also studied the statistical distribution of response times to test suitability of the protocol for real-time computing where a predictable performance is more important than a fast response time. We found that (both prevention and recovery-based) queue-based spin locks are suitable for soft real-time computing as they show only a small number of lock acquisition and release misses even under a small deadline, and most misses occur around their mean response times.

1.4 Related work

Anderson [2] first pointed out the problem of non-scalability of the traditional spin lock protocols that allow processors to spin (busy-wait) on a single shared variable, where spinning processors cause unbounded amount of network traffic. To tackle the problem, he proposed an array-based spin lock protocol where each process has a spin variable statically allocated for a lock. Each process spins on its own spin variable, generating only a constant number of communications when used in the systems with a coherent cache support. The protocol uses only fetch-and-increment operations (in addition to atomic reads and writes). Graunke and Thakkar [4] also independently developed a similar protocol using fetch-and-store.

Later, Mellor-Crummey and Scott [7] developed a queue-based protocol where each process spins with a “local” spin variable dynamically allocated to its own memory. The protocol requires both fetch-and-store and compare-and-swap operations, and generates a constant number of communications with or without a coherent cache support.

The above-mentioned protocols are designed only for non-preemptive systems. Anderson et al [3] first suggested a recovery-based scheme in which a preempted process in a critical section can be migrated to another processor to finish the critical section. In the technique, when a process is preempted in a critical section, the kernel informs another user process in the same application by sending a software signal to the user process, and then the user process yields its processor to the preempted process to push the preempted process through the critical section. Although the protocol handles pre-
emptions in critical sections, it overlooks the case where a lock is granted to a preempted process.

The effect of preemption on the performance of spin locks was also studied by Wisniewski, Kontothanasis and Scott [8, 9]. To tackle inopportune preemptions, they proposed a prevention-based spin lock protocol that prevents preemptions when processes are in an unsafe area. The protocol uses an extended kernel interface that allows communication of process state information between the user process and the kernel. Preemption is prevented by continuous updates of the process status by both user processes and the kernel through compare-and-swap operations. The kernel examines the process status before preemption and if a to-be-preempted process is in a critical section, preemption is delayed until the process finishes with the critical section. When a process grants a lock to the next process in the queue, the preempted processes are skipped over and the lock is passed only to the next running process. The skipped processes enter the lock queue again when they are resumed. This skipping also makes the protocol lose starvation freedom. Their experimental results show significant performance advantages of the protocol in a preemptive environment. However, the protocol tends to impose overhead during the normal operation regardless of whether or not the preemptions occur.

2 Algorithm

2.1 Assumptions

We assume that a system consists of a collection of processes that communicate through shared variables. Processes are sequential such that each process applies a sequence of operations to shared variables. All the memory operations are atomic; and four kinds of memory operations are supported: read(x), write(x,v), fetch-and-store(x,v) and C&S(x,v,z) where x is a shared variable and v and z are integers. read(x) returns the value of x; write(x,v) replaces the value of x with v; fetch-and-store(x,v) atomically returns the value of x and replaces it with v; and C&S(x,v,z) atomically compares the value of x with v, then if they are equal, replaces the value of x with z and returns true; and if they are not equal, it returns false.

2.2 A recovery-based spin lock (RLOCK)

We first describe how the algorithm works under non-preemptive systems. Omitting certain important details, the basic idea of the algorithm is the following. Each process joins a doubly linked list (i.e. a queue) of processes that attempt to acquire a lock, through a data structure called spinitem. Each process in the queue has its own spinitem and is aware of the address of the spinitem of its immediate successor in the queue. The process in the head currently holds the lock and executes a critical section while its successors keep reading their own spinitems (i.e. spinning). After finishing the critical section, the current lock holder releases the lock by writing to the spinitem of its immediate successor. After reading its modified spinitem, the successor acquires the lock. Pseudo-code of lock acquire and release procedures appears in Figure 1.

The linked list is constructed using an atomic fetch-and-store operation (or atomic swap). A process trying to acquire a lock performs an atomic swap to a global variable, called Lock, with the address of its spinitem. This swap returns the value of Lock, which is the spinitem address of the previous process that performed the swap operation to the Lock. Note that because of the atomicity of the swap, each process has a unique predecessor. To form the linked list, each process makes the next field of the spinitem of its predecessor point to its spinitem (Line 19).

The process at the head of the list that has not entered the critical region recognizes that it is at the head and enters the critical region. This is done in cooperation with its predecessor, if it has any, that releases the lock (i.e. leaves the critical region). The responsibilities of the process releasing the lock are to ensure that (1) if it has a successor, the successor gets the lock, and (2) if it doesn’t have a successor, then whoever joins after the releaser acquires the lock without spinning.

There is a potential data-race between the releaser and its successor because the releaser might be in the transition of finishing step (2) while its successor joins the list and start spinning. Then, the successor ends up spinning forever – a deadlock.

This race is broken using the atomicity of reads and writes. The releaser first indicates that it no longer holds the lock in its own spinitem (Line 24) and then in that of its successor (Line 26). Before the successor spins at Line 21, it reads the spinitem of its predecessor (Line 20). If the condition at Line 20 is true, the releaser must have not executed Line 24, and will eventually execute Line 26.

The compare-and-swap at Line 25 writes to the successor’s status only when the successor is not waiting for some other releaser, and is in between Lines 19 and 22. Note that it is possible that the successor may not spin and proceeds to enter the critical region at Line 20, and sometime later, it tries to acquire a lock with the same spinitem. Then it might be waiting for another releaser. The compare-and-swap checks atomically whether the successor’s status is equal to the address of the spinitem of the releaser plus one before it writes to it. Since no other spinitem has the address equal to that of the releaser and the spinitem’s structure is bigger than one byte, this ensures that the successor is not waiting for some other releaser when the releaser writes to the spinitem of the successor. This way, any value that the releaser has written to the spinitem’s status of the successor is not overwritten until the successor
procedure/, it sets
in Figure /1/. Each spinitem also has a pointer to the
entrants and leaves the critical region. The kernel also modifies the
state of the control block to indicate that the process
is preempted. When the kernel preempts a process, it executes procedure preempt() where the kernel finds
out whether the process is in a critical section, or is
already granted the lock or not, by examining the current
spinitem of the process (which is pointed to by
spinitems of the control block of the process). If the
spinitem of its successor is already
pointing to its predecessor (i.e., the
spin of its successor is already
preempted when a process releases a
lock). When a process releases a
lock/, it examines the control block of
the process/. If the process is gran
ted the lock/, it alternately uses the other spinitem (see Line 44).

To handle inopportune preemptions, processes need
to communicate with the kernel. Our kernel interface
is similar to that proposed in [8, 9]. We assume
that each kernel maintains a data structure, called the control
block, for each process running in the system. This
control block is accessible by the process as well as the
kernel. (The structure of the control block is also shown
in Figure 1.) Each spinitem also has a pointer to the
control block of its process (cb in qnode definition).

Before a process attempts to execute the acquire
procedure, it sets cspinitems of its control block to
point to its spinitem structure (a Qnode). Note that
this is done only once as long as the process
acquires the same spin lock. The kernel also modifies the state
field of the control block to indicate that the process
is preempted. When the kernel preempts a process, it
executes procedure preempt() where the kernel finds
out whether the process is in a critical section, or is
already granted the lock or not, by examining the current
spinitem of the process (which is pointed to by
spinitems of the control block of the process). If the
spinitem of its successor is already
pointing to its predecessor (i.e., the
spin of its successor is already
preempted when a process releases a
lock). When a process releases a
lock/, it examines the control block of
the process/. If the process is gran
ted the lock/, it alternately uses the other spinitem (see Line 44).

To handle inopportune preemptions, processes need
to communicate with the kernel. Our kernel interface
is similar to that proposed in [8, 9]. We assume
that each kernel maintains a data structure, called the control
block, for each process running in the system. This
control block is accessible by the process as well as the
kernel. (The structure of the control block is also shown
in Figure 1.) Each spinitem also has a pointer to the
control block of its process (cb in qnode definition).

Before a process attempts to execute the acquire
procedure, it sets cspinitems of its control block to
point to its spinitem structure (a Qnode). Note that
this is done only once as long as the process
acquires the same spin lock. The kernel also modifies the state
field of the control block to indicate that the process
is preempted. When the kernel preempts a process, it
executes procedure preempt() where the kernel finds
out whether the process is in a critical section, or is
already granted the lock or not, by examining the current
spinitem of the process (which is pointed to by
spinitems of the control block of the process). If the
spinitem of its successor is already
pointing to its predecessor (i.e., the
spin of its successor is already
preempted when a process releases a
lock). When a process releases a
lock/, it examines the control block of
the process/. If the process is gran
ted the lock/, it alternately uses the other spinitem (see Line 44).

To handle inopportune preemptions, processes need
to communicate with the kernel. Our kernel interface
is similar to that proposed in [8, 9]. We assume
that each kernel maintains a data structure, called the control
block, for each process running in the system. This
control block is accessible by the process as well as the
kernel. (The structure of the control block is also shown
in Figure 1.) Each spinitem also has a pointer to the
control block of its process (cb in qnode definition).

Before a process attempts to execute the acquire
procedure, it sets cspinitems of its control block to
point to its spinitem structure (a Qnode). Note that
this is done only once as long as the process
acquires the same spin lock. The kernel also modifies the state
field of the control block to indicate that the process
is preempted. When the kernel preempts a process, it
executes procedure preempt() where the kernel finds
out whether the process is in a critical section, or is
already granted the lock or not, by examining the current
spinitem of the process (which is pointed to by
spinitems of the control block of the process). If the
spinitem of its successor is already
pointing to its predecessor (i.e., the
spin of its successor is already
preempted when a process releases a
lock). When a process releases a
lock/, it examines the control block of
the process/. If the process is gran
ted the lock/, it alternately uses the other spinitem (see Line 44).

To handle inopportune preemptions, processes need
to communicate with the kernel. Our kernel interface
is similar to that proposed in [8, 9]. We assume
that each kernel maintains a data structure, called the control
block, for each process running in the system. This
control block is accessible by the process as well as the
kernel. (The structure of the control block is also shown
in Figure 1.) Each spinitem also has a pointer to the
control block of its process (cb in qnode definition).

Before a process attempts to execute the acquire
procedure, it sets cspinitems of its control block to
point to its spinitem structure (a Qnode). Note that
this is done only once as long as the process
acquires the same spin lock. The kernel also modifies the state
field of the control block to indicate that the process
is preempted. When the kernel preempts a process, it
executes procedure preempt() where the kernel finds
text switching from the kernel to the preempted process while the remote recovery involves two context switchings (one to wake up the kernel and the other from the kernel to the preempted process).

When a recovered process finishes the critical section, it immediately yields its control to the kernel so that the kernel resumes the process that was running before the recovery. This yielding may involve additional overhead in lock release. Anderson et al [3] discussed this issue and their solution is to have the process run a different copy of the critical section when recovered. This new critical section, which is obtained by some preprocessing, contains the original critical section and some additional statements in the end for yielding the control. Thus, the normal lock acquisition and release do not incur any overhead because of this yielding. We adopt this approach in the RLOCK.

The RLOCK incurs overhead only when a process releases a lock and when preemptions occur. But most of the overhead is incurred during preemption recoveries, but that in lock release is charged only to the releaser, but not to its successors because the checking for the preemption of the successor (Lines 26 and 27) is executed after the releaser releases the lock.

The correctness proof of the RLOCK is left to the full paper.

3 Experiments and results
3.1 The experimental environment

We are interested in comparing the performance of prevention-based and recovery-based protocols under different preemptive environments. Our experiments were performed on the KSR1, a distributed shared memory multiprocessor designed by Kendall Square Research. The model we used contains 40 processors, each of which is a 20 MHz, super-scalar, RISC chip with a peak 64-bit floating point performance of 40 Mflop/s and 32 Mbyte of memory. Processors are connected by a uni-directional ring network with a bandwidth of 1 Gbyte/s. The memory system enforces sequential consistency. A local memory access to 64 byte data requires a latency of 18 cycles (0.9 µs). A remote memory access to 128 byte data in another processor's local memory requires a latency of around 150 clock cycles (7.5 µs) (the remote accesses are always done in 128 byte units). The machine has a Unix-compatible distributed operating system (the Mach-based OSF/1). Additionally, the KSR1 provides a cache line locking mechanism in hardware, the equivalent of queue-based locks. Since the atomic primitives, such as fetch-and-store and compare-and-swap, are not available in the hardware of the KSR1, we used the cache line locking mechanism to implement the primitives. This approach is still amenable to our experiment goals since all the spin locks of interests use these primitives, and their performance comparison is of most concern.

We ran a synthetic application in which each process, which is assigned to a unique processor, executes a simple loop that consists of (1) a critical section where it updates a global counter and makes idle loops, and (2) a non-critical section where it again makes idle loops. Before entering the critical section, a process acquires a lock, and upon leaving the critical section, releases the lock. The ratio of the critical section time over the non-critical section time, and the number of contending processes are adjusted to see the effect of contention. The preemption rate, the number of times that a process is preempted in one second, is also varied to see the effect of preemption. The preempted period, the time period that a process is preempted at a time, is also varied.1

A modified version of a pseudo-kernel used in [8, 9] is used to emulate process preemption and scheduling. To preempt a running process, the pseudo-kernel sends a Unix signal to the process, which causes a trap into the process's interrupt handler. The handler makes idle loops for its preempted period. Then, the process returns from the interrupt handler routine and resumes its execution. In our experiment, the kernel runs on a dedicated processor, and thus it does not interfere with the execution of other processes. However, our pseudo-kernel implementation is communication-bound as the kernel continually reads and writes several global shared data structures to keep track of process states. This may affect the performance of non-scalable spin lock protocols (e.g., test-and-set lock) as they may perform a large number of communications, but does not compromise the goal of our experiment as some of prevention-based protocols we tested are also scalable, e.g., SmartQ (described later).

We initially implemented the remote recovery signal in the RLOCK using a Unix signal, but later discovered that the Unix signal waiting time in the KSR1, which is the time between when the signal is sent and when the KSR1 kernel recognizes the signal and preempt the running process, can be unduly large (sometimes more than 20 ms). We suspect that the KSR1 kernel is not optimized for signal handlings, especially among processes in different processors, and believe that this signal waiting time is not inherent and can be optimized immensely. As reported in [3], the typical signal waiting time in an optimized (parallel) kernel is about 1 or 2 ms. (In a fast architecture like the SPARC, the context switching time can be less than 100 µs [6].) Thus, instead of using a Unix signal for the remote recovery signal, to provide a reasonable timing for the signal waiting time, we simulated the remote recovery signal waiting time by idle loops, where the loop counter is set to match realistic signal waiting times (from 1.4 ms to

1We did the experiments over several preempted periods larger than 20000 idle loops, and found a similar performance for all cases, and thus in the following presentation, the preempted period is set to 20000 idle loops (about 30 ms).
In the experiment, we measured the time duration between when a process attempts to acquire a lock and when it releases the lock, which we call *the response time*. The response time includes the lock acquisition time, the critical section time, and the lock release time. The size of the critical section is kept small throughout the experiment (less than 50 \(\mu\text{s}\)) as most critical sections in reality are small. Furthermore, the time measurement we get will contain the cost of lock acquisition and release than that of critical section. The 90% confidence interval is applied, and we verified that less than 10% of all the measured response times fall in the interval.

We tested the following spin lock algorithms: (1) the test-and-set lock (TAS) with exponential backoffs; (2) the TAS that prevents the preemption of a process in a critical section (TASNP); (3) the ticket lock \([7, 8, 9]\) with preemption prevention (TicketNP) where the preemption of a process in a critical section is prevented and lock is granted only to a running process by using a handshaking protocol: the releaser waits for its successor to respond within a certain time interval, and if the successor does not respond in the interval, then it is assumed to be preempted and is skipped over; (4) the MCS lock (MCS), a queue-based spin lock algorithm developed by Mellor-Crummey and Scott\([7]\) with no preemption handling facility; (5) the MCS lock with preemption prevention (SmartQ) where preemption in a process in a critical section is prevented, and by communicating with the kernel, a releaser skips over preempted processes and grants the lock only to a running process; and finally (6) the RLOCK with 2500 and 5000 loop iterations for the remote recovery signal waiting (it takes about 1.4 ms for 2500 loop iterations and about 2.8 ms for 5000 loop iterations).

### 3.2 The mean response time

We first measured the overhead of the lock acquisition and release induced by preemption handling. To do this, we measured the response time of each algorithm on a non-preemptive environment over various numbers of processors. Figure 2 shows the mean response time for each algorithm under a generally high contention environment with a small critical section and an empty non-critical section. The MCS shows the best performance because it doesn’t incur any overhead for preemption handling. The performance of the RLOCK is very comparable to that of the MCS (with less than 100 \(\mu\text{s}\) difference). The SmartQ, TASNP and TicketNP suffer from the overhead of preemption prevention as they involve communicating with the kernel every time a process acquires and releases the lock.

Figures 3 and 4 show the mean response time over various preemption rates with 30 processors. Figure 3 shows a high lock contention case where processes have an empty non-critical section and a small critical section (10 \(\mu\text{s}\)). Figure 4 shows a medium lock contention case where the relative size of critical and non-critical sections is adjusted to avoid high lock contention. The length of the critical section is around 50 \(\mu\text{s}\) and the length of the non-critical section is around 1.3 ms. The ratio of the non-critical section over the critical section is around 33, which is chosen to eliminate a lock step behavior of 30 processors, and thus, to ease contention.

In both cases, the performance of the MCS gets rapidly worse as the preemption rate increases, showing large mean response times under high preemption rates. This is because the MCS has no preemption handling facility. This phenomenon is more evident in Figure 4 because the larger critical section causes more inopportune preemptions. The performance of all the prevention-based algorithms is fairly insensitive to the preemption rate, which is expected, because the algorithms pay the same amount of overhead whether a preemption occurs or not. The RLOCK shows small mean response times under low preemption rates. However, its performance degrades slowly as the preemption rate increases because more preemptions cause more inopportune preemptions, and therefore more recoveries. The response time of the RLOCK (5000) finally exceeds that of the SmartQ around 12 preemptions per second in both of the graphs. The performance difference between the RLOCK (2500) and the RLOCK (5000) also suggests that the RLOCK’s performance is fairly sensitive to the process signal waiting time.

While the TASNP suffers badly from high contention (whose performance gets worse than that of the MCS) in Figure 3, it gives better response times under medium contention in Figure 4. The TicketNP surprisingly did not perform very well in all of our experiments, contrary to the finding in \([9]\) where it shows the TicketNP gives a better performance than the SmartQ. One plausible explanation is that the Ticket lock is a backoff based protocol and suffers from high network contention because it may require many memory accesses \([7]\). Our implementation of the pseudo-kernel involves many remote memory accesses that may congest the network. Thus any algorithm that cause many network communications, such as the Ticket lock or the TAS, tends to suffer more from the larger network latency.

### 3.3 The distribution of response times

The correctness of real-time applications relies on the timeliness of each operation it performs. Any real-time application that uses spin locks as its synchronization primitive requires an estimate of the response time (i.e., deadline), and it is important that this estimate is reliable and small.

---

\(^2\)These numbers are picked conservatively with respect to the actual signal waiting time of a more optimized kernel (e.g., Solaris).
In this section, we present our analysis of the distribution of response times for all the algorithms we tested by counting the number of response times larger than a certain response time deadline, which we call missed response times. We present our data in terms of the ratio of the number of missed response times over the total number of response times counted in each run. All the experiments are done with 50 µs critical section and 1.3 ms non-critical section. In all the cases, except where indicated, 30 processors are used. The other values for the parameters are also tested, but not much difference in the distribution trend is observed.

Figures 5, 6 and 7 show the distribution of response times under 4, 10 and 20 preemptions per second respectively. The SmartQ surprisingly shows a very good distribution. In all the preemption rates, it shows a stair shape graph where the number of missed response times suddenly rises around 20000 µs deadline, maintains a relatively low plateau for a while, and then peaks at a low deadline (around 5000 µs). Thus, by setting the deadline around the peak point, a real-time application can minimize the number of missed response times while maintaining a small deadline.

The place that the plateau starts is related to the level of contention while the height of the plateau is related to the preemption rate. Figure 8 shows the shape of the distribution for the SmartQ with different preemption rates. As the rate increases, so does the height of plateau. Figure 9 shows the same information with different numbers of processors (i.e. different contention levels). As contention increases, the plateau starts earlier at a larger deadline. The height of the plateau is also affected by contention as high contention increases the probability of inopportune preemptions.

Our original intuition was that since the SmartQ doesn’t maintain strict FIFO ordering and starvation freedom, the response time can be unpredictable. That intuition is true up to the point that those processes that are preempted and skipped over indeed have large response times. However, the inopportune preemptions
The number of missed response times (4 preemptions/sec).

The number of missed response times (10 preemptions/sec).

The number of missed response times (20 preemptions/sec).

Figure 5: The distribution of missed response times.

Figure 6: The distribution of missed response times.

Figure 7: The distribution of missed response times.

Affect only those processes that are skipped over, but not the other processes in the queue. We also confirmed that the number of missed response times is close to the height of the plateau. This indicates that the SmartQ can be effectively used in soft real-time systems because only a small number of misses (that corresponds to the number of inopportune preemptions) is generated.

The RLOCK shows generally a smoothly increasing curve (similar to an exponential distribution) in the number of missed response times over the entire range of the deadlines, and a peak at a small deadline (around 5000 µs). Under high preemption rates, it shows a faster increase, and the shapes of the graph gets closer to a straight line with a slope (Figure 7). The slope seems proportional to the preemption rate, and the place where the slope starts is related to the contention level. Figures 10 and 11 show more clearly the shape for the RLOCK (2500) with different preemption rates and contention levels. Figure 13 describes the shape in a more simplified form.

This shape can be explained from the following observation. In the RLOCK, when a process gets an inopportune preemption, the overhead of recovery affects not only the preempted process, but also those processes that wait for the process in the queue. This is because the preempted process, instead of being skipped over, is recovered to execute the critical section, keeping other processes in the queue waiting. The process in the queue suffers from all the recoveries performed by its predecessors. Thus, the total overhead that a single lock acquisition pays due to inopportune preemptions is proportional to the initial position of the process in the queue. Assuming an exponential distribution of the initial positions that a process can get in the queue, we can expect the shape of the graph to be close to a smoothly increasing curve as shown in Figures 10 and 11.

While the RLOCK (5000) suffers much from high preemption rates, the RLOCK (2500) generally shows the best performance among all the protocols (Figure 7). This indicates that when a signal waiting time is small (as it is the case in most real-time operating systems), even with a small deadline of the response time,
most executions of the critical sections protected by the RLOCK can finish within the deadline.

The TASNP shows a greater sensitivity to lock contention (See Figures 5, 6 and 7). Under low preemption rates (i.e., high lock contention), its performance was the worst, creating comparatively many misses even under a high response time deadline, but it gets relatively better under high preemption rates (i.e., low lock contention). As the performance of the TASNP depends heavily on lock contention, an extensive performance study, which we do not show here because of space limit, also indicated that its performance also depends much on an appropriate backoff factor, and we also found difficult to adjust the backoff factor to have a good response time and a good distribution of response time misses.

The TicketNP also shows generally a poor performance as many response times miss even a large deadline (See Figures 5, 6 and 7). The TicketNP uses a handshaking protocol in which when a process releases the lock to the next waiting process, the waiting process must respond to the releaser within a certain time interval. Failing to respond in the time interval results in the process being skipped over because the releaser assumes that the process is preempted. Too short a time interval results in excessive numbers of processes being skipped over, and many misses, which explains why it shows many more misses than the SmartQ although its distribution has a similar shape as that of the SmartQ.

4 Conclusion and Future Study

Queue-based spin locks can show a severe performance degradation in preemptive systems when preemptions are not handled properly because preempted processes may hold the lock. In this paper, we discuss a solution for the problem that recovers the preempted process from the preemption when a preemption that can cause delay occurs, so that the process can leave an unsafe area quickly. The advantage is that the overhead is incurred only when such preemptions occur. This is unlike the other prevention-based techniques [8, 9] where the overhead is imposed whether or not such preemptions occur. Our protocol is also an improvement over the previous recovery-based technique [3] which deals only with the case where processes are preempted in critical sections.

One disadvantage of our protocol is that the preemption handling is expensive because it causes several context switches to reschedule the preempted processes. However, this overhead is far less than that has to be paid when preemption is not handled properly as in [2, 4, 7].

Our experiment confirms that our protocol imposes little overhead under normal operations, and its performance gracefully degrades when the number of preemptions (i.e., the multiprogramming level) increases. However, we found the performance of the protocol is fairly sensitive to the process signal waiting time (and context switching time). However, under modern operating systems (such as SGI), that overhead is kept small. Thus, the protocol can be efficiently used in those machines. The statistical distribution of the response times of the protocol suggests that the protocol is also apt for real-time computing as most of response time misses occur around its mean response time.

For future study, we plan to do a more extensive performance study of our protocol. In the current experiment, unfortunately, we have to emulate system signals as the actual KSR system signal waiting time is unduly large (~20ms) when compared to that in other operating systems (less than 1ms). Thus, we plan to do the experiment in other systems where operating systems are more optimized for context switching such as SGI Challenge. We also plan to do the experiment using standard benchmarks (e.g., SPLASH).

Acknowledgement: We would like to thank Graham Riley and Tim Robinson of the Center for Novel Computing at Manchester University for the valuable discussion and permission to use their KSR1 machine. We
are also grateful for many helpful comments from anonymous reviews. Finally, special thanks go to Robert Wisniewski, Leonidas Kontothanassis, and Michael Scott of the University of Rochester for making their experimental software available to us.

References


