Real-Time Scalability of Nested Spin Locks

Hiroaki Takada and Ken Sakamura

Department of Information Science,
Faculty of Science, University of Tokyo
7-3-1, Hongo, Bunkyo-ku, Tokyo 113, Japan

Abstract

When a real-time system is realized on a shared-memory multiprocessor, the worst-case performance of interprocessor synchronizations is one of the most important issues. In this paper, scalability of the maximum execution times of critical sections guarded by nested spin locks is discussed. With the simplest method, the maximum execution times become $O(n^m)$, where n is the number of contending processors and m is the maximum nesting level of locks. In this paper, we propose an algorithm with which this order can be reduced to $O(n \cdot e^m)$ and demonstrate its effectiveness when $m = 2$ through performance measurements.

1 Introduction

Spin lock is a fundamental synchronization primitive for exclusive access to shared resources on shared-memory multiprocessors. For real-time systems, two kind of spin locks are used depending on the timing requirements on them: (1) bounded spin locks, in which the maximum times that processors acquire and release a lock are bounded, and (2) priority-ordered spin locks, in which processors acquire a lock in the order of their priorities [1].

In this paper, the scalability issue of bounded spin locks is discussed. Because worst-case behavior has the primary importance in real-time systems, we focus on scalability of the maximum execution times of critical sections guarded by spin locks, under the assumption that the maximum processing time within a critical section is bounded. We call scalability of worst-case behavior as *real-time scalability*.

In general, shared resources that must be accessed exclusively by a processor are divided into some lock units in order to improve concurrency. When a processor accesses some shared resources included in different lock units, it must acquire multiple locks one by one. If FIFO spin locks are used for this kind of *nested spin locks*, the maximum execution times of a whole critical section become $O(n^m)$, where n is the number of contending processors and m is the *maximum nesting level* of locks. The strict definition of the maximum nesting level is presented in Section 2.

It is obvious that this simple method is not acceptable from the viewpoint of real-time scalability. In this paper, we propose a method in which this order can be reduced to $O(n \cdot e^m)$, which is acceptable when m can be kept small.

In Section 2, assumptions and notations adopted in this paper are described. An $O(n)$ algorithm when the maximum nesting level is two is proposed in Section 3 and its effectiveness is evaluated through performance measurements in Section 4. In Section 5, an $O(n \cdot e^m)$ algorithm for general case is presented.

2 Assumptions and Notations

A system consists of n processors supporting atomic read-modify-write operations on a single word of shared memory (e.g. test_and_set, fetch_and_store (swap), fetch_and_add, and compare_and_swap). Each processor repeatedly executes critical sections guarded by one or more locks. The maximum execution time of a critical section except for the waiting time for the locks is assumed to be bounded.

In order to avoid deadlocks, a partial order $\succ$ is defined on the set of locks in the system. A processor must acquire locks following the order. We assume that if and only if processors possibly acquire a lock L_j while holding a lock L_i, an order $L_i \succ L_j$ exists.

The nesting level λ_i is defined for each lock L_i as follows. If L_i is a minimal element (i.e. there is no L_j such that $L_i \succ L_j$), λ_i is defined to be one. Otherwise, λ_i is defined to be $max\{\lambda_j \mid L_i \succ L_j\} + 1$. We call $max\{\lambda_i\}$ as the maximum nesting level of locks in the system. Consider the example that processors in the system execute one of the two routines presented in Figure 1. In this example, $\lambda_1 = 1$, $\lambda_2 = 2$, $\lambda_3 = 3$, and the maximum nesting level in the system is three.

```
acquire_lock(L3);          acquire_lock(L2);
acquire_lock(L2);          acquire_lock(L1);
// critical section.       // critical section.
release_lock(L2);          release_lock(L3);
release_lock(L3);          release_lock(L1);
```

routine (a) *routine (b)*

Figure 1: Example of Nested Locks

In the following sections, a lock whose nesting level is i is denoted as L_i. When there are some locks with the same nesting level, they are represented as L_i, L_i', L_i'', $\cdots$.

We also assume that the two-phase protocol is adopted on each processor. In other words, once a processor releases a lock, it cannot acquire a lock until it releases all the locks it is holding. This assumption is adopted in order to simplify the evaluation of the maximum number of the critical sections that a processor must wait for. The estimation of its order is also valid without the assumption.

3 Nesting in Two Levels

In this section, we focus on nested spin lock algorithms when the maximum nesting level is two. We regard them as important because the scalable real-time kernel model we have proposed in [2] can be implemented with the maximum nesting level being two.

Problems of Simple Methods

As mentioned before, if FIFO spin locks are simply applied to the system in which the maximum nesting level of locks is two, the maximum execution times of a whole critical section become $O(n^2)$, where n is the number of contending processors.

As an example, consider the case that each processor in the system repeatedly executes one of the three routines presented in Figure 2 in random order. Below, we illustrate the case in which the number of the critical sections that a processor P_1 must wait for until it finishes an execution of routine (c) is maximized. Assume that when P_1 tries to acquire L_2 in (c), another processor P_2 has just acquired the lock and all the other processors P_3, $\cdots$, P_n are waiting for the lock in routine (c) in this order. When P_2 releases the lock, P_3 succeeds to acquire the lock. Just before P_3 tries to acquire L_1, P_2 can acquire the lock in routine (a). In this case, P_3 must wait until P_2 finishes the critical section and releases L_1, and P_1 must wait for two critical sections executed by P_2 and P_3. Similarly,

```
acquire_lock(L1);          acquire_lock(L2);
// critical section.       // critical section.
release_lock(L1);          release_lock(L2);
```

routine (a) *routine (b)*

```
acquire_lock(L2);
acquire_lock(L1);
// critical section.
release_lock(L1);
release_lock(L2);
```

routine (c)

Figure 2: Nesting in Two Levels

```
acquire_lock(L2');                 acquire_lock(L2');
acquire_lock(L1);
// critical section.               // critical section.
release_lock(L2');                 release_lock(L1);
                                   release_lock(L2');
```

routine (d) *routine (e)*

Figure 3: Nesting in Two Levels (cont.)

when P_{i-1} releases L_2, P_i succeeds to acquire the lock. Before P_i tries to acquire L_1, P_2, $\cdots$, P_{i-1} can wait for the lock in (a). P_i must wait for the executions of $i-2$ critical sections until it succeeds to acquire L_1, and P_1 must wait for $i-1$ critical sections until P_i finishes routine (c). Finally, after P_1 succeeds to acquire L_2, P_1 must wait for $n-1$ critical sections before it acquires L_1. As a result, the maximum number of the critical sections that a processor P_1 must wait for is $1+2+\cdots+(n-1)+(n-1) = n(n+1)/2-1$, thus $O(n^2)$. Because the maximum processing time within a critical section has an upper bound, the order of the maximum execution times of routine (c) is $O(n^2)$. That of routine (b) is also $O(n^2)$, while that of routine (a) is $O(n)$.

A simple method to improve this order is that precedence is given to the processor holding an outer lock. In case of Figure 2, the processor that is waiting for L_1 in routine (c) can acquire the lock with higher priority than other processors. Because the maximum number of the critical sections that a processor must wait for while trying to acquire L_1 in (c) is reduced to one with this method, the maximum execution times of both (b) and (c) are improved to $O(n)$. The maximum execution times of routine (a) remain to be $O(n)$, because a processor never waits for L_1 in (c) while another processor holds L_1 in (c), and because the lock is passed to a processor executing (a) when the processor

in (c) releases the lock.

However, this method has a problem when each processor can also execute the two routines presented in Figure 3. In this case, a processor executing routine (a) can starve while waiting for L_1. Specifically, a processor trying to acquire L_1 in (a) can be passed by a processor executing (c) and a processor executing (e) by turns, and the maximum time until it succeeds to acquire L_1 cannot be determined.

Another method is that a processor trying to acquire nested locks reserves its turn to acquire the inner lock by enqueueing itself to the wait queue of the lock, when it begins waiting for the outermost lock. This method, however, cannot be applied when which inner lock to be acquired is determined after accessing the shared resource guarded by the outer lock.

Proposed Method

To solve the problem described in the previous section, we propose the following algorithm, which can make the maximum execution times of each routine $O(n)$.

When a processor begins waiting for the outermost lock, it obtains a time stamp by reading a real-time clock. Instead of using FIFO spin locks, priority-ordered spin locks are used with the time stamps as the priorities[1] (an earlier time stamp has a higher priority). With this method, the processor that begins waiting for the outermost lock earlier can acquire each lock with higher precedence. In other words, the FIFO policy is applied to the whole critical section.

This method can reduce the order of the maximum execution times of each routine to $O(n)$ with the following reason. The maximum number of the higher priority critical sections (the critical sections executed by the processors with higher priorities than P_1) that a processor P_1 must wait for is $n-1$. This is because only the processors obtaining time stamps before P_1 can acquire locks with precedence over P_1, and because each processor can execute only one critical section with a time stamp. P_1 must also wait for some lower priority critical sections. When a processor tries to acquire an inner lock, another processor with a lower priority possibly holds the lock. This is a kind of priority inversion and occurs at most once whenever a processor begins waiting for an inner lock. Note that this priority inversion does not occur in acquiring an outer lock.

When a processor P_2 with a higher priority than P_1 acquires the outer lock on which P_1 is waiting, and when P_2 tries to acquire an inner lock, P_2 must possibly wait for a critical section executed by a lower priority processor P_3 due to priority inversion. In this case, the critical section executed by P_3 should be counted in the number of the critical sections that P_1 must wait for. As a result, an upper bound on the number of the critical sections that P_1 must wait for is $2(n-1)+1 = 2n-1$, thus the order of the maximum execution times of routine (c) is $O(n)$. Those of the other routines are also $O(n)$.

More precisely, the number of the critical sections that P_1 must wait for in routine (c) in Figure 2 becomes maximum in the following case. Assume that when P_1 tries to acquire L_2 in (c), another processor P_2 holds the lock and all the other processors $P_3, \cdots, P_n$ are waiting for the lock in routine (c) in this order. When P_2 releases the lock, P_3 succeeds to acquire the lock. Just before P_3 tries to acquire L_1, P_2 can acquire the lock in routine (a). In this case, P_3 must wait until P_2 releases L_1, and P_1 must wait for two critical sections. Similarly, when P_i succeeds to acquire L_2 and tries to acquire L_1, one of $P_2, \cdots, P_{i-1}$ possibly holds L_1, and P_1 must wait for two critical sections. Finally, after P_1 succeeds to acquire L_2, it possibly needs to wait for a critical section before it acquires L_1. As a result, the maximum number of the critical sections that P_1 must wait for is $1+2+\cdots+2+1 = 2n-2$. The result of this exact estimation is smaller than the previous estimation because the fact that P_2 does not suffer any priority inversions is counted in.

In implementing this method, following optimizations are possible.

1. In acquiring an outer lock (a lock whose nesting level is two), a FIFO spin lock algorithm can be used instead of a priority-ordered one.
2. A sequence number that a processor begins waiting for the outermost lock, which can be implemented with fetch_and_increment operation, can be used as the time stamp instead of an absolute time read from a real-time clock.

4 Performance Evaluation

In this section, the effectiveness of the algorithm proposed in the previous section (called TF, in this section) is examined through performance evaluation. Its performance is compared with the method that FIFO spin locks are simply used for all locks (called SF, in this section) and the method that precedence is given to the processor holding an outer lock (called PI, in this section).

[1] The fact that a FIFO-ordered lock can be realized with a priority-ordered lock using time stamps as priorities is pointed out by Craig [1].

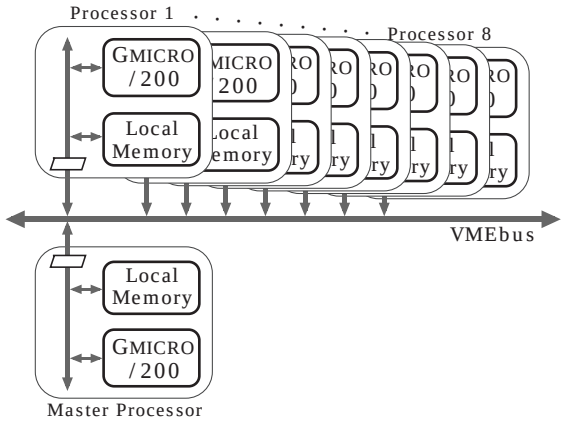

Figure 4: Evaluation Environment

Evaluation Method

We have adopted the MCS lock algorithm [3] for the FIFO spin locks and the algorithm proposed by Markatos [4] for priority-ordered spin locks. The FIFO spin lock with precedence, which is necessary to implement PI, is realized using the Markatos' spin lock algorithm. In implementing TF, we have used a FIFO spin lock algorithm for the outer locks and a priority-ordered one for the inner locks. We have also used a sequence number that a processor begins waiting for the outermost lock instead of a real-time clock.

A shared-bus multiprocessor system is used for the evaluation. The shared bus is based on the VMEbus specification, and each processor node consists of a GMICRO/200 microprocessor, which is rated at approximately 10 MIPS, and 1 MB of local memory. The local memory can be accessed from other processors through the shared bus. No coherent cache is equipped. All the program code and the data area for each processor are placed on the local memory of the processor. Global shared data is placed on the local memory of the master processor, which does not execute spin locks (Figure 4).

Since the GMICRO/200 microprocessor supports compare_and_swap instruction but not fetch_and_store nor fetch_and_increment, those operations are emulated with a compare_and_swap instruction and a retry loop. As the VMEbus has only four pairs of bus request/grant lines, processors are classified into four classes by the bus request line they use. The round-robin arbitration scheme is adopted among classes and the static priority scheme is applied among processors belonging to a same class.

Since we focus on the worst-case behavior of a system, the effectiveness of our proposal should be eval-

```
t0 := read_current_time();
prio := get_sequence_number();
acquire_lock_markatos(L1, prio);
// some shared bus accesses
//   and two empty loops (about 22μsec).
release_lock_markatos(L1);
t1 := read_current_time();
// measurement result is (t1 − t0).
```

routine (a)

```
t0 := read_current_time();
acquire_lock_mcs(L2);
// some shared bus accesses
//   and two empty loops (about 22μsec).
release_lock_mcs(L2);
t1 := read_current_time();
// measurement result is (t1 − t0).
```

routine (b)

```
t0 := read_current_time();
prio := get_sequence_number();
acquire_lock_mcs(L2);
// some shared bus accesses
//   and an empty loop (about 11μsec).
acquire_lock_markatos(L1, prio);
// some shared bus accesses
//   and an empty loop (about 11μsec).
release_lock_markatos(L1);
release_lock_mcs(L2);
t1 := read_current_time();
// measurement result is (t1 − t0).
```

routine (c)

```
for i := 1 to number_of_loop do
  case random_number() of
    1,2,3,4:
      execute routine (a);
    5:
      execute routine (b);
    6:
      execute routine (c);
  end
end
```

main routine

Figure 5: Measurement Routines with TF

uated with maximum execution times. Because maximum execution times cannot be obtained through experiments due to unavoidable non-determinism in multiprocessor systems, however, a p-reliable time, the time within which a processor finishes execution with probability p, is adopted as the performance metric instead of a maximum execution time. In this section, we show the evaluation results when p is 0.9999 (i.e. 99.99%).

Evaluation Results

At first, processors in the system repeatedly execute one of the three routines presented in Figure 2 in random order. The probability that a processor executes routine (a) is made four times larger that each of other routines. A processor accesses the shared bus several number of times and waits for a while using empty loops inside the critical section. In case of routine (c), shared bus accesses and an empty loop are also inserted between two acquire_lock operations. Without spin locks (and the routine for obtaining the sequence number in case of TF), the execution time of each critical section is about 30 μs, including the overhead for measuring execution times. As a example, pseudo code of the measurement routines with TF are presented in Figure 5.

Figure 6 presents the 99.99%-reliable execution times of routine (c). When the number of processors is large, the execution times of routine (c) is quite slower with the simplest method (SF) than our proposed method (TF). The execution times with TF increase a little more than $O(n)$. This is because the lock release times in the Markatos' lock become long as the number of processors is increased. This problem will be relieved with the PR-lock algorithm [5], but we cannot evaluate with it because double-word compare_and_swap operation is necessary to implement the PR-lock. The 99.99%-reliable execution times of routine (b) are almost same with routine (c) except that the absolute times are little shorter (Figure 7).

Figure 8 presents the 99.99%-reliable execution times of routine (a) under the same condition. Though the execution times of routine (c) are fastest with PI, those of routine (a) are slowest with the method.

The problem of PI becomes more obvious, when processors repeatedly execute one of the fives routines in Figure 2 and 3 in random order. Figure 9 presents the 99.99%-reliable execution times of routine (a) under this condition. The probability that a processor executes routine (a) is made twice larger than each of other routines. In this figure, the execution times with PI are much slower than the other methods.

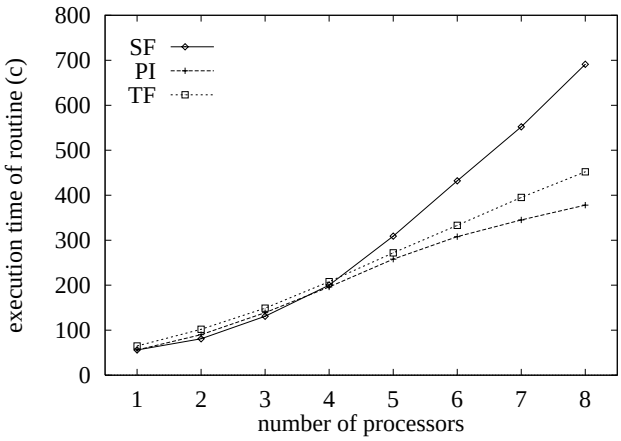

Figure 6: 99.99%-reliable Exec. Times of Routine (c)

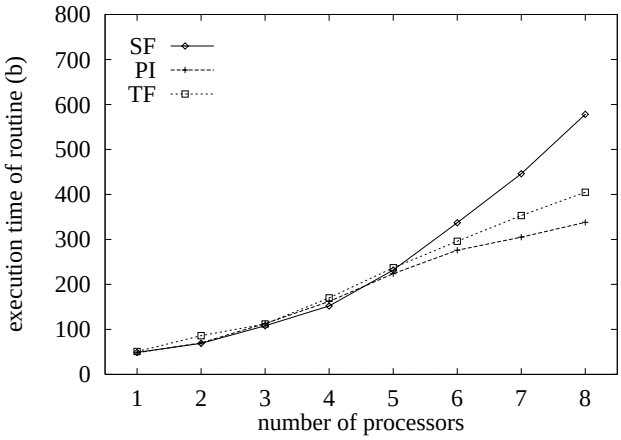

Figure 7: 99.99%-reliable Exec. Times of Routine (b)

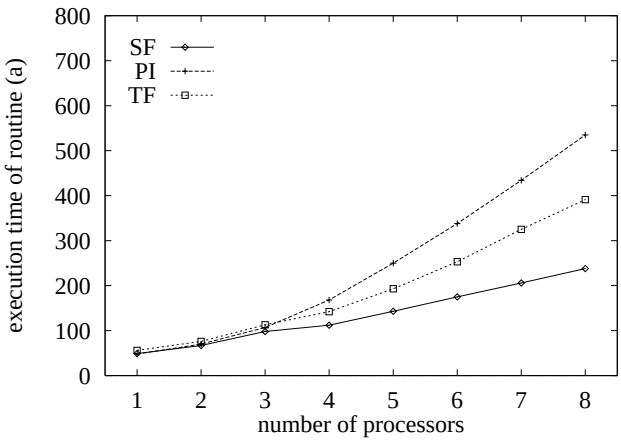

Figure 8: 99.99%-reliable Exec. Times of Routine (a)

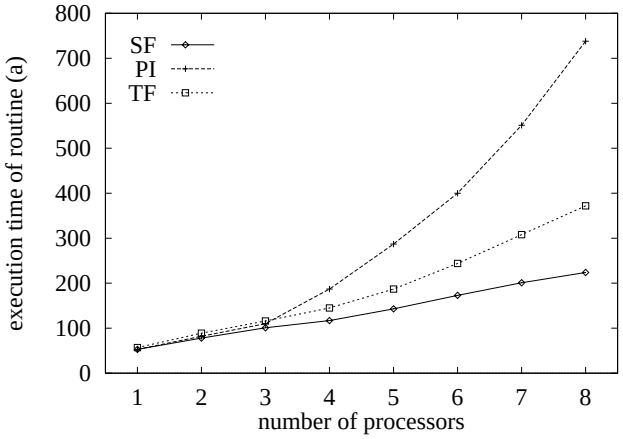

Figure 9: 99.99%-reliable Exec. Times of Routine (a)

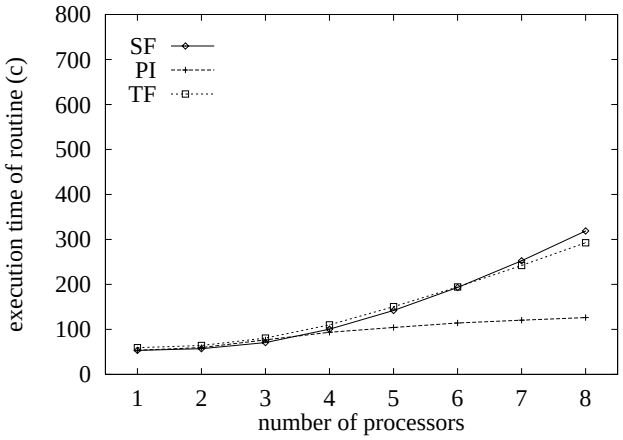

Figure 10: Average Exec. Times of Routine (c)

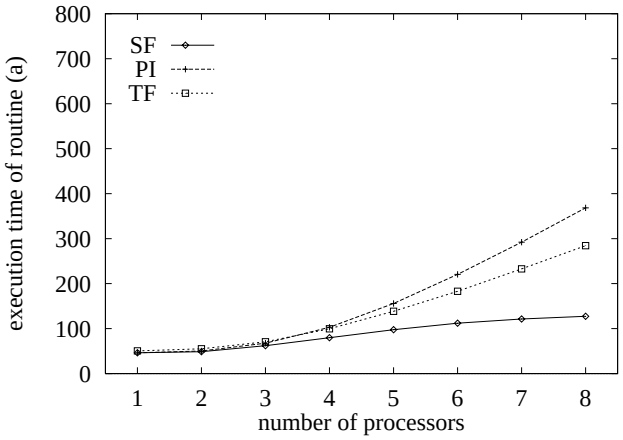

Figure 11: Average Exec. Times of Routine (a)

From these results, we can see that our proposed method is the most appropriate algorithm of the three methods from the viewpoint of real-time scalability.

Finally, in order to examine the average performance of the algorithms, we present the average execution times of routine (c) and (a) in case of three routines in Figure 10 and 10 respectively. Because the difference between SF and TF is very small in routine (c) (Figure 10), we can say that SF is more appropriate in case that improving average performance is the primary concern.

5 Nesting in Three or More Levels

When FIFO spin locks are simply used when the maximum nesting level of locks is m, the maximum execution times of a whole critical section become $O(n^m)$. An effective method to improve this order is proposed in this section.

Priority Inversion Problem

When the maximum nesting level of locks is more than or equal to three, the method proposed in Section 3 does not work effectively due to uncontrolled priority inversions.

Consider the example that processors execute one of the three routines in Figure 12 in random order. Assume the case that a processor P_1 holds L_3 and waits for L_2 in routine (c), and that another processor P_2 with a lower priority than P_1 holds L_2 and tries to acquire L_1 in (a). Processors with priorities lower than P_1 and higher than P_2 can acquire L_1 with precedence over P_2. While P_2 is waiting for those processors, P_1 must wait also and the duration of the priority inversion becomes long. As a result (we omit the detailed discussion here), the maximum execution times of (c) cannot be improved to $O(n)$. Note that this uncontrolled priority inversions do not occur when the maximum nesting level is two.

Incorporating Priority Inheritance Scheme

A priority inheritance scheme should be adopted to solve this problem. With the basic priority inheritance scheme in which a processor holding some locks inherits the highest priority of the processors that are waiting for one of the locks, the duration of priority inversions can be reduced. Since chained priority inversions cannot be avoided with this method, however, the maximum execution times of a critical section become $O(n \cdot e^m)$ with the following reason.

The maximum duration of priority inversions can be estimated as follows. Until a processor finishes the

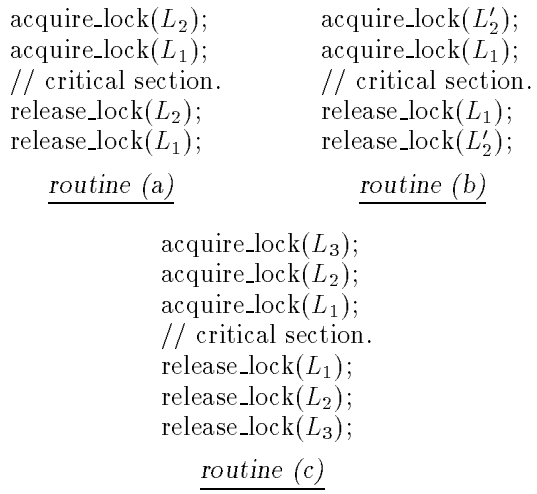

Figure 12: Nesting in Three Levels

execution of a critical section guarded by a lock L_m whose nesting level is m, it must wait for some lower priority critical sections. We denote the maximum number of these critical sections as $inv(m)$. When a processor P_1 tries to acquire L_m, another processor P_2 with a lower priority possibly holds the lock and P_1 must wait for the critical section executed by P_2. If the nesting level of the lock is one (i.e. $m = 1$), no other priority inversions can occur, thus $inv(1) = 1$. When $m > 1$, at most $inv(m-1)$ priority inversions also occur during P_2 is executing the critical section because P_2 may try to acquire another lock whose nesting level is smaller than m within the critical section. After P_1 succeeds to acquire L_m, it may also try to acquire another lock whose nesting level is smaller than m within the critical section. During its execution, at most $inv(m-1)$ priority inversions can occur. As the result, $inv(m) = 2 \cdot inv(m-1) + 1$ then $inv(m) = 2^m - 1$.

We can estimate the order of the maximum execution times of a critical section using $inv(m)$. During a processor executes a critical section guarded by L_m, at most $n-1$ higher priority critical sections are executed. Because each higher priority critical section suffers at most $inv(m-1)$ priority inversions, the maximum number of the critical sections that a processor must wait for is smaller than $(n-1)(inv(m-1)+1) + inv(m-1) = n \cdot 2^{m-1} - 1$. Note that this also includes some overestimations.

As a result, the order of the maximum execution times of critical sections is shown to be $O(n \cdot e^m)$ with the basic priority inheritance scheme. We can say that this method has real-time scalability on the number of contending processors but not on the maximum nesting level.

The priority ceiling policy can also be adopted, when there is prior knowledge on which locks are acquired in each critical section. In the concrete, when a processor acquires the outermost lock, the priority ceiling of the other locks that are required (or possibly required) by the processor within the critical section is set to the priority of the processor. When the priority ceiling of the lock that a processor tries to acquire is higher than its priority, the processor must wait with spinning even if the lock is not held by any processor[2].

In Section 3, we have mentioned the method that a processor trying to acquire nested locks reserves its turn to acquire the inner locks by enqueueing itself to their wait queue when it begins waiting for the outermost lock. When complete knowledge on all required locks in each critical section is available, the priority ceiling method is same with this method. To the contrary, if there is no knowledge on required locks at all, the priority ceiling method reduced to the situation that all shared resources in the system are guarded by a single lock, which severely degrades concurrency of the system.

6 Conclusion

In this paper, real-time scalability of nested spin locks is discussed. An algorithm with which the maximum execution times of critical sections are $O(n)$ when the maximum nesting level of locks in the system is two is proposed, and its effectiveness is demonstrated with performance evaluation. By introducing a priority inheritance scheme to the algorithm, it can be applied to the system in which the maximum nesting level is more than two.

Though this paper focuses on bounded spin locks (in other words, on the cases when each processor equally contends for nested spin locks ignoring the priority of the job it is executing), some of the results are also applicable to priority-ordered spin locks (the cases when each processor has its priority determined from the job it is executing). (1) When processors with the same priority should execute critical sections following a FIFO policy, our proposed method should be adopted. In this case, a pair of the native priority of a processor and the time stamp it obtains should be

[2]Though induced from the same policy, the behavior of "priority ceiling spin lock" is quite different from those of the priority ceiling protocol [6] or its extension for shared memory multiprocessors [7]. This is because the processor which cannot acquire a lock is blocked with those protocols, while it spins with our situation.

used as the priorities for inner locks. (2) Priority inheritance scheme is indispensable for nested priority-ordered spin locks. Note that uncontrolled priority inversions also occur when the maximum nesting level is two in case of priority-ordered spin locks.

Efficient spin lock algorithms implementing a priority inheritance scheme and performance evaluations with them are remaining as future work. We also plan to extend the algorithms to support timeouts or preemptions.

Acknowledgment

We would like to thank members of Real-Time Systems Group of Sakamura Laboratory and anonymous reviewers for their useful comments.

References

[1] T. S. Craig, "Queuing spin lock algorithms to support timing predictability," in *Proc. Real-Time Systems Symposium*, pp. 148–157, Dec. 1993.

[2] H. Takada and K. Sakamura, "Towards a scalable real-time kernel for function-distributed multiprocessors," in *Proc. of 20th IFAC/IFIP Workshop on Real Time Programming*, Nov. 1995. (to appear).

[3] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. on Computer Systems*, vol. 9, pp. 21–65, Feb. 1991.

[4] E. P. Markatos, "Multiprocessor synchronization primitives with priorities," in *Proc. of the IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.

[5] T. Johnson and K. Harathi, "A prioritized multiprocessor spin lock," Tech. Rep. TR-93-005, Department of Computer Science, University of Florida, 1993.

[6] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Computers*, vol. 39, pp. 1175–1185, Sept. 1990.

[7] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *Proc. Int'l Conf. Distributed Computing Systems*, pp. 116–123, May 1990.