Predictable Spin Lock Algorithms with Preemption

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

Both predictable interprocessor synchronization and fast interrupt response are required for real-time systems constructed using asymmetric shared-memory multiprocessors. This paper points out the problem that conventional spin lock algorithms cannot satisfy both requirements at the same time. To solve this problem, we have proposed an algorithm which is an extension of queueing spin locks modified to be preemptable for servicing interrupts [1]. In this paper, we propose an improved algorithm that minimizes the recovering overhead from an interrupt service. We also demonstrate that the proposed algorithms have required properties through performance measurement.

1 Introduction

In many applications of high performance real-time systems, a large number of external devices such as sensors, actuators, and network controllers are connected to a system and the system is required to respond to the external events from the devices within predefined and usually short time-bounds. To meet this requirement, asymmetric multiprocessors in which each device is handled by a fixed processor are often adopted.

In order to realize real-time systems using shared-memory multiprocessors, predictable interprocessor synchronization mechanisms are of primary importance. In addition to adopting a real-time scheduling algorithm with resource constraints or a real-time synchronization protocol, the execution time of the underlying mutual exclusion mechanism using spin locks must be bounded[1].

In asymmetric shared-memory multiprocessors, each processor is required to achieve fast and predictable response to interrupt requests, because external events are notified to each processor in the form of interrupts. However, each processor cannot respond to external interrupts in short latency with conventional bounded spin lock algorithms.

To solve this problem, we have proposed an algorithm which is an extension of queueing spin locks modified to be preemptable for servicing interrupts [1]. With the algorithm, an upper bound on the time to acquire and release an interprocessor lock can be given when no interrupt request occurs, and fast response to interrupt requests is achieved. However, the algorithm has a shortcoming that a processor possibly has to re-execute the lock acquiring routine from the *beginning* after it services an interrupt request. In schedulability analysis, this re-execution overhead must be added to the interrupt service time.

In this paper, we propose an improved algorithm that minimizes this overhead. We also demonstrate that the proposed algorithms have required properties through performance measurement.

2 Spin locks and interrupt latency

In this paper, we assume that 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) are supported in hardware.

In order to bound the time until a processor acquires an interprocessor lock, the duration that each processor holds the lock must be bounded as well as the number of contending processors that the processor must wait for. The latter condition can be met with ticket locks or queueing locks [2], with which the turn that a processor acquires a lock is determined when it begins waiting for the lock. To satisfy the former condition, the relationship with interrupt services must be considered.

In asymmetric multiprocessor systems, interrupt services for external devices are requested for each processor. When multiple devices are connected to a processor, interrupt requests from them are usually raised independently and the maximum time to service all of the requests becomes unbounded or very long. Consequently, in order to give a practical bound on the duration that a processor holds a lock, interrupt services should be inhibited for that duration.

On the other hand, in order to realize a system with fast response to external events, each processor must be able to service external interrupts with short latency time. Particularly, when the scalability of the system is an impor-

[1] We assume that the access time of the shared bus (or interconnection network) is bounded in this paper.

tant issue, the worst-case interrupt latency should be given independently of the number of processors in the system.

Here a problem arises in deciding whether interrupts should be disabled first or an interprocessor lock should be acquired first. When acquiring an interprocessor lock precedes disabling interrupts, interrupts may be serviced while the processor holds the lock, and the condition that interrupt services should be inhibited while a processor holds a lock is not satisfied. If acquiring a lock follows disabling interrupts, on the other hand, the interrupt mask time includes the time to acquire the lock and its upper bound heavily depends on the number of processors.

One method to solve this problem is the following. The processor first disables interrupts and tries to acquire the lock. If it fails to acquire the lock, the processor probes interrupt requests before it retries to acquire the lock. When interrupt requests are detected, it suspends trying to acquire the lock, enables interrupts, and services them.

Test-and-set locks can be extended easily with this method. Ticket locks and queueing locks, on the other hand, cannot be extended similarly.

3 Queueing locks with preemption

In all spin lock algorithms that can give an upper bound on the time until a processor acquires a lock, a processor modifies some shared variable and reserves its turn to acquire the lock when it begins waiting for the lock. When its turn comes, the lock is passed to the processor by another. If the processor simply branches to an interrupt handler while waiting for the lock, it cannot begin to execute the critical section immediately after the lock is passed to the processor, and makes the contending processors wait wastefully until the interrupt service is finished.

Consequently, when a processor begins to service interrupts while waiting for a lock, it must inform others that it is servicing interrupts and should not be passed the lock. The processor trying to release the lock checks if the succeeding processor is servicing interrupts. If the succeeding one is found to be servicing interrupts, its turn to acquire the lock is canceled or deferred, and the lock is passed to the next in line.

Original algorithm

We have applied the above scheme to the MCS lock, a list-based queueing lock algorithm [2], and proposed a queueing lock algorithm with preemption [1]. Some other spin lock algorithms can be extended similarly. Recently, R. W. Wisniewski et al. have proposed a similar algorithm for improving the average performance of multiprogrammed (non-real-time) systems [3]. Craig's algorithm can also support the same preemption scheme [4].

In the algorithm, if the processor trying to release the lock (P_0) finds that the succeeding processor (P_1) is servicing interrupts, P_0 dequeues P_1 from the waiting queue and passes the lock to a successor of P_1. When only P_1 is waiting for the lock, P_0 makes the waiting queue empty. P_0 informs P_1 that P_1 is dequeued using a shared variable. When P_1 finishes the interrupt service, it checks whether it has been dequeued during the interrupt service or not. If it has been dequeued, it re-executes the lock acquiring routine from the beginning. Otherwise, it resumes waiting for the lock.

When a processor is dequeued and re-executes the lock-acquiring routine, the waiting time after the processor first links itself to the queue until it branches to the interrupt handler is wasted. When the schedulability of the system is analyzed, this re-execution overhead should be added to the interrupt service time. Below, we present an improved algorithm which is devised to reduce this overhead.

Improved algorithm

The re-execution overhead can be reduced with the following method. When the processor releasing the lock (P_0) finds that the succeeding processor (P_1) is servicing interrupts, P_0 leaves P_1 in the waiting queue instead of dequeueing it. P_0 removes the processor to which to pass the lock from the queue using the method adopted in the prioritized queueing spin lock appeared in [5]. When P_1 finishes interrupt services, it simply resumes waiting for the lock in its original position. Therefore, the overhead which must be added to the interrupt service time in schedulability analysis is minimized.

A difficulty occurs when all processors in the waiting queue are servicing interrupts. To handle this situation, a global lock flag is introduced. If the processor trying to release the lock finds that all processors in the queue are servicing interrupts, it sets the global lock flag. A processor returning from interrupt services tries to get the global lock with the same method as with test-and-set locks. If it succeeds getting the lock, it removes itself from the waiting queue. As the processor needs to know the top processor in the queue to remove itself, the processor releasing the global lock must pass the information in some shared variable. It is also necessary for a processor to check the global lock flag once, after it links itself at the end of the queue, because it is possible that all the processors in the queue are servicing interrupts and the global lock is set.

Pseudo-code for the improved algorithm appears in Fig. 1 and 2. In these figures, the keyword **shared** indicates that only one instance of the variable is allocated and shared in the system. Other variables are allocated for each processor and located in its local memory. The right hand side of the **and** operator is assumed to be evaluated only if its left hand side is true. **Fetch_and_store** reads the memory addressed by the first parameter, returns the contents of the memory as its value, and atomically writes the second parameter to the memory. **CAS**, the abbreviation of compare_and_swap, first reads the memory pointed to by the first parameter and compares its contents

```
    type qnode = record
        next, prev: pointer to qnode;
        locked: (Released, Locked, Preempted, Dequeueing)
    end;
    type lock = record
        last: pointer to qnode;
        glock: pointer to qnode
    end;

    // global shared data.
    shared var L: lock;
    // L.last and L.glock are initialized to NIL.

    procedure dequeue(entry, pred, top: pointer to qnode)
        var succ: pointer to qnode;
        succ := entry→next;
        if succ = NIL then
            pred→next := NIL;
            if CAS(&(L.last), entry, pred) then goto release end;
#           repeat succ := entry→next until succ ≠ NIL
        end;
        pred→next := succ;
        succ→prev := pred;
      release:
        entry→next := top;
        entry→locked := Released
    end;
```

Fig. 1: Improved algorithm (1)

with the second parameter. If they are equal, the function writes the third parameter to the memory atomically and returns true. Otherwise, it returns false.

In this pseudo-code, the **glock** field of **L** serves both as the global lock flag and as the variable to pass the top processor of the waiting queue. An exponential backoff scheme is adopted to get the global lock in this code to reduce the number of shared-bus transactions. Two constant parameters α and β should be tuned for each target hardware and application.

Though there are two non-local spins (marked with #) in this pseudo-code, both of them continue during the transient state afer another processor writes the pointer to its queue node to **L.last** (successful execution of the fetch_and_store operation marked with ①) and until it writes non-NIL value to the **next** field of its predecessor (marked with ②), and their effect is not significant.

We have adopted the MCS lock as the base algorithm in this section. The FIFO version of Craig's algorithm [4] can be extended similarly.

4 Performance evaluation

The effectiveness of the two queueing spin lock algorithms with preemption, the original one in [1] (called QL/P1, in this section) and the improved one presented in Fig. 1 and 2 (QL/P2), are examined through performance evaluation. The performance of the algorithms is compared with the MCS lock without inhibiting interrupts (QL/ei), the MCS lock during interrupts inhibited (QL/di),

```
    // local data (allocated for each processor).
    var I: qnode;
    var pred, succ, top: pointer to qnode;
    var interval, i: integer;

    I.next := NIL;
    disable_interrupts;
①   pred := fetch_and_store(&(L.last), &I);
    if pred = NIL then goto acquired end;
    // enqueue myself.
    I.prev := pred;
    I.locked := Locked;
②   pred→next := &I;
    i := 1;         // check the global lock once.
    interval := ∞;          // never expires.
    while (I.locked ≠ Released) do
        if interrupt_requested and
                CAS(&(I.locked), Locked, Preempted) then
            enable_interrupts;
            // interrupt service.
            disable_interrupts;
            I.locked := Locked;
            i := 1;
            interval := α
        end;
        i := i - 1;
        if i = 0 then
            // check the global lock and try to get if it is set.
            top := L.glock;
            if top ≠ NIL and CAS(&(L.glock), top, NIL) then
                if top ≠ &I then dequeue(&I, I.prev, top) end;
                goto acquired
            end;
            i := interval;
            interval := interval × β
        end
    end;
  acquired:
    //
    // critical section.
    //
    succ := I.next;
    if succ = NIL then
        // try to make the queue empty.
        if CAS(&(L.last), &I, NIL) then goto exit end;
        repeat succ := I.next until succ ≠ NIL
    end;
    // try to pass the lock to the successor.
    if CAS(&(succ→locked), Locked, Released) then goto exit end;
    top := succ;
    repeat
        pred := succ;
        succ := pred→next;
        if succ = NIL then
            // set the global lock.
            L.glock := top;
            // check if pred is really the last processor.
            if L.last = pred then goto exit end;
            // try to withdraw the global lock.
            if ¬CAS(&(L.glock), top, NIL) then goto exit end;
#           repeat succ := pred→next until succ ≠ NIL
        end;
    until CAS(&(succ→locked), Locked, Dequeueing);
    dequeue(succ, pred, top);
  exit:
    enable_interrupts;
```

Fig. 2: Improved algorithm (2)

```
for i := 1 to NoLoop do
① acquire_lock_and_disable_interrupts;
   //
   // critical section.
   //
   release_lock;
② enable_interrupts;
   random_delay
end;
```

Fig. 3: Measurement program skeleton

and the test-and-set lock with preemption with constant delay (T&S/P)[2].

Evaluation environment

We have used a shared-bus multiprocessor system for the evaluation. The shared bus is based on the VME-bus specification, and each processor node consists of a 20 MHz GMICRO/200 microprocessor, which is rated at approximately 10 MIPS, 1 MB of local memory, and some I/O interfaces. The local memory can be accessed from other processors through the shared bus. No cache memory is equipped. The program code and the data area for each processor are placed in the local memory of the processor. G lobal shared data (e.g. **L** in Fig. 1) is placed in the local memory of the master processor, which does not execute spin locks.

The GMICRO/200 microprocessor supports the compare_and_swap instruction but not fetch_and_store. In our experiments, the fetch_and_store operation was emulated using the 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.

Measurement method

Each processor executes the code presented in Fig. 3 while periodic interrupt requests are raised on the processor. The execution time of a critical region (the region between ① and ② in Fig. 3) is measured for each execution, and its distributions when the processor services no interrupt request during the region and when it services an interrupt are collected. The interrupt latency is also measured for each interrupt service and its distribution is obtained.

Inside the critical section, a processor accesses the shared bus some number of times (for making the effect of bus traffic explicit) and waits for a while using empty loops. Without spin locks, the execution time of the critical region is about 40 μs including some overhead for obtaining the execution time of the region. In order to change timing conditions, each processor waits for a random time before it re-enters the critical region (*random_delay* in Fig. 3). The average time of the random delay is about 40 μs.

Empty loops are also included in the interrupt handler in addition to the routine for obtaining interrupt latency time. The total execution time of the interrupt handler is about 80 μs. The period of interrupt requests is about 5 ms. The exact length of the period is varied in 0–2% for each processor.

Performance metric

In real-time systems, the effectiveness of algorithms should not be evaluated with their average performance but with their worst-case execution (or response) times. However, in the case of spin lock algorithms, worst-case times cannot be obtained through experiments because of unavoidable non-determinism in multiprocessor systems. Therefore, in place of worst-case times, we have adopted p-reliable times, the time within which a processor finishes executing a critical region (or responds to an interrupt request) with probability p, as a performance metric. In the following section, we show the evaluation results when p is 0.999 (i.e. 99.9%).

Evaluation results

Fig. 4 presents the 99.9%-reliable execution time of the critical region (when no interrupt is serviced on the processor during the region) as the number of processors is increased from one to eight. With QL/P1 and QL/P2, the execution time of the critical region increases linearly with the number of processors, and the algorithms are found to be scalable. QL/ei exhibits poorer performance because preceding processors service interrupt requests during the critical region.

In Fig. 5, the interrupt latency time is nearly independent of the number of processors with QL/P1 and QL/P2. With QL/di on the contrary, the interrupt latency becomes long as the number of processors increases.

From these observations, it is demonstrated that QL/P1 and QL/P2 can give a practical upper bound on the time to acquire and release an interprocessor lock while achieving fast response to interrupt requests. The other algorithms cannot satisfy these two requirements at the same time.

The overall performance of QL/P2 is a little worse than QL/P1, because the number of shared-bus transactions is large with QL/P2 and because doubly linked queue is necessary. The advantage of QL/P2 appears in Fig. 6 which presents the 99.9%-reliable execution time of the critical region when an interrupt is serviced during the region. When the number of processors is large, the recovering overhead from interrupt services is much smaller in QL/P2 than in QL/P1.

[2] Past studies show that a test-and-set lock has good scalability with exponential backoff [2]. However, because the lock acquisition time varies widely with exponential backoff, it is inappropriate for real-time systems. This conjecture was also confirmed through our experiments.

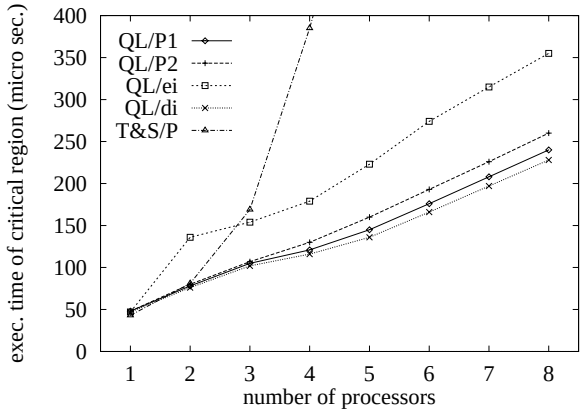

Fig. 4: 99.9%-reliable exec. time of critical region (when no interrupt is serviced)

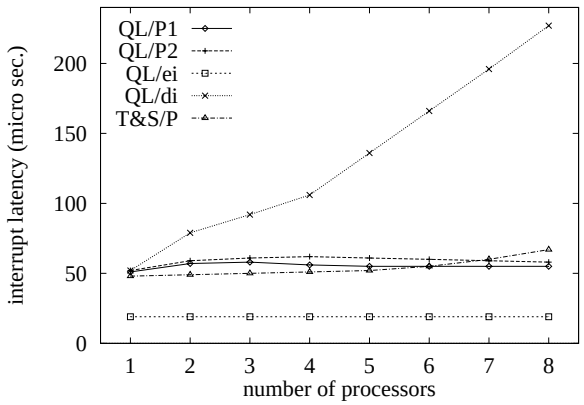

Fig. 5: 99.9%-reliable interrupt latency

Finally, in order to examine the average performance of the algorithms, we present the average execution time of the critical region (when no interrupt is serviced during the region) in Fig. 7.

5 Conclusion

Conventional spin lock algorithms cannot satisfy two important requirements for real-time systems using asymmetric shared-memory multiprocessors, predictable spin locks and fast interrupt response, at the same time. In this paper, we propose a improved spin lock algorithm that can give an upper bound on the time to acquire and release an interprocessor lock while realizing fast response to interrupt requests. To evaluate their effectiveness, we have measured their performance through experiments and confirmed that the algorithms have the required properties.

We are currently designing a real-time kernel specification called ITRON-MP and implementing it experimentally [6]. It remains as a future work to adopt the algorithms in the implementation and to evaluate the algorithms in real applications.

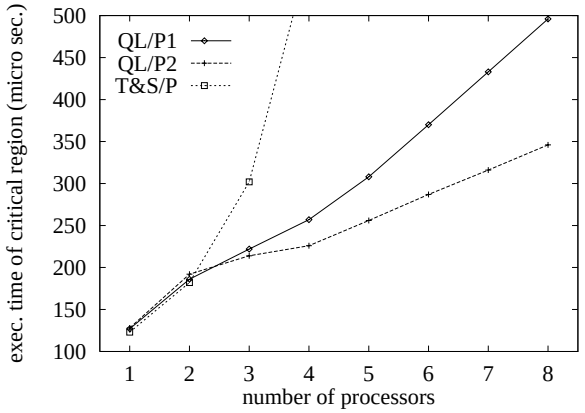

Fig. 6: 99.9%-reliable exec. time of critical region (when an interrupt is serviced)

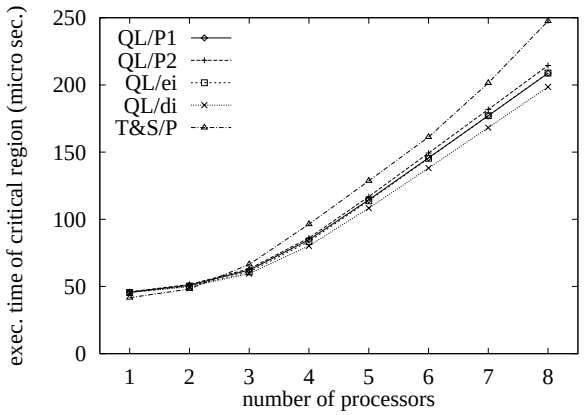

Fig. 7: Average exec. time of critical region

References

[1] H. Takada and K. Sakamura, "A bounded spin lock algorithm with preemption," Tech. Rep. 93-2, Department of Information Science, University of Tokyo, July 1993.

[2] 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.

[3] R. W. Wisniewski, L. Kontothanassis, and M. L. Scott, "Scalable spin locks for multiprogrammed systems," Tech. Rep. TR454, Computer Science Department, University of Rochester, Apr. 1993.

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

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

[6] H. Takada and K. Sakamura, "ITRON-MP: An adaptive real-time kernel specification for shared-memory multiprocessor systems," *IEEE Micro*, vol. 11, pp. 24–27,78–85, Aug. 1991.