TECHNICAL REPORT 97-01

A Novel Approach to Multiprogrammed Multiprocessor Synchronization for Real-Time Kernels

Hiroaki Takada and Ken Sakamura

January 29, 1997

Department of Information Science
Faculty of Science, University of Tokyo

7–3–1 Hongo, Bunkyo-Ku, Tokyo, 113 Japan

TECHNICAL REPORT 97-01

TITLE

A Novel Approach to Multiprogrammed Multiprocessor Synchronization for Real-Time Kernels

AUTHORS

Hiroaki Takada and Ken Sakamura

KEY WORDS AND PHRASES

multiprogrammed multiprocessor, synchronization, real-time system, SPEPP synchronization, wait-free synchronization, preemption-safe locking

ABSTRACT

In order to solve the problem of inopportune preemption in multiprogrammed multiprocessor synchronization, two strategies which are applicable to real-time systems have been investigated: preemption-safe locking and wait-free synchronization. Either of them, however, has a problem for use in the implementation of a real-time kernel. Preemption-safe locking has the drawback that the preemption overhead becomes quite large; while wait-free operations on complex data structures are generally very inefficient and are not practical.

In this paper, we propose a novel approach to multiprogrammed multiprocessor synchronization, called the SPEPP (Spinning Processor Executes for Preempted Processors) synchronization, with which the preemption overhead can be reduced to zero, while operations on complex data structures can be realized with reasonable overhead. This paper presents the two algorithms of the SPEPP synchronization, and demonstrates its effectiveness through the performance measurements of a real-time kernel implemented with the SPEPP synchronization.

REPORT DATE	**WRITTEN LANGUAGE**
January 29, 1997	English
TOTAL NO. OF PAGES	**NO. OF REFERENCES**
13	11

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT

DISTRIBUTION STATEMENT

First issue 45 copies. This technical report is available via anonymous FTP from `ftp.is.s.u-tokyo.ac.jp` (directory `/pub/tech-reports`).

SUPPLEMENTARY NOTES

DEPARTMENT OF INFORMATION SCIENCE

Faculty of Science, University of Tokyo

7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

A Novel Approach to Multiprogrammed Multiprocessor Synchronization for Real-Time Kernels

Hiroaki Takada
Department of Information Science,
School of Science, University of Tokyo

Ken Sakamura
The University Museum,
University of Tokyo

Technical Report 97-01

January 29, 1997

Abstract

In order to solve the problem of inopportune preemption in multiprogrammed multiprocessor synchronization, two strategies which are applicable to real-time systems have been investigated: preemption-safe locking and wait-free synchronization. Either of them, however, has a problem for use in the implementation of a real-time kernel. Preemption-safe locking has the drawback that the preemption overhead becomes quite large; while wait-free operations on complex data structures are generally very inefficient and are not practical.

In this paper, we propose a novel approach to multiprogrammed multiprocessor synchronization, called the SPEPP (Spinning Processor Executes for Preempted Processors) synchronization, with which the preemption overhead can be reduced to zero, while operations on complex data structures can be realized with reasonable overhead. This paper presents the two algorithms of the SPEPP synchronization, and demonstrates its effectiveness through the performance measurements of a real-time kernel implemented with the SPEPP synchronization.

1 Introduction

When a real-time system is constructed on a shared-memory multiprocessor, it is a common practice to use a real-time kernel[1] to execute multiple tasks preemptively on each processor. A real-time kernel designed for multiprocessor systems should also support some communication mechanisms among tasks on different processors. Various data structures within the real-time kernel are usually shared among processors and guarded with busy-wait mutual exclusion locks.

In realizing mutual exclusion locks used in a real-time kernel, the following difficulty exists. In order to bound the maximum execution time of a task that requires a lock, the turn that the task acquires the lock should be reserved when it begins waiting for the lock. Typically, this is realized by handing round the lock in an FCFS order. If the lock is handed to the task while the task is preempted, it cannot begin to execute the critical section immediately and holds the lock for an unnecessarily long duration. As the result, the other tasks waiting for the lock cannot acquire the lock in time. This difficulty has been discussed as one of the problems of multiprogrammed multiprocessor synchronization [1].

Two strategies have been investigated to solve this difficulty: preemption-safe locking and wait-free (or block-free) synchronization [2]. Wait-free synchronization guarantees that any task can complete an operation on a shared data structure in a finite number of steps, regardless of the execution speed of the other tasks [3], while block-free synchronization guarantees that one

[1] In this paper, the word "real-time kernel" indicates a basic run-time system for real-time systems supporting task management, priority-based preemptive scheduling, inter-task synchronization and communication, and some other functions, which is also called as a real-time monitor or a real-time executive.

of the contending tasks can. Block-free synchronization is not appropriate for real-time systems, because it cannot bound the worst-case execution time that a task executes an operation. Herlihy presented a general construction of wait-free synchronization operations, assuming a universal atomic primitive, such as compare_and_swap (CAS) or the pair of load_linked and store_conditional [3]. However, wait-free (or block-free) operations on complex data structures are very inefficient and are not practical [2].

Preemption-safe locking is a class of spin lock algorithms with which a lock is not handed to preempted tasks. The MCS lock [4], which is an FCFS-ordered queueing spin lock algorithm and thus is suitable for real-time systems, has been extended to be preemption-safe [5, 6]. With the preemption-safe version of the MCS lock, when a task that is waiting for a lock is preempted, it modifies a shared variable and informs others that it is preempted. The task releasing the lock checks the state of the succeeding task in the waiting queue. If the succeeding one is preempted, its reservation to acquire the lock is canceled and the lock is handed to the next one in line. When a task finds that its reservation has been canceled while preempted, it re-executes the lock acquisition routine from the beginning. Obviously, the maximum time of this re-execution, called the preemption overhead below, is quite long depending on the number of contending tasks. In order to bound the preemption overhead with a constant time length, we have proposed an improved algorithm with which the reservation is not canceled but is postponed [7]. When the preempted task is scheduled again, it resumes waiting for the lock in its original position.

This paper proposes a novel approach to multiprogrammed multiprocessor synchronization, with which the preemption overhead can be reduced to zero, for use in a scalable real-time kernel for function-distributed multiprocessors [8]. In case of function-distributed multiprocessors, interrupt requests are raised on each processor, and the tasks on the processor can be preempted with them. The maximum time that a task waits for a lock is extended by the preemption overhead, whenever the task is preempted by an interrupt handler. Consequently, when the schedulability of the system is analyzed, the preemption overhead should be added to the interrupt service time, and thus reducing the preemption overhead is significant for improving the schedulability of the system. Though the preemption overhead becomes zero with the general construction of wait-free operations presented by Herlihy, shared data structures in a real-time kernel is too complex to be implemented efficiently with wait-free operations.[2]

Our approach is to incorporate the realization concept of the general construction of wait-free operations into the locking-based synchronization. In detail, when a task tries to acquire a lock, the operation that the task will execute in the critical section is posted to the waiting queue for the lock. If the turn that the task acquires the lock comes while it is preempted, the posted operation is executed by another processor that is spinning on the lock. With this method, named the SPEPP synchronization (*SPEPP* stands for "*Spinning Processor Executes for Preempted Processors*"), operations on a shared data structure are executed in an FCFS order regardless of the preemptions. If one of the tasks that are waiting for the lock is not preempted, the execution of the operations continues to make progress. If all the tasks are preempted, the execution ceases to progress. It is obvious, however, that the time duration that the execution is suspended is shorter than the duration that any one of the tasks is preempted. Therefore, the extension of the maximum time until the operation of a task is finished is shorter than or equal to the duration that the task is preempted. As the result, the preemption overhead is reduced to zero.

In Section 2, two algorithms of the SPEPP synchronization are presented. We first present the basic version assuming that more than one tasks on a same processor do not acquire a same lock. We then extend the algorithm to remove the assumption. In Section 3, the effectiveness of the SPEPP synchronization is investigated through performance measurements. We have implemented three versions of real-time kernels, two with preemption-safe lockings and one with the SPEPP synchronization, and compared their performance.

[2] To the best of our knowledge, no wait-free implementation of operating systems has been reported, while some block-free implementations have [9, 10]

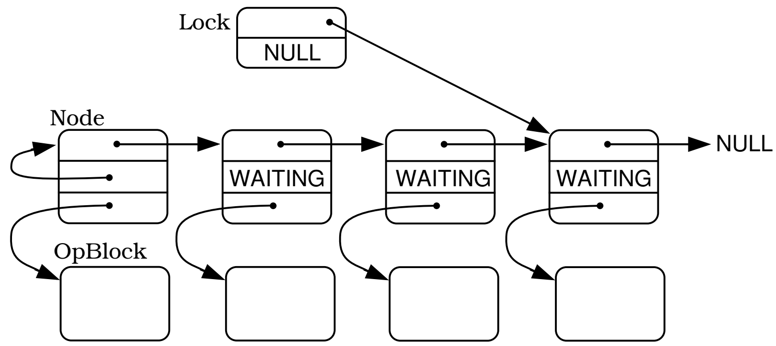

Figure 1: The Data Structures of the SPEPP Synchronization Algorithm

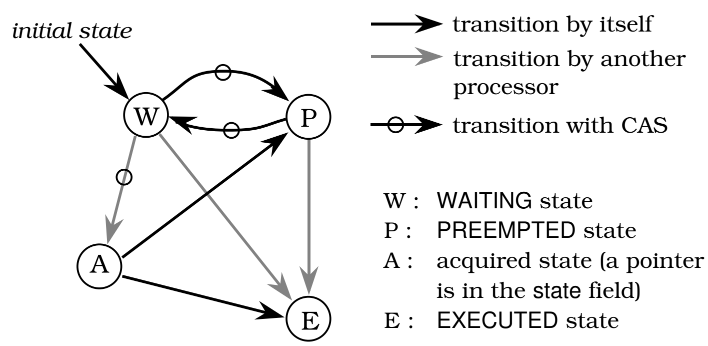

Figure 2: State Transition of a Queue Node

2 The SPEPP Synchronization

2.1 The Basic Algorithm

In this section, we present the basic version of the SPEPP synchronization algorithm with the restriction that each lock is acquired by at most one task on each processor. In the following, we assume that task switches are triggered by interrupt requests.

Pseudo-code for the algorithm appears in Figure 10, 11, and 12. In these figures, the lines beginning with "//" are comments. The keyword shared indicates that only one instance of the variable is allocated and shared in the system. Other variables are allocated for each task and located on the local memory of its host processor.[3] SWAP 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 addressed by the first parameter and compares its contents with the second parameter. If they are equal, the operation writes the third parameter to the memory atomically and returns true. Otherwise, it returns false.

Lock and Node are the data structures for a mutual exclusion lock and a queue node linked in the waiting queue, respectively. The last field of Lock points to the last node in the waiting queue, and the next field of Node points to the succeeding node in the queue. When the lock is idle and the waiting queue is empty, the last field is assigned NULL. The state field of Node includes a pointer to a queue node or one of the three special values (WAITING, PREEMPTED, and EXECUTED), and indicates the current state of the queue node. The op field points to the data block of the OpBlock type, in which the kind of operation and the parameters passed to it are stored. Also, the return values of the operation are written in this data block (Figure 1).

At first, the task trying to acquire a lock enqueues its node at the end of the waiting queue with a SWAP operation (Line 63). When the queue has been empty, the task succeeds in acquiring

[3] We assume that a processor can directly access its local memory without using the shared bus or interconnection network.

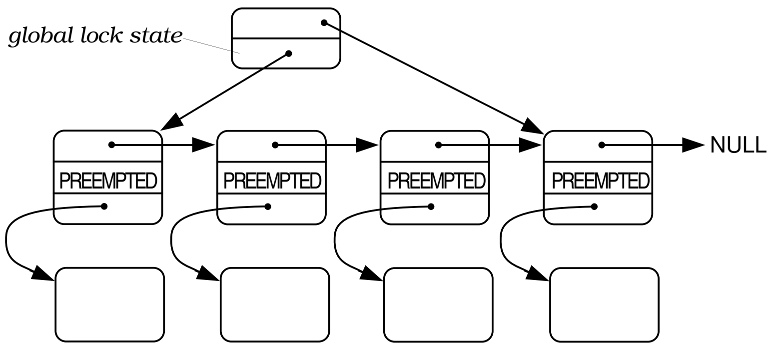

Figure 3: The Global Lock State

the lock. Otherwise, it initializes its state to WAITING (Line 69), makes the predecessor point to its queue node (Line 71), and begins to spin. When the task detects an interrupt request while spinning, it changes its state to PREEMPTED and services the request (Line 90). After the interrupt service, the task simply recovers its state from PREEMPTED to WAITING with a CAS operation (Line 82) and resumes to spin. When the lock is handed to the task, the state of the task is changed to the pointer to the head node of the waiting queue. This state is called the acquired state. If the task is at the head of the queue (in other words, if the state field points to its own queue node (Line 127)), the task executes its own operation (Line 142) and then releases the lock (Line 143). Otherwise, the task executes the operation posted to the head node (Line 129), and then proceeds to the succeeding node in the queue (Line 136), until the task's own node becomes the head of the queue (Line 127). When the operation of the task is executed by another processor, its state is changed to EXECUTED. In this case, the task simply exits the routine (Line 123). Figure 2 illustrates the state transition of the queue node of a task.

A difficult situation occurs when all the tasks waiting for a lock are preempted. In this case, there are no task to which to hand the lock, but the waiting queue should not be made empty. To cope with this difficulty, an additional field glock, called the global lock state, is introduced in the Lock data structure. When all the tasks in the waiting queue are preempted, the global lock state is assigned the pointer to the head node of the queue (Figure 3, Line 49). Each task which has resumed to spin must check the global lock state (Line 97). When a pointer is assigned to it, the task tries to acquire the lock by changing the global lock state to NULL atomically from the pointer (Line 108). Here, checking the global lock state only once is not sufficient. This is because at the moment of the checking, the task releasing the lock may be still trying to hand the lock to the succeeding ones. In order to inform that the global lock state must be checked again, a special value UPDATING is assigned to it (Line 38). In order to reduce the shared-bus traffic, we also adopt an exponential backoff scheme when checking the global lock state repeatedly (Line 103). Two constant parameters α and β should be determined for each target hardware. It is also necessary to check the global lock state, just after a task enqueues its queue node at the end of the waiting queue.

With this algorithm, the maximum duration that interrupt services are inhibited can be bounded with T plus some lock handling overhead, where T is the maximum execution time of an operation. In order to realize this feature, interrupt requests are probed while spinning on the lock (Line 88). In addition, interrupt requests must be checked after the operation requested by another task is executed (Line 130). If an interrupt request is detected at this moment, the processor hands the lock to another one (Line 132) and services the interrupt request.

On the other hand, the maximum duration that a task spins on the lock can be bounded with $(n-1)\cdot T$ plus lock handling overhead, where n is the number of contending processors for the lock, because of the restriction that at most one task on each processor acquires the lock. Consequently, the maximum execution time until a task finishes an operation[4] can be bounded with $n \cdot T$ plus lock handling overhead regardless of the occurrence of preemptions.

[4] The period during which its host processor executes other tasks is not included in this execution time.

2.2 The Extended Algorithm

The basic algorithm still works, even if the restriction on the algorithm is removed. However, the maximum duration that a task spins on the lock becomes as long as $(N-1) \cdot T$ plus lock handling overhead, where N is the number of contending tasks for the lock. Because N is much larger than n in general, the maximum execution time until a task finishes an operation becomes very long. In this section, we present an extended algorithm of the SPEPP synchronization, with which the maximum execution time until a task finishes an operation can be regarded as being bounded with $O(n)$ without the restriction.

Figure 13 presents the differences of the extended algorithm from the basic one. At first, the queue node should be prepared for each lock for each processor (instead of prepared for each task). The task trying to acquire a lock uses the queue node for the lock, even if a preempted task is using the node. When a task is scheduled again after preempted within the algorithm, it should check whether its operation has been executed during the preemption (Line 79.2). We assume that whether an operation is executed or not is recorded in the OpBlock structure and can be checked with the finished function. If the operation has not been executed, then the task checks the state of the queue node (Line 79.6). If it is EXECUTED, the queue node has been stolen by another task, and the former task should retry the operation. Otherwise, the task resumes to spin.

In order to estimate the maximum execution time until a task finishes an operation, we must consider the case when the queue node of a preempted task is stolen by another task. In this case, because the preempted task must re-execute the synchronization algorithm from the beginning, the maximum execution time to finish an operation is prolonged. Instead, the execution time that the preempting task is reduced. In more precise, when task τ_1 is preempted after spinning on a lock for the duration of T_s, the task τ_2 which steals the queue node of τ_1 can shorten its maximum waiting time for the lock by T_s. When the schedulability of the system is analyzed, we can safely count this stolen execution time of τ_1 as a part of the maximum execution time of τ_2.

When a task detects that the state of its queue node is changed to EXECUTED, it must check with the finished function whether its operation has been executed (Line 122.1). This is because what has been executed may be the operation requested by another task on the same processor. In other words, a task must possibly wait for an execution of the operation requested by another task on the same processor. As the result, with this extended algorithm, the maximum execution time until a task finishes an operation can be considered to be bounded with $(n+1) \cdot T$ plus lock handling overhead. This is a significant improvement over the simple algorithm, when n is much smaller than N.

3 Performance Measurements

We have implemented a real-time kernel in which shared data structures within the kernel are operated with the extended version of the SPEPP synchronization algorithm presented in the previous section. The application program interface of the kernel is an extension of the μITRON specification [11].

In this section, the performance of this real-time kernel is measured, and compared with the performance of two other versions of the real-time kernel implemented with preemption-safe lockings; one of them is implemented with the simple preemption-safe version of the MCS lock [5, 6] (called MCS/SPS locking in this section), and the other with the improved preemption-safe MCS lock [7] (called MCS/IPS locking).

3.1 Evaluation Environment and Performance Metric

A shared-bus multiprocessor system is used for the measurements. The shared bus is based on the VMEbus specification, and nine processor boards and a shared memory board are connected to the shared bus. Each processor board consists of a GMICRO/200 microprocessor, which is rated approximately at 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 coherent cache is equipped (Figure 4).

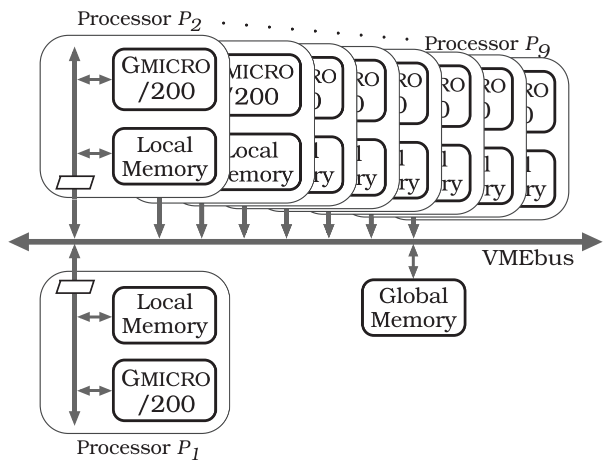

Figure 4: Evaluation Environment

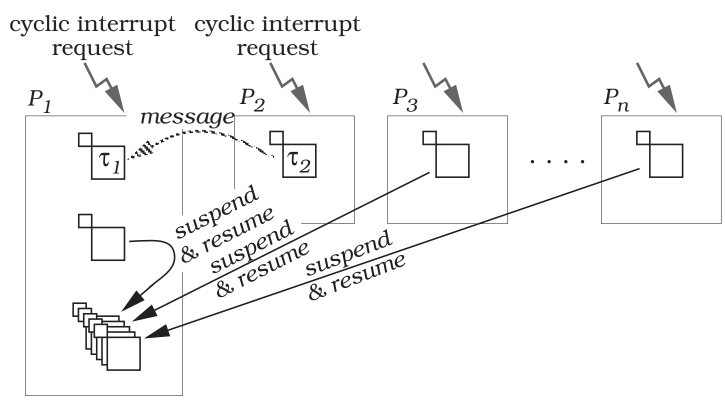

Figure 5: The Workload for the Measurements

The GMICRO/200 microprocessor supports the CAS instruction but dose not support the SWAP operation. In the measurements, the SWAP operation is emulated using the CAS instruction and a retry loop. Because 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.

Because the worst-case behavior is the primary concern in real-time systems, the effectiveness of our proposals should be evaluated with its worst-case execution (or response) times. However, worst-case times cannot be obtained through experiments because of unavoidable non-determinism in multiprocessor systems. Moreover, our evaluation environment cannot bound the execution times of various kernel services, because the access time of the shared bus has no upper bound with our environment. Therefore, in place of a worst-case time, we have adopted a p-reliable time, the time within which a processor finishes to execute (or responds) with probability p, as the performance metric. In the following section, we use the 99.9%-reliable (i.e. $p = 0.999$) execution (or response) times instead of the worst-case times.

3.2 Measurement Method

We have measured the execution times of a system call of the real-time kernels and the interrupt response times using a synthetic workload described below. A task τ_2 on processor P_2 repeatedly invokes a system call that sends a message to a task τ_1 on processor P_1, and the execution times of the system call are measured. The execution times when an interrupt request is serviced during

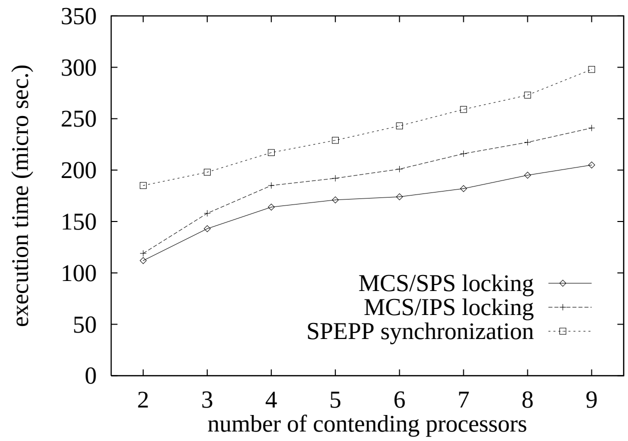

Figure 6: 99.9%-Reliable Execution Times of an Operation
(when no interrupt request is serviced)

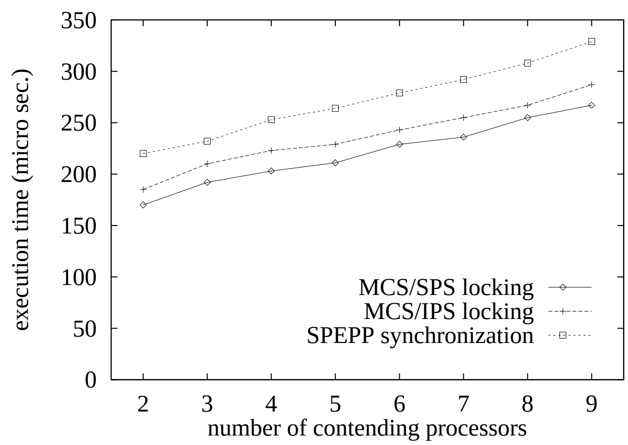

Figure 7: 99.9%-Reliable Execution Times of an Operation
(when an interrupt request is serviced)

the execution are recorded separately.

In order to interfere the message sending operation, a task on P_1 and tasks on the other processors alternately suspend and resume the execution of lower priority tasks on P_1 at random intervals. The average interval is about 1 ms. During the measurement, periodic interrupt requests are raised on each processor, and the interrupt response times are measured within the interrupt handler. The interrupt service time including the time for invoking and returning from the handler is about 33 μs. The interrupt period is around 2 ms and is varied in 0–5% for each processor, in order that the timing of interrupt requests for each processor should not be synchronized. The relation of the tasks is illustrated in Figure 5.

3.3 Measurement Results

Figure 6 and 7 present the 99.9%-reliable execution times of a message sending operation when no interrupt request is serviced and those when an interrupt request is serviced during the execution, respectively. The number of contending processors (including P_1) is changed from two (only one interfering task on P_1) to nine (eight interfering tasks). The SPEPP synchronization exhibits worse performance than the preemption-safe lockings, mainly due to the overhead of storing the parameters passed to the operation into the OpBlock structure and reading the return values from

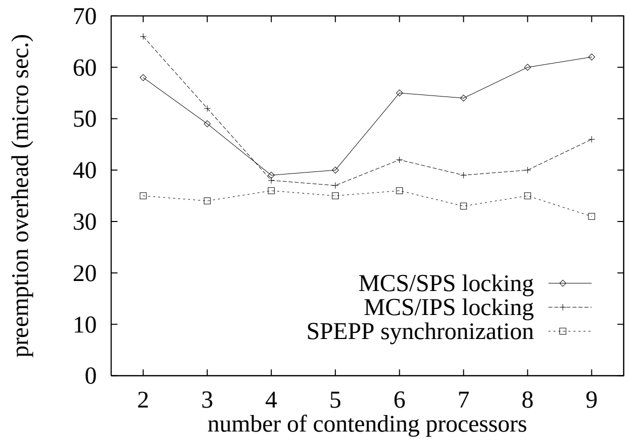

Figure 8: 99.9%-Reliable Preemption Overheads
(an interrupt service time is included)

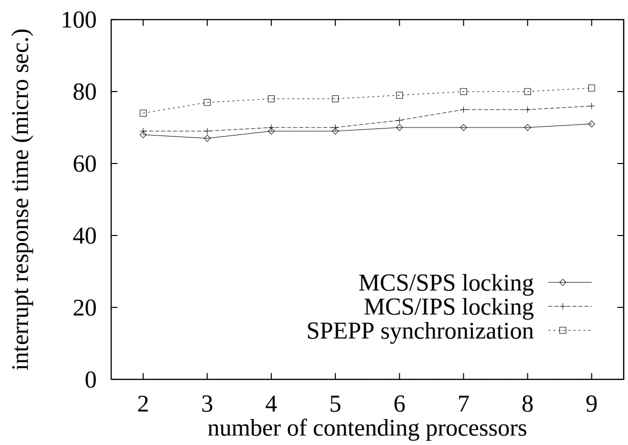

Figure 9: 99.9%-Reliable Interrupt Response Times

it.

Figure 8 presents the differences of these two graphs, i.e. the differences of the 99.9%-reliable execution times of an operation when an interrupt request is serviced and those when no interrupt request is serviced, which represent the measured preemption overheads including an interrupt service time. Considering that the interrupt service time is about 33 μs, the preemption overheads with the SPEPP synchronization are almost zero. This demonstrates that the SPEPP synchronization has the expected property.

Finally, Figure 9 presents the 99.9%-reliable interrupt response times on processor P_1. With either method, the interrupt response times are independent of the number of contending processors. The response times are a little worse with the SPEPP synchronization than the other methods.

4 Conclusion and Future Work

In this paper, we have proposed a novel approach to multiprogrammed multiprocessor synchronization, called the SPEPP synchronization, for use in the implementation of a real-time kernel. The SPEPP synchronization has the merits of both of the two conventional approaches to multiprogrammed multiprocessor synchronization: preemption-safe locking and wait-free synchronization. In the concrete, the preemption overhead can be reduced to zero with the SPEPP synchronization. It is also possible to operate on complex data structures with reasonable overhead. The two versions of the SPEPP synchronization algorithms have been described, and their timing behavior has been discussed. Finally, it is demonstrated through performance measurements that the SPEPP synchronization has the expected advantages, although its overhead is a little larger than preemption-safe lockings.

The most important future work for us is to extend our algorithm for nested locks, in order to implement a scalable full-fledged real-time kernel for function-distributed multiprocessors. The difficulty in its implementation is described in [8].

Another work to be done is to apply our approach to multiprogrammed multiprocessor synchronization of non-real-time systems. When the maximum execution times need not be bounded, various optimizations are possible. For example, in case of cache coherent multiprocessor systems, it is a promising approach that once a processor acquires a lock, it executes the operations of all the tasks that are waiting for the lock. In case of NUMA architecture, on the other hand, if the processor to which the target shared data structure is local is spinning on the lock, the lock should be handed to the processor and the processor executes for all of the waiting tasks.

References

[1] R. W. Wisniewski, L. Kontothanassis, and M. L. Scott, "High performance synchronization algorithms for multiprogrammed multiprocessors," in *Proc. 5th ACM Symposium on Principles and Practice of Parallel Programming*, pp. 199–206, July 1995.

[2] M. M. Michael and M. L. Scott, "Relative performance of preemption-safe locking and non-blocking synchronization on multiprogrammed shared memory multiprocessors," in *Proc. 11th Int'l Parallel Processing Symposium*, Apr. 1997. (to appear).

[3] M. Herlihy, "Wait-free synchronization," *ACM Trans. Programming Languages and Systems*, vol. 13, pp. 124–149, Jan. 1991.

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

[5] 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.

[6] R. W. Wisniewski, L. Kontothanassis, and M. L. Scott, "Scalable spin locks for multiprogrammed systems," in *Proc. 8th Int'l Parallel Processing Symposium*, Apr. 1994.

[7] H. Takada and K. Sakamura, "Predictable spin lock algorithms with preemption," in *Proc. Real-Time Operating Systems and Software*, pp. 2–6, May 1994.

[8] H. Takada, C.-D. Wang, and K. Sakamura, "Issues for realizing a scalable real-time kernel for function-distributed multiprocessors," in *Proc. Work in Progress Session of 17th IEEE Real-Time Systems Symposium*, pp. 23–26, Dec. 1996.

[9] H. Massalin and C. Pu, "A lock-free multiprocessor OS kernel," Tech. Rep. CUCS-005-91, Department of Computer Science, Columbia University, 1991.

[10] M. Greenwald and D. Cheriton, "The synergy between non-blocking synchronization and operating system structure," in *Proc. 2nd Symposium on Operating Systems Design and Implementation*, pp. 126–136, Oct. 1996.

[11] K. Sakamura, ed., *μITRON 3.0 Specification.* Tokyo: TRON Association, 1994. (can be obtained from "ftp://tron.um.u-tokyo.ac.jp/pub/TRON/ITRON/SPEC/mitron3.txt.Z").

```
// data structure for a queue node.
type Node = record
      next: pointer to Node;
      state: pointer to Node;
      op: pointer to OpBlock
end;

// data structure for a lock.
type Lock = record
      last: pointer to Node;
      glock: pointer to Node
end;

// globally shared variable.
shared var L: Lock;      // L.last and L.glock should be initialized to NULL.

// the subroutine to release the lock.
// top is the head node of the queue before the release.
procedure release_lock(lock: pointer to Lock, top: pointer to Node);
      var entry, next: pointer to Node;
begin
      next := top→next;
      if next = NIL then
            // tries to make the waiting queue empty.
            if CAS(&(lock→last), top, 0) then
                  // the waiting queue is empty, now.
                  top→state := EXECUTED;
                  return
            end;
            repeat next := top→next until next ≠ NULL
      end;
      top→state := EXECUTED;

      // waits if lock→glock is not NULL.
      repeat until lock→glock = NULL;

      entry := next;
      lock→glock := UPDATING;
      repeat
            // tries to hand the lock to entry.
            if CAS(&(entry→state), WAITING, next) then
                  lock→glock := NULL;
                  return
            end;
            // proceeds to the next node in the queue.
            entry := entry→next
      until entry = NULL;
      // no task to which to hand the lock.
      lock→glock := top
end;
```

Figure 10: The Basic Algorithm of the SPEPP Synchronization (Part 1)

```
    // shared variables for each task.
    var I: Node;
    var opblock: OpBlock;
    // unshared variables for each task.
    var pred, top, entry, next: pointer to Node;
    var interval, i: integer;
    // opblock is assumed to be prepared beforehand.
    disable_interrupts;
    I.next := NULL;
    // inserts I at the end of the queue.
    pred := SWAP(&(L.last), &I);
    if pred = NULL then
        // when the queue has been empty.
        goto acquired
    end;
    // when the queue has not been empty.
    I.state := WAITING;
    I.op := &opblock;
    pred→next := &I;
    goto spin;
  preemption:
    // branches here after assigning PREEMPTED to I.state
    //          when an interrupt request should be serviced.
    enable_interrupts;
    // interrupt requests are serviced here.
    disable_interrupts;
    // tries to recover I.state from PREEMPTED to WAITING.
    // does not recover I.state if it is already EXECUTED.
    CAS(&(I.state), PREEMPTED, WAITING);
  spin:
    i := 1; interval := α;
    entry := I.state;
    // spins while I.state is WAITING.
    while entry = WAITING do
        if interrupt_requested then
            // tries to change I.state from WAITING to PREEMPTED.
            if CAS(I.state, WAITING, PREEMPTED) then
                goto preemption
            end
        end;
```

Figure 11: The Basic Algorithm of the SPEPP Synchronization (Part 2)

```
            i := i − 1;
            if i = 0 then
                // checks the global lock state.
                top := L.glock;
                if top = NULL then
                    i := ∞
                else if top = UPDATING then
                    // must check the global lock state again
                    //              with exponential backoff.
                    i := interval; interval := interval × β
                else begin
                    entry := I.state;
                    if entry ≠ WAITING then
                        break
                    else if CAS(&(L.glock), top, NULL) then
                        // succeeds in acquiring the global lock.
                        entry := top;
                        goto execute
                    else
                        i := ∞
                    end
                end
            end;
            entry := I.state
        end;

        // I.state (= entry) is not WAITING, now.
        if entry = EXECUTED then
            // my operation has been executed by another processor.
            goto finish
        end;
    execute:
        // succeeds in acquiring the lock.
        while entry ≠ me do
            // executes the operation posted to entry.
            execute(entry→op);
            if interrupt_requested then
                I.state := PREEMPTED;
                release_lock(lock, entry);
                goto preemption
            end;
            // proceeds to the next node in the queue.
            repeat next := entry→next until next ≠ NULL;
            entry→state := EXECUTED;
            entry := next
        end;
    acquired:
        // executes my operation, now.
        execute(&opblock);
        release_lock(lock, me)
    finish:
        enable_interrupts
```

Figure 12: The Basic Algorithm of the SPEPP Synchronization (Part 3)

```
    // shared variables for each lock for each processor.
    var I: Node;
    // shared variables for each task.
    var opblock: OpBlock;


    // opblock is assumed to be prepared beforehand.
    disable_interrupts;
retry:
    I.next := NULL;
    // inserts I at the end of the queue.
    pred := SWAP(&(L.last), &I);


preemption:
    // branches here after assigning PREEMPTED to I.state
    //          when an interrupt request should be serviced.
    enable_interrupts;
    // interrupt requests are serviced here.
    disable_interrupts;
    entry := I.state;
    if finished(&opblock) then
        // opblock has already been executed.
        goto finish
    end;
    if entry = EXECUTED then
        // the queue node has been stolen.
        goto retry
    end;
    // steals the queue node (if another task is using it).
    I.op := &opblock;
    // tries to recover I.state from PREEMPTED to WAITING.
    // does not recover I.state if it is already EXECUTED.
    CAS(&(I.state), PREEMPTED, WAITING);
spin:


    // I.state (= entry) is not WAITING, now.
    if entry = EXECUTED then
        // the operation has been executed by another processor.
        if ¬finished(&opblock) then
            // executed operation has been the request of another task.
            goto retry
        end;
        // executed operation has been my request.
        goto finish
    end;
execute:
```

Figure 13: The Extended Algorithm of the SPEPP Synchronization