Issues for Realizing a Scalable Real-Time Kernel for Function-Distributed Multiprocessors

Hiroaki Takada[†], Cai-Dong Wang[†], and Ken Sakamura[‡]

[†] Department of Information Science,
School of Science, University of Tokyo
7–3–1 Hongo, Bunkyo-ku, Tokyo 113, Japan
Email: hiro@is.s.u-tokyo.ac.jp

[‡] The University Museum,
University of Tokyo
7–3–1 Hongo, Bunkyo-ku, Tokyo 113, Japan

Abstract

In multiprocessor systems, the worst-case execution time of a task that exclusively accesses a shared resource is unavoidably prolonged as the number of contending processors is increased. In case of function-distributed multiprocessors, because many of the tasks can be processed without synchronizing with other processors, it is advantageous that their worst-case behavior are independent of the number of processors in the system. This paper summarizes the required properties on scalable real-time kernels and discusses their realization techniques. What we have solved so far are described, and the remaining problem to be solved is presented.

1 Introduction

In many applications of high-performance real-time systems, a system is required to handle a large number of external devices and to respond to the events from the devices within predefined and usually short time-bounds. Adopting a *function-distributed multiprocessor*, in which each external device is interfaced to the local bus of a processor and is handled only by the processor, is a promising approach to satisfying this requirement (Figure 1).

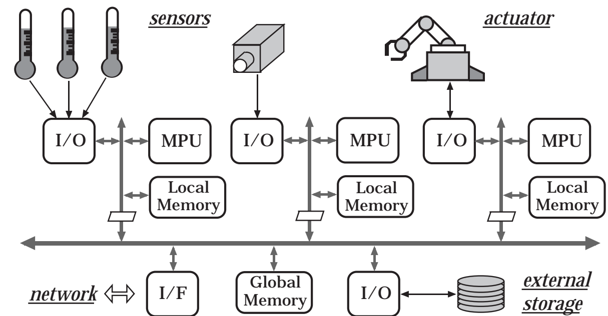

Figure 1: Function-Distributed Multiprocessor

In order to realize a real-time system on a multiprocessor environment, predictable and scalable inter-processor synchronization is among the significant issues. A real-time synchronization protocol for multiprocessors such as those in [1, 2, 3] is necessary to be adopted in the application level. In addition, the underlying synchronization mechanism used in the run-time system must also be predictable and scalable. In implementing the priority ceiling protocol for shared-memory multiprocessors in [1], for example, a spin lock is used as an underlying synchronization primitive to guard the queue of tasks that are waiting to enter an application-level critical section. Another queue (called a ready queue) of the tasks that are ready to execute on a processor should also be guarded with a spin lock. Those queues of tasks are shared among processors and thus must be accessed exclusively by a processor within the run-time system.

It is ideal from the scalability point of view that the worst-case execution time of each run-time system service is determined independently of the number of processors in the system and of the activities of other processors. Here exists an obvious limitation, however, that the worst-case execution time of a routine that exclusively accesses a shared resource is prolonged, as the number of contending processors is increased, with its linear order at least. Though multiprocessor real-time systems have been actively studied, little studies have focused on the scalability of worst-case behavior.

Our study is to ease this difficulty by taking the general design rule of function-distributed multiprocessors into account that external devices and tasks handling them are allocated to processors so that as many tasks as possible can be processed within a processor (or without synchronizing with other processors). The maximum execution times of those tasks should be independent of the number of contending processors in the system. In addition, the maximum execution times of other tasks should be bounded with its linear order.

2 Kernel Model and Lock Units

In this study, the basic model of real-time kernels[1] for function-distributed multiprocessors is specified as follows. Each task has its host processor on which it is executed,

[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.

and is called a *local task* of the processor. A task can synchronize and communicate with any task in the system with the same set of operations. A ready queue is prepared for each processor in which all the local tasks that are ready to execute on the processor are included in the descending order of priorities. Each task-independent synchronization and communication object (simply called as a synchronization object below), such as a semaphore (or a mutex) and an eventflag (or a condition variable), also has its host processor and can be accessed from any task in the system.

In designing a software system on shared-memory multiprocessors, the granularity of lock units guarding shared data structures is always a difficult issue. In implementing a real-time kernel, making lock units so small that many locks are necessary to be acquired in nested structure in some operations is not an appropriate approach, because the execution time of each operation is very short and the lock acquisition overhead is relatively large.

The simplest method to avoid nested locks is to enter all kernel data structures in one lock unit. With this method, however, because only one kernel service can be executed at the same time, the total throughput to execute kernel services cannot scale well. Thus we have premised that kernel data structures on different processors, at least, should be placed in different lock units.

Under this premise, at least two locks are necessary to be acquired in nested structure. In concrete, when a task begins waiting on a synchronization object, it first accesses the control block of the object and then accesses the task control block (TCB) of itself. When the task and the synchronization object are located on different processors, their control blocks are placed in different lock units. The same situation also occurs when a task operates on a synchronization object and wakes up another task that has been waiting on the object.

In order to determine an appropriate granularity of lock units, we have examined a real-time kernel implementation for single processors based on the μITRON 3.0 Specification [4]. As the result, two lock units are prepared for each processor; one of the locks (called the task lock) guards the TCBs and the ready queue on a processor and the other lock (called the object lock) guards the control blocks of the synchronization objects on the processor. With this granularity of lock units, at most two lock units, an object lock and a task lock in this order, are necessary to be acquired in nested structure except for some rarely used operations.

3 Required Properties

As described in Section 1, the worst-case execution times of the operations that access shared data structures are unavoidably prolonged, as the number of contending processors is increased. In case of function-distributed multiprocessors, this difficulty is greatly relieved if the worst-case execution times of the operations that can be done within a processor are independent of the number of contending processors. Processings that can be done within a processor include synchronizations and communications with another task on the same processor and interrupt services requested by the external devices attached to the local bus of the processor.

In addition, the worst-case execution times of the operations within a processor should not depend on the other processors' activities. Conversely, it is also desired that activities within a processor do not affect the timing behavior of the processings on the other processors.

These discussions can be summarized in the following four properties that scalable real-time kernels for function-distributed multiprocessors are required to satisfy [5].

(A) The maximum execution time of an operation that is to synchronize or communicate with tasks on the same processor (called a local operation) can be determined independently of the other processors' activities and the number of contending processors.

(B) The maximum execution time of an operation that is to synchronize or communicate with tasks on other processors can be determined independently of the other processors' activities and be bounded with a linear order of the number of contending processors.

(C) The maximum interrupt response time on each processor can be determined independently of the other processors' activities and the number of contending processors.

(D) The interrupt service overhead can be determined independently of the other processors' activities and the number of contending processors.

In the property (D), the interrupt service overhead is defined to be the wasted computation time caused by an interrupt service, which should be added to the interrupt service time when the schedulability of the system is analyzed. Because interrupt services are usually requested more frequently than tasks, a small increase in the interrupt service overhead can severely degrade the schedulability of the system. Therefore, this property is necessary for the scalability, though not so essential as the former three properties.

4 What has been Achieved

In this section, we focus on the case when task-independent synchronization objects are *not* supported. Without task-independent synchronization objects, only one lock unit is necessary for each processor, which guards all shared data structures on its local memory. A processor needs to acquire at most one lock unit at the same time.

Even with this limitation, the above four properties cannot be obtained with a straightforward implementation. Specifically, there are two problems for satisfying these properties; incompatibility of constant interrupt response and predictable inter-processor synchronization, and lack of scalability in local operations. Our proposed solutions to these problems are described below.

The first problem is that constant interrupt response (the property (C)) is not compatible with predictable inter-

processor synchronization (the property (B)) for the following reason. In order to bound the time until a processor acquires an inter-processor lock, it is necessary to bound the duration that each processor holds the lock as well as the number of contending processors that the processor must wait for. The latter condition can be met with a FIFO-ordered spin lock, with which the turn that a processor acquires a lock is reserved when it begins waiting for the lock. In order to satisfy the former condition, interrupt services should be inhibited while the processor holds a lock.[2] On the other hand, in order to make the maximum interrupt response independent of the number of processors, the processor must service interrupt requests while waiting for a lock. These conditions, however, cannot be satisfied with conventional spin lock algorithms.

To solve this problem, we have proposed two FIFO-ordered queueing spin lock algorithms supporting preemption [6]. With the simpler one of the algorithms, when a processor begins interrupt services while waiting for a lock, it informs others that it is servicing interrupts. The processor releasing the lock checks the state of the succeeding processor. When the succeeding one is servicing interrupts, its reservation to acquire the lock is canceled and the lock is passed to the next one in line. When a processor find that its reservation has been canceled while servicing interrupts, it re-executes the lock acquisition routine from the beginning. Clearly, this re-execution overhead, which should be included in the interrupt service overhead, depends on the number of contending processors, and the property (D) is not satisfied. With our improved algorithm in [6], when the turn to acquire a lock comes during interrupt services, the reservation is not canceled but is postponed. When the processor returning from the interrupt services, it resumes waiting for the lock in its original position. With this improved scheme, the interrupt service overhead can be reduced to a constant time length, and thus the property (D) is satisfied.

The second problem is that the worst-case execution times of local operations depend on the number of contending processors. This is because a task must acquire an inter-processor lock even when it accesses a task on the same processor. We have solved this problem by giving the precedence in acquiring a lock to its host processor [7]. In other words, a processor can acquire the lock unit that guards its local data structures with precedence over the other processors. With this scheme, the maximum execution time of a local operation becomes independent of the number of contending processors. More precisely, a task must wait for at most one critical section executed by other processors until it acquires its local lock. On the other hand, the maximum number of critical sections that a processor must wait for until it acquires a non-local lock is increased. In more precise, when a task tries to acquire a non-local lock, it must wait for $n - 1$ critical sections executed by its host processor in addition to $n - 2$ critical sections executed by the other processors (where n is the number of contending processors in the system).

Effectiveness of our proposed solutions has been demonstrated through performance measurements using an existing multiprocessor system [5]. It is confirmed through the measurements that the four properties are practically satisfied with our proposals,[3] while they cannot be met at the same time with other methods.

We have also proposed an approach to classify tasks according as their characteristics [8]. Especially, we classify the tasks that do not synchronize or communication with tasks on other processors as *private tasks*, which are managed differently from the local tasks. Task-independent synchronization objects can also be classified accordingly. Another useful class of tasks are *global tasks*, which can be executed on any processor in the system and can migrate dynamically.

5 What is Remaining to be Solved

In this section, realization techniques when task-independent synchronization objects are supported are discussed. In this case, two lock units, an object lock and a task lock, are necessary to be acquired in some operations.

The property (A) can be satisfied by classifying synchronization objects into private ones and shared ones, and by managing them differently.[4] On the other hand, the property (B) is not satisfied if each lock is realized with a simple FIFO-ordered spin lock algorithm. With this straightforward method, the maximum execution time to acquire nested spin locks becomes a square order of the number of contending processors and cannot be bounded with its linear order. We have also proposed a method with which this problem can be solved [9].

Below, two methods are presented to satisfy the properties (C) and (D), but in vain.

The First Method

In order to satisfy (C), when an interrupt is requested to a processor while it is waiting for a lock, the processor must suspend the spin-waiting and start servicing the interrupt request. When an interrupt request occurs while a processor is waiting for the outer lock, the improved preemption scheme described in Section 4 can be applied.

When an interrupt request occurs while a processor is waiting for the inner lock, it must release the outer lock

[2] There is another solution to this problem that the number of interrupt requests that are serviced while a processor holds a lock is limited to a constant number. We have not adopted this solution because the maximum execution time of some interrupt service is much longer than that of a critical section. As the result, the maximum duration that a processor holds a lock becomes very long and thus the schedulability of the system is severely degraded.

[3] Strictly speaking, some assumptions on underlying spin lock algorithm and hardware architecture are necessary to satisfy the four properties [5]. Although our software-implemented spin lock algorithm in [6] does not satisfy one of the properties in strict, the effect is so small that it can be ignored in usual applications. For very hard real-time applications, the spin lock should be implemented with hardware.

[4] This real-time kernel model is best suited to the run-time system for the priority ceiling protocol for shared-memory multiprocessors in [1].

```
retry:
    disable interrupts;
    if (¬acquire_lock(Object_Lock)) then
        enable interrupts;
        interrupt requests are serviced here;
        goto retry
    end;
    deterimine which lock to acquire next;
    if (¬acquire_lock(Task_Lock)) then
        release_lock(Object_Lock);
        enable interrupts;
        interrupt requests are serviced here;
        goto retry
    end;
    execute the operation;
    release_lock(Task_Lock)
    release_lock(Object_Lock);
    enable interrupts;
```

Figure 2: Nested Spin Locks with Preemption

before servicing the interrupt request, in addition to suspending the spin-waiting for the inner lock. Otherwise, the maximum duration that the processor holds the outer lock includes interrupt service times. The skeleton of the routine acquiring nested locks is presented in Figure 2. In this figure, the *acquire_lock* function is assumed to return false, when an interrupt is requested during the spin-waiting. With this method, the processor must re-acquire the outer lock after interrupt services. This re-acquisition overhead, which depends on the number of contending processors, should be included in the interrupt service overhead, and thus the property (D) cannot be satisfied.

The Second Method

In order to satisfy the property (D), a processor should be able to come back to the original state with a constant time length after interrupt services. In concrete, after returning from an interrupt service which is requested while waiting for the inner lock, the processor should wait for the outer lock at the top of its waiting queue instead of its end. This scheme can make the interrupt service overhead independent of the number of contending processors and satisfy (D).

With this scheme, however, the property (B) cannot be met with the following reason. Suppose the case that a processor P_1 is holding the outer lock L on which two other processors P_2 and P_3 are waiting. If an interrupt is requested on P_1 while it is waiting for the inner lock, P_1 suspends waiting for the inner lock, passes the outer lock L to P_2, and starts the interrupt service. P_2 acquires L and begins executing the critical section. Assume that P_2 is waiting for the inner lock and is still holding L when P_1 returns from the interrupt service. In this case, P_1 returns to the top of the waiting queue, i.e. in front of P_3. If an interrupt request is raised on P_2 at this moment, it passes the lock L to P_1. Again, P_2 can return to the top of the waiting queue, i.e. in front of P_3. This process can continue permanently and violates the property (B).

6 Concluding Remarks

In this paper, requirements on scalable real-time kernels for function-distributed multiprocessors are summarized in four properties, and its realization techniques are discussed. So far, we have proposed a solution in the case that only one lock unit is necessary to be acquired at the same time, while the realization method when more than one locks are acquired in nested structure is an open issue.

A possible approach to its realization is to incorporate the notion of block-free or wait-free synchronizations and to avoid nested spin locks. More precisely, the processings which needs the outer lock should be realized in a block-free or wait-free fashion [10]. Because the manipulations of a TCB and a ready queue are too complicated to realize in block-free or wait-free with reasonable performance, the inner lock should be used even with this approach. Another promising approach is to incorporate the concept in realizing wait-free synchronization to the inner lock. More precisely, the operation within the inner lock is posted to the waiting queue for the lock and is executed by another processor during an interrupt service.

We would like to report the results of these investigations in the future.

References

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

[2] V. B. Lortz and K. G. Shin, "Semaphore queue priority assignment for real-time multiprocessor synchronization," *IEEE Trans. Software Engineering*, vol. 21, pp. 834–844, Oct. 1995.

[3] I. Rhee and G. R. Martin, "A scalable real-time synchronization protocol for distributed systems," in *Proc. Real-Time Systems Symposium*, pp. 18–27, Dec. 1995.

[4] 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").

[5] H. Takada, *Studies on Scalable Real-Time Kernels for Function-Distributed Multiprocessors*. PhD thesis, School of Science, University of Tokyo, Sept. 1996.

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

[7] H. Takada and K. Sakamura, "Inter- and intra-processor synchronizations in multiprocessor real-time kernel," in *Proc. 4th Int'l Workshop on Parallel and Distributed Real-Time Systems*, pp. 69–74, Apr. 1996.

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

[9] H. Takada and K. Sakamura, "Real-time scalability of nested spin locks," in *Proc. 2nd Real-Time Computing Systems and Applications*, pp. 160–167, Oct. 1995.

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