Experimental Implementations of Priority Inheritance Semaphore on ITRON-specification Kernel

Hiroaki Takada   Ken Sakamura

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

Abstract

Using priority inheritance protocols is an effective approach to solve the problem of uncontrolled priority inversion, which is among the major sources of deadline violations in hard real-time systems. In this paper, some approaches to incorporate priority inheritance to the ITRON specification are discussed. As the result, we propose two specifications of priority inheritance semaphore functions with which the basic priority inheritance protocol can be realized, and implement both of them for evaluation. The run-time performance and memory requirements of the functions are evaluated with the implementations.

1 Introduction

Priority inversion, the state in which a higher priority task is blocked by lower priority tasks due to resource sharing, is one of the major sources of deadline violations in hard real-time systems. Priority inheritance protocols are among the promising approaches to bound the duration of priority inversion [1].

Some recent multimedia applications, especially those handling continuous media including voice and video, require the techniques devised for hard real-time systems, and then the ITRON specification is requested to support the techniques. As the first step, we discuss how to adopt priority inheritance protocols on ITRON-based systems in this paper. The necessity of priority inheritance protocols in the ITRON specification has been also pointed out by other researchers [2].

The main purpose of this research is to present a basic material for investigating future ITRON specifications. There are some approaches to incorporate priority inheritance to the ITRON specifications. We investigate on the approaches and propose two specifications of priority inheritance semaphore functions as the result. Then, we implement them in an existing ITRON-specification kernel and evaluate them from the viewpoint of run-time performance.

In Section 2, some approaches to incorporate priority inheritance to the ITRON specification are discussed after a short introduction of priority inheritance protocols. How to introduce the priority inheritance semaphore function to the ITRON specification is investigated in Section 3, and two versions of specifications are proposed. One of the specifications is considered to be ideal for application programmers, but its implementation overhead is large. Though the implementation of the other specification is compact and efficient, it has some restrictions in using the function. Both of the specifications are implemented in an existing ITRON-specification kernel and their execution times and memory requirements are evaluated in Section 4.

2 Incorporating Priority Inheritance to the ITRON Specification

2.1 Priority Inheritance Protocols

Adopting priority inheritance protocols is one of the approaches to solve the problem of uncontrolled priority inversion. A typical example of the problem is illustrated below.

Example (uncontrolled priority inversion)

Suppose that there are three tasks τ_1, τ_2, and τ_3, with τ_1 having the highest priority and τ_3 having the lowest. τ_1 and τ_3 access a shared data within mutually exclusive regions guarded by a semaphore. Suppose the case that τ_3 locks the semaphore and enters the critical section when τ_1 and τ_2 are in wait state. While τ_3 is executing in the critical section, τ_1 becomes ready state with some external event, begins to execute preempting τ_3, and then tries to lock the semaphore for accessing the shared data. Because the semaphore has already been locked by τ_3, τ_1 is blocked and τ_3 resumes to execute. At this moment, priority inversion occurs. Assume that τ_2 becomes ready state during this priority inversion. Since τ_2 has higher priority than τ_3, τ_2 preempts τ_3. As the result, the priority inversion duration is prolonged and can not be controlled.

To analyze the schedulability of the system, the maximum duration of the priority inversion must be known. It is permissible that τ_1 is blocked for the duration that τ_3 executes the critical section, because the maximum execution time of a critical section is usually bounded and known. However, if τ_1 must wait for the execution of τ_2 and other intermediate priority tasks, the duration of the priority inversion becomes very long or even unbounded.

There are some solutions to this problem. One of the simplest solutions is to disable task dispatches while τ_3 executes the critical section. `dis_dsp` system call[1] of the ITRON specification can be used to implement this method.

Another simple solution is to make the priority of τ_3 higher than τ_2 while it executes the critical section. More exactly, the priority of τ_3 is raised to the level of the highest priority task which may lock the semaphore (or access the shared data). This method is called stack resource policy [3] or highest locker protocol [4], and can be implemented using only `chg_pri` system call of the ITRON specification[2].

Another solution is to use priority inheritance protocols. The fundamental concept of priority inheritance protocols is that when a task blocks some higher priority tasks, its priority should be raised to the level of the highest priority task among the blocked ones. In other words, the task inherits the priority of the highest priority task blocked by it. The priority is put back to the original level when the task releases the blocked tasks. We call the original priority of a task as its base priority.

With the basic priority inheritance protocol [1], which is the naive realization of the concept, the priority inversion problem illustrated in the previous example is solved as follows. When τ_1 tries to lock the semaphore and is blocked, τ_3 inherits the priority of τ_1 because τ_1 is blocked by τ_3. Because the inherited priority is higher than that of τ_2, τ_3 is not preempted by τ_2 when τ_2 becomes ready to run. As the result, the duration of priority inversion is bounded by the maximum time that τ_3 executes the critical section.

Even when multiple resources are shared and some tasks lock multiple semaphores, the basic priority inheritance protocol bounds the duration of priority inversion. In this case, priority inheritance must be transitive. For example, when τ_4 inherits the priority of τ_3 and τ_3 inherits that of τ_1, the priority of τ_4 is raised to that of τ_1. Refer to [1] for further discussions.

In this paper, we discuss the ways to incorporate the concept of priority inheritance protocols to the ITRON specification, especially the basic priority inheritance protocol. We remain it as future work how to incorporate other variants of priority inheritance protocols, including the priority ceiling protocol, to the ITRON specification.

2.2 Some Approaches

From the fundamental concept of priority inheritance protocols, all synchronization and communication objects of the ITRON specification, which cause blockings, should have priority inheritance operation mode. Adding priority inheritance option to the attributes of all synchronization and communication objects and making them support priority inheritance mode is one of the approaches to incorporate priority inheritance to the ITRON specification. However, with some synchronization or communication objects, the relation between a blocked task and the task that causes the blocking cannot be determined beforehand in general. In such cases, it cannot be decided which task should inherit the priority of a blocked task. A typical example is the synchronization using an eventflag.

Another approach is that the kernel supports the mechanisms with which application programmers can implement priority inheritance protocols. Actually, priority inheritance message passing can be realized using only existing primitives of the ITRON specification, `chg_pri` system call, two priority-ordered mailboxes, and an additional task that controls priorities [4]. However, the priority inheritance mutual exclusion mechanism presented in the previous section cannot be implemented without any restrictions. In addition to some restrictions, its run-time performance will be very bad.

From these considerations, we investigate on the method to support priority inheritance binary semaphore with ITRON-specification kernels in the following sections. With priority inheritance binary semaphore, the basic priority inheritance protocol can be realized straightforwardly. We do not investigate

[1] `dis_dsp` is a newly introduced system call in the μITRON3.0 specification.

[2] The highest locker protocol has the restriction that a task which executes a critical section must not enter wait state nor be suspended.

cre_pis	Create Priority Inheritance Semaphore
del_pis	Delete Priority Inheritance Semaphore
sig_pis	Signal Priority Inheritance Semaphore
wai_pis	Wait on Priority Inheritance Semaphore
preq_pis	Poll and Request Priority Inheritance Semaphore
twai_pis	Wait on Priority Inheritance Semaphore with Timeout
ref_pis	Refer Priority Inheritance Semaphore Status

Table 1: System Calls Supporting Priority Inheritance Semaphore

on priority inheritance counting semaphore, because it is less useful than binary semaphore and requires larger memory space.

3 Specifications of Priority Inheritance Semaphore

In this section, we propose two specifications to incorporate the priority inheritance semaphore function to the μITRON3.0 specification [5].

3.1 Priority Inheritance Semaphore

Priority inheritance semaphore is the mechanism to implement the basic priority inheritance protocol.

One method to incorporate the priority inheritance semaphore function to the μITRON3.0 specification is to add a priority inheritance option to the semaphore attribute, which is passed to the kernel when a semaphore is created with `cre_sem` system call. Though no new system call is necessary with this method, it has a drawback that the overhead to check the attribute becomes large because internal processing of normal semaphore and that of priority inheritance semaphore are quite different.

Therefore, we adopt the other method that a priority inheritance semaphore is dealt as a new synchronization object. The seven new system calls presented in Table 1 are introduced to support the priority inheritance semaphore function. "`pis`" is the abbreviation of priority inheritance semaphore.

Functions of these system calls are same with those of corresponding system calls for normal semaphores, except for the following differences in addition to supporting priority inheritance (of course).

- The initial semaphore count and the maximum semaphore count, which are passed to `cre_sem` system call, are not passed to `cre_pis` system call, because only binary semaphore is supported with the priority inheritance semaphore function.
- The current semaphore count is not included in the return parameter of `ref_pis` system call with the same reason. The ID of the task that obtains the priority inheritance semaphore can be acquired with the system call, instead.
- Only priority-ordered queueing of waiting tasks is supported, because FIFO queueing is not suitable for hard real-time systems.
- A priority inheritance semaphore must be released by the task that has locked it. Otherwise, `sig_pis` reports `E_OBJ` error.

In addition to them, some other differences exist depending on the specification of the priority inheritance semaphore function, which will be mentioned in the following sections.

The priority of a task that accesses priority inheritance semaphores is changed by the kernel according as the priority inheritance rules. We do not stipulate the exact behavior when `chg_pri` system call is issued on such a task. On the other hand, the base priority of a task is not changed by the priority inheritance semaphore function. In the specifications presented below, the initial priority of a task, which is passed to the kernel when the task is created, is considered to be its base priority for simplicity. The alternative specifications in which this restriction is removed are possible. In this case, another new system call that changes the base priority of a task should be introduced. Though the specifications require additional area in the task control block (TCB), their impact on the run-time performance is very small.

3.2 Specification A

The first specification (called Specification A below) is the naive realization of the fundamental concept of the priority inheritance protocols. In the concrete, the priority of a task is preserved to be the highest priority level among its base (i.e. initial) priority and the priorities of the tasks that are waiting for one of the priority inheritance semaphores locked by the former task.

In order to implement the specification, the list of priority inheritance semaphores locked by a task must be maintained. This list is used when the new priority of a task is decided from the priority rule described above. It is actually used in following four cases. Other cases can be handled without referring to the list.

1. When a task obtains multiple priority inheritance semaphores and releases one of them (using `sig_pis` system call), the new priority of the task is calculated using the list.

2. When a priority inheritance semaphore is deleted (using `del_pis` system call), the new priority of the task that has obtained the deleted semaphore is calculated using the list.

3. When a task that has been waiting for a priority inheritance semaphore is released its wait state by timeout or `rel_wai` system call[3], the new priority of the task that obtains the semaphore is calculated using the list.

4. When the priority of a task that is waiting for a priority inheritance semaphore is changed, the new priority of the task that obtains the semaphore is calculated using the list.

Actually, some optimization is possible. In the first case, for example, the task's new priority is necessary to be calculated only when its priority before releasing the priority inheritance semaphore is equal to the priority of the task that is at the head of its waiting queue.

Because the list of locked priority inheritance semaphores is maintained, it can be implemented easily that all priority inheritance semaphores locked by a task are automatically released when the task exits or is terminated. We implement and evaluate this alternative specification in Section 4.

3.3 Specification B

Maintaining the list of the priority inheritance semaphores locked by a task, which is necessary in Specification A, is expected to degrade the run-time performance and to require more memory space. If the maintenance of the list can be omitted with some restrictions, it may be profitable for some applications. The resulting specification is called Specification B and the restrictions with the specification are discussed below.

[3]In strict, `ter_tsk` system call may also release a task's wait state.

The behavior of the priority inheritance semaphore function in Specification B is different from that in Specification A in the following points, since the list of the locked priority inheritance semaphores is not maintained.

1. When a task releases a priority inheritance semaphore, the priority of the task is put back to its base (i.e. initial) priority, even if it obtains some other priority inheritance semaphores.

2. When a priority inheritance semaphore is deleted, the priority of the task that has obtained the deleted semaphore is not changed.

3. When a task that has been waiting for a priority inheritance semaphore is released its wait state by timeout or `rel_wai` system call, the priority of the task that obtains the semaphore is not changed.

4. When the priority of a task that is waiting for a priority inheritance semaphore is changed, the priority of the task that obtains the semaphore is handled as follows. If the new priority of the former task is higher than that of the latter task, the priority of the latter task is raised to that of the former task. Otherwise, it remains unchanged.

From these differences, application programmers must be aware of the following restrictions in using the priority inheritance semaphore function.

- When a task obtains multiple priority inheritance semaphores, it must release all of them at once in a dispatch disabled region. Otherwise, the priority of the task is recovered to its base priority, even if the task blocks some other tasks with higher priority levels through the priority inheritance semaphores that are not released. As the result, unexpected priority inversion can occur.

- In the cases 2, 3, and 4 described in the above differece list, the priority of a task may not be changed to its proper priority level and may remain too high. This also causes unexpected priority inversion.

Because the latter cases do not occur with the basic priority inheritance protocol presented in [1], the protocol can be realized with the first restriction.

When the above restrictions are permissible in designing application programs, Specification B is preferable because it has smaller run-time overhead than Specification A.

At first, we have investigated on another specification in which a task's priority before it locks a priority inheritance semaphore is stored in the control block of the semaphore. When the task releases the priority inheritance semaphore, the task's priority is recovered to the level before it locks the semaphore. We have expected that the first restriction can be removed or relaxed with this specification. However, since the problem illustrated in the following example arises without the restriction, this alternative specification has little meaning.

Example

Suppose that there are three tasks τ_1, τ_2, and τ_3 and two priority inheritance semaphores S_1 and S_2. τ_1 has the highest base priority and τ_3 has the lowest. Suppose the case that τ_3 first locks S_1 and then locks S_2. While τ_3 obtains both priority inheritance semaphores, τ_1 becomes ready state with some external event, begins to execute preempting τ_3, and then tries to lock S_1. Because S_1 is locked by τ_3, τ_3 inherits the priority of τ_1 and resumes to execute. When τ_3 releases S_2, the priority of τ_3 is recovered to the level before it locks S_2, i.e. the base priority of τ_3. If τ_2 becomes ready state at this moment, τ_2 preempts τ_3 and uncontrolled priority inversion results.

4 Implementation and Evaluation

We have implemented the two specifications of the priority inheritance semaphore functions presented in the previous section in a real-time kernel based on the μITRON3.0 specification called ItIs (ITRON Implementation by Sakamura Lab), which has been developed as a research testbed. The target processor of the kernel is the TRON-specification microprocessor. Using the implementations, we compare the execution time and memory requirements of the priority inheritance semaphore functions with those of the normal semaphore function.

First, we compare the structure of the control blocks of two versions of priority inheritance semaphores and that of a normal semaphore. Figure 1 to 3 show the structure of each control block. The size of the priority inheritance semaphore control block in Specification B (24 bytes) is 4 bytes smaller than others (28 bytes).

With Specification A, a pointer for the list of locked priority inheritance semaphores is added to the task control block (TCB), and its size becomes 4 bytes larger than the original kernel (Figure 4).

The source and object code size of each module is compared in Table 2. The source codes are written in C language and the sizes presented in the table exclude comment lines and empty lines. The priority inheritance semaphore module in Specification A is a little larger than the other modules as we have expected. When the priority inheritance semaphore function is used, some routines are also added to other modules. Their object code sizes are 64 bytes in Specification A and 4 bytes in Specification B. Compared to the object code size of the normal semaphore module (1300 bytes) or that of rendezvous module (1992 bytes), those of the priority inheritance semaphore modules are considered to be reasonable.

Finally, we present the execution times of seven operations using the priority inheritance semaphore functions compared with the normal semaphore function in Table 3 and Figure 5. We use a system with GMICRO/300 microprocessor running at 20MHz clock and no-wait memories for the evaluation. In order to exclude the effect of cache, both the instruction cache and the operand cache are purged just before an execution time is measured.

The seven operations the execution times of which are measured are as follows (we call priority inheritance semaphore simply as semaphore below).

(a) The execution time of `wai_sem/pis` is measured when the semaphore can be locked immediately. No task dispatch occurs.

(b) Suppose that task τ_1 and task τ_2 have the same base priority and that τ_2 obtains semaphore S_1. Suppose also that τ_1 is running, and that τ_2 is in ready state. If τ_1 invokes `wai_sem/pis(`S_1`)`, τ_1 is blocked and τ_2 begins to execute. The execution time since τ_1 is about to invoke `wai_sem/pis` until τ_2 starts to execute is measured. A task dispatch time is included.

(c) Suppose that task τ_1 has higher base priority than task τ_2 and that τ_2 obtains semaphore S_1. If τ_1 invokes `wai_pis(`S_1`)`, τ_1 is blocked and τ_2 begins to execute with the priority of τ_1. The execution time since τ_1 is about to invoke `wai_sem/pis` until τ_2 starts to execute is measured. A task dispatch time is included.

(d) Suppose that there are three tasks τ_1, τ_2, and τ_3, with τ_1 having the highest base priority and τ_3 having the lowest. Also suppose that τ_2 obtains semaphore S_1 and is waiting for semaphore S_2

```
struct semaphore_control_block {
        QUEUE      wait_queue;  /* waiting queue for the semaphore */
        ID         semid;       /* semaphore ID */
        VP         exinf;       /* extended information */
        ATR        sematr;      /* semaphore attribute */
        INT        semcnt;      /* current semaphore count */
        INT        maxsem;      /* maximum semaphore count */
};
```

Figure 1: Semaphore Control Block

```
struct pis_A_control_block {
        QUEUE      wait_queue;  /* waiting queue for the semaphore */
        ID         pisid;       /* semaphore ID */
        VP         exinf;       /* extended information */
        ATR        pisatr;      /* semaphore attribute */
        TCB        *pistsk;     /* TCB of the task that locks the semaphore */
        struct pis_A_control_block *pislist;
                   /* pointer for the list of semaphores locked by a task */
};
```

Figure 2: Priority Inheritance Semaphore Control Block in Specification A

```
struct pis_B_control_block {
        QUEUE      wait_queue;  /* waiting queue for the semaphore */
        ID         pisid;       /* semaphore ID */
        VP         exinf;       /* extended information */
        ATR        pisatr;      /* semaphore attribute */
        TCB        *pistsk;     /* TCB of the task that locks the semaphore */
};
```

Figure 3: Priority Inheritance Semaphore Control Block in Specification B

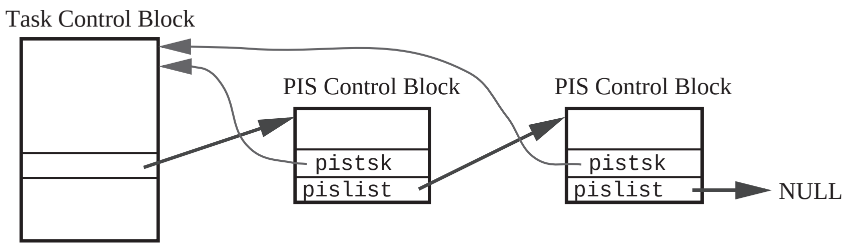

Figure 4: The List of Priority Inheritance Semaphores (in Specification A)

	Semaphore Module	PIS Module in Spec A	PIS Module in Spec B
Source Code	230 lines	341 lines	244 lines
Object Code	1300 bytes	1748 bytes	1352 bytes

Table 2: Source and Object Code Sizes

Operations		Semaphore	PIS in Spec A	PIS in Spec B
(a)	`wai_sem/pis` without switching tasks	11 μsec	12 μsec	11 μsec
(b)	`wai_sem/pis` with a task dispatch	39 μsec	39 μsec	38 μsec
(c)	`wai_pis` with a task dispatch and a priority inheritance	—	48 μsec	47 μsec
(d)	`wai_pis` with a task dispatch and two priority inheritances	—	59 μsec	58 μsec
(e)	`sig_sem/pis` without waking up tasks	9 μsec	14 μsec	11 μsec
(f)	`sig_sem/pis` with waking up a task but no task dispatch	18 μsec	27 μsec	22 μsec
(g)	`sig_sem/pis` with waking up a task, a task dispatch, and recovering the priority of invoking task	30 μsec	42 μsec	37 μsec

Table 3: Execution Times

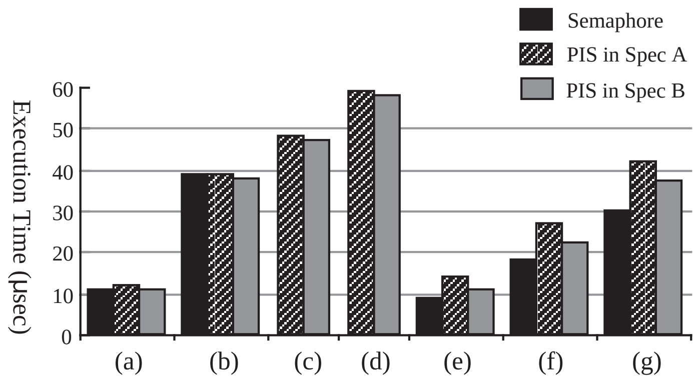

Figure 5: Execution Times

that is locked by τ_3. According as the priority inheritance rule, τ_3 inherits the priority of τ_2. If τ_1 invokes `wai_pis(`S_1`)`, τ_1 is blocked and τ_2 inherits the priority of τ_1. Because τ_2 is waiting for S_2 and S_2 is locked by τ_3, τ_3 also inherits the priority of τ_1. The execution time since τ_1 is about to invoke `wai_pis` until τ_3 starts to execute is measured. A task dispatch time is included.

(e) The execution time of `sig_sem/pis` is measured when no task is waiting for the semaphore. No task dispatch occurs.

(f) The execution time of `sig_sem/pis` is measured when a task that has lower or the same priority with the running task is waiting for the semaphore and is made ready state. No task dispatch occurs.

(g) Suppose that task τ_1 has higher base priority than task τ_2 and that τ_1 is waiting for semaphore S_1 that is locked by τ_2. According as the priority inheritance rule, τ_2 inherits the priority of τ_1. When τ_2 invokes `sig_sem/pis(`S_1`)`, τ_1 can lock the semaphore. The priority of τ_2 is recovered to its original level and τ_1 begins to execute. The execution time since τ_2 is about to invoke `sig_sem/pis` until τ_1 starts to execute is measured. A task dispatch time is included.

When no priority inheritance occurs, the execution time of `wai_pis` system call is almost same with that of `wai_sem` system call. The execution time to make a task inherit the priority of another task is about 10 μsec with both specifications. This overhead is essential to the priority inheritance semaphore function and considered to be reasonable.

The execution times of `sig_pis` with Specification A is about 40 to 50% slower than those of `sig_sem`. This is because the released priority inheritance semaphore must be removed from the list of locked priority inheritance semaphores of the invoking task, and because the priority of invoking task must be recovered if necessary. We have presented only the cases in that the task invoking `sig_pis` system call obtains no other priority inheritance semaphores. If the task obtains some other priority inheritance semaphores, the execution times are prolonged a bit. The execution times will be improved, if we impose the restriction that `wai_pis`–`sig_pis` pairs must be properly nested, in other words, the restriction that the priority inheritance semaphore that is locked last must be released first. This is because the list of locked priority inheritance semaphores can be handled LIFO (last-in first-out) manner with this restriction.

With Specification B, the execution times of `sig_pis` are also slower than those of `sig_sem`, but they are faster than those with Specification A. They remain unchanged when the task invoking `sig_pis` system call obtains some other priority inheritance semaphores.

5 Conclusions

In this paper, we have investigated on the methods to incorporate priority inheritance to the ITRON specification. As the first step, we have proposed two specifications of priority inheritance semaphore function with which the basic priority inheritance protocol can be realized. Then, we have implemented both of the specifications and evaluated them from the view point of run-time efficiency.

Specification A is considered to be ideal for application programmers, but its implementation overhead is large. The execution time of releasing semaphore operation is 40 to 50% longer than that of normal semaphore. The required memory space is thought to be reasonable. Specification B is proposed as the specification which can be implemented efficiently, but some restrictions are imposed in designing application programs. It is application dependent and is beyond the scope of this paper which of the specifications is preferable.

The method to incorporate other variants of priority inheritance protocols, including the priority ceiling protocol, to the ITRON specification remains as future work. It also remains as future work if the priority inheritance message passing function and other synchronization or communication objects supporting priority inheritance should be adopted as primitive functions of the ITRON specification.

Acknowledgments

We wish to thank the colleagues of Sakamura Laboratory for their suggestion and encouragement.

References

[1] 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.

[2] M. Fukuda and H. Nokubi, "An RTOS allowing the gradual migration," in *Proc. of the Ninth TRON Project Symposium*, pp. 107–114, IEEE CS Press, 1992.

[3] T. P. Baker, "Stack-based resource allocation policy for realtime processes," in *Proc. Real-Time Systems Symposium*, pp. 191–200, Dec. 1990.

[4] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems.* Kluwer Academic Publishers, 1993.

[5] K. Sakamura, ed., *μITRON 3.0 Standard Handbook.* Tokyo: Personal Media, 1993. (in Japanese).