Feasibility Test and Scheduling Algorithm for Dynamically Created Preemptable Real-Time Tasks

Yong-Seok Kim

Abstract

An optimal algorithm is presented for feasibility test and scheduling of real-time tasks where tasks are preemptable and created dynamically. Each task has an arbitrary creation time, ready time, maximum execution time, and deadline. Feasibility test and scheduling are conducted via the same algorithm. Time complexity of the algorithm is O(n) for each newly created task where n is the number of tasks. This result improves the previous result of O(n log n). It is shown that the algorithm can be used for scheduling tasks with different levels of importance. Time complexity of the algorithm for the problem is O(n^2) which improves the previous result of O(n^2 log n).

I. Introduction

Real-time systems are important in a wide range of applications such as defence control systems, intelligent weapons, automated manufacturing systems, power plant control systems, and so on. In such systems, each real-time task has a requirement that it should be completed within its timeing constraint called deadline. Violation of deadline may result useless output or catastropic failure of the system. Therefore, it is important to schedule real-time tasks so that each of them is completed within its deadline. For a given set of tasks, we have to test the feasibility that each of the tasks can be completed within its deadline. If the task set is feasible, we can make a scheduling. In static systems, information of every task is available before scheduling. Therefore, scheduling is static, that is, there is no addition or removal of tasks after scheduling. In this case, scheduling can be conducted off-line before execution of any task. In dynamic systems, on the contrary, each task can be created in an arbitrary time and information of a task is not available until it is created. Therefore, scheduling should be dynamic. That is, whenever a new task is created, feasibility should be tested for the task set including the new task, and if feasibile all tasks should be rescheduled. This paper presents an algorithm for feasibility test and scheduling of preemptable real-time tasks in dynamic systems.

Manuscript received July 25, 1996; accepted April 1, 1998.
The author is with the Kangwon National University.

Tasks in real-time systems can be periodic or aperiodic. For example, in process control systems, tasks periodically read input data and respond with appropriate actions. In automatic assembling systems, on the contrary, tasks respond to aperiodic events such as arriving various kinds of parts. This paper addresses scheduling problem of aperiodic tasks. For periodic tasks, rate monotonic scheduling algorithm is commonly used due to its simplicity [9]. It gives fixed priorities to tasks in the order of reverse of their periods. That is, tasks with shorter periods have higher priorities. But this algorithm can not fully utilize the processor. In the worst case, the processor utilization can be reduced to 69%.

For aperiodic tasks, full processor utilization can be achieved by scheduling tasks based on their deadlines [9]. However, scheduling algorithms for aperiodic tasks can also be applied to periodic tasks as mentioned by Cheng et al. in [2]. For example, a periodic task can be considered as a set of subtasks, where each subtask coresponds to each period of the task, and we only have to consider the subtasks within a time between zero and the least-common-multiple of the tasks' periods. If the subtasks within the interval can be scheduled by applying an algorithm for aperiodic tasks, all the other subtasks of the periodic tasks can also be scheduled.

Each task has an arbitrary ready time, maximum execution time, and deadline. A given set of tasks is said to be *feasible* if they can be scheduled such that each task is started after its ready time and completed within its deadline. A scheduling is said to be *optimal* if it satisfies ready times and deadlines of all tasks for any feasible task set. As a result of previous researches [1-7], the earliest deadline first (EDF) scheduling was shown to be optimal [3,6]. Mok and Dertouzos [5] showed that the least laxity first scheduling is also optimal. Horn [10] presented a static algorithm to schedule aperiodic tasks, but the time complexity was not given. Cheng et al. [2] claimed that Horn's algorithm is $O(n^2)$ time complexity, where n is the number of tasks. Schwan and Zhou developed a static algorithm of $O(n^2)$ time complexity. They adapted binary search tree to find an idle time slot for each task. They claimed that if the task rejection rate is low the time complexity is $O(n \log n)$. Kim [4] presented a static scheduling algorithm based on EDF scheduling. The algorithm utilizes well-known heap data structure, and improves the time complexity to $O(n \log n)$ regardless of the rejection rate.

In dynamic systems, each newly created task should be accepted if and only if the task set consisting of already accepted tasks and the new task is feasible. Recently, Kim and Lee [8] presented a dynamic scheduling algorithm of $O(\log n)$. However, it does not conduct feasibility test for a new task on its creation time but on the chance for its execution. That is, the decision of acceptance of each newly created task is delayed and made on the time that the new task has higher priority then all other tasks already accepted but not completed. Schwan and Zhou [7] presented a dynamic scheduling algorithm with time complexity of $O(n \log n)$ for each newly created task. For feasibility test and rescheduling, a known $O(n \log n)$ static scheduling algorithm is applied to a subset of tasks. In the algorithm, a reserved time slot list is used and a binary search tree is applied to find an idle time slot for each created task. The decision of whether the new task should be accepted for execution or rejected is made on the creation time. But they ignore the current scheduling information on feasibility test and rescheduling. Whenever a new task is created, all the tasks within the duration of the new task are rescheduled without utilization of the current information.

The motivation of this paper is full utilization of the current scheduling information to improve the time complexity. If the newly created task can be added successfully to the current scheduling, the task set including the new task is feasible and the resulting scheduling is accepted as the current scheduling. Otherwise, it is not feasible and the new task is rejected. To fully utilize the current information, two lists, task list and slot list, are introduced. To reduce the time complexity, they are interlinked each other, and the feasibility test and rescheduling are completed by one-pass investigation of both of them. The resulted algorithm improves the time complexity of dynamic scheduling algorithm to $O(n)$.

In real-time systems, tasks may have different levels of importance, called priorities. If a given set of tasks is not feasible we have to select a feasible subset of tasks, and tasks with higher priorities are preferable. Any accepted task should not result rejection of tasks with higher priorities than itself. There is no known algorithm explicitly addressing the problem. A strait forward algorithm for static systems can be derived by applying static scheduling algorithms [4,7] in the order of priorities. Then, time complexity will be $O(n^2 \log n)$ as shown in section 5. The second result of this paper is that the presented algoritm can solve the problem in time complexity of $O(n^2)$.

II. Description of the Algorithm

The algorithm of Schwan and Zhou [7] is based on binary search. They repeatedly apply binary search to find free time slots for each task which should be rescheduled to make a room for the newly created task. Therefore, the time complexity is $O(\log n) \cdot O(n) = O(n \log n)$. The presented algorithm improves the complexity by fully utilizing the current scheduling information which is represented as interlinked two lists. Whenever a new task is created, feasibility test is accomplished at once through a linear search of the lists. Rescheduling is accomplished through modifying the lists during the linear search. This method improves the complexity to $O(n)$.

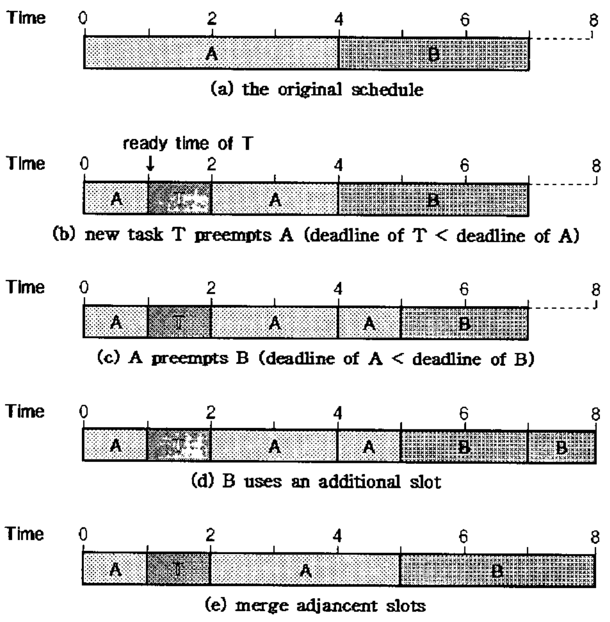

Time
0
2
4
6
8
A
B
(a) the original schedule
ready time of T
T
(b) new task T preempts A (deadline of T < deadline of A)
(c) A preempts B (deadline of A < deadline of B)
(d) B uses an additional slot
(e) merge adjancent slots

Fig. 1. Adjusting Time Solt List

The presented algorithm is based on the well-known EDF scheduling. Each task can be created in an arbitary time, and has an arbitrary ready time, maximum exection time, and deadline. Schedule information is described by interlinked two lists, a list of accepted tasks and a list of time slots. Accepted tasks are sorted on the task list based on deadlines such that the head entry has the earliest deadline. A time slot consists of begining time, ending time, and a pointer designating a task on the task list. Each time slot of the slot list represents an actual execution time slot for the task designated by the slot. Time slots are sorted on the slot list in the order of increasing time.

Rescheduling for a new task is accomplished by finding sufficient slots, from the head of the slot list, for the execution time of the task. If a slot is free, it is assigned for the new task. If the task designated by a slot has deadline later than the new task, it is also used for the new task, that is, the new task preemts the designated task based on EDF scheduling. When sufficient slots are found for the new task, the same procedure is applied for the preempted tasks in the order of deadlines to fill up the preempted time. Fig. 1 shows a typical example. If there is a task violating its deadline, the new task should be rejected since it means that the increased task set consisting of already accepted tasks and the new task is not feasible. Otherwise, the increased task set is feasibile, the new task is accepted, and the new schedule is used for execution. Since the algorithm will be completed in one pass search of the task list and slot list as shown later, time complexity is improved.

A task entry has 3 attributes, *ready* (ready time), *need* (necessary additional time to complete execution), and *deadline*, and has additional 2 attributes, *next* and *prev*, to be used for doubly linked list. A slot entry has 3 attributes, *begin* (begining time), *end* (ending time), and *task* (poniter designating a task entry), and has additional 2 attributes, *next* and *prev*, to be used for doubly linked list. As an example, Fig. 2 describes the scheduling for the case of Fig. 1 (b). *T(t)* denotes a task entry for task *T* where *t* is the attribute *need.* $S(t_1,t_2)$ denotes a slot entry where t_1 and t_2 are the attributes *begin* and *end*, respectively. The first slot *S(0,1)* is used for execution of task A, *S(1,2)* is used for T, and so forth. The entry *A(1)* means that task A requires additional slot of length 1 to complete its execution.

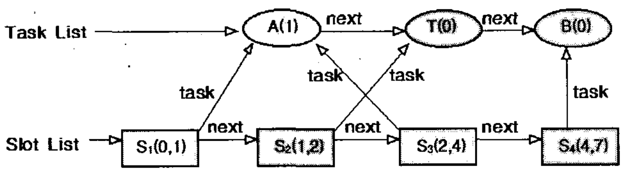
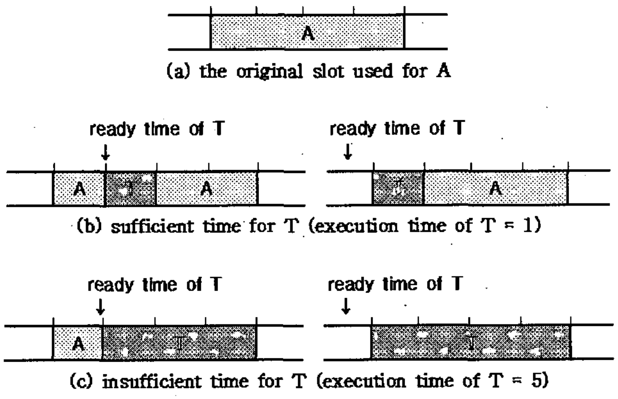


Fig. 2. Task List and Slot List

Fig. 3 describes the rescheduling algorithm for each newly created task, respectively, in a C-like language. For simplicity in description of the algorithm, an additional task *NullTask* is introduced such that *ready* = 0, *need* = infinite and *deadline* = infinite. *NullTask* will be executed virtually whenever there is no task ready to be executed. Initially, the task list has only *NullTask*, and the time slot list has only one entry such that *begin* = 0, *end* = infinite, and *task* = *NullTask.*

```
Initial values of the accepted task list (TL) and the time slot list (SL):

  TL has only one entry NullTask such that
       ready = 0, need = infinite, and deadline = infinite;
  SL has only one entry such that
       begin = 0, end = infinite, task = NullTask;

Whenever a new task N with attributes ready (ready time), need (maximum
         execution time), and deadline is created:

L1  duplicate TL and SL into TL2 and SL2, respectively, to recover them
         on rejection of N;
L2  insert N into TL;

L3  S0 = head of SL;  T0 = head of TL;
L4  while (the algorithm is not stopped) {
L5     find the first task T, from T0, such that T.need > 0;
L6     if (T == NullTask)          /* complete slot assignment for all tasks */
L7        accept N and stop the algorithm;
L8     find the first slot S, from S0, such that
          S.end > T.ready and S.task.deadline > T.deadline;
L9     if (S.begin > T.deadline)               /* violate deadline of T */
L10       reject N, recover TL and SL from TL2 and SL2, respectively, and
              stop the algorithm;
L11    if (S.begin < T.ready) {        /* T use the later part of S */
L12       split S into S1 and S2 such as
             S1.begin = S.begin, S1.end = T.ready, S2.begin = T.ready, and
             S2.end = S.end;
L13       S1.task = S.task;  S2.task = S.task;
L14       S = S2;
L15    }
L16    if (S.begin + T.need < S.end) {         /* T use the former part of S */
L17       split S into S1 and S2 such as
             S1.begin = S.begin, S1.end = S.begin + T.need,
             S2.begin = S.begin + T.need, and S2.end = S.end;
L18       S1.task = T;  S2.task = S.task;
L19       S2.task.need = S2.task.need + T.need;
L20       T.need = 0;
L21       S = S1;
L22       T0 = T.next;  S0 = S2;
L23    } else {                /* T use the whole S */
L24       S.task.need = S.task.need + (S.end - S.begin);
L25       S.task = T;
L26       T.need = T.need - (S.end - S.begin);
L27       T0 = T;  S0 = S.next;
L28    }
L29    if (S.prev.task == S.task)        /* merge two slots */
L30       S.begin = S.prev.begin, and remove S.prev from SL;
L31 }
```

Fig. 3. Description of the Algorithm


A
(a) the original slot used for A
ready time of T
ready time of T
A
T
A
T
A
(b) sufficient time for T (execution time of T = 1)
ready time of T
ready time of T
A
T
T
(c) insufficient time for T (execution time of T = 5)

Note) deadline of T < deadline of A

Fig. 4. Task Preemption

When a new task *T* is created, the algorithm examines slots for preemption from the head of the slot list. If the ready time of *T* is later than the end of the slot, or the designated task of the slot has earlier deadline than *T*, the next slot is examined. Otherwise, *T* preempts the designated task. Fig. 4 shows an example that task *T* preempts task *A*. If the slot has sufficient time for *T*, the excess time is used for the original task *A* as shown in Fig. 4 (b). Otherwise, the slot is used for T as shown in Fig. 4 (c), and the lacking time for *T* will be provided on examining following slots.

Task	Ready Time	Execution Time	Deadline
A	1	5	14
B	0	2	5
T	4	3	10

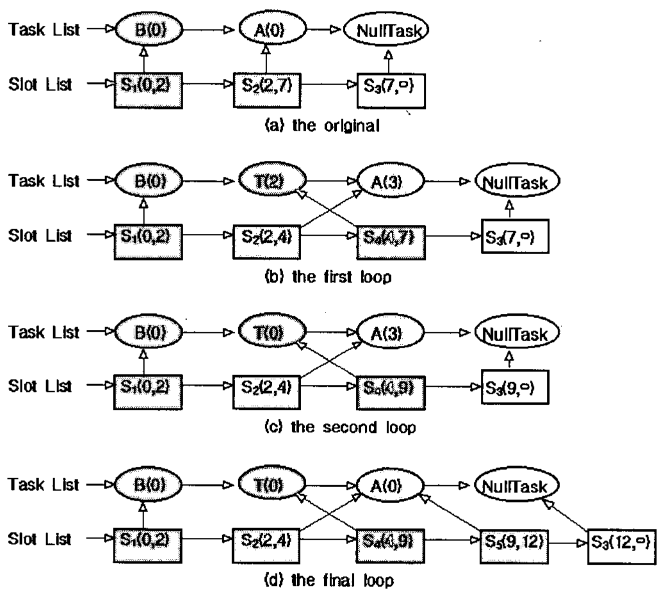


Fig. 5. Adjusting Sequence of Data Structures

For a typical example of the algorithm, an adjusting sequence of data structures is shown in Fig. 5. Fig. 5 (a) is the original schedule already accepted. Assume a new task *T* is created, which has ready time 4, execution time of 5 time units, and deadline earlier than *A*. At first, the algorithm inserts *T* into the task list and finds a candidate slot for preemption from the head of the slot list. The first slot *S(0,2)* is skipped since *T* is not ready until the end of the slot. Since the ready time 4 is on *S(2,7)* and *T* has earlier deadline than the designated task *A*, *S(2,7)* is splitted into two slots, *S(2,4)* and *S(4,7)*, by the line L12 of Fig. 3. *S(2,4)* is used for the original task *A*. *S(4,7)* is used for *T* by the line L25 as shown in Fig. 5 (b). Then, the attribute *need* of the designated task *A* is increased to 3, the preempted time, by the line L24, and the attribute *need* of *T* is decreassed to 2 by the line L26. To fill up additional 2 time units of *T*, the next slot *S(7,infinite)* is splitted into *S(7,9)* and *S(9,infinite)*, and *S(7,9)* is used for *T* by the line L18. Then, the attribute *need* of *T* becomes 0. Since the two adjacent slots, *S(4,7)* and *S(7,9)*, are used for the same task *T*, they are merged into *S(4,9)* as shown in Fig. 5 (c) by the line L40. The next task with a positive *need* attribute is *A*. The first candidate slot for *A* is *S(9,infinite)*, and it is splitted into *S(9,12)* and *S(12,infinite)*. *S(9,12)* is used for *A* to fill up the preempted time as shown in Fig. 5 (d). There is no more task with positive *need* attribute except *NullTask* and, therefore, the new task T is accepted and the algorithm ends.

III. Correctness of the Algorithm

Correctness of the presented algorithm will be proven by showing the output of the algorithm is identical to EDF which was shown to be optimal [3,6]. The task list *TL* of Fig. 3 is a sorted list of tasks based on deadline. Initially, *TL* contains *NullTask* only which is a virtual task with infinite deadline. The slot list *SL* is a list of time slots in increasing order. Initially, *SL* contains only one slot *S(0, infinite)* indicating the whole time and assigned to *NullTask*. Since each time slot assigned to *NullTask* means idle time, the whole time is free.

The algorithm maintains *TL* so that each task, except *NullTask*, on *TL* has *need* attribute as 0. That is, no task on *TL* need more time slots. Whenever a new task *N* is created, it is inserted into *TL* in the order of deadline (line L2). The *need* attribute of *N* is the execution time of *N*. On the first loop of L4, since *N* is the only task with positive *need* attribute except *NullTask*, *N* will be selected on line L5. Line L8 finds the first slot after the ready time of *N* and assigned to a task with later deadline than *N*. That is, it finds a task which can be preempted by *N*. If the slot begins after the deadline of *N* (line L9), there is no more slot assigned to tasks which can be preempted by *N*. Therefore, the whole task set is not feasible and *N* should be rejected (line L10). If the found slot overlaps the ready time of *N* (line L11), the slot is splitted into two slots by the point of the *N*'s ready time (line L12). Then the former slot is skipped and the later slot can be reassigned to *N* (line L14).

Now we have a slot which can be reassigned to *N*. If the slot is sufficient for *N* (line L16), a subslot needed for *N* is reassigned to *N* and the remaining subslot is still assigned to the original task (line L17 and L18). Otherwise, the whole slot is reassingned to *N* and the amount of shortage can be supplied on the following loops of line L4. If we can't find sufficient slots which can be reassigned for *N* on the following loops, the whole set fo tasks is not feasible and *N* will be rejected on line L10.

Whenever a slot is reassigned to *N*, the original task of the slot should be refilled the same amount of preempted time (line L19 and L24) on the following loops. Since *TL* is sorted based on deadline, the preempted tasks are located after the preempting task on *TL*. Therefore, after *N* is completed, the task found on line L5 will be the task with earliest deadline among the preempted tasks and it can be refilled in the same way of the above procedure on the following loops. Moreover, the additional slots to be refilled for the tasks should be found on the following slots of *SL*. If all tasks except *NullTask* are satisfied ultimately, the whole task set is feasible and the algorithm will be ended on line L7. Otherwise, the whole task set is not feasible, *N* is rejected, and *TL* and *SL* will be restored on line L10.

Therefore, the lines from L8 to L28 shows that *N* preempts all tasks with later deadlines for the amount of execution time. Each preempted task also preempts other tasks with later deadlines than itself for the amount of preempted time. Consequently, the algorithm always selects the task with the earliest-deadline among tasks requiring additional time slots, and finds the earliest candidate slot for the task. Therefore, the algorithm satisfies EDF scheduling which was shown to be optimal [3,6].

IV. Time Complexity

Initially, there is only one slot used for *NullTask*. Assume that all tasks of the task list are created in increasing order of deadlines. Then, there is no preemption of already accepted tasks and each task uses some parts of *NullTask* slots. Since each task splits *NullTask* slots at most two times, by the lines L12 and L17 of Fig. 3, the number of slots is at most $2n + 1$ where n is the number of tasks. For a given set of tasks, the algorithm results the same slot list for any creation sequence. Therefore, the number of slots is $O(n)$ and time complexity of L1 is $O(n)$. Obviously, time complexity of L2 is $O(n)$.

Time complexities of L5 and L8 are $O(n)$ for the loop L4 since the task list and slot list are examined in one pass search for the whole loop. Time complexity of L10 is $O(n)$. Time complexity of any other line of the loop L4 is $O(1)$ and its time complexity for the whole loop is $O(n)$ since the loop is repeated in $O(n)$ times. Therefore, time complexity of the loop L4 is $O(n)$. Consequently, time complexity of the algorithm is $O(n)$.

In addition to the on-line scheduling overhead, task switching overhead should be considered. Basically, the scheduling output of the presented algorithm is identical to the basic EDF [6]. In the worst case, whenever a new task is created, its priority is higher than the current task and it preempts the current task. Obviously, the number of resumptions is equal to the number of preemptions.

Therefore, the total number of context switchings is $2n$ and this fact was proven by Dertouzos and Mok [11]. This context switching overhead is identical to the overhead of any other algorithm based on EDF, including Schwan and Zhou's algorithm [7]. On the contrary, the least laxity first algorithm causes heavy context switching overhead [5].

V. Scheduling Tasks with Priorities

In real-time systems, tasks may have different levels of importance, called priorities. If a given set of tasks is not feasible we have to select a feasible subset of tasks and tasks with higher priorities are preferable than tasks with lower priorities. Any accepted task should not result rejection of tasks with higher priorities than itself. That is, when a task with higher priority is created and the whole set of tasks is not feasible, we have to give up some tasks with lower priorities even though they are already accepted and started.

There is no known algorithm explicitly addressing the problem. A strait forward algorithm can be derived by applying static scheduling algorithms. That is, for each task in the order of priorities, it is accepted if and only if the set of accepted tasks with the additional task is feasible. Then, time complexity of the algorithm will be $O(n^2 \log n)$ since there are known static scheduling algorithms of $O(n \log n)$ [4,7] and the algorithm will be applied n times.

The presented algorithm also can not be applied directly for the problem, but it can be used as follows. If we apply the algorithm for each task in the order of priorities instead of the order of task creation time, the scheduling meets the requirement of the problem. Then, time complexity is improved to $O(n^2)$ since the algorithm with time complexity of $O(n)$ is applied n times. In a system, tasks may not be rejected if they are already started. In that case, we can apply the algorithm for already started tasks at first, and then for each task, not started, in the order of priorities.

VI. Conclusions

An optimal algorithm is presented for feasibility test and scheduling of real-time tasks where tasks are preemptable and created dynamically. Each task has an arbitrary creation time, ready time, maximum execution time, and deadline. Feasibility test and scheduling are conducted via the same algorithm. Time complexity of the algorithm is $O(n)$ for each newly created task. This result improves the best previous result of $O(n \log n)$. It is shown that the algorithm can be used directly for scheduling tasks with different levels of importance, called priorities. Time complexity of the algorithm for the problem is $O(n^2)$ which improves the best previous result of $O(n^2 \log n)$.

References

[1] J. Blazewicz, "Scheduling dependent tasks with different arrival times to meet deadlines," E. Gelenbe (ed), Modeling and Performance Evaluation of Computer Systems North-Holland, 1976.

[2] S. -C. Cheng, J. A. Stankovic, and K. Ramamritham, "Scheduling algorithms for hard real-time systems - A brief survey," J. A. Stankovic and K Ramamritham (ed), Hard Real-Time Systems (Tutorial), IEEE, pp. 150-173, 1988.

[3] M. L. Dertouzos, "Control robotics: the procedural control of physical processes," Proc. of the IFIP Congress, pp. 807-813, 1974.

[4] Y. -S. Kim, "An optimal scheduling algorithm for preemptable real-time tasks," Information Processing Letters, no. 50, pp. 43-48, 1994.

[5] A. K. Mok and M. L. Dertouzos, "Multiprocessor scheduling in a hard real-time environment," Proc. 7th Texas Conf. Computing Systems, pp. 5.1-5.12, Nov. 1978.

[6] J. M. Moore, "An *n* job, one machine sequencing algorithm for minimize the number of late jobs," Management Science, vol. 15, no. 1, pp. 102-109, 1968.

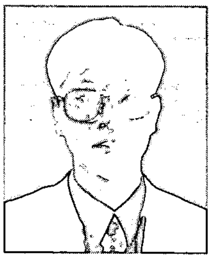
[7] K. Schwan and H. Zhou, "Dynamic scheduling of hard real-time tasks and real-time threads," IEEE Trans. on Software Engineering vol. 18, no. 8, pp. 736-748, 1992.

[8] Y. -S. Kim and H. -G. Lee, "An Optimal Algorithm for Dynamic Scheduling of Preemptable Real-Time Tasks," Journal of the Korea Information Science Society, vol. 22, no. 7, pp. 1029-1035, 1995.

[9] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in hard real-time environment," Journal of the ACM, vol. 20, no. 1, pp. 46-61, Jan. 1973.

[10] W. A. Horn, "Some simple scheduling algorithms," Naval Res. Logist. Auart., vol 21, pp. 177-185, 1974.

[11] M. L. Dertouzos and A. K. Mok, "Multiprocessor On-Line Scheduling of Hard Real-Time tasks," IEEE Trans. on Software Engineering vol. 15, no. 12, pp. 1497-1506, 1989.



Yong-Seok Kim graduated from Seoul National University, Korea, in 1984 with a B.S. in Oceanography as major and Electronics Engineering as minor. He received an M.S. and a Ph.D. in 1986 and 1989, respectively, in Electrical and Electronics Engineering from KAIST (Korea Advaced Institute of Science and Technology). During the periods of 1989-1994 and 1994-1995, he was a senior research staff at KITECH (Korea Institute of Industrial Technology) and KETI (Korea Electronics Technology Institute), respectively. He is currently an assistant professor in Kangwon National University, Korea. His research interest includes real-time operating systems, real-time communication, continuous media retrival, manufacturing message communication, fault-tolerant systems, and parallel computer architecture.