Scheduling Soft Aperiodic Tasks in Adaptable Fixed-Priority Systems*

Jongwon Lee[J], Sungyoung Lee[JJ], Hyungill Kim[JJ],

[J]Software Research Lab.
Korea Telecom, Seoul, Korea
email: jwlee@coral.kotel.co.kr

[JJ]Department of Computer Engineering
Kyunghee University, Seoul, Korea
email: {slee, hikim@nms.kyunghee.ac.kr}

Abstract

This paper presents a new type of scheduling algorithm for servicing soft deadline aperiodic tasks in adaptable fixed-priority real-time systems. The major goals of our proposed task scheduling are not only to guarantee all the deadlines of periodic tasks and to obtain fast response time for aperiodic tasks, but also to retain considerable scheduling predictability. To achieve these goals, we have adopted a new aperiodic task scheduling principle in which a normal fixed-priority assignment strategy and the information on a preassignment table built off-line are properly mingled somewhat dynamically according to the aperiodic tasks' arrivals at runtime. The paper also shows some simulation results in terms of the average aperiodic response time, verifying that the new algorithm offers significant performance improvements over other conventional joint scheduling algorithms, especially under a heavy transient overload.

1. Introduction

Real-time systems are used to control the physical processes that range in complexity from an automobile fuel ignition device to a large military defense system. Stankovic [16] pointed out that it is very important to develop an aperiodic task scheduler which works well under transient overload and gives a fast response time for aperiodic tasks while guaranteeing the deadlines of periodic tasks in unpredictable environments.

Considerable research has been done in the area of joint scheduling of periodic and soft aperiodic tasks in fixed-priority systems [2],[5],[6],[7],[14],[15],[18]. Bandwidth preserving algorithms [5],[14],[15], which create a periodic task for servicing aperiodic requests, have substantially improved the response time of aperiodic tasks compared to the conventional aperiodic task scheduling algorithms such as background processing and polling tasks. The bandwidth preserving algorithms, however, make it hard to decide an optimal bandwidth preservation capacity to cope with the stochastic arrival behavior of aperiodic tasks. Also it cannot maximally utilize the slacks for the aperiodic tasks due to a fixed value of the bandwidth preservation capacity.

Lehoczky and Ramos-Thuel [6] developed the slack-stealing algorithm by means of greedy methods, which was proved to be optimal in the sense that it simultaneously minimized the response time of all aperiodic tasks, provided that they were served in FIFO order. The slack-stealing algorithm does not create a periodic server for aperiodic task service. Rather it creates a passive task, referred to as the slack stealer, which when prompted for service attempts to make time for servicing aperiodic tasks by stealing all the processing time it can from the periodic tasks without causing their deadlines to be missed. This algorithm however, has two significant drawbacks. Firstly, it requires a relatively large amount of calculations because of its recursive checking through all the priority levels at each scheduling point. Consequently, its direct implementations may not be practical. Secondly, it cannot fully utilize all the available slacks for the largest amount of aperiodic processing because it counts on the ceiling values of periodic processing requirements.

Recently Tia, Liu and Shankar [18] improved the slack-stealing approach for soft aperiodic processing by means of non-greedy methods. They pointed out the greedy approach in [6] is not optimal when the

* This work was supported in part by KOSEF under Grant No. 95-0100-07-01-3.

class of algorithms is enlarged to include non-greedy algorithms. Subsequent extensions of the slack-stealing algorithm have been introduced by Davis, Tindell, and Burns [2]. They generalized [6] to a broad class of scheduling problems and demonstrated approximations to the full slack-stealing algorithm that are less computationally complex but still show reasonably good performance.

Meanwhile, Thuel and Lehoczky [12], [17] extended the slack-stealing algorithm for jointly scheduling hard deadline periodic tasks and hard deadline aperiodic tasks using fixed-priority methods. Works on the on-line scheduling of hard aperiodic tasks in the dynamic priority method have been reported by Chetto and Chetto [1], Schwan and Zhou [13]. Their works assume that all periodic tasks are scheduled according to the Earliest Deadline algorithm. In particular, Schwan's algorithm does not give any preferential treatment to the periodic tasks, unlike common approaches to soft aperiodic tasks in fixed-priority preemptive systems. Although they developed a few elegant dynamic scheduling methods for tasks with well-defined timing constraints, they reserved for the future work on other types of on-line scheduling algorithms which relax the hardness of deadlines.

In this paper, we are introducing the ***Critical Task Indicating (CTI) algorithm***, a new approach to the joint scheduling of periodic and soft aperiodic tasks in an adaptable fixed-priority uniprocessor real-time system, which allows temporary changes to the given periodic priority order at runtime; and we show that this method can offer reasonable improvements over the other conventional joint scheduling algorithms especially under heavy transient overload. The major goals of our proposed task scheduling are not only to guarantee all the deadlines of periodic tasks and to get the response time for soft aperiodic tasks as small as possible, but also to provide a good scheduling predictability. To achieve these goals, we have adopted a periodic task scheduling principle in which a normal fixed-priority assignment strategy [8],[9],[10] and the information on a preassignment table built off-line are properly mixed somewhat dynamically according to the aperiodic tasks' arrivals at runtime. The role of the preassignment table, called the CTI table, is to indicate the critical periodic tasks (if any) at each scheduling point that must be assigned and executed immediately to meet their deadlines at runtime.

The key to the matter is the creation of the required CTI tables in advance. To work out the problem, we have carefully developed a different kind of fixed-priority preassignment strategy, called the ***deadlinewise preassignment***, for which a periodic task set defers each task's execution start time toward the deadline at its maximum according to the given fixed-priority. The idea comes from the fact that the value of the value functions of a set of periodic tasks (generally they are step functions) is constant up to their deadlines [4]. Consequently, the slacks which were made by artificially deferring the execution time of periodic tasks toward their deadlines can be utilized by the aperiodic tasks if they arrive at those time zones. Otherwise, those slacks can be used for the periodic tasks just as per normal fixed-priority scheduling. The resulting information table which can fully be used as a CTI table in the joint scheduling looks very much like a transposed form of the normal fixed-priority preassignment table.

The major merits of the CTI algorithm are: 1) it does not require heavy computations to check if there are slacks for aperiodic tasks, since it refers only to the preassignment table at each scheduling point at runtime, 2) it gives faster aperiodic responses than the other conventional joint schedulings in fixed-priority systems because of its dynamic property of selecting periodic scheduling strategies depending on whether aperiodic tasks are ready or not at runtime, 3) its implementation is relatively easy because of its simplicity, and 4) it provides a good scheduling predictability because it uses the CTI tables containing prescheduling information.

The remainder of the paper is organized as follows. Section 2 addresses some of the general concepts which have an important effect on the development of the whole scenario. It also provides the deadlinewise preassignment concepts and their important properties. Section 3 describes the main algorithm including an example, its feasibility analysis, and some additional topics on it. Section 4 shows the simulation results compared to those of the other major aperiodic scheduling algorithms. Section 5 provides a summary and discusses some drawbacks and future plans.

2. The background of the CTI algorithm

In this section we introduce a new periodic scheduling method, called the fixed-priority deadlinewise preassignment, and its major properties to establish the minimal theoretical background for the CTI algorithm in addition to briefly discussing some of the basic environmental notions and assumptions.

2.1 Motivation

There is a common important motivation to

developing a new aperiodic task scheduling algorithm which can be easily implemented to service a mixture of hard deadline periodic and soft deadline aperiodic tasks in real-time system environments. Namely, the system should guarantee all the deadlines of periodic tasks and should obtain an average aperiodic response time which is as small as possible. As mentioned before, the slack-stealing algorithm gives a near-optimal solution to this problem in a situation where there are fixed-priority periodic tasks. We have, however, found in the verifying procedure of the algorithm that a new comparable algorithm could be developed in as much as the fact that all the given periodic tasks' instances can be shifted toward deadlines at their maximum depending on their given fixed-priority.

This deadlinewise shifting at any scheduling time interval increases the flexibility to service aperiodic task more quickly whenever there are so many aperiodic tasks requested at runtime that the CPU is always busy. But the problem is how to ensure that the scheduler effectively allocates periodic tasks when there are a small number of aperiodic tasks which lead the CPU to idle states so frequently. To deal with the problem we have decided to partially use the fixed-priority assignment methods which break up temporarily the deadlinewise shifting of the given periodic tasks at runtime.

2.2 Task Execution Model

A ***periodic task*** denoted by τ, is an infinite sequence of task instances requested at a fixed rate in a real time system environment. The request rate is defined to be its ***period***, denoted by **T**. Each of the task instances has the same magnitude of ***computation requirements***, denoted by **C**, and the ***deadline***, denoted by **D**, by which it must be completed. A ***periodic task set***, denoted by $\{\tau_1,\tau_2,\ldots,\tau_n\}$, is defined to be a set of arbitrary positive number of such periodic tasks. Any periodic task set has its ***hyperperiod*** which is the least common multiple of all the periods of the tasks in it. Note that every task in a task set is requested simultaneously at the start point of the hyperperiod and has the same deadline at the end point of it. An ***aperiodic task***, denoted by **A**, is a task having non-periodic request intervals. A ***slack*** is an available time interval, which has the length of a scheduling unit, for an aperiodic task.

2.3 Assumptions

To develop the CTI algorithm, we need some assumptions which include:

(A1) Deadline for a periodic task's instance is equal to the next request of the task.

(A2) Preemption over a periodic or an aperiodic task is always possible.

(A3) All overhead for context switching is counted into the corresponding periodic and aperiodic task's computation requirements.

2.4 Fixed-Priority Deadlinewise Preassignment Concepts

A periodic task scheduling method is classified as a fixed-priority ***deadlinewise preassignment*** if the tasks are assigned one after another according to the given fixed-priority in such a way that all the tasks are preassigned toward deadlines at their maximum. The priority preemptions in a deadlinewise preassignment take place in a manner similar to the other fixed-priority scheduling methods (e.g. rate monotonic priority assignment) except that part or all of the preempted task instance should be assigned prior to the preempting task instance.

Example 1. Suppose that a periodic task set with two tasks, τ_1 and τ_2, having the computation requirements, C_1=1 and C_2=2, and the periods, T_1=3 and T_2=5, respectively, is to be preassigned over a single hyperperiod, H=15, using the rate monotonic fixed-priority deadlinewise preassignment. The preassigning process is depicted in Figure 1. At start time, two task instances, τ_{11} and τ_{21}, arrives and assigned toward the deadlines, D_{11}=3 and D_{21}=5, respectively. At time 6, the task instance τ_{13} preempts τ_{22}. Also, at time 12, another preemption is occurs.

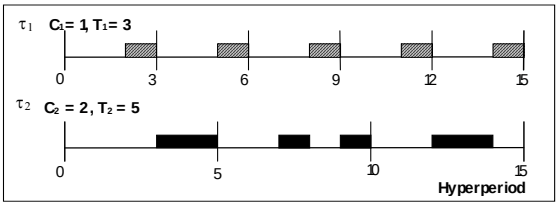

Figure 1. An example of the preassigning process using the deadlinewise preassignment.

Next we establish some of the basic notions used to analyze the feasibility of the deadlinewise preassignment for a given periodic task set. For a set of periodic tasks preassigned according to the deadlinewise preassignment, we say that an ***underflow*** occurs at **t** if a task is forced out of its given period beginning at **t** as a result of the others' preemptions.

The concept of overflow is applicable to both the on-line and the off-line fixed-priority schedulings, but on the other hand that of underflow only to the off-line fixed-priority schedulings. A ***deadlinewise critical instant*** for a given periodic task preassigned according to the deadlinewise preassignment is an instant at which the execution of a request for that task will begin, so that the largest time interval is required for its completion. The periodic interval for a periodic task having the shortest waiting time from the request to the start of the execution contains its deadlinewise critical instant. A ***deadlinewise critical zone*** for a given periodic task preassigned according to the deadlinewise preassignment is the time interval between a deadlinewise critical instant and the deadline for the corresponding request. The above two definitions are hinted in the definitions of the normal critical instant and critical time zone presented in [10]. The only difference between normal and deadlinewise is that the latter count on the deadlines instead of the requests. Any periodic task set with a fixed-priority order is ***deadlinewisely preassignable*** if no underflow occurs through all the deadlinewise critical zones for all the tasks over a single hyperperiod.

Example 2. Figure 1 shows two deadlinewise critical instants, execution start points of τ_{22} and τ_{23}, at priority level 2 and consequently two deadlinewise critical zones, **[7,10]** and **[12,15]**, respectively. Moreover, the periodic task set is deadlinewisely preassignable because no underflow occurs through all the deadlinewise critical zones. Note that every execution start point of the task instances at the priority level 1 is a deadlinewise critical instant.

Let us consider one of the properties related to a deadlinewise critical instant and the relationships between normal and deadlinewise schedulability.

Theorem 1. *For a given periodic task set with a fixed-priority order, the deadlinewise critical instant for any task occurs whenever the deadline of the task is identical to that of all higher priority tasks.*

Proof. Similar to the proof of Theorem 1 in [10] except that the preemption occurs on the basis of deadlines instead of exact request times. Consequently, the computation time zone for a task becomes the largest when its deadline is identical to that of all higher priority tasks. *

Example 3. The normal critical zone at the lowest priority level for the periodic task set given in Example 1 is the interval **[0,3]**. On the other hand, the deadlinewise critical zone for the same task is the interval **[12,15]**.

Theorem 2. *A periodic task set* $\{\tau_1, \tau_2, ..., \tau_n\}$ *with a fixed-priority order is deadlinewisely preassignable if, and only if, it is normally schedulable.*

Proof. By the definitions of deadlinewisely preassignable and normally schedulable, it is necessary and sufficient to show that for a task τ_k with its period T_k, no underflow occurs through the deadlinewise critical zone ***dcz(k)*** if, and only if, no overflow occurs through the normal critical zone ***ncz(k)***, where $1 \le k \le n$. Since the two periods on which ***dcz(k)*** and ***ncz(k)*** lay respectively are equal to T_k, it only remains to show that the length of ***dcz(k)*** is equal to that of ***ncz(k)***.

Now, by Theorem 1 in [10] and Theorem 1 above, the number of task requests through the period containing ***ncz(k)*** is equal to that of deadlines through the period containing ***dcz(k)***. Moreover, for each request for τ_k and all its higher priority tasks through the period containing ***ncz(k)***, there must exist a corresponding deadline over the period containing ***dcz(k)***, and vice versa.

Finally, by the assumption of the theorem, the computation requirements for each task that forms the lengths of ***dcz(k)*** and ***ncz(k)***, are unique. Hence, the length of ***dcz(k)*** is equal to that of ***ncz(k)***. *

Lemma 1. *For a given periodic task set with a fixed-priority order, the deadlinewise preassignment is feasible if, and only if, the fixed-priority assignment is feasible.*

Proof. The proof is directly induced from the above Theorem 2. *

3. The CTI Algorithm

A result of the deadlinewise preassignment for a given periodic task set is composed of scheduling information, a sequence of the execution start points and computation requirements of all task instances, on each instance the execution is delayed at its maximum. The information gives us the valuable knowledge that a periodic task preassigned to a scheduling point will miss its deadline under a real time scheduling situation unless it is assigned and executed immediately at that time. This means that the periodic task is critical (critical task) at that time in a real time scheduling situation. Moreover, the preassignment is the worst schedulable case with respect to the periodic task set under the condition that all the deadlines must be met. In this point of view, a new aperiodic task scheduling algorithm, called the ***CTI algorithm***, for a mixed scheduling of periodic and aperiodic tasks came to our minds.

3.1 Algorithm Description

Conceptually, the new algorithm is to assign a mixture of periodic and aperiodic tasks based on a hybrid scheduling method of the normal fixed-priority assignment and the deadlinewise preassignment information. The deadlinewise preassignment for a periodic task set through a single hyperperiod produces a scheduling table, called the ***CTI table***, to be used in the real time assignment. At runtime the table is frequently referenced by the scheduler to see if there are slacks available for aperiodic tasks.

In practice, a CTI table may be a character string for a small number of periodic tasks or may be an array of integers in general, i.e., an ordered sequence of periodic task's identifiers. For example, the CTI table for the periodic task set given in Example 1 will be a character string "001221021201221" with the length of its single hyperperiod, where '0', '1', and '2' denote identifiers of an empty slack, τ_1, and τ_2, respectively. With this simple data structure, all the execution start points and computation requirements of the periodic tasks can be fully represented in addition to the empty slacks for soft aperiodic tasks.

The behavior of the hybrid algorithm is dynamically determined at runtime depending on the information in the CTI table and arrivals of aperiodic tasks. For an arrival of an aperiodic task, the algorithm checks to see if ***(C1) there are slacks available for the aperiodic task in the CTI table at the current time, or (C2) the current critical task unit (whole or part of the task instance) already has been serviced.*** If one of the two conditions is met, it services the aperiodic task immediately. If not, it services the critical task. All of our ideas are based on the CTI table, i.e., the deadlinewise preassignment. On the other hand, if no aperiodic task arrived and no critical task is indicated, the algorithm works based on the fixed-priority assignment rule that was taken by the deadlinewise preassignment.

This dynamic property of the algorithm behavior provides that the maximum aperiodic capacity is always preserved through a single hyperperiod, although all the periodic deadlines are met strictly.

Let us examine the hybrid algorithm more systematically using a pseudocode written as in the following Figure 2. At line 1, the data structures for the algorithm including the CTI table, the timer, and the hyperperiod are initialized for a given periodic task set. Lines 2 to 9 form an infinite loop. The above two conditions, (C1) or (C2), for an aperiodic task ready (or arrived) are examined at line 3. If neither of them is met, the critical task is serviced. Otherwise, an aperiodic task which is ready (or newly arrived) is serviced immediately at line 4. If no aperiodic task is ready (or arrived) at line 4, the normal fixed-priority assignment algorithm is applied to the periodic tasks ready at that time at line 5. The pseudocode segment in line 6 takes over the control of the processor to cope with the CPU idle state whenever there is no aperiodic and no periodic task ready at that time. Although the pseudocode segment in line 7 should be ignored in a practical situation, it is necessary in a simulation. At line 8, the boundary for the hyperperiod is checked. If a boundary overflow occurs, all the global parameters such as the current pointer to the CTI table are renewed. Finally, the control goes back to line 3.

```
1  initialize data structures
2  loop begin
3      if (a critical periodic task unit not yet been
           serviced has occurred)    then service it
4      else if (aperiodic task(s) is ready or arrived) then
           service it
5      else if (periodic task(s) is ready or arrived) then
           service it
6      else process CPU idle state
7      advance timer
8      if ((timer_value MOD hyperperiod) is equal to
           zero)  then reinitialize the global parameters
9  end loop
```

Figure 2. A pseudocode of the CTI algorithm.

3.2 An Example

Suppose that there is a periodic task set with three tasks, τ_1, τ_2, and τ_3, with T_1=3, T_2=5, T_3=15, C_1=1, and C_2=C_3=2. We restrict our attention to the interval **[0,15]** which is a single hyperperiod for the task set. To obtain a CTI table for our task set, one can use the deadlinewise preassignment method described in subsection 2.4. In this example, the rate monotonic fixed-priority order has been applied. The obtaining procedure is fully depicted in Figure 3. Figure 3-A shows the deadlinewise preassignment results for each priority level. Figure 3-B shows the final result, the CTI table.

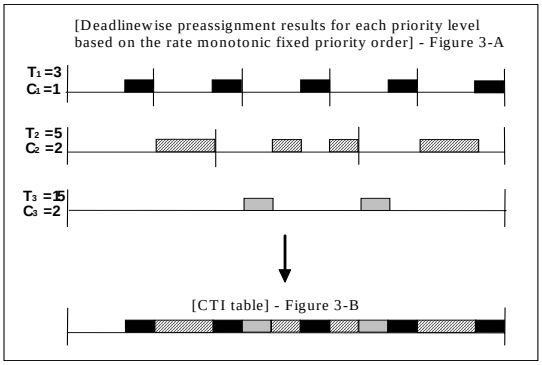

Figure 3. An example of creating a CTI table

Next we will examine the behavior of our CTI algorithm for the periodic task set given above and two aperiodic tasks with computation requirements C=1 arriving at t=5 and t=8, respectively. Figure 4 shows the mixed scheduling. The processor assigns periodic tasks by using the normal rate monotonic algorithm during **[1,5]** and **[6,8]** because there is no aperiodic task ready (or arrived) and no critical periodic task unit has not yet been serviced. The first aperiodic task A_1 has arrived at t=5 and has been serviced immediately because the current critical task unit, whole of the instance τ_{12}, had already been serviced at t=3. Similarly, the second aperiodic task A_2 has arrived at t=8 and been serviced immediately because the current critical task unit, the whole of the instance τ_{13}, had already been serviced at t=6. During the remaining interval **[9,15]**, all the periodic tasks will be scheduled in accordance with the information on the CTI table. The reason is that there always exists through the interval a critical periodic task unit which has not yet been serviced.

3.3 The Slack Discriminant

In the CTI algorithm, two consecutive conditions (C1) and (C2) given in subsection 3.1 have been used successively to test whether the current scheduling unit on the CTI table is a slack for a soft aperiodic task or not. To be more specific, the condition (C1) will be evaluated first simply by looking up the CTI table unit corresponding to the current value of the scheduling timer. If the unit denotes a slack identifier (i.e., a character '0' or an integer 0 depending on its data structure), then the required test comes to an end. However, if the unit denotes an identifier of a periodic task, it may be a slack or not. For these cases, the condition (C2) will be evaluated in sequence. This second condition will easily be checked based on the

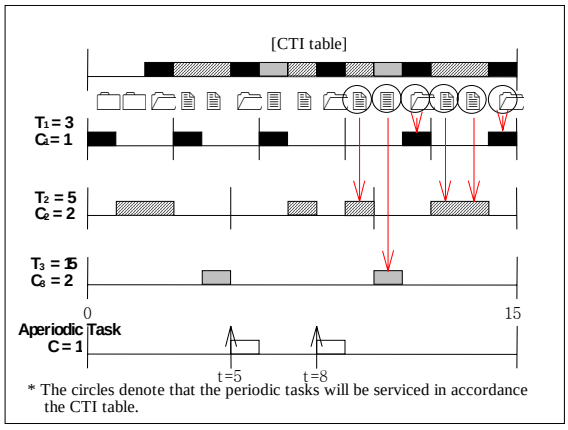

Figure 4. An example of the behavior of CTI algorithm

two kinds of special computation counters for each periodic task: one for cumulating ***all the computation processing completed*** and the other for cumulating ***all the computation requirement on the CTI table*** until the current scheduling time. Namely, the difference of the two counters for the task currently indicated by the CTI table makes it possible to check if the task has already been serviced. To provide a clearer slack identifying mechanism for soft aperiodic tasks, we need to formalize these to a slack discriminant that can be used for distinguishing a slack from the CTI table at the current scheduling point. Suppose that all the periodic tasks are sorted depending on those priorities so that the mapping from the set of task numbers (identifiers) to the set of positive integers $Y = \{1, 2, ..., n\}$ is one-to-one and onto Y where n denotes the number of periodic tasks. The resulting slack discriminant for CTI algorithm is

$$\boldsymbol{f}(\boldsymbol{t}) = \begin{cases} \text{slack} & \text{if } CTI[\boldsymbol{t}] = 0 \\ & \text{or } CP[CTI[\boldsymbol{t}]] - CR[CTI[\boldsymbol{t}]] > 0 \\ \text{critical task} & \text{otherwise} \end{cases}$$

where t denotes current scheduling time, *CTI* denotes CTI table(an array of integers, each element of which represents a slack or a periodic task), $CP[i]$ and $CR[i]$ (i represents an identifier of a periodic task) respectively denote all the computation processing done and all the computation requirement for each periodic task until t.

Based on the above slack discriminant, we have prepared a version of specified CTI algorithm in C language style for its practical use (see Figure 5.) The time complexity of the algorithm is O(**n**). Note that

the time complexity for constructing the CTI table off-line varies depending on its fixed-priority assignment method. For example, if one takes the rate monotonic fixed-priority scheme, then the time complexity will be $O(\mathbf{n}\log_2\mathbf{n})$ [9]. For a detailed explanation of the algorithm, the comment lines in the figure will help.

3.4 Feasibility Analysis

The major virtue of the fixed-priority assignment algorithms is its simplicity and ability to handle many practical problems, stability under transient overload, low scheduling overhead, and so on [3],[9],[10],[11]. In this subsection, we will analyze the feasibility of the CTI algorithm to show that almost all of these qualities supported by the fixed-priority algorithms could also be retained for periodic tasks.

Lemma 2. *For a given periodic task set with a fixed-priority order, the CTI algorithm is feasible if, and only if, the normal fixed-priority algorithm is feasible.*

Proof. By the definition of the *CTI* algorithm and the assumption of the lemma, it is directly induced that applying only the normal fixed-priority algorithm is the best schedulable case and applying only the CTI table (i.e. deadlinewise preassignment) is the worst schedulable case with respect to a given periodic task set. In this point of view, feasibility of the hybrid algorithm for a given periodic task set adheres to the worst schedulable case at least. Therefore, by the above Lemma 1, we have proved the lemma. *

Note that the above Lemma 2 gives us that the *CTI* algorithm could be applicable for all the periodic task set schedulable using a fixed-priority algorithm. And hence, our algorithm also provides almost all of the fixed-priority algorithm's nice features, especially its high schedulable utilization for periodic tasks. Moreover, it provides stability and predictability for mixed scheduling.

3.5 Problem of Discrete Unit Time Scheduling

The CTI algorithm developed and simulated in this paper clearly depends on the unit time scheduling policy that each unit of periodic computation must be checked in the interval over which a periodic or an aperiodic task is active. For the tasks with a small number of computation units, it is not a big problem because the checking process involves only referring to the CTI table, and this takes extremely small computational overhead. However, for tasks with a relatively large number of computation units (e.g., 100 units) it may lead to a drastic degradation of the efficiency of the algorithm.

In this respect, we have reviewed some of the solutions to this problem. The best candidate among them is to slightly modify the algorithm as pertaining the following three basic rules:

Rule 1. For a critical task at a scheduling point that consists of two or more units of computations, do not interrupt its service procedure until all of its computation units are consumed.

Rule 2. For an aperiodic task allowed to be serviced at a scheduling point, calculate the sum of the available units (i.e., slacks) to the next critical task unit and allocate whole or part of its computation units. Then it is not necessary to check each unit of CTI table over the interval.

Rule 3. For a periodic task which is not critical at a scheduling point, service it until a new aperiodic task's arrival or a critical task's occurrence.

Rule 3 may require a somewhat different interrupt handling mechanism of the algorithm that deals with arrivals of aperiodic tasks.

3.6 Hard Aperiodic Task Scheduling

The purpose of this subsection is to provide a brief concept of an extension to the CTI algorithm so that it can manipulate hard deadline aperiodic tasks instead of ones with soft deadlines. Although a number of extension methods may be possible, the one using the CTI table is the most plausible candidate because it will definitely guarantee simplicity and good performance by reusing the original CTI table as a slack search domain.

Since a hard aperiodic task has its own deadline, an extension should provide an additional function to decide the acceptance of the task on its arrival. In other word, if the total number of slacks available from the current or the reserved point (for those already arrived and accepted before) to the deadline of the hard aperiodic task is less than its total computation requirements, then it must be rejected so that the others including periodic tasks can use the

```
#define N                       /* # of periodic tasks given to be scheduled */
#define H                       /* hyperperiod(least common multiple of all periods) */
int CTI[H-1];                   /* CTI table */
int CP[N];                      /* all the periodic computation processing done until now */
int CR[N];                      /* all the periodic computation requirement until now */
int t = 0;                      /* time counter */
boolean CT_occurred;            /* flag indicating critical task occurrence */

build(CTI);                                 /* build CTI table */
for (i = 1; i <= N; i++)                    /* initialize periodic computation counters */
  CP[i] = CR[i] = 0;
while (TRUE) {                              /* repeat forever */
  if As_arrived()                           /* if aperiodic tasks have arrived, */
    insert(new_As, A_queue);                /* insert these into the aperiodic queue. */
    if Ps_arrived() {                       /* if periodic tasks have arrived, */
        insert(new_Ps, P_queue);            /* insert these into the periodic queue. */
        adjust_element_order(P_queue);      /* sort queue elements based on fixed-priority */
    }
    CT_occurred = TRUE;                     /* assume a critical task has occurred */
    if (CTI[t] <> 0) {                      /* if CTI table does not indicate a slack, */
        if ((CP[CTI[t]] - CR[CTI[t]]) > 0)  /* if the periodic computation processing */
                                            /* done is greater than the requirement, */
            CT_occurred = FALSE;            /* the indicated task had already been processed */
        CR[CTI[t]]++;                       /* cumulate periodic comp. requirement */
    }
    if CT_occurred {                        /* if a critical task has occurred, */
        P = remove_any(CTI[t], P_queue);    /* get it from the queue (not on the front) */
        service(P);                         /* process it */
        CP[P.id]++;                         /* cumulate periodic comp. processing done */
    }
    else if (!empty(A_queue)) {             /* else, if an aperiodic task is ready, */
        A = remove(A_queue);                /* get it from the queue (on the front) */
        service(A);                         /* process it */
    }
    else if (!empty(P_queue)) {             /* else, if a periodic task is ready, */
        P = remove(P_queue);                /* get it from the queue (on the front) */
        service(P);                         /* process it */
        CP[P.id]++;                         /* cumulate periodic comp. processing done */
    }
    else cpu_idle();                        /* otherwise, CPU is idle */
    t++;                                    /* advance timer */
  if (t == H) {                             /* if a hyperperiod has finished, */
      t = 0;                                /* reset timer and */
      for (i = 1; i <= N; i++)              /* reinitialize periodic comp. counters */
          CR[i] = CP[i] = 0;
  }
}
```

Figure 5. CTI algorithm in C

slacks. To do this, the decision making mechanism has to search slacks through the CTI table. Therefore, another slack discriminant for hard aperiodic tasks that discriminates slacks at each search point from the CTI table is required.

The new slack discriminant will be as follows:

$$g(st) = \begin{cases} \text{slack} & \text{if } CTI[st] = 0 \\ & \text{or } CP[CTI[st]] - SCR[CTI[st]] > 0 \\ \text{critical task} & \text{otherwise} \end{cases}$$

It is based on the counter *CP* that was used in the first slack discriminant in subsection 3.3 and a new computation counter *SCR* representing ***all the computation requirement searched***. Also, it substitutes a search timer *st* for the scheduling timer *t*.

In addition, the problem of search space limitation caused by the hyperperiod bound must be carefully examined. Our proposed solution to this problem is to search the CTI table similarly to a circular list. It is only necessary to count the hyperperiodic distance from the current scheduling point to the current searching point. Another problem to be considered here is the arrival queue handling of hard aperiodic tasks. Thuel and Lehoczky [17] mentioned the problem that the selection of a proper priority level for hard aperiodic processing in their algorithm involves a tradeoff, and they indicated adequate heuristics as an optimal solution to this problem. From our viewpoint, this is a kind of management of a hard aperiodic arrival queue (prior to the acceptance decision making process) since every hard aperiodic task should be assigned a priority and queued depending on its degree of importance. The queuing theory approach would be desirable to handle the problem.

4. Simulation Results

In this section, we have prepared some of the aperiodic response time performances as the result of simulations on the four different aperiodic task scheduling algorithms including background processing, sporadic server, slack-stealing, and our CTI algorithm. Because of the similarity between the bandwidth preserving algorithms, only the sporadic server which is superior to others such as the deferrable server and priority exchange has been simulated as a representative.

The task sets applied to the simulations consist of 10 different periodic tasks, each of which has randomly generated period and computation requirements. All aperiodic tasks have been generated by using both an exponential distribution function for their computation requirements and Poisson arrival function for their arrivals at runtime. Aperiodic workloads could be easily coordinated by modifying the exponential scale parameter value and the arrival rate in Poisson function.

In order to provide a fairly subjective observation ground for the simulation, we have constructed a model of simulation in a very similar manner with that shown in the slack-stealing algorithm. Consequently, we have arranged three different periodic task sets with 40%, 70%, and 90% of CPU utilization ratios to simulate various workloaded real-time scheduling. These are summarized in Table 1. Note that the sizes of sporadic servers have been fixed to 50% and 20% of CPU utilizations respectively for the task sets with 40% and 70% of CPU utilization ratios because some of the periodic deadlines would be missed if the server sizes grew larger. In the following subsections, the results of aperiodic response time performances for each of the task sets is briefly analyzed and discussed.

4.1 Low Periodic Workload

Figure 6 illustrates the simulation results on the task set with 40% of periodic workload in terms of average aperiodic response time. Aperiodic workload is scaled from 0% to 60% of CPU utilizations. While background processing has significantly delayed average response time throughout all the aperiodic workloads, the others have similarly comparable response performances under relatively low aperiodic workload. However, the performance of sporadic server is remarkably downgraded due to its limited server capacity whenever the aperiodic workload goes over 40%. Moreover, our CTI algorithm outperforms slightly the optimal algorithm over almost all the aperiodic workloads, especially after 55%.

4.2 Medium Periodic Workload

Figure 7 illustrates the simulation results on the task set with 70% of periodic workload. Aperiodic workload is scaled from 0% to 30% of CPU utilizations. The simulation results go similarly with that of the above case with 40% of the periodic workload except that the slopes of curves steepen, i.e., performances decline surprisingly, according as the total workload approaches 100%.

	With 40% Periodic Workload		With 70% Periodic Workload		With 90% Periodic Workload	
Task ID	Period	Computation	Period	Computation	Period	Computation
1	33	2	100	2	100	2
2	105	7	280	8	280	14
3	21	3	2100	30	2100	108
4	60	4	440	29	440	29
5	55	4	350	14	350	14
6	70	1	210	8	210	30
7	22	3	35	3	35	8
8	315	10	70	4	70	11
9	180	12	2200	46	2200	231
10	540	23	300	9	300	12

Table 1. Sample periodic task sets used in the simulations

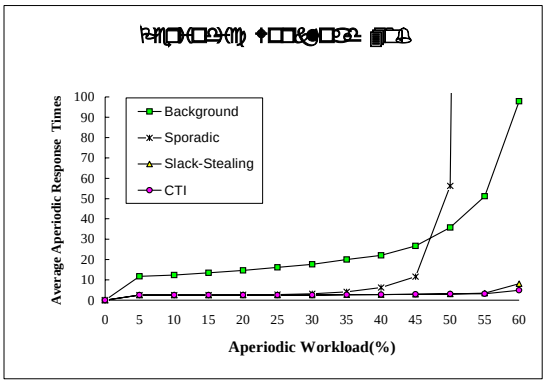

Figure 6. Simulation results on the task set with 40% of periodic workload.

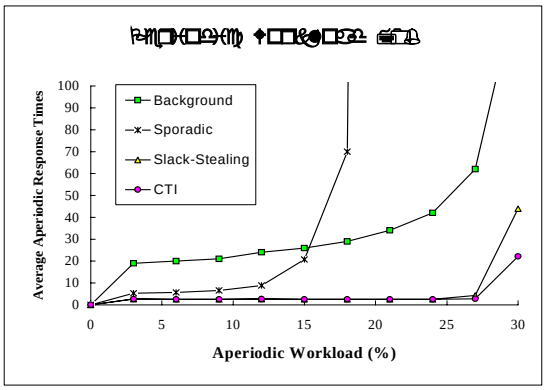

Figure 7. Simulation results on the task set with 70% of periodic workload.

4.3 High Periodic Workload

One of the important factors determining average aperiodic response time performance would be the periodic workload compared to other cases.

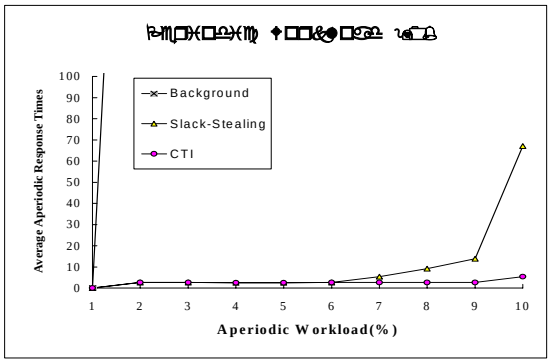

Figure 8. Simulation results on the task set with 90% of periodic workload.

In general, the performance of almost all the aperiodic scheduling algorithms goes down as the periodic workload goes up, in spite of having the same total workload. The fact is clear at a high total workload. A simple comparison of Figure 7 with Figure 8, presenting performances for the task set with 90% of periodic workload, shows it very well. Note that the sporadic server has been omitted because of its server capacity limit.

4.4 Overall Evaluation

One of the most important results from the simulations is that the CTI algorithm has outperformed all of the other joint scheduling algorithms without regard to the periodic workload ratios. Moreover, it

shows stable low average aperiodic response times although the total workload ratio approaches to 100%. The major reason is that the algorithm gives full flexibility to service aperiodic tasks as promptly as possible by delaying periodic tasks at their maximum.

5. Conclusions

This paper has presented the CTI algorithm for the joint scheduling of hard deadline periodic and soft deadline aperiodic tasks. The new algorithm assigns given tasks based on the hybrid scheduling method of a fixed-priority assignment and its deadlinewise preassignment. The deadlinewise preassignment for a periodic task set through a single hyperperiod produces a scheduling table, called CTI table, which is to be frequently referenced by the scheduler at runtime to check if there are slacks for aperiodic tasks. The major benefits of the CTI algorithm lie in its computational simplicity and predictability, since it uses the CTI table containing prescheduling information on the given periodic task set.

The simulation of the algorithm shows a performance as good as that of the slack-stealing algorithm in most cases and even better in the case of having a heavy transient overload. Moreover, the algorithm is reasonably simple to implement compared to the other joint scheduling algorithms and also addresses the scheduling predictability in some sense since it maintains the CTI table which has been built off-line.

However, the algorithm has some drawbacks in implementation environments. One is that the fixed-priority system on which the algorithm will work must be adaptable for the fixed-priorities. In other words, the system should allow temporary changes to the tasks' priority order on which each task's assigning order depends at runtime. This is due to the use of CTI tables formed from the deadlinewise preassignments that break up the original on-line fixed-priority assignment rule. Another shortcoming comes from the space complexity on the CTI tables for task sets with large numbers of periodic tasks.

Finally, our major ongoing and future works consist of 1) developing a hard aperiodic task scheduling algorithm using the properties of the CTI table, 2) relaxing the deadline constraints that currently limited us to the assumption (A1) given in subsection 2.3, and 3) taking advantage of the additional spare capacity produced by real schedulings because real tasks very rarely take their worst case execution times.

References

[1] H. Chetto and M. Chetto, "Some Results of the Earliest Deadline Scheduling Algorithm", IEEE Transactions on Software Engineering, vol. 15 (10), pp. 466-473, 1989.

[2] R. I. Davis, K. W. Tindell, and A. Burns, "Scheduling Slack Time in Fixed Priority Preemptive Systems", Proceedings of the IEEE Real-Time System Symposium, pp. 222-231, December 1993.

[3] M. Gonzalez Harbour, M. Klein, and J.P. Lehoczkey, "Fixed-priority Scheduling of Periodic Tasks with Varying Execution Priority", Proceedings of the IEEE Real-Time System Symposium, pp. 116-128, December 1991.

[4] E. D. Jensen, C. D. Locke, and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Operating Systems", Proceedings of the IEEE Real-Time System Symposium, pp. 112-122, December 1985.

[5] J.P. Lehoczky, L. Sha, and J.K. Strosnider, "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments", Proceedings of the IEEE Real-Time Systems Symposium, pp. 261-270, San Jose, CA, December 1987.

[6] J.P. Lehoczky and S. Ramos-Thuel, "An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems", Proceedings of the IEEE Real-Time Systems Symposium, pp. 110-123, December 1992.

[7] J.P. Lehoczky and S. R. Thuel, Scheduling Periodic and Aperiodic Tasks using the Slack Stealing Algorithm (Chapter 8), Advanced Real-Time Systems, (ed. S. Son) Prentice Hall, Englewood Cliffs, NJ, 1994.

[8] J.P. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior", Technical Report, Department of Statistics, Carnegie Mellon University, Pittsburgh, PA, 1987.

[9] J.Y.-T. Leung and J. Whitehaed, "On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks", Performance Evaluation, 2:253-250, 1982.

[10] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environments", Journal of the Association for Computing Machinery 20(1):46-61, January 1973.

[11] A.K. Mok, Fundamental Design Problems of Distributed Systems for the Hard Real-Time

Environment, Ph.D. Thesis, M.I.T., 1983.
[12] S.Ramos-Thuel and J.P. Lehoczky, "On-Line Scheduling of Hard Deadline Aperiodic Tasks in Fixed-priority Systems", Proceedings of the IEEE Real-Time System Symposium, pp. 160-171, December 1993.
[13] K. Schwan and H. Zhou, "Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads", IEEE Transactions on software engineering, vol. 18, No. 8, August 1992.
[14] B. Sprunt, J.P. Lehoczky, and L. Sha, "Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System", Technical Report CMU/SEI-890TR-11, April 1989.
[15] B. Sprunt, J.P. Lehoczky, and L. Sha, "Exploiting Unused Periodic Time for Aperiodic Service Using the Extended Priority Exchange Algorithm", Proceedings of the IEEE Real-Time System Symposium, pp. 251-258, December 1988.
[16] J.A. Stankovic and K. Ramamrithm, "What is Predictability for Real-Time System?", in the International Journal of Time-Critical Computing Systems Vol. 2, No. 4, pp. 247-254, November 1990.
[17] S. R. Thuel and J.P. Lehoczky, "Algorithms for Scheduling Hard-Aperiodic Tasks in Fixed-priority Systems using Slack Stealing", Proceedings of the IEEE Real-Time System Symposium, pp. 22-33, December 1994.
[18] T.-S. Tia, J. W.-S. Liu, and M. Shankar, "Algorithms and Optimality of Scheduling Aperiodic Requests in Fixed-Priority Preemptive Systems", Technical Report, University of Illinois, 1994.