Scheduling of Hard Aperiodic Tasks in Hybrid Static/Dynamic Priority Systems

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

Software Research Lab.[J]
Korea Telecom, Seoul, Korea
Department of Computer Engineering[JJ]
Kyunghee University, Seoul, Korea

Abstract

In this paper, we present a preemptive joint scheduling of hard deadline periodic and hard deadline aperiodic tasks on a uniprocessor real-time system. The scheduling has extended the Critical Task Indicating (CTI) algorithm [4] of which simulation study shows a considerable performance improvement over the other soft aperiodic task schedulings, especially under a heavy transient overload. Since a hard deadline aperiodic task has its own deadline, the proposed algorithm has a decision making mechanism that performs the on-line acceptance/rejection test upon its arrival. For simplicity and good performance, the algorithm reuses the original CTI table being used in the CTI algorithm as a slack search domain. Moreover, by searching the CTI table similarly to a circular list, it has removed the problem of search space limitation caused by the hyperperiod bound.

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] addressed that it is very important to develop an aperiodic task scheduler which works well over the transient overload and serves a fast response time for aperiodic tasks while guaranteeing the deadlines of periodic tasks in the unpredictable environment.

Over the last decade, the problem of jointly scheduling hard deadline periodic tasks and soft deadline aperiodic tasks using fixed-priority methods [5],[14],[15] has been investigated by many researchers in the real-time systems community. Recently, Lehoczky and Ramos-Theul [6] have developed a slack stealing algorithm which has proved to be optimal in the sense that it simultaneously minimizes the response time of all aperiodic tasks, provided they are 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 some drawbacks. Firstly, it requires a relatively large amount of calculation. Consequently, a direct implementation may not be practical. Secondly, it can not fully utilize all the available slacks for the largest amount of aperiodic processing due to the ceiling value of periodic processing requirements.

The slack stealing approach was generalized by Davis et. al. [2] to a wide class of scheduling problems. Also it has been improved by Tia et. al. [18] by means of a non-greedy approach while the slack stealing algorithm as defined in [6] is a greedy method. In [18], they showed that the total slack available during certain specific intervals can be larger than that available to the slack stealing algorithm if certain lower priority tasks are serviced before the available slack is used. They also pointed out that the greedy approach in [6] is not optimal when the class of algorithms is enlarged to include non-greedy algorithms.

A different type of approach to the algorithm, called the

Critical Task Indicating (CTI) algorithm, jointly scheduling hard deadline periodic and soft deadline aperiodic tasks was introduced by Lee, Kim, and Lee [4] (the authors). Goals of the algorithm are not only to guarantee the deadline of all periodic tasks and to get the response time for soft aperiodic tasks as small as possible, but also to achieve a considerable scheduling predictability. In order to achieve these goals, the authors adopted a hybrid manner of fixed-priority and dynamic-priority methods. By the fixed-priority methods, the spirit of a predictability can be achieved, and by the dynamic-priority methods using a deadlinewise preassignment reference table, the objective of the fast response time for the soft aperiodic tasks can be retained.

The deadlinewise preassignment for a periodic task set in a single hyperperiod produces a scheduling table, so called the critical task indicator (CTI table), which will be referenced by the scheduler to see if there are slacks available for aperiodic tasks at run time. The role of a CTI table is to indicate a critical task (if any) at each scheduling point that must be assigned and executed immediately to meet its deadline.

The key to the matter of the CTI algorithm converges to the creations of the required CTI tables in a priori. To work this out, the authors carefully developed a different kind of fixed-priority preassignment strategy, deadlinewise preassignment, for which a periodic task set defers each task's execution start time toward the deadline at its maximum. This idea comes from the fact that the value functions of a set of hard periodic tasks are generally step functions. Thus, the values of the functions are constant upto their deadlines [3] which means the execution time of tasks can be deferred till their deadlines if necessary. Consequently, the slacks which were made by artificially deferring the execution time can be utilized by the aperiodic tasks if they arrive at those time zones. Otherwise, the slacks can be used for the periodic tasks by the normal fixed-priority scheduling.

The result of the deadlinewise preassignment for a given periodic task set has scheduling information including the sequence of starting points of execution and the computation requirement of all task instances, on each instance whose execution is delayed at its maximum. This information gives us the valuable knowledge that a periodic task preassigned to a scheduling point would 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 a critical task at that time in a real time scheduling circumstance. The CTI algorithm, consequently, is reasonably simple to implement and offers an improvement to aperiodic response time over the slack stealing algorithm, especially under the transient overload.

Meanwhile, considerable research has been done in the area of the joint scheduling of hard deadline periodic tasks and hard deadline aperiodic tasks with respect to either fixed-priority or dynamic priority systems. Recently, Thuel and Lehoczky [7],[11],[17] have developed an extension of the slack stealing algorithm [6], which provides the largest amount of processing capacity for aperiodic tasks subject to guaranteeing the deadlines of the periodic tasks, in fixed-priority systems. The algorithm tests acceptance for hard aperiodic tasks for guaranteeing tasks at any priority level while it assumes that the periodic deadlines must all be met.

Work on the on-line scheduling of hard aperiodic tasks in dynamic priority methods has been reported by Chetto and Chetto[1], and Schwan and Zhou [12]. Their work assumes that all periodic tasks are scheduled according to the Earliest Deadline algorithm [10]. Especially a point to note is that Schwan and Zhou's algorithm does not give any preferential treatment to the periodic tasks, unlike common approaches to soft aperiodic tasks in fixed-priority preemptive systems. Every task is subject to an acceptance-rejection test upon arrival. The algorithm, however, may lead to an undesirable implementation overhead if the real-time workload is mainly periodic.

In this paper, we present an extended CTI algorithm for the jointly scheduling the hard deadline periodic tasks and hard deadline aperiodic tasks based on the CTI table. The tasks are scheduled in a way of mixed scheduling of a static and dynamic priority algorithm. The proposed algorithm is not only to guarantee all the deadlines of periodic tasks, but also performs an on-line acceptance test upon arrival of aperiodic tasks. The algorithm offers a less computational complexity than those of the other on-line schedulings of hard deadline aperiodic tasks in fixed-priority systems. Moreover, the algorithm demonstrates a remarkable scheduling predictability since it maintains the CTI table which has scheduling information and has been built off-line.

The remainder of this paper is organized as follows. The next section introduces the background of the CTI algorithm including general terms, notations, assumptions, and description of how to build a CTI table. Section 3 describes the CTI algorithm including a feasibility analysis. Section 4 discusses the extension of the CTI algorithm to maintain the accept-reject test for hard aperiodic tasks. Section 5 addresses some open issues and problems on the CTI approach. Finally, section 6 concludes the paper.

2. The background of the CTI Algorithm

In this section, we briefly review the background of the CTI algorithm including the basic notions, notations, and its major properties.

2.1 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 $\{\tau_1,\tau_2,...,\tau_n\}$, 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.

2.2 Assumptions

To develop the CTI scheduling 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.

2.3 Fixed-Priority Deadlinewise Preassignment Concepts

A periodic task scheduling method is classified to 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 similar manner of the other fixed-priority scheduling methods (e.g. rate monotonic priority assignment) except that the 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}, are arrived 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 has occurred.

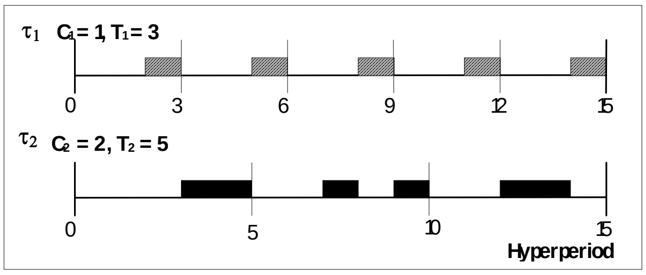

Figure 1. An example of the preassigning process using the deadlinewise preassignment.

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 to its completion. The period of a periodic task having the shortest waiting time from the request to the start of the execution contain 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 at 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. The proofs of the Theorems and Lemmas are shown in [4].

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.

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.

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.

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 would 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 mind.

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 the works are done 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 of which the fixed-priority order 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 to the processor to cope with the CPU idle state whenever there

```
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 (an aperiodic task(s) is ready or arrived) then service it
5          else if (a 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 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.

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 2.3. 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.

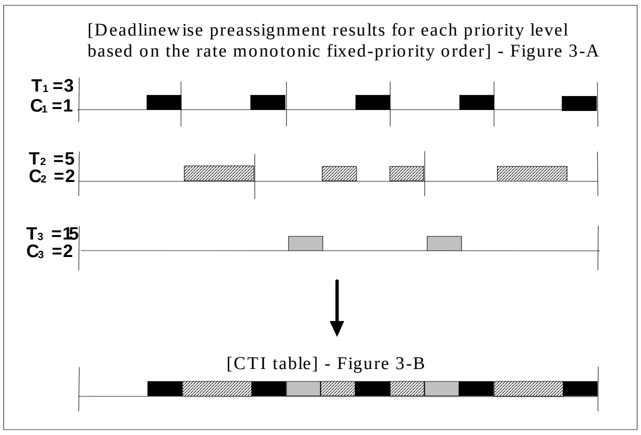
[Deadlinewise preassignment results for each priority level based on the rate monotonic fixed-priority order] - Figure 3-A
T₁ =3
C₁ =1
T₂ =5
C₂ =2
T₃ =15
C₃ =2
[CTI table] - Figure 3-B

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 occurred. The first aperiodic task A_1 has arrived at t=5 and 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.

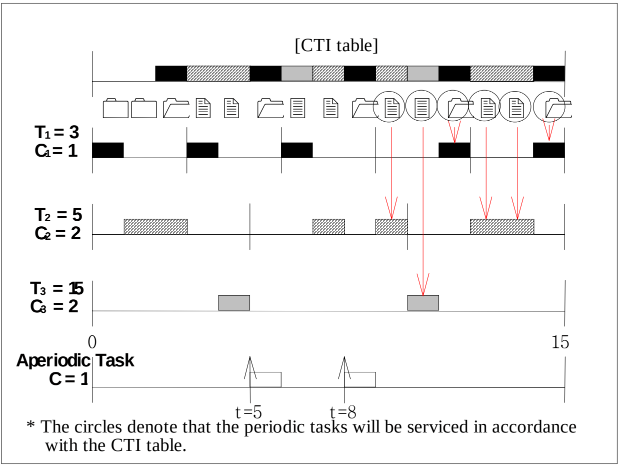
[CTI table]
T₁ = 3
C₁ = 1
T₂ = 5
C₂ = 2
T₃ = 15
C₃ = 2
0
15
Aperiodic Task
C = 1
t=5
t=8
* The circles denote that the periodic tasks will be serviced in accordance with the CTI table.

Figure 4. An example of the behavior of CTI algorithm

3.3 Feasibility Analysis

The major virtue of the fixed-priority assignment algorithms comprises of its simplicity and ability to handle many practical problems, stability under transient overload, low scheduling overhead, and so on [8],[9],[10]. 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.

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 algorithms' nice features, especially its high schedulable utilization for periodic tasks. Moreover, it provides stability and predictability for mixed scheduling.

4. The Accept-Reject Decision Algorithm

The purpose of this section is to extend the CTI algorithm to 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. In the following, we assume that all hard aperiodic tasks are known to their execution times and deadlines when they have arrived to the system.

4.1 The Slack Discriminants

In the CTI algorithm, two consecutive conditions (C1) and (C2) given in subsection 3.1 are 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 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 current scheduling time. Namely, the difference between 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 extend the above conditions to support a slack search mechanism for hard aperiodic tasks, first 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 first 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 $\boldsymbol{t}$.

Since a hard aperiodic task has its own deadline, a new version of the CTI algorithm 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 (for the arrived and accepted before) point 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 slacks. To do this, the decision making procedure has to search slacks through the CTI table. Therefore, another slack discriminant that discriminates slacks at each search point from the CTI table is required.

The second slack discriminant is as follows:

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

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

4.2 The Extended Algorithm

To develop an accept-reject decision making mechanism for hard aperiodic tasks, the original CTI algorithm dealing with soft aperiodic tasks should be specified in detail. For that purpose, we have prepared the original CTI algorithm and its extension written in C programming language style (see Figure 5 and 6). It mainly uses the data structures supporting the first and second slack discriminants given above. The rectangled areas (1), (2), and (3) in Figure 5 have been extended to the areas (1'), (2'), and (3') respectively in Figure 6. The routine (2') contains the slack search and accept-reject decision making components. Note that the integer counter *HD* has been introduced to handle the problem of search space limitation caused by the hyperperiod bound. For a detailed explanation, the comment lines in the figures will help.

4.3 Some Comments

The extended CTI algorithm leaves some aspects to be enhanced. One is its recovery procedure from a rejection of a hard aperiodic task. Currently, it restores the very previous state (returning back to the point at which the search began) after a rejection by backwarding the search steps done that caused the waste of CPU time twice. It is a minor problem and can be fixed simply by using additional temporary counters. Another to be considered is the arrival queue handling of hard aperiodic tasks. The extended CTI algorithm assumes that the hard periodic arrival queue is only in FIFO order. 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 indicated adequate heuristics as an optimal solution to this problem. At our viewpoint, it is a kind of hard aperiodic arrival queue (prior to the acceptance decision making process) handling problems, since every hard aperiodic task should be assigned a priority and queued depending on its degree of importance. The queueing theory approach would be desirable to handle the problem.

5. Discussion

In this section, we discuss some open issues and problems on the CTI approach.

5.1 Major Differences with the Conventional Reverse Schedulings

Under the strong assumption (A1) given in the subsection 2.2, a deadlinewise preassignment for a periodic task set according to their fixed-priority basis can also be obtained by rotating the normal fixed-priority preassignment in a 180° on the axis of the beginning or the ending point of the hyperperiod. Because of this transposed property for a special case, the CTI approach may be considered as the same reverse schedule introduced by Chetto and Chetto[1], and Shih and Liu [13]. Some of the properties such as laying out the schedule in reverse time order to determine the maximum delay in executing periodic tasks without causing their deadlines to be missed has already been shown in [1] and [13].

We, however, would like to point out the major differences between the pervious works and our approach in terms of application scope (constraint) and pursuing goal. In the light of application scope, our approach can fully be used to relax the constraint of Chetto and Chetto's scheduling method. While the reverse schedule of EDL (Earliest Deadline as Late as possible) proposed by the authors of [1] is restricted to the fact that the deadlines

```
#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;                   ...(1)
while (TRUE) {                                            /* repeat forever */
    if As_arrived()                                       /* if aperiodic tasks have arrived, */
        insert(new_As, A_queue);       ...(2)             /* 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++)       ...(3)             /* reinitialize periodic comp. counters */
            CR[i] = CP[i] = 0;
   }
}
```

Figure 5. Critical Task Indicating Algorithm in C (Soft Aperiodic Version)

```
int SCR[N];                  /* all the periodic computation requirement searched */
int S;                       /* # of slacks searched */
int HD = 0;                  /* distance between t and st in the unit of hyperperiod */
int st = 0;                  /* search timer */

for (i=1; i <= N; i++)                                  …(1')   /* initialize periodic comp. counters */
    CR[i] = CP[i] = SCR[i] = 0;

while (As_exist()) {                                                /* while there exist aperiodic tasks arrived...*/
    if ((HD == 0) && (st < t)) st = t;                              /* if t and st lies on the same hyperperiod...*/
    if (HD < 0) {                                                   /* if t outdistances st in one or more hyperperiod...*/
        HD = 0;                                                     /* reinit. the hyperperiodic distance value */
        for (i = 1; i <= N; i++) SCR[i] = 0;                        /* reinit. all the periodic comp. requirement...*/
        st = t;                                                     /* move search start point to the current... */
    }
    S = 0;                                                      /* initialize searched slack counter */
    X = st + new_A.D;                                           /* set up search deadline */
    while ((st < X) && (S < new_A.C)) {                             /* while (st < deadline) and (total slacks */
                                                                    /* found < new aperiodic comp. requirement)*/
        if (CTI[st%H] == 0) S++;                                    /* if CTI table indicates a slack, cumulate it */
        else {                                                  /* otherwise, i.e., if CTI table indicates a task, */
            if (SCR[CTI[st%H]] < CR[CTI[st%H]])                     /* backed up the comp. counter to... */
                SCR[CTI[st%H]] = CR[CTI[st%H]];
            if ((CP[CTI[st%H]] - SCR[CTI[st%H]])> 0)                /* if the periodic task had already been */
                S++;                                      …(2')     /* serviced, it is a slack */
            SCR[CTI[st%H]]++;                                       /* increase searched comp. requirement */
        }
        st++;                                                       /* increase search timer */
        if ((st%H) == 0) HD++;                                      /* if st gets to a hyperperiod bound, ... */
    }
    if (S == new_A.C) {                                             /* if the required slacks found, */
        insert(new_A, A_queue);                                     /* accept the aperiodic task */
        st = st % H;                                                /* set up new start point for next search */
    }
    else {                                                          /* otherwise, */
        reject(new_A);                                          /* reject the aperiodic task requested */
        for (i = 0; i < new_A.D; i++) {                             /* restore the previous state */
            if ((st%H) == 0) HD--;
            st--;
            if (CTI[st%H] != 0) SCR[CTI[st%H]--;
        }
    }
}

if (t == H) {                                                   /* if t gets to a hyperperiod bound, */
    t = 0;                                                      /* reinitialize it and */
    if (HD > 0)                                                     /* if the hyperperiodic distance is positive */
        for (i=1; i <= N; i++)                                  /* take out comp. requirement from the */
            SCR[i] -= CR[i];                            …(3')   /* corresp. searched comp. requirement */
    HD--;                                                       /* decrease hyperperiodic distance */
    for (i=1; i <= N; i++)                                          /* reinit. periodic comp. counters */
        CR[i] = CP[i] = 0;
}
```

Figure 6. CTI Accept-Reject Decision Algorithm in C of periodic tasks must be same as the periods of periodic tasks, an extension of our approach will remove the limitation in a way to build a CTI table using deadlinewise preassignment according to the EDF (Earliest Deadline First) priority order. By using this deadlinewise preassignment, our CTI approach does not need to reverse the scheduling domain, so that the deadlines of the periodic tasks are not necessarily to be same as the periods of those periodic tasks. Further, Chetto and Chetto have introduced the EDL scheme as an acceptance test mechanism for hard aperiodic tasks while we have suggested the CTI approach as a mechanism for mainly reducing computational complexity in calculating the slacks.

5.2 Priority Inversion

In the course of the scheduling by using the CTI algorithm, we may be faced with the priority inversion problem. When there is no aperiodic task request, the periodic tasks are scheduled according to their fixed-priorities, i.e., by the normal fixed-priority scheduling scheme. But when there are aperiodic requests, the original fixed-priority assignment order of the periodic tasks may be inverted. This priority inversion phenomenon seems to hurt the merits of the fixed-priority scheduling method. We, however, have to carefully examine the tradeoffs between the advantages of the fixed-priority scheme and the expenses of the heavy computational complexity to find slacks in the joint scheduling. Since the cost of calculating the available slacks for aperiodic tasks in the fixed-priority system is very expensive, we must consider an alternative scheduling mechanism which may violate the fixed-priority order, but is remarkably simple in calculating the slacks. For example, while the complexity of computing all the slack values in the static slack stealing algorithm is $O(n^2)$, that of the soft aperiodic CTI algorithm is O(1) because of its use of the one or two step slack discriminant. In this respect, we have adopted the CTI algorithm which gains huge benefits from using simple slack calculation method at the expenses of the merit of fixed-priority scheduling. It also meets the goals of joint scheduling which are not only to guarantee all the deadlines of periodic tasks and to obtain the fast response time for aperiodic tasks, but also to get the implementation simplicity and the scheduling predictability.

5.3 Discrete Unit Time Scheduling

The CTI algorithm clearly depends on the unit time scheduling policy that causing 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 is involved only on referring to the CTI table that taking extremely small computational overheads. However, for the tasks with relatively large number of computation units it may lead to a drastic degradation of the efficiency of the algorithm.

The best solution 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 out.

Rule 2. For an aperiodic task allowed to be serviced at a scheduling point, calculate 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.

5.4 Other Issues

Since tasks rarely take worst case execution times, a pre-allocation of processor utilization based on these execution times may result in an undesirable waste of processing time. Some of the previous approaches [2],[6],[15] have taken advantage of the additional spare capacity, so called gain time, produced by pre-allocated tasks to improve average system performance. While Lehoczky and Ramos-Thuel [6] presented an easy way of reclaiming gain time for their static slack stealing algorithm, Davis et. al. [2] mentioned three major methods of identifying gain time to provide a less pessimistic worst case execution time for a task in addition to reclaiming gain time to their dynamic slack stealing algorithm.

Unfortunately, the currently running version of the CTI algorithm does not provide the ability to reclaim gain times of tasks because of its time-driven scheduling property (i.e., discrete unit time scheduling policy). However, an extension of the algorithm that may successfully reflect the three basic rules suggested in subsection 5.3 can easily be modified to efficiently reclaim gain time. An expected reclaiming mechanism would be simple enough to just add gain time to the total sum of the available slacks found at a scheduling point in a similar manner of the static slack stealing reclaimer.

On the other hand, the proposed algorithm somewhat suffers from misindicating of a non-critical task as a critical one which may lead to the result that the algorithm can not find the available slacks at the scheduling point unless otherwise. Let us consider two periodic task T_1 and T_2, with T_1 having the higher priority. Suppose there is no aperiodic request initially. The first instant of T_1 finishes executing. An aperiodic task comes. Now, according to the CTI table, T_2 may now be critical because the deadlinewise preassignment scheduled is constructed with T_2 first, followed by T_1. But T_1 has already finished executing, and the CTI algorithm may not detect this. So instead of pushing T_2 back and schedule the aperiodic request, the scheduler schedules T_2 instead, thinking that it is critical when in fact it is not critical because T_1 has finished and there may be more time later to execute T_2.

This phenomenon indicates the proposed algorithm may not be optimal. This is, however, obviously not an usual case. Furthermore, our simulation study showed the misindicating problem never affect to the overall performance significantly. Rather, in many cases, we found that the CTI algorithm can probabilistically find more slacks for aperiodic tasks compared to the static slack stealing algorithm. An exact example and simulation study on this matter will be shown on another work that is now under preparing by the authors.

6. Summary

This paper discusses the problem of jointly scheduling hard deadline periodic and hard deadline aperiodic tasks in hybrid static/dynamic priority systems. The paper develops a hard aperiodic tasks' acceptance test algorithm based on the CTI algorithm. The deadlinewise preassignment for a periodic task set in a single hyperperiod produces a scheduling table, called CTI table, which is frequently referenced by the scheduler whether there are slacks for hard aperiodic tasks. The CTI algorithm shows a good performance as much as the slack stealing algorithm in most cases and even better than the case of a transient overload.

The proposed algorithm using the CTI table, consequently, keeps the benefits of the CTI algorithm as well as performs well to test acceptance-rejection for hard aperiodic tasks at run time. Moreover, it offers remarkable scheduling predictability since the algorithm refers the CTI table which has considerable approximated scheduling information and has been built off-line.

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] 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.

[4] J. Lee, H. Kim, and S. Lee, "Scheduling Soft-Aperiodic Tasks in Adaptable Fixed-Priority Systems", Submitted for publication.

[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), Advances in 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] 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.

[12] 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.

[13] W.-K. Shih and J.W.S. Liu, "On-line Scheduling of Imprecise Computations to Minimize Error", Proceedings of the IEEE Real-Time System Symposium, pp. 280-290, December 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?", 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.