Scheduling Aperiodic and Sporadic Tasks in Hard Real-Time Systems *

Seonho Choi Ashok K. Agrawala
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, MD 20742
{seonho,agrawala}@cs.umd.edu

May 7, 1997

Abstract

The stringent timing constraints as well as the functional correctness are essential requirements of hard real-time systems. In such systems, scheduling plays a very important role in satisfying these constraints. The priority based scheduling schemes have been used commonly due to the simplicity of the scheduling algorithm. However, in the presence of task interdependencies and complex timing constraints, such scheduling schemes may not be appropriate due to the lack of an efficient mechanism to schedule them and to carry out the schedulability analysis. In contrast, the time based scheduling scheme may be used to schedule a set of tasks with greater degree of schedulability achieved at a cost of higher complexity of off-line scheduling. One of the drawbacks of currently available scheduling schemes, however, is known to be their inflexibility in dynamic environments where dynamic processes exist, such as aperiodic and sporadic processes. We develop and analyze scheduling schemes which efficiently provide the flexibility required in real-time systems for scheduling processes arriving dynamically. This enables static hard periodic processes and dynamic processes(aperiodic or sporadic) to be jointly scheduled.

*This work is supported in part by ONR and ARPA under contract N66001-95-C-8619 to the Computer Science Department at the University of Maryland. The views, opinions, and/or findings contained in this report are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency, ONR or the U.S. Government.

1 Introduction

In this paper we develop an approach to addressing the problem of *incremental* scheduling of tasks in a hard real-time system.

Real-time systems are characterized by the timing constraints on the execution of tasks. Usually, the real-time computer systems are used to control or interact with a physical system and the timing constraints on the execution of task instances are imposed by the requirements of the physical system. In a hard real-time system, failure to conform to any timing constraint is considered a catastrophic failure.

Clearly, in order to ascertain that the timing constraints will be satisfied, it is essential that the resource requirements for task instances be known and made available in a timely manner. When, for a hard real-time system, an absolute guarantee is required, it is necessary that the resources for the worst case execution be reserved.

Traditionally, the scheduling problem considered for real-time systems is that of generating a schedule for n tasks. In practice, however, a system may have to accept additional tasks during its operation. Here, we study the problem of *incremental scheduling* in *dynamic* time-based environment. We assume that we are given a set of n tasks, $\mathcal{T}$(and all their task instances), along with a schedule for their execution. We consider adding a task to the schedule. To add a new task, we have to first analyze the acceptability of it. If this task can not be scheduled without violating constraints of any of the tasks in $\mathcal{T}$ then this task is not accepted. If this can be scheduled, we not only accept the task, but also add it to the schedule.

One approach to real-time scheduling is to assign priorities to tasks statically or dynamically and use those priorities to schedule the tasks at runtime. The schedulability tests can be carried out with little overhead. But, this approach does not provide the capability to schedule the task set with complex timing constraints, such as complicated precedence relations, jitter constraints, relative timing constraints, etc.

The time-based scheduling approach can schedule any set of tasks with complex timing constraints. Traditionally, this is done by statically allocating resources(time intervals) to the task instances at pre-runtime. Recently, much research effort has been devoted to reduce the complexity of the off-line scheduling algorithms and some systems have been successfully implemented using this approach [24, 29, 31, 14, 30]. In this proposal, we develop a time-based scheduling scheme in which the scheduling of resource usages may be done dynamically at run-time and we employ this approach to incrementally schedule arriving tasks.

In Section 2 the incremental scheduling problem is formally defined within a time-based scheduling scheme. The related work is presented in Section 3 and the results on incremental scheduling of aperiodic and sporadic tasks are presented in Section 4. Finally, a conclusion follows in Section 5. The appendix contains the proof of the main theorem in Section 4.2.

2 Problem Description

The main problem which is to be addressed in this paper is about how to incrementally accept and schedule tasks while not sacrificing the schedulability of the tasks already accepted.

A task in a real-time system may invoke its corresponding *task instances* by informing the system of the release time, deadline, and execution time of the task instance. Tasks in real-time systems

may be classified into *single instance* task and *multiple instance* task. Single instance task, which is also called *aperiodic* task, invokes its task instance only once, and multiple instance task invokes its instance repeatedly. Multiple instance tasks are further divided into *periodic* tasks and *sporadic* tasks. Periodic task invokes its instances at regular time intervals(period), whereas sporadic task invokes its instances at any time instants with a defined minimum inter-arrival time between two consecutive invocations.

Any arriving task belongs to one of these classes. A periodic task P is characterized by an invocation of a sequence of task instances. The following characteristics are assumed to be known at the arrival time, $\mathcal{A}^p$, of the periodic task, P.

- task invocation time $\mathcal{I}^p$ from which the task starts to invoke its instances.
- task termination time $\mathcal{X}^p$ when the task is terminated.
- period p
- invocation time of the j-th task instance is defined to be $\mathcal{I}^p_j = \mathcal{I}^p + (j-1)p$
- relative deadline d^p which implies that the absolute deadline of j-th task instance is $\mathcal{I}^p_j + d^p$.
- worst case execution time c^p

A hard aperiodic task A invokes its task instance only once. A has the following set of parameters:

- arrival time of the request, $\mathcal{A}^a$
- ready time $\mathcal{R}^a$ from which the task instance can start its execution.
- relative deadline d^a which implies that the absolute deadline is $D^a = \mathcal{R}^a + d^a$
- worst case execution time c^a

A sporadic task S is characterized by an invocation of a sequence of task instances with the minimum inter-arrival time apart. The following characteristics are assumed to be known at the arrival time, $\mathcal{A}^s$, of the sporadic task, S.

- task invocation time $\mathcal{I}^s$ from which the task instances can be invoked.
- task termination time $\mathcal{X}^s$ when the task is terminated.
- minimum inter-arrival time δ
- invocation time of the j-th task instance, $\mathcal{I}^s_j$, can be any time instant satisfying $\mathcal{I}^s_j \geq \mathcal{I}^s_{j-1} + \delta$
- relative deadline d^s $(\leq \delta)$ which implies that the absolute deadline of the j-th task instance is $\mathcal{I}^s_j + d^s$.
- worst case execution time c^s

In addition to these, system may be called upon to handle non-realtime tasks which don't have deadlines; Instead, they require as fast completion time as possible.

For a set of task instances to be scheduled, traditional time-based scheduling scheme first finds a complete schedule for them in a given scheduling window. This schedule contains a static start time, s_i, for each task instance, which is decided based on the worst case execution time c_i and reflects all task dependencies. However, to enhance the scheduler with the ability to schedule dynamically arriving tasks, it may change s_i at runtime, while conforming to all constraints, such as release time r_i, deadline d_i, precedence relations, relative constraints, etc. Clearly, this additional information has to be kept for each task instance with the schedule. If a new task arrives, based on the current schedule it needs to be decided whether this new task can be accepted by the system, and if it can be accepted, a new schedule has to be constructed to incorporate this new task.

In hard real-time environment, tasks may be executed in preemptive or non-preemptive manner. When a task is executed non-preemptively it begins execution at time s_i and is assured CPU access for the time, c_i, without any interruption or preemption. In preemptive execution, the task execution may be preempted at some defined time instant, and resumed at a later time instant. Note that the task preemption and resumption times may be dynamically decided.

We extend the static time-based scheduling scheme into dynamic time-based scheduling scheme that enables any dynamically arriving aperiodic, periodic, or sporadic task to be incrementally scheduled. In traditional static time-based scheduling scheme, every resource requirement is met by assigning explicit start times to the task instances. But, in this dynamic time-based scheduling scheme, the start times no longer have to be statically determined. Instead, the schedule includes a mechanism for determining the time when a task instance will be started or resumed based on the information available prior to its start time.

3 Related Work

3.1 Scheduling

The scheduling algorithms in hard real-time systems may be classified in several ways. One way is to classify them in terms of how the scheduling is done. Priority-based scheduling schemes resolve the resource(CPU) contention between different tasks by making use of the fixed or dynamic priorities of the tasks. Another scheduling approach is a time-based scheduling scheme in which the explicit time line is used to schedule the tasks. In traditional time-based scheduling schemes, all resource requirements are satisfied by statically assigning time intervals to the task instances at pre-runtime.

Each approach has its own advantages and disadvantages. Even though scheduling in priority based approach can be done in a simple manner, it lacks the capability of scheduling tasks with complex constraints such as precedence relations, relative timing constraints, while the time-based approaches have that capability. In this paper, we develop a dynamic time-based scheduling scheme to provide the flexibility commonly required in dynamic real-time systems, i.e. incremental scheduling of dynamic tasks, such as aperiodic, periodic, and sporadic tasks. In this dynamic scheme, the actual execution times of the tasks may be decided at run-time for example by constructing parameterized start or resume times of the tasks.

3.1.1 Fixed priority scheduling

In this scheme, fixed priority is assigned to each task and it is used at runtime to resolve the resource conflicts. A task with a higher priority can preempt any lower priority task and thus the currently executing task has the highest priority among the tasks currently active(released). It is well known that rate monotonic scheduling algorithm is optimal for scheduling a set of independent periodic tasks with deadlines at the end of their periods [21]. It is optimal in a sense that it can schedule any set of tasks if that is schedulable by any fixed priority scheduling scheme. Any set of n tasks are schedulable according to rate monotonic scheduling scheme if the total utilization of the tasks doesn't exceed $n(2^{1/n} - 1)$ which converges to $ln(2) \cong 0.69314718$ as n goes to ∞. This is a sufficient condition for a given set of tasks and not a necessary condition. The exact schedulability condition which is necessary and sufficient is found in [17] with the statistical simulation results showing that in general the utilization of the schedulable task set is higher than $ln(2)$.

A deadline monotonic scheduling algorithm is shown to be optimal for a set of tasks which have deadlines less than or equal to their periods. It assigns priorities according to their deadlines, the shorter the deadline, the higher priority is assigned regardless of their periods [20, 2]. For a set of tasks with arbitrary deadlines, it is shown that the optimal priority assignment can't be done in a simple priority assignment method, but requires a pseudo polynomial time algorithm [28].

A synchronization protocol becomes necessary when tasks use shared resources such as semaphores. Sharing resources may lead to a possible *priority inversion* when a higher priority task is blocked due to the lower priority task possessing the required resource by a higher priority task. This priority inversion may cause an unbounded blocking times. To solve this problem, several synchronization protocols have been developed. In a priority ceiling protocol [23], a priority ceiling is first assigned to each semaphore, which is equal to the highest priority of the tasks that may use this semaphore. Then, a task, τ, can start a new critical section only if τ's priority is higher than all priority ceilings of all the semaphores locked by tasks other than τ. In stack-based protocol [3], the concept of *preemption level* is used instead of the priorities to derive the protocol suitable for both fixed priority and dynamic priority based systems. Also, sharing of multiple-unit resources becomes possible with this protocol.

Hard and non-realtime aperiodic tasks can be scheduled within a fixed priority scheduling scheme. One approach is to utilize the aperiodic server concept in which a certain percentage of the processor utilization is reserved for servicing the aperiodic tasks. Several algorithms have been developed and their performances have been compared [18, 25]. Another approach is slack stealing approach which tries to utilize as much processor time as possible by postponing the execution of hard periodic task executions as long as the schedulability of the hard tasks is not affected [10, 19, 27]. The optimal slack stealing algorithm is found to be pseudo polynomial [10] and several approximation algorithms have been devised [9].

3.1.2 Dynamic Priority Scheduling

The priorities of tasks in dynamic priority scheme are decided at runtime. This means that the task instances from the same task may have different priorities at runtime while in the fixed priority scheme the same priority is used for scheduling them. The *earliest deadline first*(EDF) scheduling algorithm which assigns the highest priority to a task instance with the closest deadline is known to

be optimal for a set of periodic or aperiodic tasks [21, 11]. The necessary and sufficient schedulability condition for a set of independent tasks with their deadlines equal to their periods is that the total processor utilization of the tasks should be less than or equal to 1 [21]. A dynamic priority ceiling protocol [4] and a stack-based protocol [3] have been developed for dynamic priority systems to enable the use of shared resources and to bound the blocking times. Note that the stack based resource allocation protocol may be used for both fixed priority and dynamic priority scheduling algorithms. Also, in [3], it is shown that the stack-based protocol provides a better schedulability test than that of dynamic priority ceiling protocol.

An aperiodic task scheduling problem has been studied under the assumption that only hard periodic tasks exist [16, 15]. The $O(N)$ acceptance test for a hard aperiodic task is given when a set of independent periodic tasks is scheduled by EDF where N is the total number of task instances in an LCM[1] of the periods of periodic tasks [7, 6, 8]. Aperiodic scheduling schemes for EDF have been proposed and studied and the Improved Priority Exchange Algorithm is shown to perform well [26].

3.1.3 Static Time-based Scheduling

In a static time-based scheduling scheme, a calendar for a set of task instances is constructed at pre-runtime. At runtime this calendar is referred to execute each task instance at a scheduled time instant. Through off-line scheduling, we can schedule any set of tasks with various constraints, such as complex precedence relation, relative timing constraints, and other synchronization constraints. Even though the complexity of the off-line scheduling is NP-Complete in general, the scheduling can be done in a reasonable amount of time in most cases using techniques such as branch and bound or heuristic search algorithms [29, 12, 5, 32]. It has been shown that the complexity of non-preemptive scheduling can be dramatically reduced in many cases by *decomposition scheduling* approach where task instances are decomposed into a sequence of subsets, which are scheduled independently [31]. Also, the time based scheduling scheme can efficiently schedule task sets with relative timing constraints which can't be easily accommodated in priority-based systems [14, 5]. Because of these reasons, it is claimed that the time-based scheduling scheme is the most appropriate scheduling approach for *hard* real-time systems [30].

The aperiodic scheduling problem in time-based scheduling scheme has been addressed in the paper [13]. The initial schedule is assumed to be given and arriving aperiodic tasks are scheduled at runtime. First, the deadlines of task instances, τ_j, in the time-based schedule are sorted and the schedule is divided into a set of *disjoint execution intervals*, I_i. Then, the *spare capacities* are defined for these intervals, which may be used to schedule arriving aperiodic tasks. Several tasks with the same deadline constitute one interval and the end of an interval, $end(I_i)$, is defined to be the deadline of the last task in the interval. The earliest start time of an interval is defined to be the minimum of the earliest start times of its constituting tasks. And, the start time of the interval, $start(I_i)$ is defined to be the maximum of its earliest start time or the end of the previous interval. The length of an interval, $\mid I_i \mid$ is defined to be $end(I_i) - start(I_i)$. Then, the spare capacity is defined recursively as:

$$sc(I_i) \quad = \quad \mid I_i \mid - \sum_{\tau_j \in I_i} C_j + \min(sc(I_{i+1}), 0)$$

[1] An LCM is a least common multiple of the periods.

$$sc(I_e) \quad = \quad | I_e | - \sum_{\tau_j \in I_e} C_j$$

where C_j denote the worst case execution time of τ_j and I_e is the last interval in the schedule. Note that the spare capacity may have a negative value reflecting the fact that the borrowed spare capacity from the previous interval is used to schedule the task instances in the current interval. Figure 1 shows an example case of this. In this example, the spare capacities for I_2 and I_1 are found to be:

$$sc(I_2) = 2 - 3 = -1$$

$$sc(I_1) = 8 - 3 + \min(-1, 0) = 4$$

These spare capacities are used to schedule arriving aperiodic tasks and adjusted whenever the aperiodic tasks are accepted.

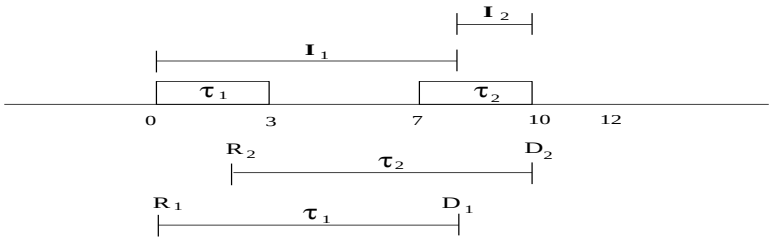

Figure 1: Example case.

However, no consideration is given how to obtain correct spare capacities when the deadlines of the task instances are not in increasing order in the schedule. For example, no correct spare capacity can be obtained in the example case shown in Figure 2.

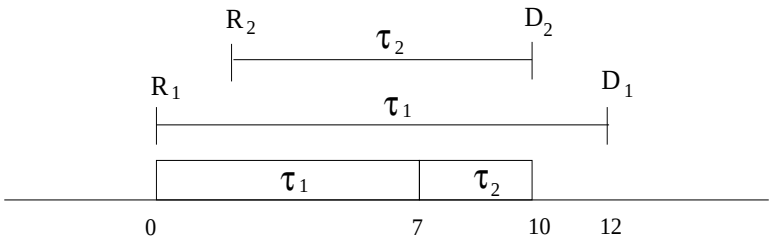

Figure 2: No spare capacities can be found.

According to the algorithm presented in that paper, $[0, 12]$ is an execution interval of spare capacity 2, which is not correct.

Moreover, the spare capacities of the task instances have to be adjusted every time a new task is accepted and scheduled, which introduces more overhead.

4 Dynamic Time-based Scheduling Schemes

Two dynamic time-based scheduling schemes are presented here. In Section 4.1, a mechanism is presented to incrementally schedule periodic and aperiodic tasks. In Section 4.2, a mechanism is

presented to incrementally schedule periodic and sporadic tasks. In both sections, it is assumed that a valid schedule of task instances is initially given with start times of the task instances. Based on this schedule, we develop an efficient scheduling algorithm to incrementally schedule arriving aperiodic task instances with the schedulability of the existing task instances not affected. And then, another scheduling mechanism is presented in the following section to incrementally schedule dynamically arriving periodic or sporadic tasks while not affecting the schedulability of the already accepted periodic and sporadic tasks.

4.1 Aperiodic Task Scheduling

In this section, a mechanism is presented to schedule arriving aperiodic tasks. The key idea of this mechanism is to make use of the fact that the task executions may be dynamically shifted to the left or to the right in a time line as long as the timing constraints of the tasks can be satisfied. All task instances in this section are assumed to be preemptible.

4.1.1 Task Model

We assume that an initial schedule of task instances is given in a scheduling window $[0, L]$ and this schedule is used by dispatcher at run-time. Let $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_N\}$ be a set of task instances in the initial schedule. It is assumed that τ_i is scheduled before τ_{i+1} in the schedule. Each task instance τ_i has the following parameters in the schedule:

- release time R_i
- absolute deadline D_i ($D_i \leq L$ for all $1 \leq i \leq N$)
- worst case execution time C_i
- runtime variable e_i denoting the processing time already spent for τ_i at any time instant
- runtime variable ω_i denoting the latest start(or resume) time of τ_i, which is a function of the current time t and the value of e_i
- earliest start time $est(i)$
- latest start time $lst(i)$

A hard aperiodic task A is defined the same way as in Section 2 except that the ready time is assumed to be equal to its arrival time, i.e, $\mathcal{A}^a = R^a$. Also, the task instances in $\mathcal{T}$ are assumed to be *preemptible* by an aperiodic task and any aperiodic task is assumed to be *preemptible* by a task instance in $\mathcal{T}$.

The values of $est(i)$, $lst(i)$, $i = 1, 2, \ldots, N$, are found as follows:

$$
\begin{aligned}
est(1) &= R_1 \\
est(i) &= \max(R_i, est(i-1) + C_i) \quad \text{for } i = 2, 3, \ldots, N \\
lst(N) &= D_N - C_N \\
lst(i) &= \min(D_i, lst(i+1)) - C_i \quad \text{for } i = N-1, N-2, \ldots, 1
\end{aligned}
$$

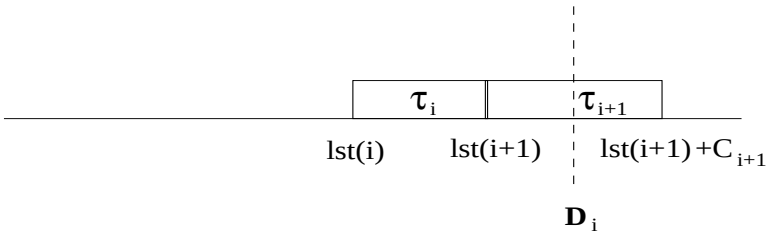

Figure 3: Deriving $\omega_i(0)$ recursively

If $D_i \leq lst(i+1)$, then $lst(i)$ value will be decided from D_i. And if $D_i > lst(i+1)$, then $lst(i)$ will be decided from $lst(i+1)$. Fig 3 shows an example of these relationships.

Also, Fig 4 shows an example set of task instances with their $est(i)$ and $lst(i)$.

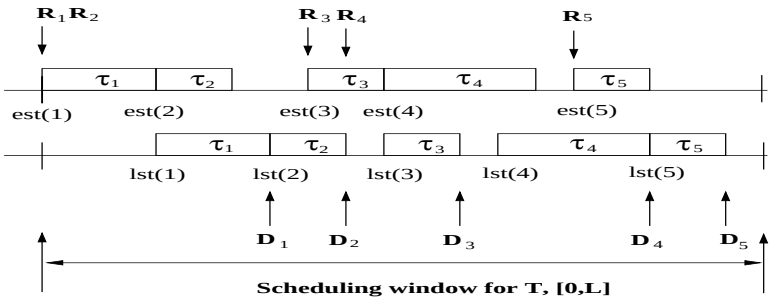

Figure 4: $est(i)$ and $lst(i)$ for an example task set

Note that the run-time variable e_i is initialized to 0 and ω_i to $lst(i)$.

$\mathcal{T}$ and a set of arriving aperiodic tasks A_1, ..., A_i are said to be *feasible* if and only if there exists a schedule which satisfies all the timing constraints on $\mathcal{T}$ and aperiodic tasks. The *optimality* of a scheduling algorithm is defined as:

Definition 1 (Optimality) *A scheduling algorithm is* optimal *if and only if it can schedule any feasible task instance set $\mathcal{T}$ and arriving aperiodic tasks.*

4.1.2 Scheduling of Non-realtime Tasks

We can efficiently schedule any non-realtime tasks in a sense that maximum processor time can be found and used to service non-realtime tasks at any time instant by delaying as much as possible the executions of task instances already accepted. The non-realtime tasks are assumed to be processed by using FIFO scheduling policy.

At a current time instant t_1, let τ_j denote a task instance in $\mathcal{T}$ which is just finished or partially executed. Also, let t_0 denote the last time instant when the dispatcher took the control before t_1, and let t_2 denote the run-time variable denoting the future time instant when the dispatcher should take the control. The dispatcher takes the control whenever a non-realtime task or a task instance in $\mathcal{T}$ is finished, or whenever $t_1 = t_2$ holds. Then, at a current time instant t_1 when a dispatcher takes the control:

If τ_j is executed in $[t_0, t_1]$
  then
    let $e_j = e_j + t_1 - t_0$
    let $\omega_j = \omega_j + t_1 - t_0$
If τ_j is finished
  then
    let $j = j + 1$
let $t_2 = \omega_j$
If $t_1 < \omega_j$
  then
    if there exists a non-realtime task pending,
      then give the processor to the first non-realtime task in the queue
      else if $R_j \leq t_1$,
        then give the processor to τ_j
        else let the processor be idle
  else
    give the processor to τ_j

If there exists no non-realtime task pending, the next(or partially executed) task τ_j is executed if it is possible, i.e., the release time of it is reached. Whenever there exists a non-realtime task waiting in the queue, and the latest start(or resume) time, ω_j, is not reached for τ_j the non-realtime task will be executed(after preempting τ_j if it is already started) until it finishes or ω_j is reached. If it continues its execution until ω_j, the non-realtime task is preempted and τ_j will resume its execution or start its execution. In other words, the non-realtime tasks have higher priorities until the latest start(or resume) time of τ_j is reached.

Example case is shown in Fig 5.

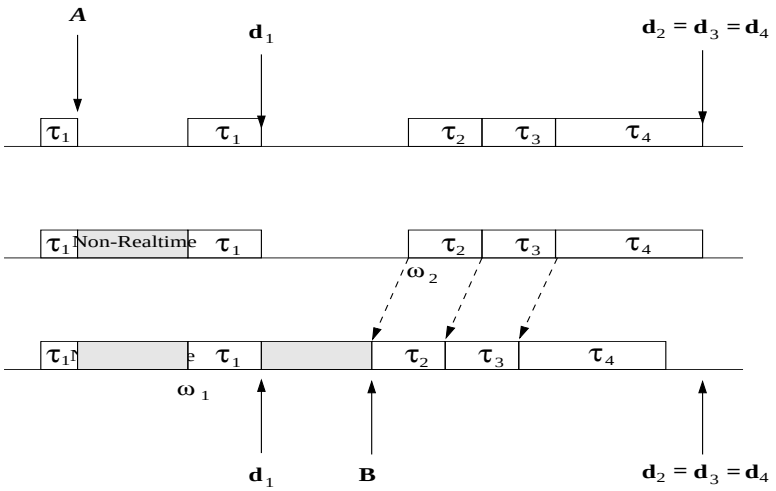

Figure 5: Joint scheduling of a non-realtime and $\mathcal{T}$

4.1.3 Acceptance Test for A Hard Aperiodic Task

Some aperiodic tasks may have hard deadlines within which the execution of it should be completed from the arrival time of it. An acceptance test is performed to check if the requested hard aperiodic task can be finished within its deadline. The relative deadline of A is assumed to be less than or equal to the scheduling window size L. The approach taken in this section treats arriving aperiodic task instances in FIFO order. This assumption will be removed in the next section.

Similar scheduling policy is used for this case to the one in the previous section except that aperiodic tasks are accepted and scheduled instead of non-realtime tasks. However, at the arrival time of an aperiodic task, an acceptance test is performed to check if this new task can be scheduled or not.

The acceptance test algorithm follows. Assume that τ_i is the next or partially executed task when the hard aperiodic task, A, arrived at time R^a.

At the arrival time, R^a, of an aperiodic task, A:

$TotalCapacity = \omega_i - R^a$
$k = i + 1$
While ($TotalCapacity < c^a$ and $lst(k) \leq R^a + d^a$)
    begin
    $TotalCapacity = TotalCapacity + lst(k) - lst(k-1) - C_k$
    $k = k + 1$
    If ($TotalCapacity \geq c^a$)
      then Return(Success)
    end
$TotalCapacity = TotalCapacity + \max(0, R^a + d^a - lst(k-1) - C_{k-1})$
If ($TotalCapacity \geq c^a$)
  then Return(Success)
  else Return(Fail)

At the arrival time of an aperiodic task, R^a, the acceptance test can be done in $O(M)$ time within this framework where M denotes the total number of task instance $\tau_j (i \leq j)$ which satisfies $R^a \leq lst(j) \leq R^a + d^a$. In this case, the total amount of available processor time for A in $[R^a, R^a + d^a]$ can be found by the following formula:

$$\begin{aligned} \Omega(R^a, R^a + d^a) &= \omega_i - R^a \\ &\quad + \sum_{k=i}^{j'-1} (lst(k+1) - lst(k) - C_k) + \max(0, R^a + d^a - lst(j') - C_{j'}) \end{aligned} \tag{1}$$

where $j' (i \leq j')$ is the last index satisfying $\omega_{j'} \leq R^a + d^a$. Note that we can pre-calculate the values, $lst(k+1) - lst(k) - C_k$ at pre-runtime and use them at runtime to reduce the runtime complexity of the acceptance test algorithm.

Example case is depicted in Fig 6 where $j' = 5$.

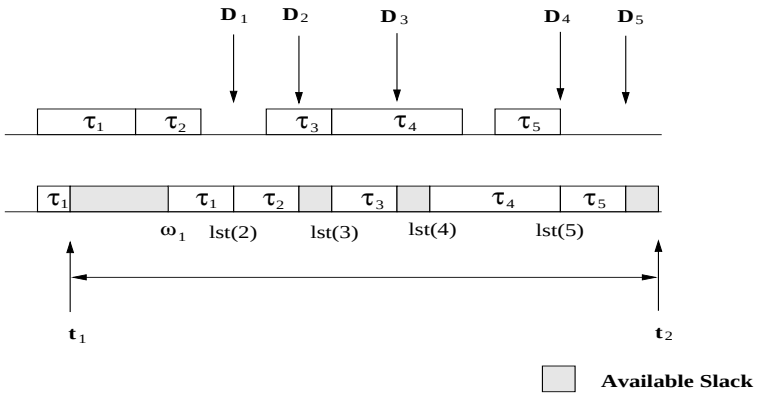

Figure 6: Obtaining a maximum slacks within a scheduling window of a hard aperiodic task A

4.1.4 Acceptance Test for A Set of Hard Aperiodic Tasks

In this section, we address the problem of scheduling aperiodic tasks when several such tasks may arrive at any time instants. In this generalized scheduling model, we need to decide which scheduling policy is to be used for resolving the resource conflicts between the task instances in $\mathcal{T}$ and the aperiodic tasks, as well as the conflicts among the aperiodic tasks. For example, we can assign higher priorities to aperiodic tasks than the task instances in $\mathcal{T}$ as long as the latest start times of them are not reached, and use an earliest deadline first scheduling algorithm among the aperiodic tasks. However, this algorithm is not optimal as you can see from Fig 7. In this figure, the example task set is shown which is not schedulable according to the above approach. But, there exists a feasible schedule for this task set as is shown at the bottom of this figure. In the following subsections, we develop an optimal scheduling algorithm.

Deriving Virtual Deadlines and Virtual Release Times

As a first step, we derive a virtual deadline and a virtual release time for each task instance τ_i in $\mathcal{T}$. This process is necessary to enforce the total order on $\mathcal{T}$ when we employ EDF scheduling policy to resolve the resource conflicts in an unified manner for all the task instances.

A virtual deadline of τ_i is defined by the following recursive equation where D_i^o is the original deadline of τ_i:

$$\begin{aligned} D_N &= D_N^o \\ D_i &= min(D_{i+1} - C_{i+1}, D_i^o) \text{ for } i = N-1, N-2, \ldots, 1 \end{aligned}$$

If a virtual deadline is missed by some task τ_i, then either the deadline of that task itself is missed or at least one of the following tasks misses its deadline. It is clear that the virtual deadline is always less than or equal to the original one and the virtual deadline D_i is always less than D_{i+1} by a difference of at least C_{i+1}, i.e. $D_i \leq D_{i+1} - C_{i+1}$.

Also, a virtual release time of τ_i is defined by the following recursive equation where R_i^o is the original release time of τ_i. Fig 8 explains the virtual release time and deadlines of the example tasks. Virtual release time is necessary to impose a total order on $\mathcal{T}$ when an EDF scheduling

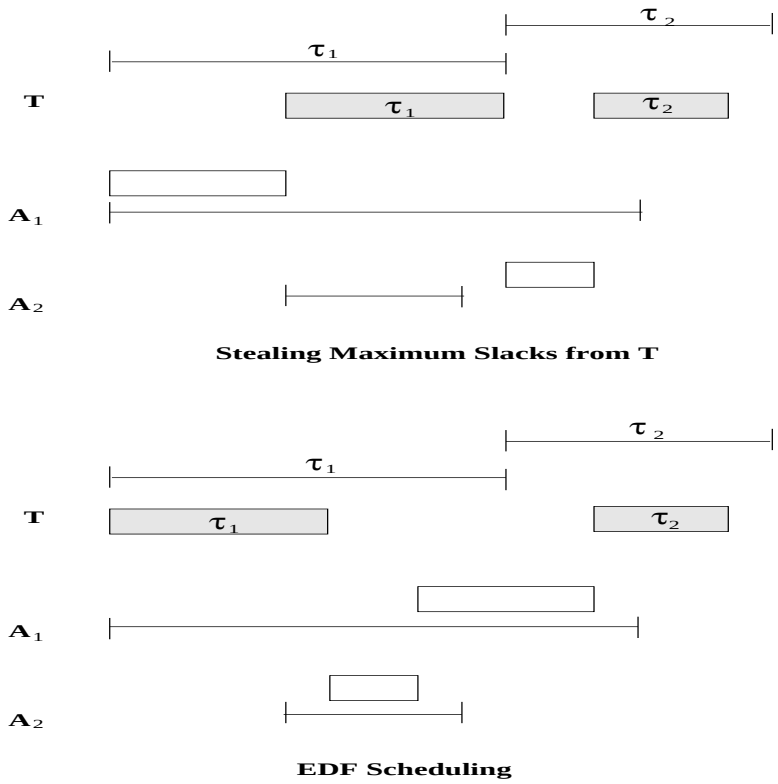

Figure 7: Example Schedules

algorithm is used to schedule the tasks.

$$\begin{aligned} R_1 &= R_1^o \\ R_i &= max(R_{i-1}, R_i^o) \text{ for } i = 2, 3, \ldots, N \end{aligned}$$

This reduction of scheduling window of each task to $[R_i, D_i]$ from $[R_i^o, D_i^o]$ by the introduction of the virtual deadline is the result of imposing total order on $\mathcal{T}$.

The following proposition establishes the equivalence between the original task set and the transformed task set with virtual deadline and release times in terms of the schedulability when an EDF is used to schedule $\mathcal{T}$ and an additional set of aperiodic tasks. Here, it is assumed that the total order of the task instances in $\mathcal{T}$ should be kept.

Proposition 1 *$\mathcal{T}$ and a set of additional aperiodic tasks are schedulable by EDF if and only if $\mathcal{T}$ with virtual deadlines and release times is schedulable with the additional aperiodic tasks by EDF.*

Proof Proof can be derived from the theorem in [8].

Optimal Scheduling Algorithm

In this section, the optimal scheduling algorithm is presented and its optimality is proved. We assume that the task instances in $\mathcal{T}$ have virtual deadlines and virtual release times instead of the

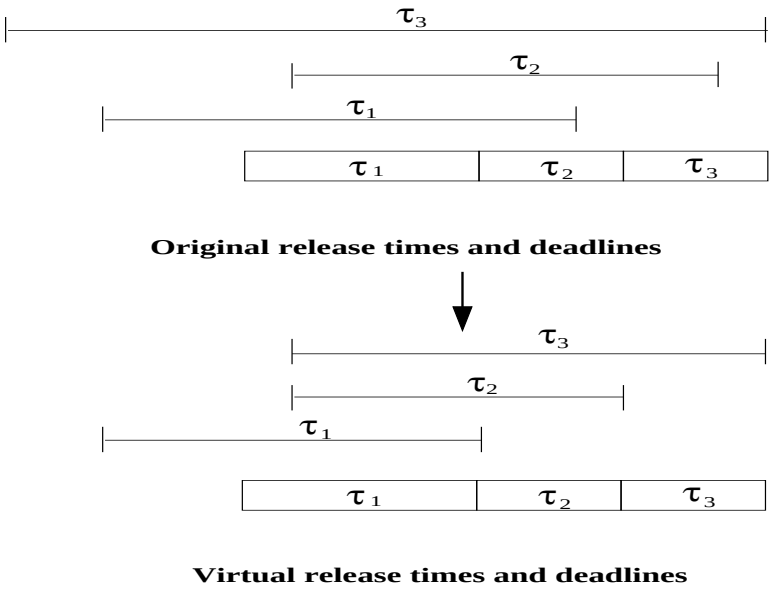

Figure 8: Deriving virtual deadlines and release times

original ones. The optimal scheduling algorithm assigns a higher priority to a task instance with a closer deadline in an unified manner.

At any time instant t, let $\mathcal{A}^{old}(t) = \{A_1^{old}, A_2^{old}, \ldots, A_m^{old}\}$ denote a set of *active* aperiodic tasks. Here, active aperiodic task is the aperiodic task that was accepted before t and still needs to be executed. It is obvious that the deadlines of these aperiodic tasks are greater than t. The tasks in $\mathcal{A}^{old}(t)$ are assumed to be sorted in their increasing order of deadlines. In addition, A_t^{new} denotes a newly arrived aperiodic task at time t. The first step of testing the acceptability of A_t^{new} is to insert A_t^{new} into $\mathcal{A}^{old}(t)$, thus producing $\mathcal{A}(t) = \{A_1, A_2, \ldots, A_{m+1}\}$ in which the tasks are sorted according to their deadlines in increasing order. Also, let $e_i^a(t)$ denote the processor time already spent for A_i up to time t. Obviously, $e_i^a(t) = 0$ if $A_i = A_t^{new}$. At this point, we derive the following lemmas and theorem which proves the optimality of the EDF scheduling algorithm proposed above.

The following lemma specifies the necessary condition for $\mathcal{A}(t)$ to be schedulable. Here, let D_i^a $(1 \leq i \leq m+1)$ denote a deadline of the i-th aperiodic task, A_i, in $\mathcal{A}(t)$.

Lemma 1 *Let $\mathcal{A}(t)$ denote a set of aperiodic tasks defined above. If there exists a feasible schedule for $\mathcal{A}(t)$, then*

$$\forall\ 1 \leq i \leq m+1\ ::\ \Omega(t, D_i^a) \geq \sum_{j=1}^{i} (c_j^a - e_j^a(t)) \tag{2}$$

Proof Suppose (2) is not satisfied for some $1 \leq k \leq m+1$, then

$$\Omega(t, D_k^a) < \sum_{j=1}^{k} (c_j^a - e_j^a(t))$$

This means that the processor demand in $[t, D_k^a]$ required by $\mathcal{A}(t)$ exceeds the maximum processor time in $[t, D_k^a]$ available for $\mathcal{A}(t)$. The un-schedulability of $\mathcal{A}(t)$ follows.

Lemma 2 *Let $\mathcal{A}(t)$ denote a set of aperiodic tasks defined above. Then $\mathcal{A}(t)$ can be scheduled under the proposed EDF if*

$$\forall\ 1 \leq i \leq m+1\ ::\ \Omega(t, D_i^a) \geq \sum_{j=1}^{i}(c_j^a - e_j^a(t))$$

Proof The proof can be easily derived from the theorem 3.2 and 3.3 in the paper [6].

Theorem 1 *Let $\mathcal{A}(t)$ denote a set of aperiodic tasks defined above. Then the proposed EDF scheduling algorithm is optimal and the schedulability condition is:*

$$\forall\ 1 \leq i \leq m+1\ ::\ \Omega(t, D_i^a) \geq \sum_{j=1}^{i}(c_j^a - e_j^a(t))$$

Proof From Lemma 1 and Lemma 2, this theorem follows.

Clearly, the condition of the above theorem can be checked within $O(M+m)$ by utilizing the formula (1) where M denotes the total number of task instances in $\mathcal{T}$ whose deadlines are greater than t and less than or equal to D_{m+1}^a, i.e., the task instances in $\mathcal{T}$ which may be executed within the range $[t, D_{m+1}^a]$. The first step is to insert the newly arrived aperiodic task into the set of active aperiodic tasks so that the aperiodic tasks are ordered in increasing order of their deadlines. Then, the maximum slack times, $\Omega(t, D_i^a)$, are found from $i = 1$ to $i = m+1$ by making use of $\Omega(t, D_{i-1}^a)$ already found.

If multiple number of aperiodic tasks arrive at t, we have to give priorities to these aperiodic tasks to decide which one has to be accepted and scheduled first. In this case, the above acceptance test is repeated for each aperiodic task from the one with highest priority to the one with lowest importance. The total complexity in this case is $O(K(M+m))$ where K denotes the number of aperiodic tasks arrived at t.

4.2 Sporadic Task Scheduling

One of the drawbacks of time-based scheduling scheme is that the sporadic task scheduling becomes very difficult. The algorithm to transform a sporadic task to an equivalent pseudo-periodic task has been proposed by *Al Mok* [22]. From the definition of the sporadic tasks, the events which invoke the sporadic task instances may occur at any time instants with the minimum inter-arrival time, δ. And, once the task is invoked, it has to be finished within its relative deadline from the invocation time, d^s. The first step of the transformation is to decide the relative deadline of the pseudo-periodic task, d^p, which is less than or equal to d^s. And then, the period, prd^p, of the pseudo task is found from the equation $prd^p = \min(d^s - d^p + 1, \delta)$. This is justified from the worst case scenario which can be seen in Figure 9.

However, this approach may lead to significant under-utilization of the processor time, especially when d^s is small compared to δ, since a great amount of processor time has to be reserved statically at pre-runtime for servicing dynamic requests from sporadic tasks. This is well explained in Fig 10 through a simple example where an equivalent periodic task is to be found from a sporadic task

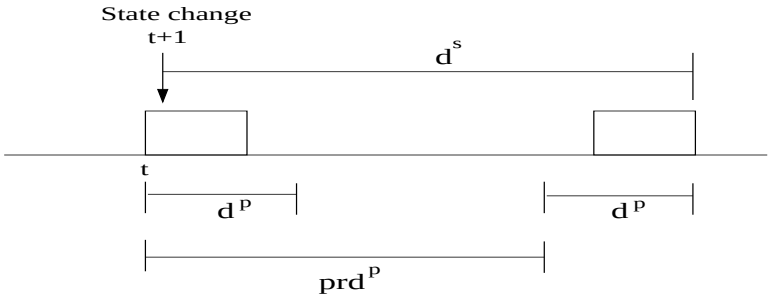

Figure 9: Worst Case for Deadline Determination

whose worst case execution time is $c^s = 4$, whose relative deadline is $d^s = 8$, and whose minimum inter-arrival time is $\delta = 8$. If we employ *Mok*'s algorithm, the corresponding periodic task has a worst case execution time $c^p = c^s = 4$, a relative deadline $d^p = 4(\leq d^s)$, and a period $prd^p = \min(d^s - d^p + 1, \delta) = 5$. The processor utilization of this new periodic task is $4/5 = 0.8$.

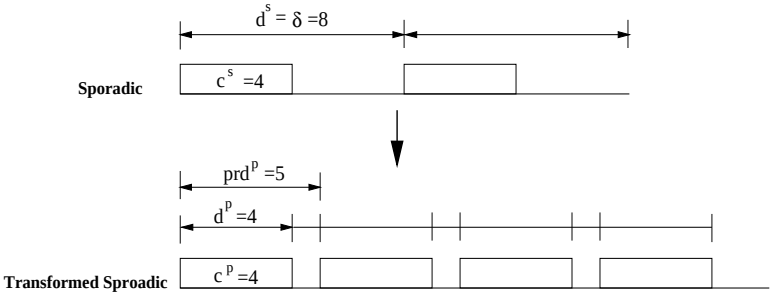

Figure 10: Under-utilization of the transformed sporadic task

In our proposed scheduling approach, the incremental scheduling of hard periodic tasks and sporadic tasks may be decomposed into two steps. We assume that the initial schedule of task instances is given in a scheduling window $[0, L]$ as in the previous sections. Then, the release times and deadlines of those task instances are transformed into virtual ones as was done in Section 4.1. And at runtime, every time new sporadic task arrives, the schedulability check is performed to see if the already accepted tasks and this new sporadic tasks can be scheduled using the EDF scheduling algorithm. And at runtime, the hard task instances from the schedule and the sporadic tasks are scheduled according to EDF. This can be viewed as merging two task instance streams, one from hard tasks and the other from sporadic tasks.

4.2.1 Extended Task Model

As in Section 4.1.1, an initial schedule of task instances is assumed to be given in an scheduling window $[0, L]$ and denoted as Γ. Let $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_N\}$ be a set of task instances in Γ where τ_i appears earlier than τ_{i+1} in Γ. Each τ_i has a following set of parameters in the schedule.

- virtual release time R_i
- virtual deadline $D_i(\leq L)$

- worst case execution time C_i

deadlines and virtual release times are obtained as in Section 4.1.4 from the original ones.

Let $\mathcal{S} = \{S_1, S_2, \ldots, S_{m_s}\}$ be a set of sporadic tasks which have to be scheduled with $\mathcal{T}$. For each sporadic task S_i, the minimum inter-arrival time δ_i, the maximum execution time c_i^s, and the relative deadline d_i^s ($\leq \delta_i$) are assumed to be given. It is also assumed that the S_is are ordered in increasing order of their relative deadlines, d_i^s, i.e., $d_i^s \leq d_{i+1}^s$. The objective of this section is to develop an optimal scheduling algorithm and its schedulability test for $\mathcal{T}$ and $\mathcal{S}$ together.

Some additional terms are defined in the following:

- Extended scheduling window for $\mathcal{T}$ and $\mathcal{S}$, $[0, LCM]$, where LCM is the least common multiple of L and the minimum inter-arrival times of the tasks in $\mathcal{S}$.
- $N^{'}$ denotes the total number of hard task instances scheduled in $[0, LCM]$. $N^{'} = N(LCM/L)$ where $[0, L]$ is the original scheduling window of Γ
- Extended schedule in an extended scheduling window $[0, kLCM]$ is found by repeating k times the schedule Γ and denoted as $k\Gamma$.

We need to check the schedule in an extended window $[0, 2LCM]$ to verify a schedulability of $\mathcal{T}$ and $\mathcal{S}$ according to the following scheduling model.

4.2.2 Scheduling Model

The conflicts among the tasks in $\mathcal{T}$ are resolved naturally from the total order among the tasks given in Γ. This can be done by using an earliest deadline first scheduling algorithm and by using the virtual deadlines introduced earlier since $R_i \leq R_{i+1}$ and $D_i < D_{i+1}$. But, the mechanisms to resolve the resource contention between tasks from $\mathcal{S}$ and those from $\mathcal{T}$ should be provided to enable them to be scheduled at run-time. We assume that those contentions are also resolved through the same scheduling algorithm(EDF), leading to an uniform scheduling policy for $\mathcal{S}$ and $\mathcal{T}$.

We denote a subset, $\{\tau_a, \tau_{a+1}, \ldots, \tau_b\}$, of $\mathcal{T}$ in $[0, LCM]$ as Ψ if:

- $1 \leq a \leq b \leq N$
- $est(j+1) = est(j) + C_j$ for $j = a+1, a+2, \ldots, b-1$
- $est(a) > est(a-1) + C_{a-1}$ if $1 < a$
- $est(b+1) > est(b) + C_b$ if $b+1 \leq N$

In this case, we can divide the set of task instances in $[0, LCM]$ into disjoint subsets, Ψ_1, Ψ_2, ..., Ψ_ψ, satisfying the above conditions. Let $est(\Psi_i)$ denote the earliest start time of the first task instance in Ψ_i and let $eft(\Psi_i)$ denote the earliest finish time of Ψ_i. Figure 11 shows an example case.

In addition, we define $\Phi(t_1, t_2)$ $(0 \leq t_1 < LCM \wedge t_1 < t_2 < 2LCM)$ as the maximum slack time obtainable in $[t_1, t_2]$ under the assumption that from time 0 up to time instant t_1 task instances only from $\mathcal{T}$ have been executed with their maximum execution times, i.e., tasks have started at their

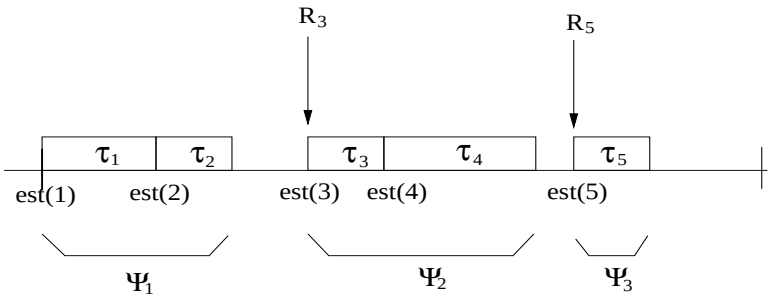

Figure 11: Ψ found for an example task set

earliest start times and spent their worst case execution times. Then, $\Phi(t_1, t_2)$ can be obtained as follows. First step is to find task instance τ_i satisfying:

$$est(i-1) + C_{i-1} \leq t_1 \ \wedge \ t_1 \leq est(i) + C_i$$

If $t_1 \leq est(1) + C_1$, then let $i = 1$. Then,

$$\begin{aligned} \Phi(t_1, t_2) \ &= \ lst(i) - t_1 + \max(0, t_1 - est(i)) \\ &\quad + \sum_{k=i}^{j'-1} (lst(k+1) - lst(k) - C_k) + \max(0, t_2 - lst(j') - C_{j'}) \end{aligned} \tag{3}$$

where $j'(i \leq j')$ is the last index satisfying $lst(j') \leq t_2$. This process is similar to the one used in the acceptance test of aperiodic task in Section 4.1. Example case is depicted in Figure 12.

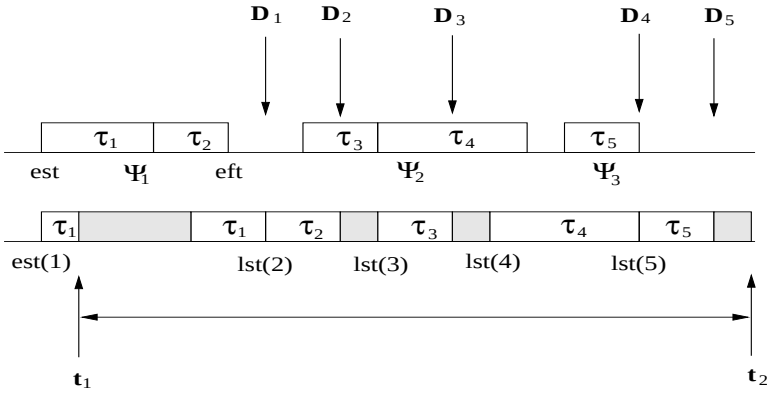

Figure 12: $\Phi(t_1, t_2)$ for an example task set

4.2.3 Schedulability Test

The following proposition specifies the necessary condition for $\mathcal{T}$ and $\mathcal{S}$ to have a feasible schedule.

Proposition 2 *If there exists a feasible schedule for $\mathcal{T}$ and $\mathcal{S}$, then*

$$\forall i \in [1, \psi] \ :: \ \forall t \in [est(\Psi_i), est(\Psi_i) + LCM]$$

$$:: \ \Phi(est(\Psi_i), t) \geq \sum_{k=1}^{m_s} c_k^s \cdot \lfloor \frac{(t - est(\Psi_i) + \delta_k - d_k^s)}{\delta_k} \rfloor \tag{4}$$

Proof: This is proved in the appendix.

The following theorem specifies the sufficient and necessary schedulability condition of the task set $\mathcal{T}$ and $\mathcal{S}$. The extended schedule in $[0, 2LCM]$ is assumed to be given.

Theorem 2 *$\mathcal{T}$ and $\mathcal{S}$ are schedulable according to EDF if and only if*

$$\begin{aligned} &\forall i \in [1, \psi] \ :: \ \forall t \in [est(\Psi_i), est(\Psi_i) + LCM] \\ &:: \ \Phi(est(\Psi_i), t) \geq \sum_{k=1}^{m_s} c_k^s \cdot \lfloor \frac{(t - est(\Psi_i) + \delta_k - d_k^s)}{\delta_k} \rfloor \end{aligned} \tag{5}$$

Proof: By proposition 2 and proposition 7.

From the above proposition and a theorem, we can know that EDF is optimal for scheduling $\mathcal{T}$ and $\mathcal{S}$. Finally, we obtain an equivalent condition to (5) of the theorem 2, which enables us to reduce the complexity of the schedulability check. This corollary specifies that only the time instant which is equal to a deadline of some task instance in $\mathcal{S}$ needs to be examined at or after $est(\Psi_i)$ in checking the condition (5) of the theorem 2.

Corollary 1 *The following two conditions are equivalent to each other:*

$$\begin{aligned} (1) \quad &\forall i \in [1, \psi] \ :: \ \forall t \in [est(\Psi_i), est(\Psi_i) + LCM] \\ &:: \ \Phi(est(\Psi_i), t) \geq \sum_{k=1}^{m_s} c_k^s \cdot \lfloor \frac{(t - est(\Psi_i) + \delta_k - d_k^s)}{\delta_k} \rfloor \\ (2) \quad &\forall i \in [1, \psi] \ :: \ \forall d_j \in [est(\Psi_i), est(\Psi_i) + LCM] \\ &:: \ \Phi(est(\Psi_i), d_j) \geq \sum_{k=1}^{m_s} c_k^s \cdot \lfloor \frac{(d_j - est(\Psi_i) + \delta_k - d_k^s)}{\delta_k} \rfloor \end{aligned}$$

where d_j is the deadline of some task instance in $\mathcal{S}$.

Therefore, the total complexity of the schedulability check algorithm is reduced to $O(M')$ where $M' = \psi(N' + \sum_{i=1}^{m_s}(LCM/\delta_i)) + \sum_{i=1}^{m_s}(LCM/\delta_i)\log(\sum_{i=1}^{m_s}(LCM/\delta_i))$. The first step is to obtain the deadlines(d_j) of the task instances from $\mathcal{S}$ in the window $[0, LCM]$ and sort them in increasing order. Then, for each $est(\Psi_i)$ $(1 \leq i \leq \psi)$, the second condition of the above corollary is checked in $O(N' + \sum_{i=1}^{m_s}(LCM/\delta_i))$ for the deadlines obtained in the first step. This process is similar to the one used in Section 4.1.4.

5 Conclusion

As was seen in the previous sections, the time-based scheduling scheme can be extended to achieve the flexibility by providing mechanisms such as incremental non-realtime task scheduling, incremental aperiodic task scheduling, and incremental sporadic task scheduling. The schemes are obtained

for cases when preemptions are allowed. For non-preemptible tasks, similar results can be obtained as those presented in this paper.

We believe that this dynamic time-based scheduling scheme is a suitable framework for scheduling dynamic tasks especially in the presence of complex task dependencies and complex timing constraints such as relative timing constraints. The scheduling problem of tasks with relative timing constraints has been addresses in our paper [1]. The start time range of task instances are parameterized in terms of start or finish times of already executed task instances, and the parametric functions are evaluated to obtain the valid range of task instance start times. This enables to dynamically adjust the task instance start times without affecting the schedulability of tasks. On-line scheduling of dynamic tasks based on this dynamic dispatching framework seems to be a promising approach for incorporating both static tasks with complex timing constraints, and dynamic tasks such as non-realtime or aperiodic tasks.

References

[1] Ashok K. Agrawala, Seonho Choi, and Leyuan Shi. Designing temporal controls. Technical Report CS-TR-3504, UMIACS-TR-95-81, Department of Computer Science, University of Maryland, July 1995.

[2] N. C. Audsley. Deadline monotonic scheduling. YCS 146, University of York, Department of Computer Science, October 1990.

[3] T. P. Baker. A Stack-Based Resource Allocation Policy for RealTime Processes. In *IEEE Real-Time Systems Symposium*, 1990.

[4] M. Chen and K. Lin. Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems. *Real-Time Systems*, 2(4):325–346, 1990.

[5] S. T. Cheng and Ashok K. Agrawala. Allocation and scheduling of real-time periodic tasks with relative timing constraints. Technical Report CS-TR-3402, UMIACS-TR-95-6, Department of Computer Science, University of Maryland, January 1995.

[6] H. Chetto and M. Chetto. Scheduling Periodic and Sporadic Task in a Real-Time System. *Information Processing Letters*, 30(4):177–184, 1989.

[7] H. Chetto and M. Chetto. Some Results of the Earliest Deadline First Algorithm. *IEEE Transactions on Software Engineering*, SE-15(10):1261–1269, October 1989.

[8] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic Scheduling of Real-Time Tasks under Precedence Constraints. *Real-Time Systems*, 2:181–194, 1990.

[9] R. I. Davis. Approximate slack stealing algorithms for fixed priority pre-emptive systems. Technical Report YCS 217 (1993), Department of Computer Science, University of York, England, November 1993.

[10] R. I. Davis, K. W. Tindell, and A. Burns. Scheduling slack time in fixed priority pre-emptive systems. In *IEEE Real-Time Systems Symposium*, pages 222–231. IEEE Computer Society Press, December 1993.

[11] M. Dertouzos. Control Robotics: the Procedural Control of Physical Processes. *Proceedings of the IFIP Congress*, pages 807–813, 1974.

[12] G. Fohler and C. Koza. Heuristic Scheduling for Distributed Real-Time Systems. MARS 6/89, Technische Universitat Wien, Vienna, Austria, April 1989.

[13] Gerhard Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, December 1995.

[14] R. Gerber, W. Pugh, and M. Saksena. Parametric Dispatching of Hard Real-Time Tasks. *IEEE Transactions on Computers*. To appear.

[15] T.M. Ghazalie and T.P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 9:31–67, 1995.

[16] Homayoun and P. Ramanathan. Dynamic priority scheduling of periodic and aperiodic tasks in hard real-time systems. *Real-Time Systems*, 6(2), March 1994.

[17] J. P. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *IEEE Real-Time Systems Symposium*, pages 166–171, Dec. 1989.

[18] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *IEEE Real-Time Systems Symposium*, pages 261–270, Dec. 1987.

[19] John P. Lehoczky and Sandra Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *IEEE Real-Time Systems Symposium*, pages 110–123. IEEE Computer Society Press, December 1992.

[20] J.Y. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[21] C. L. Liu and J. Layland. Scheduling Algorithm for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM.*, 20(1):46–61, Jan. 1973.

[22] A. K. Mok. *Fundamental Design Problems for the Hard Real-time Environments*. PhD thesis, MIT, May 1983.

[23] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[24] T. Shepard and J. A. M. Gagne. A Model of The F-18 Mission Computer Software for Pre-Run Time Scheduling. In *IEEE* 10^{th} *International Conference on Distributed Computer Systems*, pages 62–69, May 1990.

[25] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1):27–60, June 1989.

[26] Marco Spuri and Giorgio C. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *IEEE Real-Time Systems Symposium*, pages 2–11. IEEE Computer Society Press, December 1994.

[27] Sandra R. Thuel and John P. Lehoczky. Algorithm for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. In *IEEE Real-Time Systems Symposium*, pages 22–33. IEEE Computer Society Press, December 1994.

[28] K. Tindell, A. Burns, and A. Wellings. An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks. *Real-Time Systems*, 6(2), March 1994.

[29] J. Xu and D. L. Parnas. Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Transactions on Software Engineering*, SE-16(3):360–369, March 1990.

[30] J. Xu and D. L. Parnas. On Satisfying Timing Constraints in Hard-Real-Time Systems. In *ACM SIGSOFT'91 Conference on Software for Critical Systems*, pages 132–146, December 1991.

[31] X. Yuan, M. Saksena, and A. Agrawala. A Decomposition Approach to Real-Time Scheduling. *Real-Time Systems*, 6(1), 1994.

[32] W. Zhao and K. Ramamritham. Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constraints. *Journal of Systems and Software*, pages 195–205, 1987.

A Proof of Theorem 2

The proof of theorem 2 is presented here.

Proposition 3 *If $\mathcal{T}$ and $\mathcal{S}$ are schedulable then the following condition is satisfied:*

$$\forall i \in [1, \psi] \ :: \ \forall t \in [est(\Psi_i), est(\Psi_i) + LCM]$$
$$:: \ \Phi(est(\Psi_i), t) \geq \sum_{k=1}^{m_s} c_k^s \cdot \lfloor \frac{(t - est(\Psi_i) + \delta_k - d_k^s)}{\delta_k} \rfloor$$

Proof: Suppose that $\mathcal{S}$ and $\mathcal{T}$ are schedulable and the above condition doesn't hold. Let t_v be the first time instant at which the condition is not satisfied. That is, the following is satisfied for some $i_v \in [1, \psi]$:

$$\Phi(est(\Psi_{i_v}), t_v) < \sum_{k=1}^{m_s} c_k^s \cdot \lfloor \frac{(t_v - est(\Psi_{i_v}) + \delta_k - d_k^s)}{\delta_k} \rfloor$$

However, from this we can conclude that the task set is not schedulable when all the sporadic tasks start to be invoked at time $est(\Psi_{i_v})$ with their minimum inter-arrival times. This is because the processor demand by $\mathcal{S}$ in $[est(\Psi_{i_v}), t_v]$ exceeds the processor time in $[est(\Psi_{i_v}), t_v]$ available for tasks in $\mathcal{S}$. Therefore, if $\mathcal{T}$ and $\mathcal{S}$ are schedulable, the condition is satisfied.

We define a *busy period* for the given task, α, which belongs to $\mathcal{T}$ or $\mathcal{S}$ and denote it as $BP_\alpha = [a, f_\alpha]$ where f_α is the actual finish time of the task α at run-time. Let D_α denote a deadline of α. Then, let β be the *last* task satisfying the following conditions:

(1) $\beta \in \mathcal{T}$ or $\beta \in \mathcal{S}$
(2) β starts its execution before f_α.
(3) β starts its execution at its release time r_β.
(4) no idle period exists between r_β and f_α.
(5) no task whose deadline is greater than D_α is executed between r_β and f_α.

Then, the following proposition claims that the task β exists for any given task α.

Proposition 4 *If EDF is used at run-time to schedule $\mathcal{T}$ and $\mathcal{S}$, for any given task α ($\in \mathcal{T}$ or $\in \mathcal{S}$), the task β($\in \mathcal{T}$ or $\in \mathcal{S}$) exists.*

Proof: It is clear that at the end of the last idle period before f_α, the conditions (1), (2), (3), and (4), hold for some task β_0 whose release time is equal to the end of that idle period. If there is no idle period before α, then let β_0 denote the first task which starts its execution at time 0. Let β_1 denote the *last* task which starts its execution between the end of the idle period and f_α, and which satisfies all of the conditions from (1) to (4). In this case, β_1 is the last task in $[r_{\beta_0}, f_\alpha]$ which starts its execution at its release time. In other words, every task executed between r_{β_1} and $f_\alpha]$ has started its execution some time after its release time except β_1.

Suppose that the condition (5) is not satisfied in $[r_{\beta_1}, f_\alpha]$ and let γ denote a task whose start time is between r_{β_1} and f_α, and which has a deadline D_γ greater than D_α. But, because D_α is less than D_γ and EDF is used to schedule the tasks, a contradiction has occurred. γ should never have

been executed between r_γ and f_α since the task α has a higher priority than γ. Therefore, task instance β_1 satisfies the condition from (1) to (5).

Then, the start time a of the busy period for α is defined to be r_{β_1} which is found in the above proof procedure. Example busy period is depicted in Fig 13.

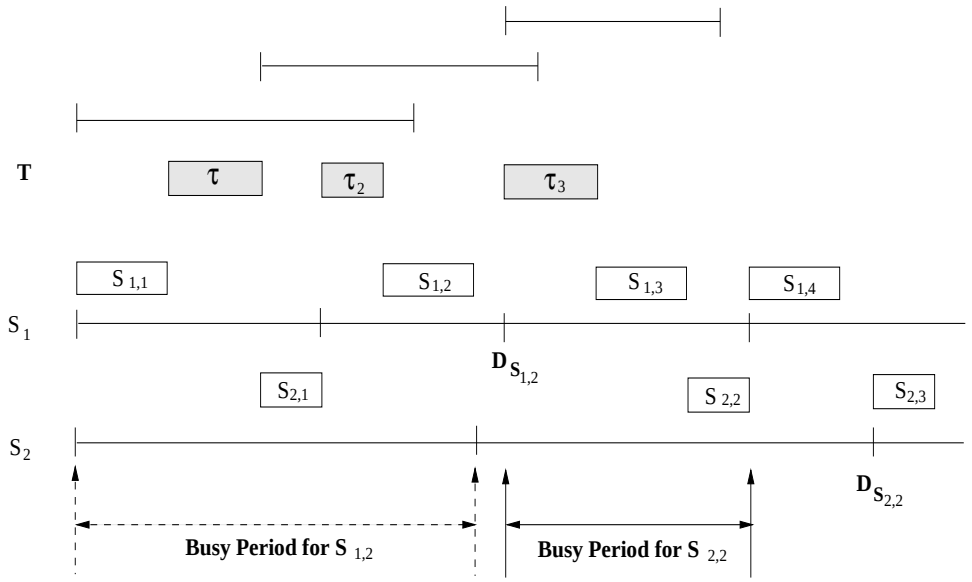

Figure 13: Busy period

Here, the *earliest finish time* of τ_i is defined as $est(i) + C_i$.

Proposition 5 *The following is satisfied for every* $i \in [2, \psi + 1]$*:*

$$\forall t_1 \in [eft(\Psi_{i-1}), est(\Psi_i)] \ :: \ \forall l > 0 \ :: \ \Phi(t_1, t_1 + l) \geq \Phi(est(\Psi_i), est(\Psi_i) + l) \tag{6}$$

Proof: If the time interval $[est(\Psi_i), est(\Psi_i) + l]$ is shifted to the left by the amount of $est(\Psi_i) - t_1$ which results in a new time interval $[t_1, t_1 + l]$, the slack time is increased by the amount of $est(\Psi_i) - t_1$ and decreased with the amount less than or equal to $est(\Psi_i) - t_1$. This is depicted in Figure 14.

Proposition 6 *The following is satisfied for every* $i \in [1, \psi]$*:*

$$\forall t_1 \in (est(\Psi_i), eft(\Psi_i)) \ :: \ \forall l > 0 \ :: \ \Phi(t_1, t_1 + l) \geq \Phi(est(\Psi_i), est(\Psi_i) + l) \tag{7}$$

Proof: If the time interval $[est(\Psi_i), est(\Psi_i) + l]$ is shifted to the right by the amount of $t_1 - est(\Psi_i)$ which results in a new time interval $[t_1, t_1 + l]$, the maximum slack time, Φ is increased or at least remains the same as can be seen from Figure 15. This proves the proposition.

Proposition 7 *If* $\mathcal{T}$ *and* $\mathcal{S}$ *satisfy the condition of proposition 2, then they are schedulable by EDF scheduling algorithm.*

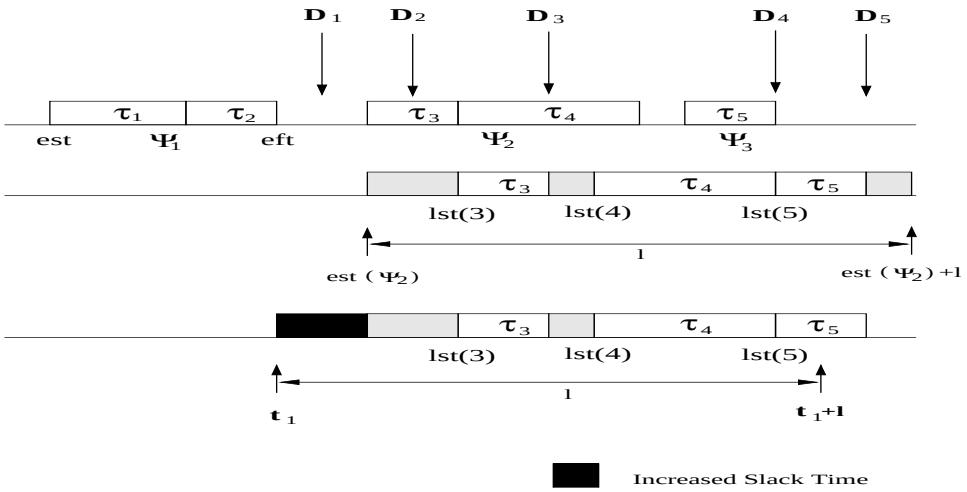

Figure 14: Φ is increased or remains the same in the shifted interval

Proof:

Suppose that the condition is satisfied for $\mathcal{S}$ and $\mathcal{T}$ and some task can't be finished within its deadline. Let's call that task α ($\alpha \in \mathcal{T}$ or $\alpha \in \mathcal{S}$) and the deadline of that task D_α. And, let $BP_\alpha = [t_i, f_\alpha]$ denote a busy period for α. In this case, the actual finish time of α, f_α, is greater than D_α.

Then there are two cases to be considered.

Case 1: $D_\alpha - t_i > LCM$.

Note that the maximum processor demand in $[t_i, t_i + LCM]$ by task instances from $\mathcal{S}$ is less than or equal to $\Phi(t_i, t_i + LCM)$ from the condition 4. In this case, at $t_i + LCM$ a new task instance starts its execution whose release time is equal to $t_i + LCM$. Then, it is obvious that the start time of the busy period, t_i, should be greater than or equal to $t_i + LCM$, which is a contradiction.

Case 2: $D_\alpha - t_i \leq LCM$.

Let τ_ι be the first task in $[t_i, D_\alpha]$ which belongs to $\mathcal{T}$. First, suppose that this exists. Then, let Ψ_j denote the task group containing τ_{iota}. From the definition of a busy period we know that the release time of τ_ι, r_ι, is greater than or equal to t_i. Then from proposition 5 and 6,

$$\forall l > 0 \ :: \ \Phi(t_i, t_i + l) \geq \Phi(est(\Psi_j), est(\Psi_j) + l)$$

This means that if the tasks in $\mathcal{S}$ starts to invoke their task instances from t_i with their minimum inter-arrival times, then they are schedulable with $\mathcal{T}$. This implies that the task instances invoked at or after t_i are schedulable since the worst case scenario is that every $S_i \in \mathcal{S}$ starts to be invoked at t_i with δ_i inter-arrival time, which is proven to be schedulable. This contradicts to the assumption that α misses its deadline at D_α.

Second, suppose that τ_ι doesn't exist. In this case all the task instances executed in the interval $[t_i, D_\alpha] \subset [t_i, f_\alpha]$ are from $\mathcal{S}$. It is clear in this case from the condition 4 that

$$\forall l > 0 \ :: \ \Phi(t_i, l) \geq \sum_{k=1}^{m_s} c_k^s \cdot \lfloor (l - t_i + \delta_k - d_k^s)/\delta_k \rfloor$$

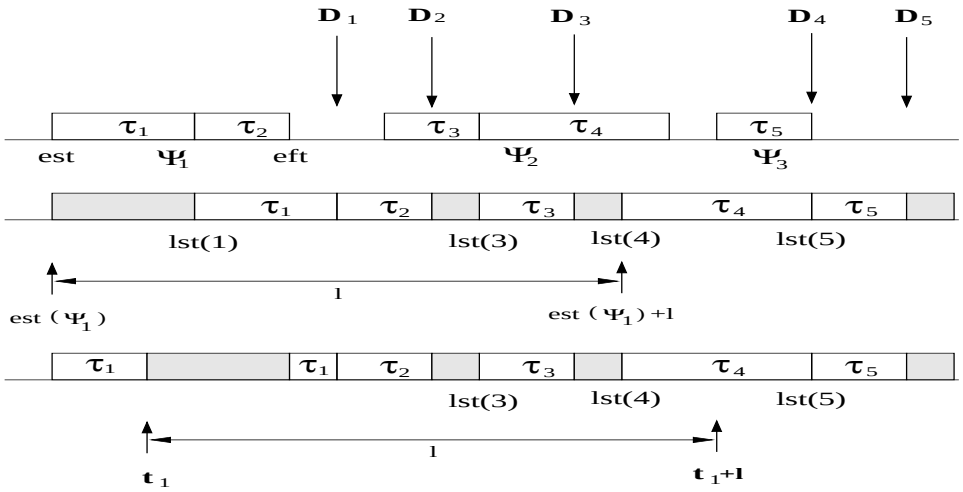

Figure 15: Φ is increased or remains the same in the shifted interval

From this, we can conclude that every task instance in $[t_i, D_\alpha]$ is schedulable, which contradicts to the assumption that α misses its deadline at D_α.