Submitted to IEEE Transactions on Software Engineering

A Linear-Time Optimal Acceptance Test for Scheduling of Hard Real-Time Tasks

Too-Seng Tia Jane W.-S. Liu Jun Sun Rhan Ha

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, IL 61801

November 15, 1994

Abstract

This paper describes an optimal acceptance test for scheduling sporadic requests together with periodic tasks on an earliest-deadline-first basis. The deadlines of periodic requests must be met. Sporadic requests are accepted only if their deadlines can also be met. The time complexity of our acceptance test is $O(n+m)$, where n is the number of periodic tasks and m is the number of sporadic requests that have previously been accepted but not completed. The proposed algorithm is a significant improvement over previous algorithms which are pseudo-polynomial in time complexity, i.e., $O(N+m)$ where N is the number of periodic requests in a hyperperiod.

1 Introduction

A real-time system typically has a mixture of off-line and on-line workloads. Each off-line request for execution $R_i(r_i, c_i, d_i)$ is characterized by its *ready time* r_i, *worst-case computation time* c_i, and (absolute) hard *deadline* d_i. These parameters are known before the system begins execution. Together, the off-line requests support the normal functions of the system. In addition, the system must also respond to unexpected requests, e.g., operator commands and recovery actions. These requests arrive sporadically. Each sporadic request $S_i(A_i, C_i, D_i)$ is characterized by its *arrival time* A_i, *worst-case computation time* C_i, and hard *deadline* D_i. The parameters of a sporadic request become known when the request arrives.

The commonly adopted approach to scheduling this mixed workload is to ensure that all off-line requests meet their deadlines and to accept as many sporadic requests, which can be guaranteed to meet their deadlines, as possible. When a sporadic request arrives, the scheduler performs an acceptance test. The test succeeds if the scheduler can schedule the request to meet its deadline without causing any off-line request or previously accepted sporadic request to miss its deadline. If the test succeeds, the scheduler accepts the sporadic request and schedule it; otherwsie the scheduler rejects the request.

In this paper, we consider the problem of designing an efficient acceptance test for the important special case where off-line requests are instances of periodic tasks [1]. In this case, there are an infinite number of off-line requests. The set $\{R_i(r_i, c_i, d_i)\}$ of these requests is partitioned into n disjoint subsets. Each subset is a periodic task, denoted by $T_i(c_i, p_i, \phi_i)$ or T_i for short. All requests in T_i have the same worst-case computation time c_i. Their ready times occur periodically with period p_i, the *period* of T_i. The length of time $(d_j - r_j)$ between the ready time r_j and deadline d_j of every request $R_j(r_j, c_j, d_j)$ in $T_i(c_i, p_i, \phi_i)$ is equal to the relative deadline ϕ_i of T_i. Because off-line requests become ready periodically, they are called periodic requests in the

literature. Hereafter, we will use this term.

A number of acceptance tests [2, 3, 4, 5] have been proposed for the case where all the requests are independent and are scheduled preemptively on the earliest-deadline-first (EDF) basis. Because of the optimality of the EDF algorithm [1, 6], most of the known acceptance tests (e.g., [2, 3, 4]) based on this principle are optimal in the following sense: A sporadic request is accepted if and only if it can be scheduled without causing any missed deadline, and if it is rejected, all other acceptance tests must also reject it.

Chetto and Chetto [2] proposed an acceptance test that takes $O(N)$ time: N is the total number of requests in each hyperperiod of the n periodic tasks in the system, the length of each hyperperiod being the least common multiple of the periods $p_1, p_2, \cdots, p_n$ of the n periodic tasks. They prescheduled the periodic requests and whenever a sporadic request arrive, the acceptance test determines the location and duration of processor idle times in the schedule. One restriction of the test is that Chetto and Chetto assumed that whenever the acceptance test is done, there is no uncompleted sporadic request in the system. Silly, Chetto and Elyounsi [3] extended the test to deal with an arbitrary number of previously accepted but uncompleted sporadic requests. The extended test takes $O(N + m)$ time. The acceptance test proposed by Schwan and Zhou [4] has a worst-case time complexity of $O(N + m)$. Their algorithm maintains a slot list which is used to search for available time to schedule the periodic and sporadic requests. The main drawback of their approach is that both periodic and sporadic requests are subjected to the test, leading to unnecessarily high overheads. In an attempt to reduce the time complexity of the acceptance test, McElhone [5] proposed five acceptance tests which trade off time complexity with accuracy. Except for the test which takes $O(N + m)$ time, the other tests are pessimistic and not optimal. The fastest test has a time complexity of $O((n + m)^2)$.

In this paper, we proposed an optimal acceptance test which handles an arbitrary number

of sporadic requests. Only sporadic requests, and not periodic requests, are subjected to this test. Our test has a time complexity of $O(n+m)$ which is significantly lower than $O(N+m)$ for most real-time systems. In fact, since N depends on the periods of the tasks, previous optimal acceptance tests are pseudo-polynomial in time complexity. The efficiency of our test is attained by carefully exploiting the properties of EDF scheduling and periodic tasks, as well as by maintaining simple and easy to update data structures.

Following this introduction, Section 2 states additional assumptions and introduces notation used later in the paper. Section 3 presents our acceptance algorithm for the first sporadic request. Section 4 generalizes the algorithm to handle multiple sporadic requests. Section 5 describes two extensions to this algorithm. Section 6 concludes the paper.

2 Assumptions and Notation

Again, the system contains a set of n independent periodic tasks. We refer to this set by $\mathbf{T} = \{T_i(c_i, p_i, \phi_i) : 1 \leq i \leq n\}$. Unless stated otherwise, ϕ_i is assumed to be less than or equal to p_i. Let t_c denote the current time. The request in T_i whose ready time is before or at t_c and whose deadline is after t_c is called the current request of T_i. Because $\phi_i \leq p_i$, every periodic task T_i has only one current request at any time.

We index all the periodic requests in a hyperperiod of periodic tasks in $\mathbf{T}$ in ascending order of their deadlines. With a slight abuse of notation, we refer to the set of N periodic requests in each hyperperiod also as $\mathbf{T}$, i.e., $\mathbf{T} = \{R_i(r_i, c_i, d_i) : 1 \leq i \leq N\}$.

We let $\mathbf{S} = \{S_i(A_i, C_i, D_i) : 1 \leq i \leq m\}$ denote the stream of m sporadic requests that have been accepted but uncompleted at the time when an acceptance test is to be carried out. The requests in $\mathbf{S}$ are indexed in ascending order of their deadlines. In our discussion, we will often use S_i and R_i to refer to an individual sporadic request and periodic request, respectively.

Without loss of generality, we assume that the system begins execution at time 0. As stated earlier, the scheduler orders all requests according to their deadlines in ascending order and gives the requests with earlier deadlines higher priorities. When a periodic request has the same deadline as a sporadic request, the periodic request has higher priority. Ties among periodic requests and ties among sporadic requests are broken arbitrarily.

3 Acceptance Test for the First Sporadic Request

Our acceptance test relies on information on slack of each request in the system. It has the following two steps:

1. When a sporadic request S_i arrives, we first determine the amount of slack available before its deadline D_i. If the amount of slack is less than C_i, then S_i is rejected since there is not enough time to schedule S_i before its deadline.

2. If there is enough slack, then we proceed to check whether accepting S_i would cause any request in the system whose deadline is after D_i to miss its deadline. S_i is accepted if no other deadline is missed, otherwise it is rejected.

Preprocessing

Before the system begins execution, the scheduler computes for each periodic request $R_i(r_i, c_i, d_i)$ the slack δ_i, which is the maximum amount of time available before d_i to execute sporadic requests without causing R_i to miss its deadline. Let $\mathbf{H}_i$ denote the set of all periodic requests whose deadlines are before d_i, and let $\mathbf{L}_i$ denote the the set of all periodic requests whose deadlines are

after d_i. Before the execution of any requests begins,

$$\delta_i = d_i - c_i - \sum_{R_j \in \mathbf{H}_i} c_j = \delta_{i-1} + d_i - d_{i-1} - c_i. \tag{1}$$

The values of δ_i for $1 \leq i \leq N$ are stored in a slack table. This table can be computed in a forward scan of the periodic requests in $O(N)$ time.

Let $\omega_{i,j}$ denote the minimum slack among all periodic requests whose deadlines fall between the deadlines d_i and d_j of R_i and R_j inclusive, i.e.,

$$\omega_{i,j} = \min_{\forall k,\ d_i \leq d_k \leq d_j} (\delta_k) \tag{2}$$

$\omega_{i,j}$ can be computed and recorded in $O(N^2)$ time and requires $O(N^2)$ space. By doing so, we trade off space for time as the $\omega_{i,j}$'s allow us to easily determine which periodic requests are the most affected if any sporadic request is accepted.

For example, suppose that there are three requests in $\mathbf{T}$: $R_1(0,1,3)$, $R_2(0,1,5)$, and $R_3(0,3,6)$. Their slacks are: $\delta_1 = 3 - 1 = 2$, $\delta_2 = 5 - 1 - 1 = 3$ and $\delta_3 = 6 - 3 - 1 - 1 = 1$. We note that although $\delta_1 = 2$, we cannot use two units of time to execute sporadic requests before time 3. Doing so would make R_3 miss its deadline because δ_3 is only 1. This information is given by the values $\omega_{1,2} = \min(2,3) = 2$, $\omega_{1,3} = \min(2,3,1) = 1$ and $\omega_{2,3} = \min(3,1) = 1$.

Updating of Slack Information

During run-time, a number of events may occur to render the values in the slack table obsolete. We can get the correct values of the slacks quickly based on the following observation:

Observation 1. *At any time t, the slack δ_i of R_i is reduced from the value given by Eq. (1) by (a) processor idle time before t, (b) the total computation time of accepted sporadic requests with*

deadlines before d_i, (c) the total computation time of the executed portions of accepted sporadic requests with deadlines after d_i and (d) the total computation time of executed portions of periodic requests with deadlines after d_i.

Observation 1 is correct since the only other type of scheduling events that can happened before time t and is not included in the above statement is the execution of periodic requests with deadlines before d_i, but this has been accounted for in Eq. (1).

Because of Observation 1, we do not need to update the slack of every request in the system as requests become ready, preempted or complete. We only need to maintain the following run-time information in order to compute the updated slacks when we want to perform an acceptance test.

1. I: The cumulative amount of time the processor is idle before the current time t_c.
2. $f_i, 1 \leq i \leq n$: The computation time of the completed portion of the current request of each periodic task T_i.
3. $F_i, 1 \leq i \leq m$: The computation time of the completed portion of each yet-to-be-completed sporadic request S_i.
4. SP: The total amount of time spent executing all completed sporadic requests.

This information can be updated in $O(1)$ time when a request starts, completes or is preempted and when an acceptance test is done. At the start of each hyperperiod, the run-time variables I, SP and f_i, for $1 \leq i \leq n$ need to be re-initialized to zero. That is, if $\mathcal{H}$ is length of the the hyperperiod of the set $\mathbf{T}$, then the variables are re-initialized at time $x\mathcal{H}$, where x is an integer.

Acceptance Testing

To simplify the discussion, we first describe the acceptance test when the first sporadic request S_1 arrives and then generalize the test to deal with other sporadic requests in the next section.

We assume that any sporadic request S_k has a deadline D_k within the same hyperperiod that it arrives, i.e. $x\mathcal{H} \leq A_k < D_k \leq (x+1)\mathcal{H}$ for some integer x greater than or equal to zero. We will show how this assumption can be removed in Section 5.

When the first sporadic request arrives, the first step of the acceptance test determines whether there is enough slack to complete $S_1(t_c, C_1, D_1)$ before its deadline. Let R_p be the periodic request whose deadline d_p is closest to D_1 among all periodic requests whose deadlines are before D_1. Since the set of candidates for R_p consists of only one instance of each periodic task, R_p can be found in $O(n)$ time. To compute the total amount of time available to S_1, we need to consider the following factors:

1. From d_p to D_1, S_1 has the highest priority among all requests and hence this interval of length $(D_1 - d_p)$ is available to S_1.

2. R_p and the requests in $\mathbf{H}_p$ have earlier deadlines and hence higher priorities than S_1, and we need to subtract their worst-case computation times from the time available to S_1. This quantity is already accounted for in the expression for δ_p in Eq. (1).

3. Before S_1 arrives, part of the interval $(0, t_c]$ may have been spent in the following manner:

   (a) The processor may be idle for some time before t_c. The amount of idle time is given by I.

   (b) A portion of some periodic requests in $\mathbf{L}_p$ may have executed before t_c. Because only the current request of each periodic task can have a deadline later than d_p and yet start executing before t_c, we can determine this quantity using $\sum_{\forall l,\ D_1 < d_l} f_l$.

Hence, S_1 can complete before its deadline if and only if its slack Δ_1 is no less than zero, i.e.,

$$\Delta_1 = \delta_p + (D_1 - d_p) - I - C_1 - \sum_{\forall l,\ D_1 < d_l} f_l \geq 0. \tag{3}$$

Clearly, this decision can be done in $O(n)$ time.

The second step of the acceptance test determines whether accepting S_1 would cause any periodic request to miss its deadline. A request will miss its deadline if and only if its updated slack is less than zero. We note that only requests in $\mathbf{L}_p$ can be affected by the inclusion of S_1. The slack δ_k of each request R_k in $\mathbf{L}_p$ is reduced by the quantity $(I + C_1)$ since it is no longer available to R_k. In addition, we also need to subtract from δ_k the computation time of the portion of any request R_i with deadline d_i after d_k which has been executed before t_c because this time is not taken into account in the precomputed slack δ_k. Hence, S_1 can be accepted if and only if the following is true:

$$0 < \delta_k - I - C_1 - \sum_{\forall l,\ d_k < d_l} f_l, \qquad \text{for all} \quad R_k \in \mathbf{L}_p. \tag{4}$$

A naive way to implement this check is to determine whether the inequality in Eq. (4) is satisfied for every R_k in $\mathbf{L}_p$. This would require at least $O(N)$ time. To improve the $O(N)$ time complexity, we make the following observation:

Observation 2. *Consider the set of periodic requests* $\mathbf{V} = \{R_i, R_{i+1}, \cdots, R_j\}$ *and suppose that* $R_l \in \mathbf{V}$ *has the least amount of slack among all requests in* $\mathbf{V}$. *If the slack of every request in* $\mathbf{V}$ *is reduced by the same amount, then a request* $R_k \in \mathbf{V}$ *will miss its deadline only if* R_l *will also miss its deadline.*

This obviously valid observation tells us that we should partition all the periodic requests with deadlines after D_1 into $y + 1$ sets, for $0 \leq y \leq n$, each containing consecutive requests, such that the slacks of all the requests in a set are to be subtracted by the same amount. Hence, instead of examining every request to see whether it misses its deadline, we examine the one request with the minimum slack in each of the $y + 1$ sets. In this way, we reduce the time complexity of our

acceptance test.

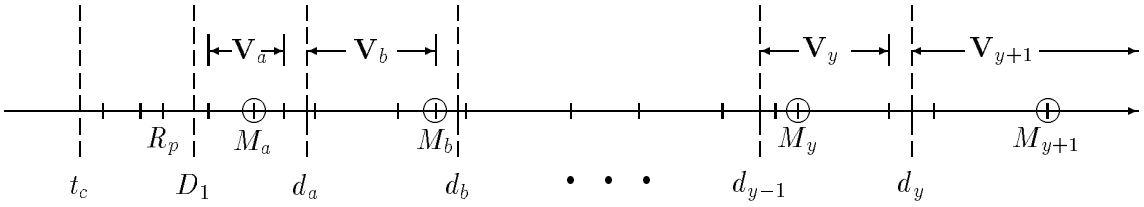

Figure 1: Partitioning the periodic requests after D_1.

To further explain this approach, we suppose that at time t_c, there are y current periodic requests $\{R_a, R_b, \cdots, R_y\}$, sorted in ascending order of their deadlines, that have deadlines after D_1 and have started executing before t_c. There are at most n such requests, and we can determine them (and their relative order) by examining the priority queue of ready requests that the system maintains. Figure 1 shows a timeline, and the ticks on the timeline represent the deadlines of some periodic requests. We partition all the periodic requests after D_1 into $(y+1)$ sets $\mathbf{V}_a, \mathbf{V}_b, \cdots, \mathbf{V}_{y+1}$ as shown in Figure 1. $\mathbf{V}_a$ contains the periodic requests $\{R_{p+1}, R_{p+2}, \cdots, R_{a-1}\}$; again, R_p is the periodic request with deadline closest to D_1 among all periodic requests with deadlines before D_1. $\mathbf{V}_b$ contains the periodic requests $\{R_a, R_{a+1}, \cdots, R_{b-1}\}$. The sets $\mathbf{V}_c$ to $\mathbf{V}_y$ are defined in a similar fashion as $\mathbf{V}_b$. The set $\mathbf{V}_{y+1}$ contains the periodic requests $\{R_y, R_{y+1}, \cdots, R_N\}$. The sets $\mathbf{V}_a, \mathbf{V}_b, \cdots, \mathbf{V}_{y+1}$ can be obtained in $O(n)$ time since $y < n$, all the requests in $\mathbf{T}$ are presorted and we know the index of every periodic request.

The circles on the timeline in Figure 1 denote the periodic requests with the minimum amount of slacks in $\mathbf{V}_a, \mathbf{V}_b, \cdots, \mathbf{V}_{y+1}$. We denote these requests by $M_a, M_b, \cdots, M_{y+1}$, respectively. The slack for each of these requests can be found in $O(1)$ time by examining the values of $\omega_{i,j}$'s that we have precomputed. Specifically, before the system begins to execute, the slack δ_{M_a} of M_a is equal to $\omega_{p+1,a-1}$, the slack δ_{M_b} of M_b is equal to $\omega_{a,b-1}$, and so on. We know from Observation

2 that accepting S_1 will not cause any requests in $\mathbf{V}_a$ to miss its deadline if $\omega_{p+1,a-1} - I - C_1 - \sum_{\forall l,\ d_a \leq d_l} f_l \geq 0$, will not cause any requests in $\mathbf{V}_b$ to miss its deadline if $\omega_{a,b-1} - I - C_1 - \sum_{\forall l,\ d_b \leq d_l} f_l \geq 0$, and so on for the sets $\mathbf{V}_c, \mathbf{V}_d, \cdots \mathbf{V}_{y+1}$. Although the amounts to be deducted from the slack of each $M_i, a \leq i \leq y+1$ differs, they can be computed recursively, e.g.,

$$\sum_{\forall l,\ d_a \leq d_l} f_l = f_a + \sum_{\forall l,\ d_b \leq d_l} f_l = f_a + f_b + \sum_{\forall l,\ d_c \leq d_l} f_l = \cdots.$$

In this way, we can determine in $O(n)$ time whether any of the requests after D_1 will miss its deadline.

4 Acceptance Test for Multiple Sporadic Requests

We now modify the acceptance test to handle subsequent sporadic requests. When $S_k(t_c, C_k, D_k)$ arrives, some previously accepted sporadic requests may not be completed. To determine whether there is enough slack to schedule S_k, we modify the left hand side of the inequality (3) to account for three additional factors:

1. SP: This amount of time has been consumed by sporadic requests which have completed.
2. $\sum_{\forall i,\ D_i < D_k}(C_i)$: This amount of time is reserved for sporadic requests which have not been completed and have deadlines before D_k.
3. $\sum_{\forall i,\ D_k < D_i}(F_i)$: This amount of time has been consumed by sporadic requests that have not completed and have deadlines after D_k.

SP, C_i, and F_i are determined at run-time. The slack Δ_k for S_k at t_c is given by:

$$\Delta_k = \delta_p + (D_k - d_p) - I - C_k - SP - \sum_{\forall i,\ D_k < d_i} f_i - \sum_{\forall i,\ D_i < D_k} C_i - \sum_{\forall i,\ D_k < D_i} F_i. \qquad (5)$$

where d_p and δ_p are the deadline and initial slack of the periodic request R_p whose deadline is closest to D_k among all requests whose deadlines are earlier than D_k. S_k will meet its deadline if and only if its slack Δ_k is greater than or equal to zero. This computation can be performed in $O(m+n)$ time. If S_k is accepted, its slack Δ_k is stored for later use.

If S_k can meet its deadline, the next step is to determine whether accepting S_k will cause other requests to miss their deadlines. All periodic and sporadic requests with deadlines earlier than D_k will not be affected by the acceptance of S_k. The slack of a previously accepted but uncompleted sporadic request S_l whose deadline is after D_k is decreased by C_k if S_k is accepted. Hence we subtract C_k from Δ_l if S_l is accepted. S_l will miss its deadline if and only if $(\Delta_l - C_k < 0)$. No other computation is needed for S_l because when we accepted S_l and computed its slack Δ_l according to Eq.(5), we have already accounted for all the factors. Since S_l's acceptance, the processor cannot be idle or executes requests with deadlines after D_l that can further reduce Δ_l as long as S_l is not completed. Hence, to determine whether any previously accepted sporadic request will miss its deadline takes $O(m)$ time.

To determine whether any periodic request will miss its deadline, we follow a similar procedure for the case of the first sporadic request. In addition to the periodic requests that have deadlines later than D_k but have already started executing when S_K arrives, we also need to consider the sporadic requests that have deadlines later than D_k and have already started executing. Together, these requests partition all periodic requests with deadlines after D_k into y disjoint sets, $\mathbf{V}_1, \mathbf{V}_2, \cdots, \mathbf{V}_y$, where $y \leq n+m$. For each set $\mathbf{V}_i$, let X_i and Y_i be the requests that have the earliest deadline and latest deadline among all requests in $\mathbf{V}_i$, respectively. By definition of $\omega_{i,j}$'s, the smallest slack of all periodic requests in each set $\mathbf{V}_i$ is equal to ω_{X_i,Y_i}. To determine whether any of the periodic requests in $\mathbf{V}_i$ will miss its deadline, we subtract from ω_{X_i,Y_i} the cumulative idle time I, the total computation time of the executed portions of all requests that

have deadlines after Y_i's deadline (i.e., $\sum_{\forall l, d_{Y_i} < d_l} f_l + \sum_{\forall l, d_{Y_i} < D_l} F_l$), and the total computation time of the all sporadic requests which have completed or have deadlines earlier than X_i's deadline (i.e., $SP + \sum_{\forall l, D_l < d_{X_i}} C_l$). The resultant quantity is the updated slack δ'_{M_i} of the periodic request M_i with the smallest slack in $\mathbf{V}_i$. S_k will not cause any request to miss its deadline if and only if the updated minimum slack is greater than or equal to zero for all $\mathbf{V}_i$'s, i.e.,

$$\delta'_{M_i} = \omega_{X_i, Y_i} - I - SP - \sum_{\forall l, d_{Y_i} < d_l} f_l - \sum_{\forall l, d_{Y_i} < D_l} F_l - \sum_{\forall l, D_l < d_{X_i}} C_l \geq 0 \tag{6}$$

for all $\mathbf{V}_i, 1 \leq i \leq y$. If S_k is accepted, its slack Δ_k and the slack Δ_l of every sporadic request S_l with deadline after D_k are updated for future use. The slack for any periodic request is not updated with the value computed in Eq. (6) because it is much faster and easier to recompute them when needed rather than to keep track of what has or has not been updated.

As in the case of accepting the first sporadic request, although the amounts to be deducted from each $\mathbf{V}_i$ is different, these amounts can be computed recursively. Hence, this computation can be performed in $O(n + m)$ time. This time complexity can be achieved as we do not need to sort the current periodic requests and uncompleted sporadic requests in ascending order of their deadlines for the sake of acceptance test. This assumption is reasonable since the system already maintains a sorted list in its EDF priority queue of all requests awaiting execution. The entire acceptance test for S_k is summarized by the pseudo code in Figure 2. It takes $O(n+m)$ time. The optimality of our acceptance test is stated in the following theorem. The proof of this theorem follows trivially from the discussion given above.

Theorem 1. *Upon the arrival of a sporadic request S_k, the acceptance test described above accept S_k if and only if it is possible to schedule S_k without causing any periodic requests or previously accepted sporadic requests to miss its deadline.*

Input

- $\delta_i, 1 \leq i \leq N$: Precomputed initial slack of all periodic requests in a hyperperiod of length $\mathcal{H}$.
- $\omega_{i,j}, 1 \leq i, j \leq N$: Precomputed minimum slack of all requests with deadlines in $[d_i, d_j]$

Request Scheduling and Acceptance Test

- Initialize $I; f_i, 1 \leq i \leq n$; and SP to zero at time $t = 0, \mathcal{H}, 2\mathcal{H}, \cdots$
- Execute the requests (periodic and sporadic) in the system on the EDF basis
- Whenever a request starts execution, completes or is preempted, update I, f_i, F_i, or SP appropriately
- When a sporadic request $S_k(t_c, C_k, D_k)$ arrives at t_c
  - 0. Update I, f_i, F_i, or SP appropriately
  - 1.a. Find the periodic request R_p whose deadline is closest to D_k among all requests whose deadlines are before D_k
  - b. Compute Δ_k according to Eq. (5)
  - c. If $(\Delta_k < 0)$ Then REJECT S_k and goto step 3
  - 2.a. For (all uncompleted sporadic request S_l with deadline after D_k) Do
    - i. If $(\Delta_l - C_k < 0)$ Then REJECT S_k and goto step 3
  - b. Partitions the periodic requests with deadlines after D_k into disjoint sets $\mathbf{V}_1, \mathbf{V}_2, \cdots, \mathbf{V}_y$ as described in Section 4.
  - c. For (each $\mathbf{V}_i$) do
    - i. Compute the updated minimum slack δ'_{M_i} in $\mathbf{V}_i$ according to Eq. (6)
    - ii. If $(\delta'_{M_i} < 0)$ Then REJECT S_k and goto step 3
  - d. ACCEPT S_k
  - 3. Continue to execute the request that was executing when S_k arrives

Figure 2: Pseudo code of request scheduling and acceptance test.

5 Extensions

The acceptance test as presented in the previous sections works even if periodic requests have jittered ready times because it does not make use of the values of ready times to compute the slacks. We now present two extensions to handle sporadic requests whose arrivals and deadlines span more than one hyperperiod and to account for variation in the computation time.

Sporadic Requests Spanning More Than One Hyperperiod

Suppose a sporadic request $S_k(t_c, C_k, D_k)$ arrives in the x-th hyperperiod and has deadline which falls in the y-th hyperperiod, i.e. $x\mathcal{H} \leq t_c < y\mathcal{H} \leq D_k \leq (y+1)\mathcal{H}$. The amount Δ_k of time available to schedule S_k is the sum of the following three terms:

1. Amount in the x-th hyperperiod. This is given by δ_N minus the amount of idle processor time and the amount of time allocated to sporadic requests with deadlines in the x-th hyperperiod and to the sporadic requests with deadlines after D_k but have started before t_c.

2. Amount from the $(x+1)$-th to the $(y-1)$-th hyperperiods. This is equal to $(y-x-1)\delta_N$ minus the amount of time allocated to sporadic requests with deadlines within these hyperperiods.

3. Amount in the y-th hyperperiod. If R_p is the request whose deadline is closest to D_k among all requests with deadline before D_k in the y-th hyperperiod, then this amount is given by $(\delta_p + D_k - d_p)$ minus the amount of time allocated to sporadic requests with deadlines earlier than D_k.

If C_k is less than the amount computed above, then S_k is schedulable. To check whether S_k can now be accepted, we can use the exact procedure described in the previous section to determine whether any periodic or sporadic request with deadlines after D_k will miss its deadline. In this step, we only need to check the periodic requests in the y-th hyperperiod.

Reclaiming Unused Computation Time

Our method of computing slack times makes use of the worst-case computation times. It ensures that all requests will meet their deadlines in the worst-case. However, the actual computation time of a request may be less than its worst-case computation time. We can use a mechanism

to reclaim these unused computation time so that they can be made available to schedule more sporadic requests. For this purpose, we maintain the following information during run-time:

1. U: This is the cumulative unused computation time for all periodic and sporadic requests with deadlines before the current time t_c.

2. $u_i, 1 \leq i \leq n$: This is the amount of unused computation time of the current periodic request of T_i.

3. $v_i, 1 \leq i \leq m$: This is the amount of unused computation time for the sporadic request S_i whose deadline is after t_c.

U, u_i or v_i are updated each time a request completes. When we update the slack of any periodic or sporadic request, we simply add U and all u_i's and v_j's for all requests that have deadlines before that particular periodic or sporadic request.

6 Conclusion

In this paper, we have presented an optimal acceptance test for determining whether newly arrived sporadic requests can be scheduled with existing periodic and sporadic requests in the system using the earliest-deadline-first algorithm with all requests meeting their deadlines. Our algorithm has linear time complexity which is a significant improvement over previous algorithms which are pseudo-polynomial. Unlike previous algorithms which maintain complicated data structures or precomputed schedule, the data structures maintained by our algorithm are simple and easy to maintain.

We are currently applying variations of our algorithm to make the earliest-deadline-first algorithm more deterministic during overload conditions. In our model, the system has a mixture of periodic requests and both hard and soft deadline sporadic requests. In addition, each request has

an *importance* value associated with it. During overload conditions, requests will miss their deadlines in order of (increasing) importance. The data structures and algorithms are being modified as a means to detect when an overload condition occurs and to decide how to deal with them.

References

[1] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment," in *J. Assoc. Comput. Mach.*, vol. 20(1), pp. 46–61, 1973.

[2] H. Chetto and M. Chetto, "Some Results of the Earliest Deadline Scheduling Algorithm," *IEEE Transactions on Software Engineering*, vol. 15(10), pp. 1261–1269, Oct. 1989.

[3] M. Silly, H. Chetto, and N. Elyounsi, "An Optimal Algorithm for Guaranteeing Sporadic Tasks in Hard Real-Time Systems," in *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pp. 578–585, 1990.

[4] K. Schwan and H. Zhou, "Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads," *IEEE Transactions on Software Engineering*, vol. 18(8), pp. 736–748, Aug. 1992.

[5] C. McElhone, "Adapting and Evaluating Algorithms for Dynamic Schedulability Testing," Tech. Rep. YCS 225, Department of Computer Science, University of York, England, 1994.

[6] M. L. Dertouzos, "Control Robotics: The Procedural Control of Physical Processes," in *Proceedings of the IFIP Congress*, 1974.