Robust Aperiodic Scheduling under Dynamic Priority Systems

Marco Spuri Giorgio Buttazzo Fabrizio Sensini

Scuola Superiore S. Anna
via Carducci, 40 - 56100 Pisa - Italy
giorgio@sssup1.sssup.it, spuri@sssup2.sssup.it

Abstract

When hard periodic and firm aperiodic tasks are jointly scheduled in the same system, the processor workload can vary according to the arrival times of aperiodic requests. In order to guarantee the schedulability of the periodic task set, in overload conditions some aperiodic tasks must be rejected.

In this paper we propose a technique that, in overload conditions, adds robustness to the joint scheduling of periodic and aperiodic tasks in systems with dynamic priorities. Our technique is based on an aperiodic server, called Total Bandwidth server, already proven effective in a previous work. Here the algorithm is first extended to efficiently handle firm aperiodic tasks and then integrated with a robust guarantee mechanism that allows to achieve graceful degradation in case of transient overloads. Extensive simulations show that the proposed new algorithm is effective in all workload conditions.

1 Introduction

Real-time systems must be able to handle not only periodic tasks, but also aperiodic tasks, that is, tasks with irregular arrival times. Periodic tasks are generally used to implement activities such as sensory acquisition or control loops, which need to be executed at constant rates to insure system stability. Hence, periodic tasks often have hard deadlines that must be met under all anticipated circumstances.

On the other hand, aperiodic tasks are usually employed to implement less demanding and less critical activities. For this reason they can have soft deadlines (a deadline is soft if when missed does not compromise the security of the system), firm deadlines (a deadline is firm if the execution of the task is useful for the system only if completed within the deadline) or no deadlines at all.

When aperiodic tasks do not have deadlines, the goal of the system is to minimize the average response time of their instances. A common approach that does not jeopardize the schedulability of the hard tasks is to introduce in the system a special purpose process called *server*, whose computation time, or better *capacity*, is used to server the aperiodic requests. The server is usually scheduled by a specific algorithm designed in such a way that timing faults do not occur within the critical task set and at the same time the cpu is allocated to the server as soon as possible, in order to improve the aperiodic response time.

A number of algorithms that solve this problem in fixed priority systems can be found in the literature [5, 9, 10, 13]. Less attention has been dedicated to the same problem in the context of dynamic priority systems. Only recently a few works have appeared [7, 15].

Much less attention, in our opinion, has been devoted to systems with firm aperiodic tasks. In this case the problem is more difficult, since tasks that are going to be late must be rejected in advance, in order not to waste time in useless computations, try to guarantee as much work as we can, and not jeopardize the schedule of critical tasks.

A similar framework has been faced within the Spring system [14], even though here the main feature was to maximize the number of completed tasks and not the overall value of the system. In our model we associate an importance value to each aperiodic task: the value is gained only if the task is completed in time. In this framework, the goal of the scheduling algorithm is to maximize the value of the system.

The solution to this problem is not trivial, since when there are aperiodic tasks whose arrival times are not known a priori, the load of the system can vary greatly from time to time. In particular, there can be transient overloads, in which not all tasks can be completed in time. In these conditions the algorithm has to make some choices in order to establish who is

1052-8725/95 $04.00 © 1995 IEEE

going to miss, who is going to complete. Depending on these choices the overall value of the system can consequently vary.

In this paper we introduce a mechanism which provides a good solution of the problem under earliest deadline scheduling. The mechanism is based on an improved version of the Total Bandwidth server [15, 16], that we have extended with a robust strategy which improves the performance during transient overloads.

With respect to the original formulation, the new TB server includes two novel features: the reclaiming of unused computation time and the possibility of adopting any preemptive scheduling policy for the aperiodic task set (the older version was non-preemptive). The improvement due to the new features is illustrated in the discussion of our simulations.

The strategy adopted to add robustness had been already proven effective in a previous work [3]. It adopts a planning EDF-based strategy, together with rejection and recovery policies. Also in this new framework, the strategy introduced graceful degradation and effectiveness under any load condition.

2 Assumptions and Terminology

In the definition of our algorithms we will consider the following assumptions:

- all periodic tasks $\tau_i : i = 1, ..., n$ have hard deadlines;
- each periodic task τ_i has a constant period T_i and a constant worst case execution time C_i, which is considered to be known, as it can be derived by a static analysis of the source code;
- the arrival time of each aperiodic task is <u>unknown</u>;
- the worst case execution time of each aperiodic task is considered to be known at its arrival time;
- all aperiodic tasks have firm deadlines and can be rejected (for the new definition of the improved TB server we will temporarily assume aperiodic tasks without deadlines).

For the sake of clarity, all properties of the proposed algorithms will be proven under the above assumptions. However, they can easily be extended to handle less restrictive assumptions such as deadlines different from periods and inclusion of sporadic tasks.

The notation used throughout the paper is the following:

J denotes a set of active aperiodic tasks J_i ordered by increasing deadline, J_1 being the task with the shortest absolute deadline.

r_i denotes the arrival time of task J_i, i.e., the time at which the task is activated and becomes ready to execute.

C_i denotes the maximum computation time of task J_i, i.e., the worst case execution time (*wcet*) needed for the processor to execute task J_i without interruption.

$\bar{C}_i$ denotes the actual computation time of task J_i.

d_i denotes the absolute deadline of task J_i, i.e., the time before which the task should complete its execution in order to be useful for the system.

m_i denotes the deadline tolerance of task J_i, i.e., the maximum time that task J_i may execute after its deadline, and still produce a valid result.

v_i denotes the task value, i.e, the relative importance of task J_i with respect to the other tasks in the set.

f_i denotes the finishing time of task J_i, i.e., the time at which task J_i completes its execution and leaves the system.

E_i denotes the exceeding time, i.e., the possible lateness of task J_i in case of overload.

In our model, an aperiodic task J_i is thus completely characterized by specifying its worst case execution time C_i, its deadline d_i, its deadline tolerance m_i, and its value v_i. Within this framework tasks are scheduled based on the deadline assigned by a server, guaranteed based on C_i, d_i, m_i, and rejected based on v_i.

Throughout our discussion, we will assume that the set of periodic tasks is scheduled on a uniprocessor system by the Earliest Deadline First (EDF) scheduling algorithm. Similarly, the aperiodic tasks are scheduled within a TB server, described in the following section, also with earliest deadline policy. Groups of tasks with precedence constraints can also be handled by EDF by modifying their deadlines and release times so that both deadlines and precedence relations are met [1, 4].

3 The Total Bandwidth Algorithm

In [15] Spuri and Buttazzo proposed several algorithms to joint schedule soft aperiodic tasks and hard periodic tasks in earliest deadline scheduled systems.

Among these algorithms, the Total Bandwidth server showed the best performance/cost ratio. In this section, the algorithm is briefly recalled and later extended with new features that further improve its behaviour. The new formulation, also useful in a framework with firm aperiodic tasks, will then be integrated in Section 4 with a mechanism that adds robustness to the algorithm with respect to transient overloads in the system.

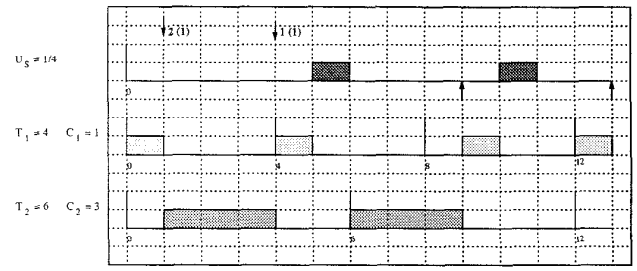


Figure 1: Total Bandwidth Server example.

3.1 The Original Formulation

The name of the Total Bandwidth server comes from the fact that, each time an aperiodic request enters the system, the total bandwidth (in terms of cpu execution time) of the server, whenever possible, is immediately assigned to it. This is done by simply assigning a suitable deadline to the request and to schedule it according to the EDF algorithm together with the periodic tasks in the system. The assignment of the deadline must be done in such a way that on one hand it is the shortest possible to improve the aperiodic responsiveness, but on the other hand it must not jeopardize the schedule of periodic tasks.

The definition of the TB server is the following. When the k-th aperiodic request arrives at time $t = r_k$, it receives a deadline

$$d_k = \max(r_k, d_{k-1}) + \frac{C_k}{U_S},$$

where C_k is the maximum execution time of the request and U_S is the server utilization factor (*i.e.*, its bandwidth). By definition $d_0 = 0$. The request is then inserted into the ready queue of the system and scheduled by EDF, as any periodic instance.

Note that we can keep track of the bandwidth already assigned to other requests by simply taking the maximum between r_k and d_{k-1}. Intuitively, the assignment of the deadlines is such that in each interval of time the ratio allocated by EDF to the aperiodic requests never exceeds the server utilization U_S, that is, the processor utilization of the aperiodic tasks is at most U_S. This is formally proven in [16], where the definition and the formal analysis of this algorithm, as well as several others, can be found. Hence, to state the schedulability of a task set, it is sufficient to add the utilization of the server to that of the other critical tasks, and verify that $U_P + U_S \leq 1$.

In Figure 1, an example of schedule produced with a TB server is shown. The first aperiodic request, arrived at time $t = 1$, is serviced (*i.e.*, scheduled) with deadline $d_1 = r_1 + \frac{C_1}{U_S} = 1 + \frac{2}{0.25} = 9$. Since there are more urgent periodic instances in the system, the aperiodic activity is executed at time $t = 5$. Similarly, the second request receives the deadline $d_2 = \max(r_2, 9) + \frac{C_2}{U_S} = 13$. Also in this case the request is not serviced immediately. Instead, it is serviced only at time $t = 10$. Note that the delay occurs in spite of the early completion of the first aperiodic request, whose actual computation time is 1 (indicated in parentheses in figure) instead of 2. Later we will see how the response time is improved with the new formulation of the TB server.

In spite of its simplicity, the TB server shows one of the best performance results among the several servers described in [16]. In this paper we have investigated efficient algorithms to solve the problem of the joint scheduling of both hard periodic and soft aperiodic tasks. One of the algorithms, EDL, was shown to be optimal. In normal conditions the TB server showed a performance comparable to that of EDL, and was the best among the practical algorithms. Considering it has a very simple implementation and low run-time overhead, it is an ideal candidate for actual systems. For example, we have chosen TB as the algorithm for aperiodic scheduling in the HARTIK kernel [2].

3.2 Adding Resource Reclaiming

In the original formulation, the behaviour of the TB server strictly depends on the estimated maximum execution time of each aperiodic task, since the deadlines are assigned based on this value. This may be a drawback if the value is overestimated and it is much greater than the mean execution time. If this happens, the deadlines assigned by the server to aperiodic requests are farther than necessary, and this may delay their execution.

Of course, the algorithm cannot be clairvoyant, in the sense that it cannot predict the actual execution time of any aperiodic task. However, whenever a task completes earlier, the actual execution time can be

used to keep track of the actual processor bandwidth taken so far by the aperiodic load. Hence, the main idea behind the proposed reclaiming technique is to correct the assigned deadline as follows. Whenever a request completes earlier, its actual execution time is used to compute the deadline that could have been assigned to it if its execution time had been known in advance. This value is then used to compute the deadline for the next request. In the following, $\bar{d}_i$ denotes the "corrected" deadline of the i-th task.

More formally, let $\bar{d}_0 = 0$ and $f_0 = 0$ by definition. The server keeps a queue of aperiodic instances ready to execute (we do not make any assumption on the particular policy used to sort the aperiodic queue). At any time, only the first task, referred to as *active*, has a deadline assigned by the server and is then scheduled by the system. In particular, the i-th task to be executed receives a deadline d'_i equal to:

$$d'_i = \bar{r}_i + \frac{C_i}{U_S},$$

where C_i is the maximum execution time of the task and U_S is the server utilization factor. $\bar{r}_i$ is the "corrected" release time of the task and is computed as:

$$\bar{r}_i = \max(r_i, \bar{d}_{i-1}, f_{i-1}),$$

where r_i is the actual release time of the task, that is, its arrival time, $\bar{d}_{i-1}$ and f_{i-1} are the corrected deadline and the completion time of the previous task, respectively. At the task completion, the corrected deadline $\bar{d}_i$ is computed as:

$$\bar{d}_i = \bar{r}_i + \frac{\bar{C}_i}{U_S},$$

where $\bar{C}_i$ is the actual execution time of the task. Being $\bar{C}_i \leq C_i$, we have $\bar{d}_i \leq d'_i$, that is, we try to reclaim the unused computation time by assigning a shorter deadline to the next request.

In Figure 2 the same example of Figure 1 is handled with the new formulation of the TB server. As shown, nothing changes for the first aperiodic task. However, due to its early completion, the computation of the deadline assigned to the second one takes advantage of this. The new value of this deadline is 10, instead of 13, the value computed with the older formulation. In this case the task can be executed immediately and the response time is then considerably improved.

To prove the schedulability of the TB server with this new formulation we first show that the actual aperiodic processor utilization cannot exceed U_S, and then that the overall utilization can be up to 100%.

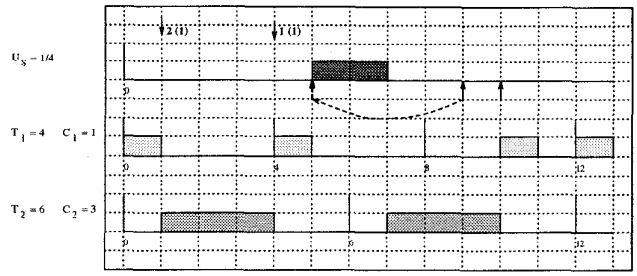


Figure 2: Example with the new formulation of the Total Bandwidth Server.

Lemma 1 *In each interval of time $[t_1, t_2]$, if $\bar{C}_{ape}$ is the total execution time actually demanded by aperiodic requests arrived at t_1 or later and served with deadlines less than or equal to t_2, then*

$$\bar{C}_{ape} \leq (t_2 - t_1)U_S.$$

Proof. By definition

$$\bar{C}_{ape} = \sum_{t_1 \leq r_k, d'_k \leq t_2} \bar{C}_k.$$

Since the index k indicates the order of execution, there must be two indexes k_1 and k_2 such that

$$\sum_{t_1 \leq r_k, d'_k \leq t_2} \bar{C}_k \leq \sum_{k=k_1}^{k_2} \bar{C}_k.$$

It follows that

$$\bar{C}_{ape} \leq \sum_{k=k_1}^{k_2} \bar{C}_k = \sum_{k=k_1}^{k_2} (\bar{d}_k - \bar{r}_k)U_S = U_S \sum_{k=k_1}^{k_2} (\bar{d}_k - \bar{r}_k),$$

which, since $\bar{d}_{k-1} \leq \bar{r}_k$, becomes

$$\bar{C}_{ape} \leq U_S(\bar{d}_{k_2} - \bar{r}_{k_1}).$$

Finally, being $\bar{d}_{k_2} \leq d'_{k_2} \leq t_2$ and $\bar{r}_{k_1} \geq r_{k_1} \geq t_1$, we have

$$\bar{C}_{ape} \leq U_S(t_2 - t_1).$$

□

We can now prove the claimed result.

Theorem 1 *Given a set of n periodic tasks with processor utilization U_P and a TB server with processor utilization U_S, the whole set is feasibly scheduled* if *and* only if

$$U_P + U_S \leq 1.$$

Proof. "If". Suppose there is an overflow at time t. The overflow must be preceded by a period of continuous utilization of the processor. Furthermore, from a certain point t' on, only instances of tasks (periodic or aperiodic) ready at t' or later and having deadlines less than or equal to t are run. Let $\bar{C}$ be the total execution time demanded by these instances. Since there is an overflow at time t, we must have

$$t - t' < \bar{C}.$$

We also know that

$$\begin{aligned} \bar{C} &\leq \sum_{i=1}^{n} \left\lfloor \frac{t-t'}{T_i} \right\rfloor C_i + \bar{C}_{ape} \\ &\leq \sum_{i=1}^{n} \frac{t-t'}{T_i} C_i + (t-t')U_S \\ &\leq (t-t')(U_P + U_S). \end{aligned}$$

It follows that

$$U_P + U_S > 1,$$

a contradiction.

"Only If". If an aperiodic request enters the system periodically, say each $T_S > 0$ units of time, and has execution time $C_S = \bar{C}_S = T_S U_S$, the server behaves exactly as a periodic task with period T_S and execution time C_S. Being the processor utilization $U = U_P + U_S$, by Theorem 7 of [11] we can conclude that $U_P + U_S \leq 1$. □

3.3 A Preemptive Implementation

In the description of the TB server we have implicitly assumed that the aperiodic requests are serviced non-preemptively. So far, we have only left unspecified the policy for sorting the queue of the pending requests, with the implicit hypothesis that the request currently serviced cannot be preempted. However, a preemptive implementation can be easily built as follows. When a new request is issued, it is inserted into the server ready queue. If it must preempt the current active request, we brake this one into two requests. One is the part which has already run and the other one is the part to be still executed. For the first one we behave like for a normal aperiodic completion, that is, we compute its corrected deadline, which will then be used to assign a deadline to the new active request. The second one, whose maximum execution time is $C - \bar{C}$, is kept in the server ready queue and treated like a new request.

In the rest of this paper, we will always refer to this new formulation with resource reclaiming and preemption features.

4 Robust Aperiodic Scheduling

In this section we first extend the TB server technique to deal with firm aperiodic requests and then we introduce a robust guarantee mechanism capable of achieving graceful degradation.

The extension of the TB server to firm aperiodic tasks is straightforward. As illustrated in Section 3.2, the TB server assigns to each aperiodic request a deadline d' with which the request is then scheduled. The earliest deadline scheduling mechanism guarantees the completion of the task within this deadline. This fact can be used in the following way: assuming d is the actual deadline of the task, if $d' \leq d$ the task is guaranteed, otherwise it is rejected. The approach will be better described in a later section.

Although useful, this form of extension to handle firm aperiodic tasks is not yet enough. In fact, the arrival rate of aperiodic tasks may vary during system life. Consequently, the load of the system can change significantly and the system can experience a number of transient overloads.

If the scheduling algorithm is not able to deal with these situations, we may have undesired results, such as the so called *domino effect*, in which a missed deadline causes a series of subsequent deadlines to be also missed.

There are two alternatives to solve the problem. One is to assign an unnecessary large bandwidth to the server, in order to limit the occurrences of overloads. In this way we normally waste a lot of processor time, when the actual load does not reach large values. The second alternative is to introduce overload awareness in our scheduling algorithm. This is the approach we have followed and we will describe in what follows.

An effective strategy explicitly developed to handle overload conditions is the RED algorithm [3]. The main idea of RED is to use the Earliest Deadline First algorithm in a planning mode, so as to predict deadline misses and depict the size of the overload, its duration and its overall impact on the system. By using this information, the algorithm is able to achieve optimal performance in normal conditions and graceful degradation in overload conditions. A better description of the RED strategy will be given in the following section.

Unfortunately, the RED algorithm cannot be easily used for the joint scheduling of hard periodic and firm aperiodic tasks. However, we can use the RED strategy within a TB server. In this way we can achieve the goal of optimal joint scheduling in normal conditions and robustness during transient overloads.

4.1 The RED Algorithm

Although the EDF algorithm has been shown to be optimal under many different conditions [6, 8], if overload occurs, tasks may miss deadlines in an unpredictable manner, and in the worst case, the performance of the system can approach zero effective throughput [12].

To increase flexibility in expressing time constraints and to enhance the performance of the system in overload conditions, Buttazzo and Stankovic [3] proposed to separate deadline and importance by introducing two additional parameters into the task model: a task *value*, which reflects the importance of the task in the set, and a deadline *tolerance*, which is the amount of time by which a specific task is permitted to be late.

The algorithm they proposed, called RED, tries to increase the cumulative value in overload conditions by using a rejection strategy that removes tasks based on their values. The rejection policy searches for a subset J^* of least value tasks to reject in order to make the current set schedulable. If J^* is returned empty, then the overload cannot be recovered, and the newly arrived task is not accepted.

The concept of deadline tolerance comes from the fact that in many real applications, such as robotics, the deadline timing semantics is more flexible than scheduling theory generally permits. For example, most scheduling algorithms and accompanying theory treat the deadline as an absolute quantity. However, it is often acceptable for a task to continue to execute and produce an output even if it is late – but not too late.

The general framework in which RED operates is illustrated in Figure 3. Notice that, if a task cannot be guaranteed by the RED algorithm at its arrival time, it is not removed forever, but it is temporarily rejected in a queue of non guaranteed tasks, called *Reject Queue*, ordered by decreasing values, to give priority to the most important tasks. As soon as the running task completes its execution before its worst case finishing time, a recovery strategy tries to reaccept the highest value task in the Reject Queue having positive laxity.

All rejected tasks with negative laxity are removed from the system, and inserted in another queue, called *Miss Queue*, containing all late tasks, whereas all tasks that complete within their timing constraints are inserted in a queue of regularly terminated jobs, called *Term Queue*.

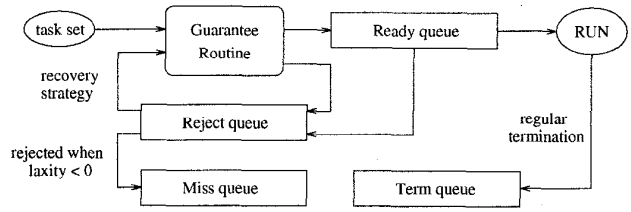


Figure 3: RED Scheduling Block Diagram.

4.2 Implementation of the RTB Server

The RED algorithm represents a useful choice to add robustness to our deadline scheduled system. However, if we need a more general framework in which both hard periodic tasks and soft aperiodic tasks are managed, RED is no longer applicable. On the other hand, a TB server is designed to handle a similar situation, with the major drawback of treating only soft tasks without deadlines. The two techniques can be usefully combined to achieve robustness in the aperiodic load scheduling, still guaranteeing at any time the feasibility of the critical load schedule.

The basic idea is very simple. By definition, the TB server assigns a deadline to each aperiodic task and schedules the task on the basis of this deadline, thus guaranteeing its completion by that time. Hence, we can simply compare this value with the actual deadline of the task: if it is less than or equal to it the task is guaranteed, otherwise it isn't. Of course we can also easily introduce a tolerance by adding it to the actual deadline.

In particular, at each new arrival of a task J_a the algorithm RTB, illustrated in Figure 4, is executed.

The algorithm starts by initializing t at the current time value and E, the maximum exceeding time, to 0. The new task J_a is then inserted into the queue of the server. If it becomes the new head of the queue a preemption occurs, thus, as remarked in Section 3.3, the event is treated like a termination for the previous task which has executed for a time $\bar{C}$ (if the task has not actually completed it is still kept in the queue with its maximum remaining computation time). This value, as well as $\bar{r}$, is used to compute the corrected deadline $\bar{d}$. Note that the values of $\bar{r}$ and $\bar{d}$ are also updated by the routines that dispatch and terminate the aperiodic tasks.

The maximum between t and $\bar{d}$ is then used to initialize the computation of the deadlines assigned by the server to the ready aperiodic tasks. During the computation, the maximum exceeding time is also determined. If this value is zero the task is accepted

```
Algorithm RTB(J, J_a)

begin
        t = current_time();
        E = 0; /* Maximum Exceeding Time */
        J' = J ∪ {J_a}; /* Insert J_a in the
                          ordered task list */
        k = position of J_a in the task set J';
        if (k = 1) and (|J'| > 1) then
                d̄ = r̄ + C̄/U_S;
        d'_0 = max(t, d̄);

        for i = k to |J'| do {
                d'_i = d'_{i-1} + C_i/U_S;
                if (d_i - d'_i + m_i < -E) then
                        E = -(d_i - d'_i + m_i);
        }

        if (E = 0) then return ("Guaranteed");
        else {
                J* = set of least value tasks
                selected by the rejection policy;
                if (J* is not empty) then {
                        reject all task in J*;
                        return ("Guaranteed");
                }
                else return ("Not Guaranteed");
        }
end
```

Figure 4: The RTB Algorithm

as guaranteed, otherwise a policy is used to possibly select one or more tasks to be rejected. If the set returned by the rejection policy is not empty the new task is guaranteed, otherwise it is rejected. Note that in case of rejection the task is not yet refused by the system, but kept in a reject queue until either it is recovered or its laxity becomes negative. There is in fact the possibility that by reclaiming the unused computation time of some other task we may recover it at a later time.

4.3 Rejection Strategy

If a new task cannot be guaranteed, the RTB algorithm tries to find lower valued tasks to be rejected in order to guarantee the new one. For the sake of generality, in the description of the algorithm we have left unspecified the rejection policy. Let us describe a possible choice.

Suppose we want to select the i-th task in the server queue for rejection. The deadlines of all following tasks are then advanced:

$$d'_i = d'_{i-1} + \frac{C_i}{U_S}, \qquad d'_{i+1} = d'_i + \frac{C_{i+1}}{U_S}, \quad \ldots$$

become

$$d''_{i+1} = d'_{i-1} + \frac{C_{i+1}}{U_S}, \qquad d''_{i+2} = d''_{i+1} + \frac{C_{i+2}}{U_S}, \quad \ldots$$

with

$$\begin{aligned} d'_{i+1} - d''_{i+1} &= d'_i - d'_{i-1} = \tfrac{C_i}{U_S}, \\ d'_{i+2} - d''_{i+2} &= d'_i - d'_{i-1} = \tfrac{C_i}{U_S}, \\ &\ldots \end{aligned}$$

That is, all the deadlines from d'_{i+1} on can be advanced by $\frac{C_i}{U_S}$. Hence, a simple rejection policy is to find the least valued task J_i, preceding the first exceeding time, such that $\frac{C_i}{U_S} \geq E_{max}$, where E_{max} is the maximum exceeding time. The task J_i is then selected for rejection only if its value is less than the value of the new task J_a.

4.4 Recovery Strategy

When a task completes before its maximum execution time, its spare time can be reclaimed to execute sooner the pending requests. This gives us a chance to recover rejected tasks which still have a positive laxity.

The details of our recovery strategy are the following. At the completion of a task at time t, the exit routine computes the corrected deadline $\bar{d} = \bar{r} + \frac{\bar{C}}{U_S}$. The deadlines of all tasks in the server queue can thus be advanced by $d - \max(t, \bar{d})$, since d'_h, the deadline assigned to the head of the queue, becomes $\max(t, \bar{d}) + \frac{C_h}{U_S}$ (it was $\max(t, \bar{d}) + \frac{C_h}{U_S}$), d'_{h+1} becomes $d'_h + \frac{C_h}{U_S}$, and so on.

This means that we can recover a task from the reject queue if the sum of the computation time saved by all tasks completing within its laxity, is greater than or equal to the maximum exceeding time caused by the rejected task. Our attention, of course, is on tasks with larger values.

5 Experimental Results

In this section we will briefly discuss the results of the simulations we have carried out in order to evaluate the performance of the algorithms described in the paper. Our first concern has been to measure the improvement introduced in the performance of the TB server by the new formulation, that is, by the reclaiming of unused computation time.

A second set of experiments has been concerned with the main issue of this paper, that is, the evaluation of the RTB algorithm's robustness with respect to

overload conditions in the system. In this case the algorithm has been compared with a plain version of the TB server implementing an EDF policy for the aperiodic tasks, and a version with a first level of guarantee.

In what follows, we will often distinguish between *nominal* and *actual* loads. With the former term we indicate loads computed by using worst case execution times. Vice versa, with the latter one we indicate the same quantity computed by using actual execution times. Furthermore, unless stated otherwise loads are expressed as absolute values. That is, values less than 1 represent feasible conditions, while values greater than 1 represent overloaded conditions.

The different loads were generated by simulating a poisson aperiodic arrival, with random maximum execution times chosen uniformly in a suitable range. Similarly, the values assigned to the aperiodic tasks were chosen randomly with a uniform distribution.

5.1 Old vs New TB Formulation

Two versions of the TB server, obtained with the old and the new formulation, respectively, have been compared trying to understand the impact on the performance of the resource reclaiming technique. To achieve this goal we have set up a simulation of a system with 10 periodic tasks, for a global periodic load $U_P = 0.7$. The aperiodic load was obtained by generating the arrival of 80000 aperiodic tasks, whose nominal load was equal to 0.25. The average ratio of the maximum execution time actually utilized by the aperiodic tasks was varied between 0.5 and 1.0, thus giving an actual aperiodic load varying from 0.125 to 0.25. The bandwidth U_S assigned to the servers was equal to 0.3.

In Figure 5 the resulting mean aperiodic response times are reported for the two versions of the algorithm, referred to as TB-94 and TB-95, respectively. The reported values are normalized with respect to the maximum computation times. The graph clearly shows that the new version of TB takes advantage of the reclaiming technique under any condition. Only when the actual computation times are near their maximum values the performance is comparable, as expected. In all other cases the new version shows a significant improvement.

5.2 Robust TB Evaluation

In this second set of experiments we have evaluated the performance of the robust algorithm, RTB, described previously. In all experiments we have compared the algorithm with a "plain" version of TB and a version with a "raw" guarantee.

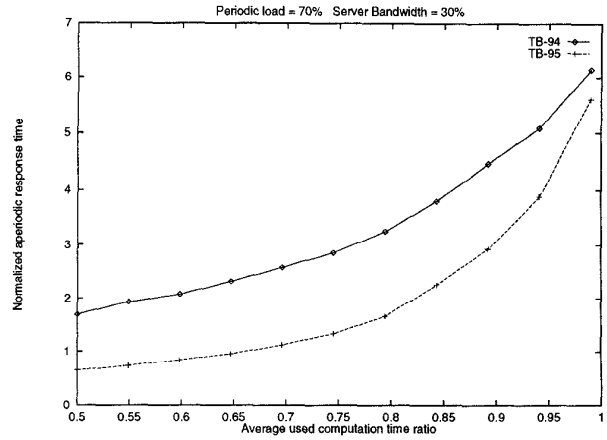


Figure 5: Old vs New version of the TB server.

The former, simply referred to as TB in the following, is obtained by using the definition of TB without any guarantee on the completion of the aperiodic tasks. When a task arrives, it is inserted into the server queue, sorted by increasing deadline, and scheduled, when it is the most urgent task, according to the deadline assigned by the server.

The latter version, instead, is obtained as follows. Each time a new task arrives the server computes the deadline it will assign to it and to all other less urgent tasks arrived earlier. Similarly to the RTB algorithm, during the computation of the deadline also the maximum exceeding time is determined. At the end of the computation, if the maximum exceeding time is 0 the new task is guaranteed, otherwise it is rejected.

In the first experiment the three algorithms have been compared in three situations of increasing periodic load. In the second experiment we have varied the unused computation time ratio of the aperiodic tasks, thus measuring the impact of the reclaiming technique.

5.2.1 Experiment 1: Hit Value Ratio vs Increasing Periodic Load

In this experiment we have compared the three mentioned algorithms in three situations with periodic loads U_P equal to 0.2, 0.5 and 0.8, respectively. The bandwidth U_S of the servers has been always set to the remaining processor capacity. The nominal load of the aperiodic tasks has been varied from 0.5 to 3.0, with an actual load varying from 0.25 to 1.5 (obtained

with actual execution times in average equal to half of the corresponding maximum values). The results of the simulations are reported in Figure 6. In the vertical axes of the graphs is represented the Hit Value Ratio, that is, the ratio of the value achieved by the system to the global value of the task set.

As shown by the three graphs, the behaviour of the algorithms is similar in all conditions, with TB showing the best performance until we have actual overload, and RTB being the best otherwise. Note that even in underload conditions the performance of RTB is comparable to that of TB.

In underload conditions we have a difference between the plain and the guaranteed versions. Furthermore, with large periodic loads the improvement of the robust version in overload conditions is smaller, compared to that shown with the other settings. The reason for this behaviour is the intrinsic pessimism in the guarantee routine. When the server bandwidth is small, the deadline assigned to aperiodic tasks may be significantly large. This value is then used as an upper bound of the completion time of the task, hence the algorithm may be quite pessimistic and unnecessarily reject some tasks that can actually complete in time. In the same situation the RTB algorithm is helped by the reclaiming strategy, while the performance of the GTB algorithm is compromised.

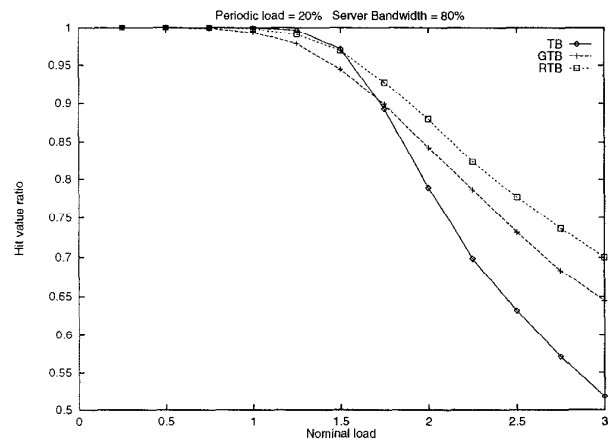
On the other hand, when the bandwidth of the server is large (the periodic load is low) the robust algorithms are less pessimistic and the improvement is larger. In particular, RTB shows the robustness and graceful degradation features we claimed previously.

5.2.2 Experiment 2: Hit Value Ratio vs Unused Computation Time

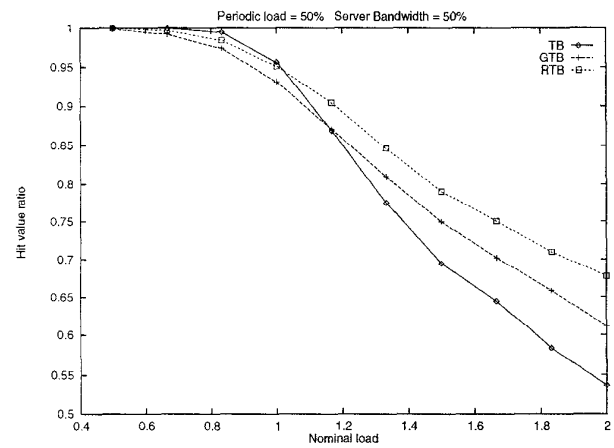
In this last experiment our intention was to identify the impact of the reclaiming technique used in the RTB algorithm. In order to do this, we have set up an experimental framework with a periodic load $U_P = 0.5$, given by ten tasks. The bandwidth assigned to the servers was equal to 0.5. The nominal load of the aperiodic task set was equal to 3.0. However, during the experiment we have varied the ratio of the average used computation time from 0.1 to 0.9, thus giving an actual aperiodic load between 0.3 and 2.7.

The result of this experiment is illustrated in Figure 7. As previously, in the vertical axis we have represented the Hit Value Ratio.

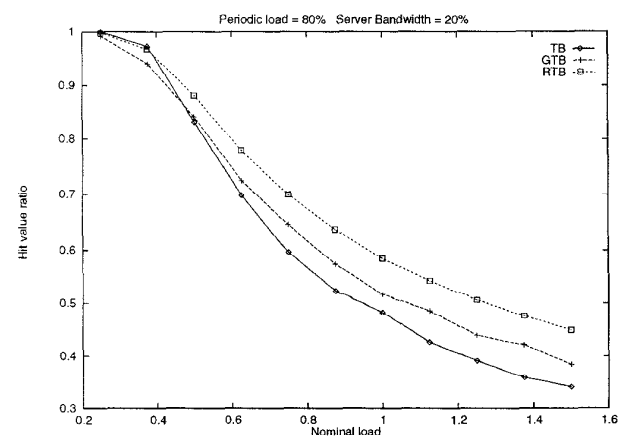
In the graph we can see that for large values of the used computation time ratio, i.e., for large loads, the RTB offers the best performance. This confirms the effectiveness of the reclaiming strategy.



(a)



(b)



(c)

Figure 6: Performance with increasing periodic load.

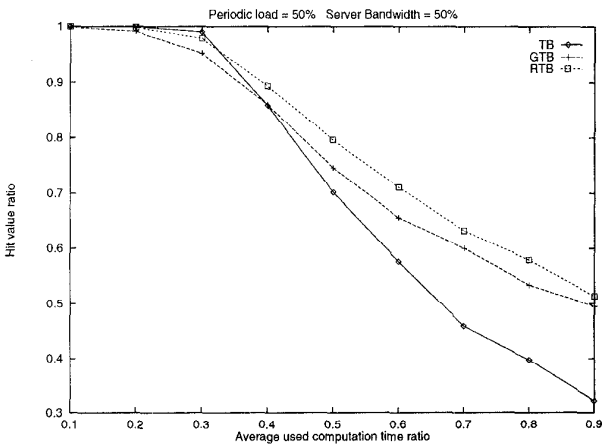


Figure 7: Impact of the reclaiming strategy.

For smaller values, however, the results of the plain version of TB are slightly better than those of the robust version. Also in this case the difference is due to the pessimism of the guarantee routine, not completely compensated by the reclaiming strategy. The confirmation of this analysis comes from the performance of the GTB version, worse than that of TB in this condition.

6 Discussion and Conclusions

In this paper we have investigated the problem of the joint hard periodic and firm aperiodic scheduling under dynamic priority systems. In particular, we have focused our attention on the problem of achieving graceful degradation and acceptable performance during transient system overloads.

The solution we have proposed is based on the integration of an efficient aperiodic server, called Total Bandwidth server, and a technique including a rejection and a reclaiming strategies, for the addition of robustness. The resulting algorithm, called Robust TB, have been tested with a number of experimental simulations.

The experiments have shown the effectiveness of the RTB algorithm. Most of it is due to the reclaiming strategy, while the acceptance test and relative rejection strategy are sometimes a bit too pessimistic, especially when the bandwidth assigned to the server is small. We believe this is the weakest part of the algorithm and it needs more attention if we want to improve the algorithm.

References

[1] J. Blazewicz, "Scheduling dependent tasks with different arrival times to meet deadlines," In E. Gelenbe, H. Beilner (eds), Modelling and Performance Evaluation of Computer Systems, Amsterdam, North-Holland, 57-65, 1976.

[2] G. Buttazzo, "HARTIK: A Real-Time Kernel for Robotics Applications," *Proc. of Real-Time Systems Symposium*, 201-205, 1993.

[3] G. Buttazzo and J. Stankovic, "RED: A Robust Earliest Deadline Scheduling Algorithm," *Proc. of 3rd International Workshop on Responsive Computing Systems*, Austin, 1993.

[4] H. Chetto, M. Silly, T. Bouchentouf, "Dynamic Scheduling of Real-Time Tasks under Precedence Constraints," *The Journal of Real-Time Systems* 2, 181-194, 1990.

[5] R.I. Davis, K.W. Tindell, A. Burns, "Scheduling Slack Time in Fixed Priority Pre-emptive Systems", *Proc. of Real-Time Systems Symposium*, 222-231, 1993.

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

[7] T.M. Ghazalie and T.P. Baker, "Aperiodic Servers In A Deadline Scheduling Environment," *The Journal of Real-Time Systems*, to appear.

[8] J. Jackson, "Scheduling a Production Line to Minimize Tardiness," Research Report 43, Management Science Research Project, University of California, Los Angeles, 1955.

[9] J.P. Lehoczky and S. Ramos-Thuel, "An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems," *Proc. of Real-Time Systems Symposium*, 110-123, 1992.

[10] J.P. Lehoczky, L. Sha, J.K. Strosnider, "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments," *Proc. of Real-Time Systems Symposium*, 261-270, 1987.

[11] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard real-Time Environment," *Journal of the ACM* 20(1), 40-61, 1973.

[12] C. D. Locke, "Best Effort Decision Making For Real-Time Scheduling," PhD Thesis, Computer Science Dept., CMU, 1986.

[13] B. Sprunt, L. Sha, J. Lehoczky, "Aperiodic Task Scheduling for Hard-Real-Time Systems," *The Journal of Real-Time Systems* 1, 27-60, 1989.

[14] J. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Systems," *IEEE Software*, 8(3), 62-72, May 1991.

[15] M. Spuri, G. Buttazzo, "Efficient Aperiodic Service under Ealiest Deadline Scheduling," *Proc. of Real-Time Systems Symposium*, 2-11, 1994.

[16] M. Spuri, G. Buttazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems," TR ARTS Lab 94-06, Scuola Superiore S.Anna, Pisa, 1994, submitted to *The Journal of Real-Time Systems*.