Scheduling Real-Time Multi-Task Applications in an Open System

Giuseppe Lipari and Giorgio Buttazzo
Scuola Superiore S. Anna, Pisa
{lipari,giorgio}@sssup.it

Abstract

This paper[1] focuses on the problem of providing isolation and real-time execution in scheduling multi-thread applications on a single processor. The proposed scheduling algorithm can be applied in a wide range of different situations, such as in a multimedia real-time systems, where applications may require a guaranteed level of Quality of Service, or in a network link, where different flows of packets may require a certain level of service. This algorithm does not require the exact knowledge of the service times and interarrival times of the tasks: hence, it is especially suited for soft real-time and multimedia environments.

1 Introduction

Modern computer systems are powerful enough to execute concurrent applications, some of them may be composed by several concurrent tasks. In many cases, applications may require timing constraints to exhibit the desired performance. Such constraints can be hard (as in control systems) or soft (as in multimedia systems). Such a multi-application scenario is also typical in network systems, where different flows of packets belonging to the same user, or to the same group of users, must be scheduled with a certain level of service.

In order to provide a predefined Quality of Service (QoS) to soft real-time applications and guarantee hard timing constraints, operating systems must support suitable resource reservation mechanisms and scheduling algorithms. Unfortunately, traditional operating systems do not meet this goal because scheduling decisions are not based on task parameters, like *deadlines* or *periods*, so system resources are not managed according to the requirements of soft real-time streams. As a consequence, an application may have an unpredictable timing behavior and experience discontinuity during its execution depending on peak load situations.

On the other hand, the use of a hard real-time system for handling soft real-time applications can be inappropriate, because it may be very difficult to provide a precise estimation of worst-case execution times (WCETs) for multimedia tasks, even if the application always runs on the same hardware. Moreover, in a hard real-time environment, if the WCET of a task is underestimated, the task could cause the failure of all system tasks. Rather, in a multi-application system, it would be desirable that only the failed application is affected.

In [7], Kaneko et al. propose a scheme based on a periodic process (the multimedia server) dedicated to the service of all multimedia requests. This allows to nicely integrate multimedia tasks together with hard real-time tasks; however, being the server only one, it is not easy to control the QoS of each application.

In [1], Abeni and Buttazzo presented the Constant Bandwidth Server, which is able to schedule hard and soft real-time tasks together, guaranteeing the deadlines of the formers, and a minimum service to the latters. The algorithm is also able to isolate the behavior of each task from the others, even in the case in which the parameters (WCET and minimum interarrival time) are not known a priori. However, in this work, only single-task applications are considered.

In [4], Liu and Deng describe a scheduling hierarchy which allows hard real-time, soft real-time, and non real-time applications to coexist in the same system, and to be created dynamically. According to this approach, which uses the EDF scheduling algorithm as a low-level scheduler, each application is handled by a dedicated Total Bandwidth Server. This solution can be used to isolate the effects of overloads at the application level, but requires the knowledge of the WCET even for soft and non real-time tasks.

Recently, some scheduling algorithms have been proposed [10, 5] to mix some form of real-time support with a notion of fairness. In these works, the isolation is realized allocating the processor to applications in a weighted fair manner. These methods are effective in scheduling soft

[1]This work has been carried out under the financial support of the Ministero dell'Universitá e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the Project "Design Methodologies and Tools of High Performance Systems for Distributed Applications".

real-time tasks, and nicely integrate with the existing software: for example, in [5] the authors realized a two-level scheduling algorithm and it is possible to select the higher-level scheduling policy that best meets the application requirements. However, this approach may be unsuitable for guaranteeing applications consisting of hard real-time tasks, because the method is quite sensitive to the temporal quantum seen by an application, which depends on the percentage of processor reserved to it.

A very interesting approach to the problem of scheduling different applications in an open system can be found in [6]. It is based on a two-level scheduler: the global scheduler is based on EDF, whereas it is possible to select the most appropriate local scheduling strategy. With this method, it is easy to guarantee real-time execution, but no isolation is provided among applications. Hence, an accurate guarantee test must be performed on each application and a misbehaved task can affect the schedulability of other applications.

In this paper, we propose a scheduling methodology based on reserving a fraction of the processor bandwidth to each application (in a way similar to the Constant Bandwidth Server [1]). Each application is scheduled by a dedicated server, the Bandwidth Sharing Server, which assigns suitable deadlines to the served tasks whenever the reserved time is consumed. The proposed method does not require the exact knowledge of the WCETs.

The paper is organized as follows: Section 2 describes the main goals of this work; Section 3 introduces the terminology and the assumptions used throughout the paper; Section 4 formally presents the algorithm; Section 5 illustrates the properties of the algorithm; Section 6 shows some simulation results; and Section 7 presents our conclusions.

2 Goals of this research

The main goal of this paper is to handle multi-task applications on a single processor in such a way that:

1. the behavior of each application does not affect the performance of the other applications in the system (*bandwidth isolation property*);

2. it is possible to guarantee the schedulability of hard real-time applications when the worst-case computation times (WCETs) and the minimum interarrival times (MITs) of the tasks are known a-priori (*hard schedulability property*).

The *bandwidth isolation* property ensures that if an application demands more service time than expected, the schedulability of the others is not affected, but only the "wrong" application slows down.

On the other hand, the *hard schedulability* property ensures that, if the task parameters are correctly estimated, an a-priori guarantee can be performed and the application executes as it was running alone on a slower processor.

It is worth to point out that, for our purposes, it is not important to achieve isolation among tasks within an application, but only among tasks belonging to different applications. Thus, a task τ_i belonging to application $\mathcal{A}$, which requires too much computation time, can only delay the other tasks in $\mathcal{A}$.

3 System model

A task τ_i is a finite or infinite sequence of requests (jobs) for a shared resource (e.g., the CPU). Each job $J_{ij} = (a_{ij}, c_{ij})$ is characterized by a request time (or arrival time) a_{ij} and a computation time c_{ij}.

In addition, each task has a parameter T_i which is its *nominal period* used to assign each instance a deadline

$$d_{ij} = a_{ij} + T_i.$$

Notice that, although the exact sequence of arrival times and computation times is not known a priori, T_i is a well know parameter, assigned at design time.

The meaning of T_i depends on the task type. If the task is critical (hard), T_i represents the *relative deadline*, that is the interval of time by which a job <u>must</u> complete in order for the application to be correct. On the other hand, if the task is non critical (soft), T_i represents the interval of time by which a job <u>should</u> complete, if there is enough resource available.

Thus, assigning critical tasks a sufficient fraction of the resource bandwidth to allow a priori guarantee is a responsibility of the system designer. On the contrary, if the application tasks are not critical, the system designer can also decide to assign resources based on average values, so that deadlines can be respected with a certain probability (QoS guarantee). As hard tasks are managed exactly like soft tasks, from now on we do not distinguish between them.

An application is a set of tasks:

$$\mathcal{A} = \{\tau_1, \tau_2, \ldots, \tau_n\}.$$

In the following, symbol $\tau_i^{\mathcal{A}}$ indicate the i-th task of application $\mathcal{A}$. Each application $\mathcal{A}$ is assigned a bandwidth $U^{\mathcal{A}}$ which is the fraction of processor time that the application is allowed to use. Our system consists of a set of applications Γ which share the same resource. We assume that

$$\sum_{\mathcal{A} \in \Gamma} U^{\mathcal{A}} \leq 1.$$

All tasks are scheduled using the Earliest Deadline First (EDF) scheduling algorithm. To achieve bandwidth isolation, each application is handled by a dedicated server,

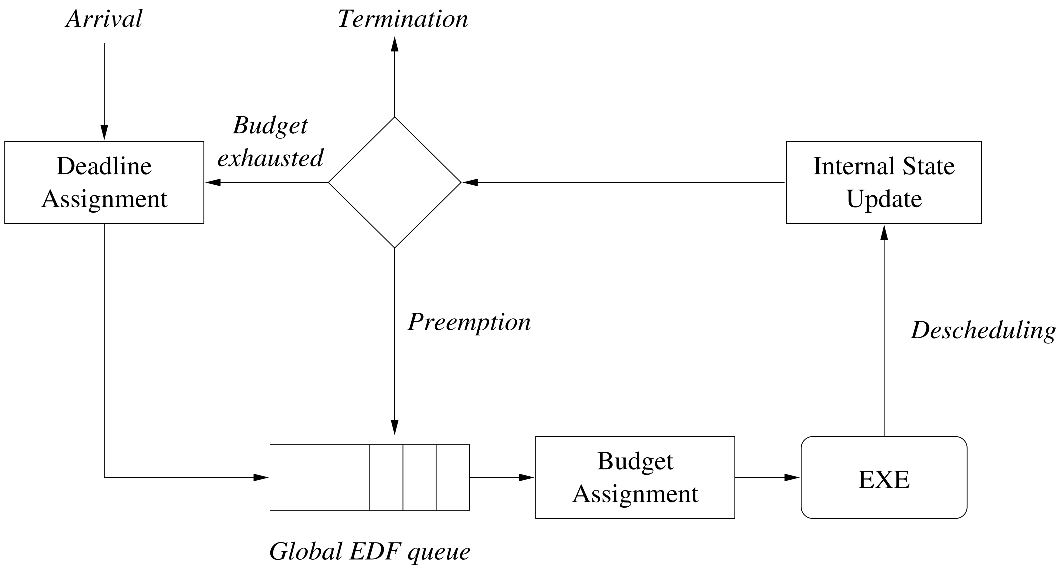

Figure 1. The Bandwidth Sharing Server.

namely the Bandwidth Sharing Server (BSS), which assigns each request a deadline for scheduling purposes (*scheduling deadline*).

4 Description of the server

In order to bound its bandwidth $U^{\mathcal{A}}$, each application $\mathcal{A}$ is handled by a dedicated server, called Bandwidth Sharing Server (BSS). The structure of the BSS is depicted in Figure 1. Whenever a task issues a request, it is assigned a scheduling deadline and it is inserted in a global EDF queue. Then, the first task from the global queue is selected to be executed and a budget is calculated which represents the maximum time that the task is allowed to execute continuously. When a task leaves the processor, three cases are possible:

- The task has finished. In this case, it is suspended until the next activation.
- The budget is exhausted. In this case a new scheduling deadline is assigned to the task and it is inserted again in the EDF queue.
- It has been preempted by another task with an earliest deadline. In this case it is simply put again in the EDF queue.

To calculate the budget such that the bandwidth isolation property and the hard schedulability property hold, the BSS uses some internal data structures. These data are private to each server (and, hence, to each application).

4.1 Data Structures

For each application $\mathcal{A} = \{\tau_1^{\mathcal{A}}, \ldots, \tau_{N^{\mathcal{A}}}^{\mathcal{A}}\}$, the dedicated BSS server stores the amount of bandwidth still available at the current time in a *list of residuals* $L_R^{\mathcal{A}}$. For each task of $\mathcal{A}$, this list contains one or more of the following type of elements:

$$l = (R, d)$$

where

- d is the task's deadline;
- R is the residual execution time available in interval $[a, d]$ (where a is the task's arrival time); that is, the maximum time that the tasks in $\mathcal{A}$ are allowed to demand in $[a, d]$.

Thus, an element l specifies for the interval $[a, d]$ the amount of execution time available in it. The goal of the server is to update the list such that in every interval of time the application cannot use more than its bandwidth. From now on, symbol l_k will denote the element in the k-th position of the list.

List $L_R^{\mathcal{A}}$ is ordered by not decreasing deadlines d. Furthermore, the algorithm assigns and modifies the residuals R such that the following property holds:

Property 1

$$\forall l_j, l_k \in L_R^{\mathcal{A}} \qquad d_j < d_k \Rightarrow R_j \leq R_k.$$

This property basically says that the total execution time allowed in an interval must be greater than the execution time allowed in any contained interval.

In the following sections we present the BSS algorithm in two successive steps. To simplify the description, Section 4.2 describes the basic server mechanism and presents a simple example of a produced schedule, whereas Section 4.3 refines the algorithm to take into account some problems that arise when elements are not inserted in the list in the right order.

4.2 The algorithm

In this section, we present a simplified version of BSS, called BSS-A, aimed at simplifying the presentation of the general algorithm.

Algorithm BSS-A

Rule A1: Job arrival. Whenever a job J_i issues a request, if its task is not already active (i.e., there is not another job from the same task which has not yet finished) then it is assigned a deadline $d_i = a_i + T_i$; otherwise, the job must wait until the previous job has finished, and at that time (t) it can be assigned a deadline $d_i = max(t, a_i + T_i)$. In any case, it is put in the EDF queue.

Rule A2: Job scheduling. Each time a job J_i with deadline d_i is selected to be executed for the first time, a new element l is created and inserted in the list. First, the list is scanned in order to find the right position for the new element. Let k be such a position, that is:

$$\exists l_{k-1}, l_k \quad d_{k-1} < d_i \leq d_k$$

Now, the residual R_i is computed as:

$$R_i = min\{T_i U^{\mathcal{A}},\ (d_i - d_{k-1})U^{\mathcal{A}} + R_{k-1},\ R_k\} \quad (1)$$

At this point, the new element is completely specified as $l = (R_i, d_i)$ and can now be inserted at position k, so that the k-th element becomes now the $(k+1)$-th element, and so on.

In the case the job resumes execution after a preemption, then an element already exists in the list. Let $l = (R_i, d_i)$ be such an element. In both cases, job J_i is assigned a budget $B = R_i$, and it is not allowed to execute more.

Rule A3: Job leaves processor. Each time a job J_i leaves the processor for any reason (it has finished or it has been preempted or the budget is exhausted), the list must be updated. Suppose that it has executed for E units of time from the instant when it was scheduled. Then, the algorithm picks the corresponding element in the list, say the k-th element, and updates the residuals in the following way:

$$\begin{array}{llll} \forall l_j & j \geq k & R_j = R_j - E \\ \forall l_j & j < k & R_j = min(R_j, R_{j+1}) \end{array}$$

At this point, three cases must be handled:

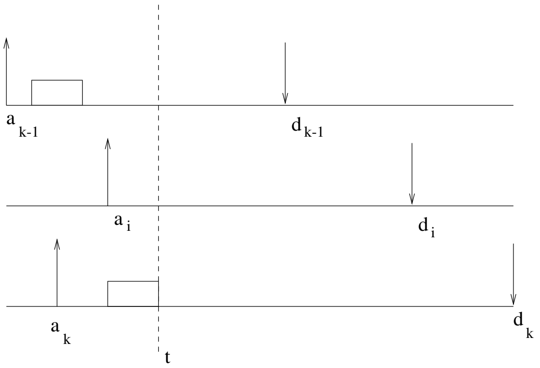

Figure 2. Computation of R_i

- the job was **preempted** by a task with earliest deadline: in this case, nothing must be done;
- the job has **finished**: in this case, a new element must be created for the next job of this task;
- the budget was **exhausted**: in this case, the job deadline is postponed at $d_i = d_i + T_i$ and the job is put again in the EDF queue. When it will be executed, another element must be created, just as if it was scheduled for the first time (see Rule A2).

It is worth to do some consideration on Equation (1). A typical situation is shown in Figure 2: when at time t task τ_i is the earliest deadline task, the algorithm must compute a new residual such that in any interval the application cannot use more than its share. The residual must not exceed the share, which is $T_i U^{\mathcal{A}}$ in interval $[a_i, d_i]$, $R_{k-1} + (d_i - d_{k-1})U^{\mathcal{A}}$ in interval $[a_{k-1}, d_i]$, and R_k in interval $[a_k, d_k]$. It can be shown that if R_i is the minimum among these values, then the application will not use more than its share in any other interval (for details, see [9]).

An example of schedule produced by algorithm BSS-A is illustrated in Figure 3. In this example there are two applications, $\mathcal{A} = \{\tau_1^{\mathcal{A}}\}$ and $\mathcal{B} = \{\tau_1^{\mathcal{B}}, \tau_2^{\mathcal{B}}\}$, each of them is assigned a bandwidth of 0.5. Let us focus our attention on application $\mathcal{B}$. Assume that periods $T_1^{\mathcal{B}} = 8$ and $T_2^{\mathcal{B}} = 20$ are assigned to the two tasks of application $\mathcal{B}$.

At time $t = 0$, $L_R^{\mathcal{B}}$ is empty and task $\tau_2^{\mathcal{B}}$ arrives with a request of 4 units of computation time: then, it is assigned a deadline at 20 and it is inserted in the global EDF queue. Since no other task is in the system, it is selected to execute and a new element is created and inserted in list $L_R^{\mathcal{B}}$:

$$L_R^{\mathcal{B}} = \{(10, 20)\}.$$

It means that all the tasks of $\mathcal{B}$ with deadline less than or equal to 20 are allowed to execute at most for 10 units of time. Then, it is assigned a budget of 10.

At time $t = 3$, task $\tau_1^{\mathcal{B}}$ issues a request of 6 and it is assigned a deadline at 11. Since it is now the earliest deadline

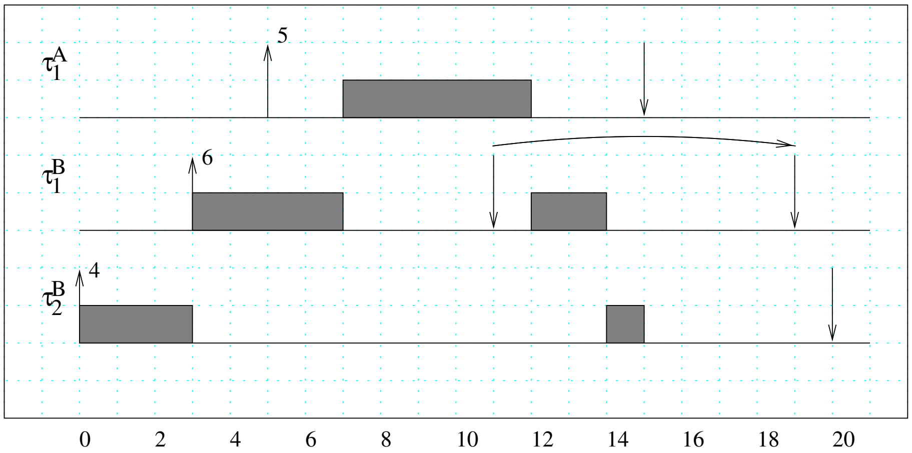

Figure 3. Two applications scheduled by the BSS.

task, a preemption occurs. Thus, task $\tau_2^{\mathcal{B}}$ leaves processor and the list is updated by decreasing the residual of 3 units of time:

$$L_R^{\mathcal{B}} = \{(7, 20)\}.$$

Then, a new element for task $\tau_2^{\mathcal{B}}$ is created and inserted in the list:

$$L_R^{\mathcal{B}} = \{(4, 11), (7, 20)\}.$$

Finally, it is assigned a budget of 4. At time $t = 7$ task $\tau_1^{\mathcal{B}}$ leaves processor because its budget is exhausted. Its deadline is postponed at $d_1 = 11 + 8 = 19$, and the list is updated as follows:

$$L_R^{\mathcal{B}} = \{(0, 11), (3, 20)\}.$$

Please, notice that $\tau_1^{\mathcal{B}}$ has also consumed the residual of task $\tau_2^{\mathcal{B}}$. Then, task $\tau_1^{\mathcal{A}}$ starts executing, but it belongs to application $\mathcal{A}$, so nothing changes for application $\mathcal{B}$. At time $t = 12$, task $\tau_1^{\mathcal{B}}$ is again the earliest deadline task, so it can execute. The list is updated as follows:

$$L_R^{\mathcal{B}} = \{(0, 11), (3, 19), (3, 20)\}.$$

Notice that the new residual is not 4, which is the maximum possible, but it is 3, because it is bounded by the residual in the interval $[0, 20]$.

Some things are worth to notice for algorithm BSS-A:

- elements are inserted in list L_R (and residuals are calculated) only when the task instances begin executing for the first time;
- the actual order of execution does not depend on the arrival order of the tasks, but depends on the presence of other tasks in the system, even of other applications.

4.3 Ordering the insertions in the list

As the order of insertion depends upon the order of execution, algorithm BSS-A does not properly compute residuals R in certain situations. Consider Figure 4 where there is an application $\mathcal{A}$ which consists of 3 tasks, with $U^{\mathcal{A}} = 0.5$, and an application $\mathcal{B}$ which consists of only one task, with $U^{\mathcal{B}} = 0.5$.

Nothing happens to list $L_R^{\mathcal{A}}$ until time $t = 10$, when task $\tau_3^{\mathcal{A}}$ starts execution. At this point, element $l = (4, 18)$ is inserted in $L_R^{\mathcal{A}}$. At time $t = 11$, when $\tau_3^{\mathcal{A}}$ completes, list is updated as $L_R^{\mathcal{A}} = \{(3, 18)\}$. Now task $\tau_1^{\mathcal{A}}$ is ready to execute, and a new element is calculated and inserted in the list:

$$L_R^{\mathcal{A}} = \{(3, 18), (4, 20)\}.$$

Notice that the residual calculated for $\tau_1^{\mathcal{A}}$ is too low. In fact, $\tau_1^{\mathcal{A}}$ could be assigned 9 units of residual without exhausting the bandwidth allowed in interval [0,20]. This problem is due to a wrong order of insertion of elements in the list. In this example, the algorithm does not take interval [0,8] into account; if the element corresponding to task $\tau_1^{\mathcal{A}}$ is calculated and inserted in the list before the element corresponding to task $\tau_3^{\mathcal{A}}$, all the residuals are properly calculated.

To insert elements in L_R in the proper order, we need another list to keep track of the arrival times of the tasks. Let L_A be such a list, whose elements are pairs (τ_i, a_i), denoting that task τ_i arrived at time a_i. The list is ordered by arrival times.

We now introduce algorithm BSS-B which modifies some of the rules of algorithm BSS-A to properly order the insertions in the list.

Algorithm BSS-B

Rule B1: Job arrival. The task identifier and its arrival time are saved in the list $L_A^{\mathcal{A}}$, which is ordered by increasing

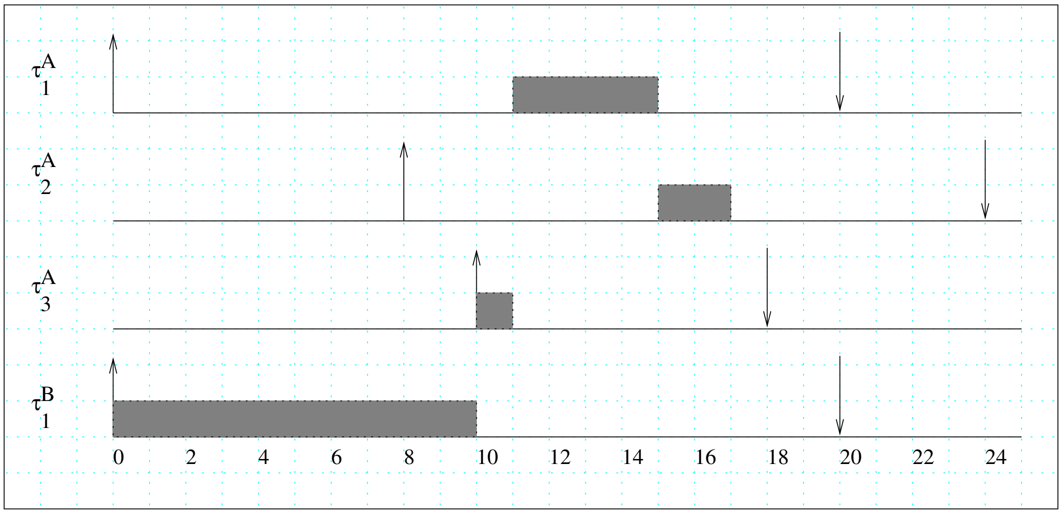

Figure 4. A critical execution order.

arrival times. The deadline is assigned to the job as in Rule A1.

Rule B2: Job scheduling. If job J_i is executed for the first time, then

- $Lastd = \infty$
- for each element $(\tau_i, a_i) \in L_A^A$ (order by increasing arrival time): if the job deadline d_i is less than or equal to $Lastd$ then
  - create and insert an element for τ_i in L_R^A as in Rule A2;
  - extract (τ_i, a_i) from L_A^A;
  - $Lastd = d_i$.

In the case the job resumes execution after a preemption, then an element already exists in the list. Let $l = (R_i, d_i)$ be such an element; in both cases, job J_i is assigned a budget $B = R_i$, and it is not allowed to execute more.

Rule B3: Job leave processor. Same as Rule A3.

Using algorithm BSS-B, it is easy to see that in the example in Figure 4, at time $t = 10$ element $(10, 20)$ for task τ_1^A is inserted first; element for τ_2^A is not inserted as its deadline is greater than $Lastd$; and finally element $(4, 18)$ for τ_3^A is inserted.

It can be proved that using algorithm BSS-B the order of insertion of elements in list L_R^A does not depend on the execution order of the jobs and the residual R is always the maximum possible.

4.4 Implementation

The final version of the algorithm has some additional rule for deleting elements of the list that are not useful any longer. Due to space limitations, we report here only some of these rules without proof.

Property 2 *If at time t there exists an element $l_k = (R_k, d_k)$ with $d_k < t$, then any l_j with $j < k$ can be safely deleted from the list.*

Property 3 *If there exists an element $l_k = (0, d_k)$, then any element l_j with $j < k$ can be safely deleted.*

Property 4 *If element $l_k = (R_k, d_k)$ with $R_k = T_i U^{\mathcal{A}}$ is inserted in the list, then element l_{k-1} can be safely deleted.*

Given these rules, it can be shown that the maximum number of elements in list L_R^A is always less than or equal to $T_{max} = max_{\tau_i \in \mathcal{A}}(T_i)$. Hence, BSS algorithm has complexity $\mathcal{O}(T_{max})$, which is pseudo-polynomial in the input.

However, if the T_is assigned to the tasks are not too different from the average interarrival times, then the average length of the list is proportional to the number of tasks in the application. So, in practical situations, the overload caused by the server is not very high.

5 Formal Properties

The formal properties of the server are analyzed using the *processor demand approach* ([3] pag. 102). The *demand* of a task τ_i in an interval of time $[t_a, t_b]$ is defined as:

$$D_i(t_a, t_b) = \sum_{\substack{a_{ij} \geq t_a \\ a_{ij} + T_i \leq t_b}} c_{ij}.$$

The demand of a hard application $\mathcal{A}$ is defined as the sum of the demands of all the tasks in $\mathcal{A}$:

$$D^{\mathcal{A}}(t_a, t_b) = \sum_{\tau_i \in \mathcal{A}} D_i(t_a, t_b).$$

Please note that the demand of an application does not depend upon the adopted scheduling algorithm, but only on its temporal requirements.

During its execution, a job scheduled by the BSS algorithm can have different deadlines: each time the available fraction of the processor bandwidth is exhausted, the deadline of the current executing task is postponed. An example is depicted in Figure 3 where task $\tau_1^{\mathcal{B}}$ requires 6 units of computation time in the interval $[3, 11]$. It is easy to see that the request cannot be satisfied, because the task would use 6/8 of the processor utilization, while it is allowed to use only 4/8. Thus, at time $t = 7$ the job execution is suspended and, according to the BSS algorithm, its deadline is postponed at $d_1 = d_1 + T_1^{\mathcal{B}} = 19$. From our point of view, this situation is equivalent to having 2 distinct jobs: j_{11}^* having request time $a_{11}^* = 3$, computation time $c_{11}^* = 4$ and deadline $d_{11}^* = 11$; and j_{12}^* with request time $a_{12}^* = 3$, computation time $c_{12}^* = 2$ and deadline $d_{12}^* = 19$. In other words, job $J_1^{\mathcal{B}}$ is split into jobs j_{11}^* and j_{12}^*. In general, a job $J_i = (a_i, c_i, d_i)$ can be split into n_i jobs j_{ij}^* with $a_{ij}^* = a_i$, $d_{ij}^* = a_i + jT_i$, and $\sum_{j=1}^{n_i} c_{ij}^* = c_i$.

Given a schedule $\sigma(t)$ produced by the BSS, it is useful to define the *scheduling demand* of task $\tau_i^{\mathcal{A}}$ in the interval $[t_a, t_b]$ as:

$$SD_i^{\mathcal{A}}(t_a, t_b) = \sum_{\substack{a_{ij}^* \geq t_a \\ d_{ij}^* \leq t_b}} c_{ij}^*,$$

and the *scheduling demand* of application $\mathcal{A}$ as

$$SD^{\mathcal{A}}(t_a, t_b) = \sum_{\tau_i \in \mathcal{A}} SD_i^{\mathcal{A}}(t_a, t_b).$$

Please note that the scheduling demand of an application depends upon the scheduling algorithm adopted and upon the actual application requirements. To better understand the difference between the demand and the scheduling demand of an application, consider the previous example (Figure 3):

$$D^{\mathcal{B}}(3, 11) = D^{\mathcal{B}}(3, 19) = 6,$$
$$SD^{\mathcal{B}}(3, 11) = 4, \quad SD^{\mathcal{B}}(0, 12) = 6.$$

Given these definitions, the following results can be proven (the formal proofs of these theorems are not reported here and can be found in [9]).

Theorem 1 *Let $[x, y]$ be an interval of time such that all the task of $\mathcal{A}$, which arrive before x and have deadline less than or equal to y, finish before x. Then:*

$$SD^{\mathcal{A}}(x, y) \leq (y - x)U^{\mathcal{A}} \quad (2)$$

Theorem 2 *An application $\mathcal{A}$ with bandwidth U^A, served by BSS, is feasible (i.e., each job completes before its absolute deadline $d_{ij} = a_{ij} + T_i$) if:*

$$\forall t_1, t_2 \quad t_1 < t_2 \quad D^{\mathcal{A}}(t_1, t_2) \leq (t_2 - t_1)U^{\mathcal{A}}.$$

Theorem 1 ensures the *Bandwidth Isolation Property*: it allows to mix different applications in the same computer system and guarantees that the temporal behavior of an application is not affected by, nor interferes with the others. This property is very useful in highly dynamic systems, where is difficult to obtain a priori information about applications. Even if the request times and the execution times of the tasks of an application are not known in advance, it is guaranteed that the other applications are not affected.

The *Hard Schedulability Property* stated in Theorem 2 allows to guarantee hard real-time tasks within a single application, even though other soft real-time applications run concurrently in the same system.

6 Simulation results

The performance of the BSS algorithm has been compared with the one of the Constant Bandwidth Server [1] on the same task set with different soft loads. The experimental set consists of two different applications. The first one contains 5 tasks with fixed periods and computation times, for a total utilization factor of 0.5. The second application contains 3 soft tasks having variable interarrival times and computational requests, with average cumulative load ranging from 0.470 to 0.498 and with a peak cumulative load ranging from 0.8 to 1.2. Notice that, while the average load in the system is always less than 1, the peak load can be higher than 1.

In the first experiment, each soft task is handled by a different CBS with a bandwidth of $\frac{0.5}{3}$. In the second experiment, the three soft tasks are handled by a single BSS with a bandwidth of 0.5. In both cases, we measured the average tardiness of one soft task normalized with respect to its desired period, defined as:

$$E = \frac{\max\{0, f_i - d_i\}}{T_i}.$$

where f_i is the actual finishing time of the job.

As shown in Figure 5, the BSS performs far better than the CBS. This is mainly due to the fact that the BSS takes advantage of handling together the 3 soft tasks. In facts, the probability that the three tasks are in overload at the same time is not high: each CBS can only reclaim the spare time of the early completions of the task that it handles, whereas the BSS can reclaim the spare time of any of the three tasks.

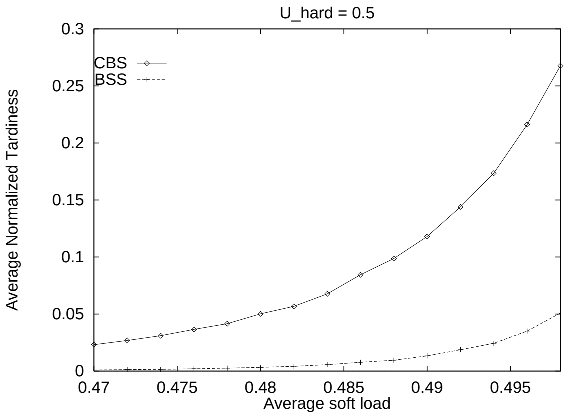

Figure 5. Experimental evaluation.

7 Conclusions

In this work, we presented a mechanism for scheduling real-time multi-task applications on a uniprocessor system. The main goal of this research is to reserve and guarantee a bandwidth to each application, independently from the execution requests of the tasks. This is achieved by managing each application through a dedicated server which assigns suitable deadlines to application tasks and put them in a single EDF queue. If all the parameters are correctly estimated, our service method guarantees that the resulting schedule is equivalent to the one obtained by EDF. On the contrary, if the parameters of some task are not correctly estimated, only the tasks belonging to the application which the failed task belongs to will be affected.

In the next future, we plan to implement the proposed mechanism in the HARTIK kernel [2, 8] and use it as a basic support for multimedia applications.

References

[1] L. Abeni and G.C. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 4–13, December 1998.

[2] G.C. Buttazzo. Hartik: A real-time kernel for robotics applications. In *IEEE Real-Time Systems Symposium*, December 1993.

[3] G.C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Boston, 1997.

[4] Z. Deng and J. W. S. Liu. Scheduling real-time applications in open envirovment. In *IEEE Real-Time Systems Symposium*, San Francisco, December 1997.

[5] P. Goyal, X. Guo, and H. M. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *2nd OSDI Symposium*, October 1996.

[6] J. Jehuda, G.Koren, and D.M. Berry. A time sharing architecture for complex real-time systems. In *Proceedings of the First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 9–16, November 1995.

[7] H. Kaneko, J.A. Stankovic, S. Sen, and K. Ramamritham. Integrated scheduling of multimedia and hard real-time tasks. In *Proceedings of Real-Time Systems Symposium*, December 1996.

[8] G. Lamastra, G. Lipari, G.C. Buttazzo, A. Casile, and F. Conticelli. Hartik 3.0: A portable system for developing real-time applications. In *Real-Time Computing Systems and Applications*, October 1997.

[9] G. Lipari. An algorithm for resource reservation in multi-applications systems. Technical Report RETIS TR-98-01, Scuola Superiore S. Anna, 1998.

[10] I. Stoica, H. Abdel-Wahab, K. Jeffay, S.K. Baruah, J.E. Gehrke, and C.G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *IEEE Real Time System Symposium*, 1996.