Giuseppe Lipari and Sanjoy Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. Proceedings of the Real-Time Technology and Applications Symposium, pp 166-175, Washington, DC. May 2000. IEEE Computer Society Press.

Efficient Scheduling of Real-Time Multi-Task Applications in Dynamic Systems*

Giuseppe Lipari, Sanjoy K. Baruah
lipari@sssup.it, baruah@cs.unc.edu

Abstract

The Bandwidth Sharing Server *(BSS) is a scheduling strategy for multi-task real-time applications that provides the dual properties of performance guarantees and inter-application isolation. We describe here the BSS-I algorithm, an evolution of the BSS, aimed at (i) improving the run-time performance and (ii) generalizing the kinds of performance guarantees that can be offered.*

1 Introduction

The thread model of concurrent programming is very popular and it is supported by most of the operating systems. In this model, each application (or process) has its own address space and applications communicate by means of operating system primitives. Context switch among applications is often an expensive operation. An application can be multi-threaded, i.e. it can consist of several concurrent tasks: different tasks belonging to the same application share address space and other resources. Context switch among tasks of the same application is faster and the communication easier.

In many cases, applications may require timing constraints to exhibit the desired performance. Such constrains can be hard (as in control systems) or soft (as in multimedia systems). In order to provide a predefined Quality of Service (QoS) to soft real-time applications and guarantee hard timing constrains, operating systems should support suitable resource reservation mechanisms and scheduling algorithms. Unfortunately, traditional operating systems do not meet this goal because scheduling decisions are not based on real-time parameters, like *deadlines* or *periods*. As a consequence, applications may have an unpredictable timing behavior and experience discontinuity under peak load situations.

On the other hand, using a traditional real-time system is not always appropriate. In fact, many of these systems were designed for guaranteeing execution to one single real-time application composed of hard real-time tasks. Everything is pre-determined and every constraint guaranteed a-priori under worst-case conditions.

Our interest is in being able to provide scheduling support in highly-dynamic systems where new applications may join the system at any instant, and currently active applications may leave the system. Furthermore, the behavior of each application cannot be completely a-priori characterized. In such systems, our scheduling goals are to provide:

- some kind of concrete *performance guarantee* to each admitted application, and
- inter-application *isolation* – each application should be protected from the potential misbehavior of other applications.

In [8], the *Bandwidth Sharing Server* (BSS) algorithm has been proposed to provide isolation and precise real-time execution to hard and soft applications. In this paper we will describe the BSS-I algorithm, an evolution of the BSS, that improves it in two ways:

- extending the local selection mechanism in order to handle different schedulers at the task level (section 5);
- introducing a new data structure called *Incremental AVL tree* to store the BSS internal status, in order to lower complexity (section 7).

2 Previous work

The problem of scheduling hard and soft multi-task applications in dynamic systems has already been addressed. A classic methodology for providing isolation

*Supported in part by the National Science Foundation (Grant Nos. 9704206 and 9972105).

and real-time execution is to assign each application a *server* with certain parameters which specify its desired performance. Each time a new application is activated in the system, an acceptance test is run: whether the application is accepted in the system or not depends on the server parameters and on the available system resources. The server parameters represent a *contract* between the application and the system: the goal of the system-wide scheduler is to provide each application the expected level of service. However, it is incumbent upon the server, and not the system-wide scheduler, to ensure that the tasks that comprise the application performs as expected and do not request more than specified.

Several service mechanisms have been proposed under RM [6, 5, 1, 12] and under EDF [9, 3, 11, 10], but each of them can handle only one single aperiodic task.

A very interesting approach to our problem can be found in [4]: the application level scheduler is EDF, whereas it is possible to select the most appropriate scheduling strategy at the task level. With this method it is possible to guarantee real-time execution for each application independently from the others, but an accurate guarantee test must be performed on each application as a misbehaved task can affect the schedulability of all the system.

In [2], Liu and Deng describe a two level 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 global scheduler, each application is handled by a dedicated TBS server [10] and it is also possible to choose more appropriate local scheduling strategies. The algorithm makes a distinction between *predictable* and *non-predictable* applications: an application is said to be predictable if it consists of periodic tasks or non-preemptable tasks. For non-predictable applications, it is necessary to specify a *scheduling quantum*, that is the minimum amount of time between two successive events in the application. Due to this mechanism, non-predictable applications can be guaranteed only paying some penalty in processor utilization.

In our opinion, this is a serious limitation of the algorithm in [2], because most of the applications in an open system are event-driven and preemptable. Moreover, in both the previous approaches, no isolation is provided: it is not clear what happens if a task is misbehaving and wants to execute more or arrives at an higher rate than specified.

In [8], the *Bandwidth Sharing Server* (BSS) algorithm has been proposed to provide isolation and precise real-time execution to hard and soft applications. Every application is handled by a custom server with a fixed bandwidth and a system-level scheduler, based on EDF, selects the application to execute. The BSS algorithm solves some of the problems of the previous approaches:

- no information on the application is needed other than the desired bandwidth;
- the scheduling strategy is *de-coupled* from the guarantee algorithm: no information is needed on the execution time or arrival rate in order to schedule the tasks. These information are only needed before run-time, in order to calculate the bandwidth to assign to the application server;
- there is no need to distinguish between predictable and non-predictable application; in particular, no processor bandwidth need to be wasted when scheduling hard-real-time tasks with a preemptive algorithm;
- the BSS *isolates* applications: a misbehaving task cannot affect the guaranteed performance of another application; hence, the BSS is particularly suitable for scheduling soft-real-time applications, i.e. applications for which we have no exact information on the worst case execution time or arrival rate of the tasks.

However, the BSS suffers two major limitations: its high complexity and the fact that the only scheduling policy supported at the task level is EDF. To deal with this limitations, in the following sections we present the BSS-I algorithm: in section 5 the local selection mechanism is extended in order to handle different schedulers at the task level; in section 7 a new data structure, called *Incremental AVL tree*, is introduced in order to lower complexity.

3 Definitions

In our model, a task τ_i is a finite or infinite sequence of requests for a shared resource (e.g. the CPU). Each job $J_{ij} = (a_{ij}, c_{ij}, d_{ij})$, is characterized by a request time (or arrival time) a_{ij}, a computation time c_{ij} and a deadline d_{ij}. There is no constraint on the arrival times: therefore, a task can be time-driven (periodic) or event-driven (sporadic or even aperiodic).

The meaning of d_{ij} depends on the task type. If the task is hard-real-time, d_{ij} represents the *absolute deadline*, that is time by which a job must complete in order for the application to be correct. If the task is soft-real-time, d_{ij} represents the time by which a job should complete, if there is enough resource available.

Informally, we can define the percentage of jobs that complete before the deadline as a measure of the Quality of Service (QoS) of a soft task: the more the jobs that complete before d_{ij}, the higher is the QoS experienced by the soft task.

Thus, it is the responsibility of the system designer to assign critical tasks a sufficient fraction of the processor bandwidth such that no deadline is missed. On the other end, if the application tasks are not critical, the system designer can also decide to assign resource shares basing on average values, so that deadlines can be met with a certain probability (QoS guarantee).

An application is a set of tasks:

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

In the following, symbol $\tau_i^{\mathcal{A}}$ will 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 the processor time that the application is allowed to use. Our system consists of a set of applications $\mathcal{S}$ which share the same resource. We assume that

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

4 The BSS-I Algorithm

In this section we give an overview of the BSS-I algorithm. The exposition is somewhat different from the one in [8], because we need to extend the algorithm to handle different scheduling policies. For details on the original algorithm, please refer to [8].

In Figure 1, the general system architecture is outlined. Each application **A** is handled by a dedicated *application server* $S^{\mathcal{A}}$ and it is assigned a share (or fraction or processor utilization, or *bandwidth*) U^A, with the assumption that the sum of the shares of all the applications in the system cannot exceed 1. The server maintains a queue of ready tasks: the ordering of the queue depends on the local scheduling policy.

Each time a task is ready to be executed in application **A**, the server $S^{\mathcal{A}}$ calculates a **budget B** and a **deadline d** for the entire application. The active servers are then inserted in a global EDF queue, where the *global scheduler* selects the earliest deadline server to be executed. It will be allowed to execute for a maximum time equal to the server budget. In turn, the corresponding server selects the highest priority task in the ready queue to be executed according to the local scheduling policy.

The server deadline is assigned by the server to be always equal to the deadline of the earliest-deadline task in the application (notice that, as per the description above, the task selected to be executed is chosen according to the local scheduler policy and might not be the earliest deadline task). The budget calculation will be described in the next section.

To better understand the dynamics of the system, in the following we describe the system events, the interface that the global scheduler module exports towards the application server and the interface that each server exports toward the global scheduler.

The interface that the kernel exports toward the server consists of only two functions:

activate(Server S, Budget B, Deadline d) : inserts server **S** in the global EDF queue with deadline **d** and budget **B**. If **S** becomes the earliest deadline server, then it is selected to execute (its *schedule()* function is called).

suspend(Server S) : extracts server **S** from the global EDF queue. If **S** was executing, then its *deschedule()* function is called; a new server is selected to be executed (its *schedule()* function is called).

Each server exports the following interface to the global scheduler:

schedule() : the server is selected to execute. In turn the highest priority task is chosen to be executed according to the local scheduler policy.

deschedule(time e) : the application is no longer executing, and the processor was held for **e** consecutive units of time; the server updates its internal data accordingly.

budgetExhausted(time e) : the budget is over and the application has been suspended. If the application consists of hard tasks, then the server raises an exception. If the application consists of soft tasks, the server calculates a new tuple (budget, deadline) and activates the application again.

The dynamics of the system are described by the following set of events:

Task arrival: a new task instance is released in an application. If the task is already active, the arrival is buffered. If the task is not active, it is inserted in the local ready queue: this could cause a *local preemption*. Also, if the newly activated task is also the earliest deadline task in the application, then the server

1. suspend itself calling the **suspend(S)** function;

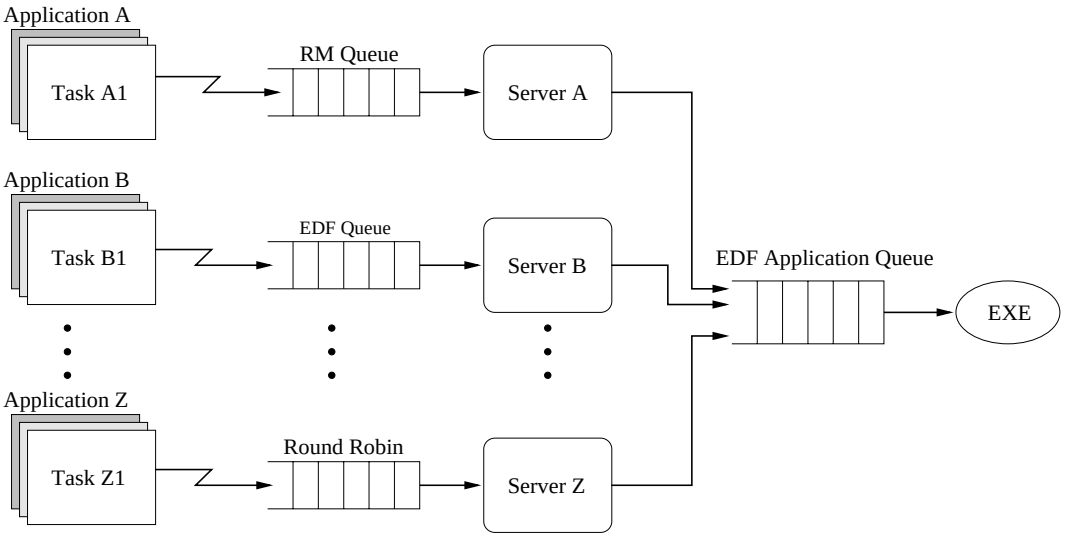

Figure 1. System Architecture.

2. calculates a new tuple **(B, d)**, where **d** is the earliest deadline among those of the active tasks in the application and **B** is calculated according to the algorithm described in section 4.1;
3. activates itself calling the **activate (S, B, d)**.

Since the server deadline has changed, a global preemption could occur. In this case, the global scheduler calls the **deschedule(E)** function of the preempted server, where **E** is the amount of time that the server held the processor; then it calls the **schedule()** function of the preempting server to signal that it has gained the processor.

Task end: a task instance finishes execution. If there is some buffered instance for this task, it is activated and inserted in the ready queue. If the ready queue becomes empty, then the server suspend itself, waiting for the next task arrival. Otherwise, the next task in the ready queue is selected to execute. If the finished task was the earliest deadline task in the application, then the server deadline changes: again the server **suspend** itself, calculates a new tuple **(B,d)**, and **activate** itself. Just as before, if the server deadline changes, a global preemption can occur.

Budget Exhausted: the server budget is exhausted. The global scheduler suspends the executing server and calls its **budgetExhausted()** function. In this case the server cannot execute any longer with the current deadline, otherwise some other application could be affected. We have two cases:

1. If the application consists of hard tasks, then something catastrophic could happen if the task misses its deadline. This can be considered a *fault*: therefore, an exception must be raised.
2. If the application is soft, we can simply *degrade* the level of service of the application lowering its priority. Hence, the server *postpone* its deadline, a new budget is calculated and the server activate itself again. In section 5 we discuss different methods of postponing the server deadline.

4.1 Budget calculation

In the BSS algorithm [8], in order to calculate the budget, every server uses a private data structure called *list of residuals*. We briefly recall here this mechanisms to explain how the budget for each server is calculated. In section 7 we will introduce a novel data structure, called Incremental AVL Tree with the same properties of the list of residuals but with a lower complexity.

For each task of an application **A**, the list of residuals $L^{\mathcal{A}}$ contains one or more of the following type of elements:

$$l = (B, d)$$

where

- d is the task's deadline;

- B is the budget available in interval $[a, d]$ (where a is the task's arrival time); that is, the maximum time that application **A** is 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 $L^{\mathcal{A}}$ is ordered by non-decreasing deadlines d. For the list to be consistent, the budgets must be assigned such that they are non-decreasing. Intuitively, this means that the total execution time allowed in an interval is never smaller than the execution time allowed in any contained interval.

The server assigns the application a pair (budget, deadline) corresponding to the element $l = (B, d)$ of the earliest deadline task in the application, regardless of the local scheduling policy. Only in the case the local scheduling policy is EDF, this element corresponds to the first task in the ready queue.

Two main operations are defined on this list: *adding* a new element and *updating* the list after some task has executed.

4.2 Adding a new element

A new element is created and inserted in the residual list when a newly activated task becomes the earliest deadline task among the ready tasks in the application. Let d_i be its deadline: 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 budget B_i is computed as:

$$B_i = min\{D_i U^{\mathcal{A}},\ (d_i - d_{k-1})U^{\mathcal{A}} + B_{k-1},\ B_k\} \quad (1)$$

where $U^{\mathcal{A}}$ is the bandwidth (share) assigned to application **A** and D_i is the task's relative deadline. At this point, the new element is completely specified as $l = (B_i, d_i)$ and can now be inserted at position k, so that the k-th element becomes now the $(k+1)$-th element, and so on.

The basic idea behind Equation (1) is that the budget for the newly arrived task must be constrained such that in any interval the application does not exceed its share. A typical situation is shown in Figure 2: when at time t task τ_i becomes the earliest deadline task, the

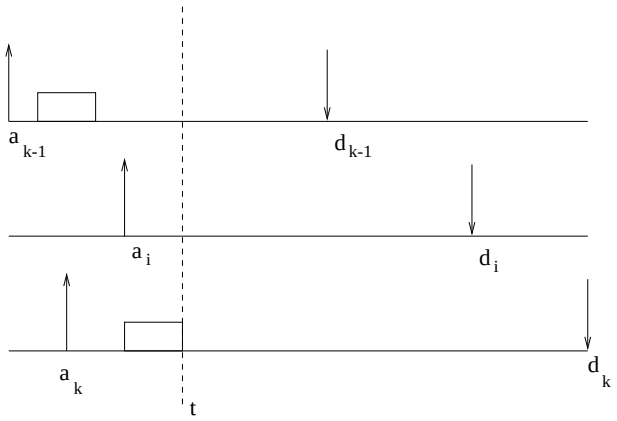

Figure 2. Computation of B_i

algorithm must compute a new budget: it must not exceed the share in interval $[a_i, d_i]$, which is $D_i U^{\mathcal{A}}$; it must not exceed the share in interval $[a_{k-1}, d_i]$ which is $B_{k-1} + (d_i - d_{k-1})U^{\mathcal{A}}$, and must not exceed the share in interval $[a_k, d_k]$ which is B_k. It can be shown that, if B_i is the minimum among these values, then the application will not use more than its share in any other interval.

4.3 Updating the list

Every time the application leaves the processor, (i.e. the *deschedule(time e)* is called), the list must be updated. It could happen for any of the following reasons:

- the task has finished execution;
- the budget has been exhausted;
- the application has been preempted by another application with an earlier deadline;
- *suspend()* has been called while the application was executing.

Then, the algorithm picks the element in the list corresponding to the actual deadline of the server, say the k-th element, and updates the budgets in the following way:

$$\forall l_j \quad j \geq k \quad B_j = B_j - e$$
$$\forall l_j \quad j < k \ \wedge \ B_j > B_k \ \rightarrow \ \textit{remove element } l_j$$

4.4 Deleting elements

We also need a policy to delete elements from the list whenever they are not necessary any longer. At time t, element l_k can be deleted if the corresponding task's instance has already finished and

- either $d_k \leq t$;
- or $B_k > (d_k - t)U^{\mathcal{A}}$.

It can be seen from Equation 1 that in both cases element l_k is not taken into account in the calculation of the budget. In fact, suppose that element l_i is being inserted just after l_k. Then

$$D_i U^{\mathcal{A}} = (d_i - t)U^{\mathcal{A}} < B_k + (d_i - d_k)U^{\mathcal{A}}$$

and $B_k + (d_i - d_k)U^{\mathcal{A}}$ cannot be chosen in the minimum. Suppose now that element l_i is being inserted just before l_k. Then

$$D_i U^{\mathcal{A}} = (d_i - t)U^{\mathcal{A}} < (d_k - t)U^{\mathcal{A}} < B_k$$

and B_k cannot be chosen in the minimum. Since l_k cannot contribute to the calculation of any new element, then it can be safely deleted.

5 Support for different scheduling policies

In [8], EDF was the only supported local scheduling policy. In this case, for each application, the server selects its currently active task with the earliest deadline; hence the server deadline is always coincident with the deadline of the first task in the ready queue.

In this work, it is possible to choose a different local scheduling policy for each server. The algorithm remains the same except that the task selected to be executed might not be the earliest deadline task. However, the server is always assigned a pair (budget,deadline) equal to element $l = (B, d)$ in the residual list corresponding to the earliest deadline task.

To clarify the mechanism, consider the example in Figure 3 in which two applications are scheduled by the BSS algorithm: application **A** consists of two tasks, $\tau_1^{\mathcal{A}}$ and $\tau_2^{\mathcal{A}}$ and it is served by a server with a bandwidth of 0.5 and with a Deadline Monotonic scheduler. Application **B** consists of only one task and it is served by a server with a bandwidth of 0.5 (since there is only one task, the local scheduling policy doesn't matter).

Let us concentrate our attention on application **A**:

- an instance of task $\tau_1^{\mathcal{A}}$ arrives at time $t = 0$ with deadline $d_1 = 10$ and an (as yet unknown) execution requirement $c_1 = 3$ units. Then the server calculates a budget $B_1 = 5$ and inserts a new element in the residual list:

  $$L^{\mathcal{A}} = \{(5, 10)\}$$

  Then the server invokes **activate(A,5,10)**, and the global scheduler puts it in the global ready queue. However, since the server of application **B** has an earlier deadline, the **schedule()** function is not called until time $t = 3$;

- At time $t = 3$ the global scheduler signals the server of application **A** that it can execute;

- At time $t = 4$ an instance of task $\tau_2^{\mathcal{A}}$ arrives with deadline $d_2 = 12$ and an execution requirement of $c_2 = 5$. According to the DM scheduler, since task $\tau_2^{\mathcal{A}}$ has a smaller relative deadline than task $\tau_1^{\mathcal{A}}$, then a *local preemption* is done. However, since the earliest deadline in application **A** is still $d_1 = 10$, the server budget and deadline are not changed.

- At time 8 the budget is exhausted: application **A** has executed for 5 units of time. The global scheduler suspends the server and invokes its **budgetExhausted()** function. The server first updates the list:

  $$L^{\mathcal{A}} = \{(0, 10)\};$$

  then it postpones its deadline: suppose the deadline is postponed by $T = 10$ units of time: $d_1' = d_1 + 10 = 20$, (see next paragraph for alternative rules for postponing the deadline). Now the earliest deadline in the application is $d_2 = 12$, and the server calculates a new budget equal to:

  $$B_2 = (d_2 - d_1)U^{\mathcal{A}} + B_1;$$

  and inserts it into the list:

  $$L^{\mathcal{A}} = \{(0, 10); (1, 12)\};$$

  Finally, it invokes **activate(A,1,12)**. Since it is again the earliest deadline server in the global ready queue, it is scheduled to execute.

- At time $t = 9$ task $\tau_2^{\mathcal{A}}$ finishes. The server updates the list:

  $$L^{\mathcal{A}} = \{(0, 10); (0, 12)\};$$

  Now the earliest deadline in application **A** is $d_1' = 20$. Then the server calculates a new budget and inserts it in the list:

  $$L^{\mathcal{A}} = \{(0, 10); (0, 12); (4, 20)\};$$

  finally, it invokes **activate(A, 4,20)**. Since it is not the earliest deadline server, another server is scheduled to execute.

It is important to notice that at time $t = 8$ the earliest deadline in the application has been postponed, and

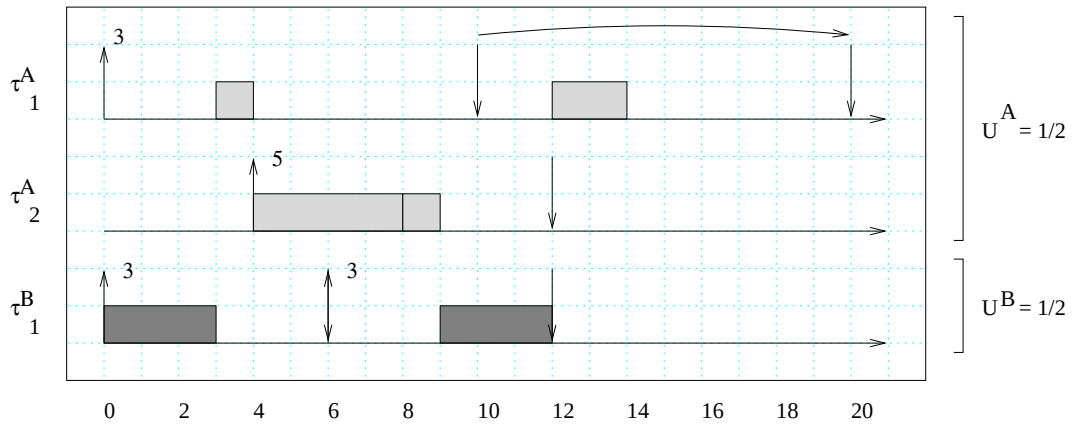

Figure 3. An example of schedule produced by the BSS algorithm: the two tasks in application A are scheduled by Rate Monotonic

this deadline can in general be different from the deadline of the executing task. Notice also that this framework is very general: it is possible to choose any kind of local scheduler. In particular, we can let tasks share local resources with any concurrency control mechanism, from simple semaphores to the more sophisticated Priority Ceiling or Stack Resource Policy.

Some consideration must be done for non-real-time applications whose tasks have no deadline. It also is possible to schedule these applications with the BSS-I algorithm if we provide a way to assign a deadline (and a budget) to the application. A simple way is to define an additional parameter T_N with the following meaning: each time a task of a non-real-time application is released at time t_a, it is assigned a deadline at $t_a + T_N$. This deadline has no meaning of criticality, but is only a way to give a priority to the application. The smaller the value of T_N, the higher the responsiveness of the non-real-time application, and the higher the run-time overhead for the system in terms of context-switch times, handling of deadline postponements, etc. Notice that, no matter the value of T_N, the other real-time applications in the system will not be influenced as long as the total utilization factor in the system is less than 1 (see section 6).

5.1 Rules to postpone deadlines

When the application budget is exhausted, the server cannot execute any longer with the current deadline otherwise some other application could be affected: thus, its deadline is postponed by a certain amount. Even though postponing the server deadline is necessary, it could cause a global preemption, as we are lowering the priority of the executing server. There are several ways to postpone the deadline depending on the desired level of service and on the local scheduling algorithm.

In [8], the proposed rule was to postpone the deadline of the executing task by an amount equal to the task's relative deadline; then the server selected the next task in the ready queue and calculated a new budget for it. This rule is intuitive and flexible, however it needs some adjustment for the case of local schedulers different from EDF. In fact, the earliest deadline in the application must be postponed in order to lower the server priority, and in general this deadline can be different from the deadline of the executing task. For example, in a static priority scheduler, after an event of budget exhaustion, the priority of the executing task remains the same but the server executes with a longer deadline.

Another possibility is to postpone the server deadline by a small fixed amount T_D. Suppose for example that, even if the budget is exhausted, the executing task needs to execute only a little more to complete. Thus, postponing the deadline by T_D could give the task the small amount of budget that is needed to complete without causing a preemption. However, if the task has an highly variable execution requirement, this rule could result in a large amount of unnecessary preemptions.

A good compromise between the two previous rules is to postpone the task deadline by an amount that increases exponentially: the first time a task instance exhausts its budget, its deadline is postponed by T_D; if the same instance exhausts its budget a second time, its deadline is postponed by $2T_D$; if it exhaust its budget

a third time, its deadline is postponed by $4T_D$; and so on. In this way, the system automatically adapts to the application needs. It is an engineering issue to select the most appropriate T_D in order to minimize the number of preemptions and the completion times of the soft tasks.

6 Formal properties

The BSS-I has two important properties:

- The **Bandwidth Isolation Property** says that, no matter the local scheduling algorithm, the execution times and the arrival rates of the tasks, no application misses its server's current deadline.
- The **Hard Schedulability Property** says that if an application **A** is schedulable when executed alone on a processor with speed $U^{\mathcal{A}}$, then it is schedulable when handled by a BSS-I with bandwidth $U^{\mathcal{A}}$ together with other applications on a processor with speed 1.

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 end, 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. Of course, the schedulability condition for an application depends on the local scheduling algorithm and on the application bandwidth. Here are some examples of schedulability conditions for different local schedulers:

Earliest Deadline First: Application **A**, which consists of periodic hard real-time periodic tasks, is schedulable if and only if:

$$\forall L > 0, \sum_{i=1}^{n} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i \leq U^{\mathcal{A}} L$$

where C_i, D_i and T_i are respectively the worst-case execution time, the relative deadline and the period for task i.

Rate Monotonic: Application **A**, which consists of periodic or sporadic tasks with deadlines equal to periods, is schedulable if:

$$\forall i = 1, \ldots, n \quad \sum_{j=1}^{i} \left\lceil \frac{T_i}{T_j} \right\rceil C_j \leq T_i U^{\mathcal{A}}.$$

Stack Resource Policy with EDF: Application **A**, which consists of periodic tasks with deadlines equal to periods, is schedulable if:

$$\forall i = 1, \ldots, n \quad \sum_{j=1}^{i} \frac{C_j}{T_j} + \frac{B_i}{T_i} \leq U^{\mathcal{A}}$$

where B_i is the maximum blocking factor for task i.

However, recall that the Hard Schedulability Property is valid for any local scheduling policy. The proofs for the Bandwidth Isolation Property and the Hard Schedulability Property are rather complex and cannot be reported here due to space limitations. We will give only an overview of the method used to prove the hard schedulability property. The complete proofs can be found in [7].

Proof Sketch. Suppose that the application is schedulable with the specified scheduling algorithm on a processor $P^{\mathcal{A}}$ with speed $U^{\mathcal{A}}$. It's easily seen that the application is schedulable by a BSS-I server that executes alone on processor $P^{\mathcal{A}}$ with assigned bandwidth equal to 1. Moreover, no task will finish after the computed server deadline. Let $\sigma^{\mathcal{A}}$ be the schedule produced for application $\mathcal{A}$ on processor $P^{\mathcal{A}}$ as described above and let us compute these schedules for every application in the system. We define the system tick T_s such that every event in the system is multiple of T_s. Now, let us consider the original system. It must be executed on a processor P with speed 1. Let us build a (feasible) global schedule σ in the following way: every tick we schedule application $\mathcal{A}$ for $T_s U^{\mathcal{A}}$, application $\mathcal{B}$ for $T_s U^{\mathcal{B}}$, and so on. This is equivalent to merging all the schedules using a global GPS scheduler. Since σ is feasible, and since EDF is optimal, then there will exist a feasible EDF schedule for the entire system.

7 Improving complexity bounds

It is easy to see that if the list of residuals $L^{\mathcal{A}}$ is implemented as a linear list, both the *insert* and the *update* operations take time $O(N)$, where N is the maximum number of elements in the list. In [8], we showed that N is bounded by $O(D_{max})$, where D_{max} if the maximum relative deadline among the tasks of the application; hence the complexity of an operation on the list of residuals is $O(D_{max})$ which is pseudo-polynomial in the input. To obtain a more efficient algorithm, we decided to implement the residual list as a novel data structure that we call *Incremental AVL tree*.

An AVL tree is a binary tree with the following properties:

- It is ordered: for each node n, every element in its left subtree is "less" than the element in node n,

and each element in its right subtree is not "less" than the element in node n, where "less" is a relation of ordering defined on the elements.

- It is balanced: for each node n the height of the left subtree differs from the height of the right subtree by 1, 0 or -1.

- the usual operations of insertion, deletion and search of an element take time $O(\log(N))$ where N is the number of elements in the tree.

An Incremental AVL Tree is introduced here as a particular kind of AVL tree in which elements are tuples (b, d) that correspond to elements in the residual list. The tree is ordered by increasing values of d. While in a residual list we store the budgets B, in an incremental AVL tree, we store the budgets in a *relative* way.

For each node n, we denote with *n.parent*, *n.left* and *n.right* respectively its parent node, its left child and its right child, and with $n.b$ and $n.d$ the relative budget and the deadline. The budget B in each node can be calculated as:

$$B = n.b + ref(n)$$

where *ref()* is a function defined recursively on n:

- if n is the *root* node, *ref(n)* returns 0;

- if n is the *left child* of its parent node, *ref(n)* returns *ref(n.parent)*;

- if n is the *right child* of its parent node, *ref(n)* returns *n.parent.b + ref(n.parent))*;

Of course, the insertion, update and deletion operations for an AVL tree must be re-written to take into account the fact that the contents of a node depends on its position in the tree. Due to space constraints, we cannot report the complete implementation of this data structure. In the following we will give some hint on how these operations can be implemented. More details can be found in [7].

Insertion. When inserting a new element, the algorithm explores the tree in order to find the correct position. It starts descending from the *root* of the tree: if the deadline of the element to insert is greater than or equal to the deadline of the current node, the algorithm inserts it in the right subtree; if the deadline of the element to insert is less than the deadline of the current node, the algorithm inserts it in the right subtree. As the algorithm descends through the tree, it keeps track of the relative budget according to the previous rule; when finally it finds the correct position for insertion, the new element is assigned the relative budget.

After an insertion, the tree could result unbalanced. Suppose for example that the right subtree of node n has an height equal to h and the left subtree has an height equal to $h + 2$. In an AVL tree, to re-balance the tree, we need to perform one or two *rotations*, depending on the structure of the left subtree. In an Incremental AVL tree, the relative budget of the nodes to be rotated must be updated accordingly.

Update. After the server has consumed E units of execution time with deadline d, the data structure must be updated. As explained in section 4.3, we need to find the element with deadline equal to d, and subtract E from the budget of this element and from all the following ones. Moreover, we have to ensure the consistency of the list deleting all the elements that have a deadline smaller than d and a budget greater than B. This can result in a great number of elements to delete. To minimize the number of operations we use a technique called "lazy deletion": briefly, we avoid to delete an element during an update operation and we update only the elements in the path from the root to the node with deadline d. The other elements will be updated the next time the algorithm descends the tree.

Deletion. To minimize the number of operations to perform, the algorithm deletes only when the current time is greater than the deadline of the root element. At that point, it deletes at once the root element and its left subtree. As a consequence, the right child becomes the new root. Of course, doing lazy deletion, we keep in the tree more elements than necessary. However, this does not increase the overall complexity of the algorithm. It can be shown that the maximum number of elements in the tree is $O(2D_{max})$, where $D_{\max}$ is the largest relative deadline in the application.

7.1 Complexity

It has been shown that the complexity of the insert and update operations in a balanced binary tree is $O(log(N))$, with N number of elements in the tree. According to the discussion in the previous section, $N = O(2D_{max})$. Since the complexity of implementing the ready queue in each server can be as low as $O(log(N_{\mathcal{A}}))$ (where $N_{\mathcal{A}}$ is the number of tasks in application **A**), the overall complexity of the BSS algorithm is $O\left\{log(N_{MAX}) + log(N_{App}) + log(D_{MAX})\right\}$ where N_{App} is the number of applications and D_{MAX} and N_{MAX} are respectively the maximum relative deadline and the maximum number of tasks among all the applications.

Currently, we are doing simulation experiments to

compare the average and the maximum complexity of the insert and update function in the incremental AVL tree and in the linear list, as a function of the number of tasks and of the number of applications.

8 Conclusions and future work

If general-purpose computers are to support both real-time and non-real-time applications, it is important that (i) *performance guarantees* be provided to individual real-time applications, and (ii) each application be *isolated* from the potential misbehavior of other active applications.

The BSS approach [8] is a provably optimal approach to processor scheduling that can provide both performance guarantees and inter-application isolation. The major drawback of BSS was its computational complexity. Also, each application was required to schedule its own tasks according to earliest deadline first algorithm.

In this paper, we have described an evolution of the BSS algorithm, called BSS-I, aimed at improving it in two ways:

- generalizing the server mechanism; we managed to de-link each application's internal scheduling strategy from the BSS specification, so that it is now possible to choose an arbitrary scheduling discipline for the application;

- proposing a new data structure to implement the lists of residuals that are maintained by BSS. By storing these lists of residuals as balanced binary trees rather than as linked lists/arrays, we have obtained a significant reduction of the complexity, and therefore reduced the run-time overhead.

As future work, we are considering several issues. Until now, applications have been considered to be independent. In the future we plan to further extend the BSS algorithm to permit different application to access global-shared resources. We believe that this problem is crucial for an implementation of techniques based on bandwidth allocation in a real operating system.

Another interesting issue is the reclaiming of unused bandwidth. In fact, during the system life, some application can be temporarily idle; also, there can be time intervals in which the system load is strictly less than 1. Then, the active applications could take advantage of this unused bandwidth improving their quality of service and performance expectation.

References

[1] N.C. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(8):284–292, Sep 1993.

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

[3] T.M. Ghazalie and T.P. Baker. Aperiodic servers in a deadline scheduling environment. *Journal of Real-Time System*, 9, 1995.

[4] 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.

[5] J.P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1992.

[6] J.P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1987.

[7] G. Lipari. *Resource Reservation in Real-Time Systems.* PhD thesis, Scuola Superiore S.Anna, Pisa, Italy, under preparation.

[8] G. Lipari and G.C. Buttazzo. Scheduling real-time multi-task applications in an open system. In *Proceeding of the 11th Euromicro Workshop on Real-Time Systems*, York, UK, June 1999.

[9] M. Spuri and G. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1994.

[10] M. Spuri and G.C. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Journal of Real-Time Systems*, 10(2), 1996.

[11] M. Spuri, G.C. Buttazzo, and F. Sensini. Robust aperiodic scheduling under dynamic priority systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1995.

[12] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Journal of Real Time Systems*, 6(2):133–151, Mar 1994.