Serving Non Real-Time Tasks in a Reservation Environment

Luca Abeni

University of Trento
Via Sommarive 14, I-38100 POVO (TN) - Italy
luca.abeni@unitn.it

Claudio Scordino

Computer Science Dept., University of Pisa
Pisa, Italy
scordino@di.unipi.it

Giuseppe Lipari

Scuola Superiore Sant'Anna
Pisa, Italy
lipari@sssup.it

Luigi Palopoli

University of Trento
Via Sommarive 14, I-38100 POVO (TN) - Italy
palopoli@dit.unitn.it

Abstract

Resource reservations have been proven to be an effective technique to support hard and soft real-time applications in open systems, and some implementations for Linux have already been proposed in the past. However, such implementations generally focus on providing guarantees to real-time applications, disregarding the performance of non real-time activities. In this paper, the problems encountered while using a reservation-based scheduling algorithm in Linux are described, showing why the original algorithm is not suitable for scheduling non real-time activities. Then, the properties required for properly scheduling non real-time tasks are described, and a novel algorithm (called HGRUB) is analysed, showing how it effectively addresses the presented issues. The performance of HGRUB are then evaluated (by using our implementation in Linux) and compared with the performance of traditional reservation systems.

1 Introduction

Recently, considerable effort has been devoted to support real-time activities in such general-purpose operating systems (GPOSs) as Linux. This kind of real-time support is important for a new class of time-sensitive applications (audio/video players, video on-demand servers, teleconferencing systems, IPTV streamers, but also CD/DVD burners, soft modem drivers and video-games), but is also useful for supporting more "traditional" real-time applications running on embedded boards.

As an example of this trend, there is a strong on-going effort to reduce the kernel latencies in Linux: Ingo Molnar and other Linux kernel developers have recently proposed a kernel patch called preempt-rt [1] to reduce the maximum interrupt latency down to a few tens of microseconds. Because of these improvements, the ideal goal of supporting both real-time and non real-time applications on the same OS has

1

become achievable. However, reducing interrupt latency is not *per se* sufficient without the support of appropriate scheduling strategies. Indeed, a GPOS is an open system, where applications can dynamically activate at any time, and the scheduler cannot make any restrictive assumption on the characteristics of the programs.

Resource Reservations [14] have emerged as an effective technique to support time-sensitive applications on such GPOSs as Linux (for example, see Linux/RK [10, 15]). This technique provides support for time-sensitive applications by allowing the integration of classical real-time techniques, developed to meet timing constraints on RTOSs, with the general-purpose allocation strategies used on GPOSs.

CPU reservations have been traditionally implemented by using a dedicated aperiodic server (in particular, the Deferrable Server [18]) per reserved task [14, 15]. Unfortunately, this implementation strategy generates problems when tasks block and unblock dynamically [18]. This particular problem has been solved by the Constant Bandwidth Server (CBS) class of algorithms [2], which uses dynamic priorities to correctly cope with dynamic aperiodic arrivals.

However, most of the existing work in this area mainly focused on providing guarantees to real-time applications, disregarding the performance of non real-time activities. As a result, the proposed solutions strongly penalise batch and interactive processes: for example, although algorithms like the CBS can provide a predictable QoS to real-time activities [3], they cannot be used to schedule generic tasks, because of the so called *"Greedy Task"* and *"Short Period"* problems. The first contribution of this paper is to identify such problems, and to formally define the properties needed for serving such applications (namely, the worst-case fairness property).

As a second contribution, a scheduling algorithm (HGRUB - Hard GRUB) that provides the desired property is designed to address the issues presented above. This algorithm harmonically blends three different ideas: provides the same performance guarantees to real-time applications as a traditional reservation algorithm (like the CBS), efficiently reclaims unused CPU time, and provide some kind of fairness to non real-time applications. These results are achieved by combining a hard reservation behaviour [15] with a reclaiming mechanism [9].

Finally, the algorithm has been implemented in the Linux kernel, and used to schedule real-time applications along with interactive and batch tasks, confirming that HGRUB is able to properly serve both real-time tasks (providing the same QoS guarantees provided a "traditional" reservation-based algorithm) and non real-time tasks (providing worst-case fairness and good throughput to batch applications, and responsiveness to the interactive ones).

The rest of the paper is organised as follows. Section 2 introduces the scheduling model that will be used throughout the paper, and briefly recalls the basic Resource Reservation concepts. Section 3 presents the problems address in this paper, and introduces the HGRUB algorithm as an effective solution. A formal description of such algorithm is contained in Section 4, whereas Section 5 presents some experimental results. Finally, Section 6 presents the conclusions, open issues and future work.

2   Background

This section defines the objectives of this work, and introduces the formal notation and scheduling model used in the paper. Then, the Resource Reservation approach is presented, together with a brief overview of the related work.

2.1   Scheduling model

Although the techniques and principles described here are quite general, this paper only focuses on CPU scheduling. To simplify the description, the following definitions are introduced:

Definition 1 *A real-time task[1] τ_i is a stream of jobs (or instances) $J_{i,1}$, $J_{i,2}$, $J_{i,3}$, ..., where $J_{i,j}$ becomes ready for execution (arrives) at time $r_{i,j}$ ($r_{i,j} \leq r_{i,j+1} \quad \forall i,j$), requires a computation time $c_{i,j}$, and finishes at time $f_{i,j}$.*

Each job $J_{i,j}$ is also characterised by a deadline $d_{i,j}$, that is respected if $f_{i,j} \leq d_{i,j}$, and is missed if $f_{i,j} > d_{i,j}$. A real-time task τ_i is said to be *periodic* if $r_{i,j+1} = r_{i,j} + P_i$, where P_i is the *task period*. In an open system, tasks are often aperiodic, and it is not possible to assume strict relations between $r_{i,j}$ and $r_{i,j+1}$.

Definition 2 *A task τ_i with no deadlines associated to its instances is said to be* non real-time. *Non real-time tasks can be interactive or batch.*

- A non real-time task τ_i is *interactive* if it blocks and unblocks reacting to some external event. Usually, it requires a (relatively) short computation time between consecutive blockings.

- A non real-time task is said to be *batch* if it is (almost) continuously active, and it never (or very rarely) blocks. Scientific computation programs are the typical example of such tasks.

---

[1] In this paper we use the word "task" to indicate a generic scheduling entity, that can be either a thread or a process.

The goal of a scheduler for these tasks is to fairly allocate the processor to all batch programs, so that they can all progress continuously.

2.2 Objectives

The objective of this paper is to identify a scheduling algorithm that can be used for scheduling both hard real-time, soft real-time and non-real-time tasks. More formally, such scheduling algorithm must comply with the following specifications:

1. Provide timely service to *hard* and *soft* real-time tasks. In other words, it must provide both deterministic real-time guarantees and some form of more "relaxed" guarantees (for example, based on probabilistic deadlines)

2. Reduce the response time of interactive tasks as much as possible

3. Fairly allocate the CPU bandwidth to batch tasks

4. Maximise the throughput. More specifically, the scheduler should never idle the processor when there is some active task; moreover, it should avoid unnecessary preemptions and context switches.

It is important to point out that this paper focuses on the scheduling algorithm, and not on the policies for assigning the scheduling parameters. In particular, this paper does not address the problem of dynamically assigning and modifying the scheduling parameters for soft real-time and non-real-time tasks. Many papers deal with heuristic algorithms and feedback control schemes for adapting the scheduling parameters to the need of the application [5]. Instead, we want to completely separate the problem of scheduling from the problem of selecting the scheduling parameters [2].

While the goal in scheduling real-time tasks can be precisely defined (all deadlines must be respected, or some QoS metric dependent on the deadlines must be maximised), the goal in scheduling interactive and batch tasks is more fuzzy. To help in identifying such goal, the next sections highlight some bad behaviours obtained when applying resource reservations to interactive and batch tasks.

Since reservation-based schedulers have been originally designed to integrate real-time and non real-time tasks (and they seem to provide most of

the properties mentioned above) this paper focuses on such class of algorithms.

2.3 Resource Reservations

The basic idea behind the Resource Reservation technique is to *reserve* a fraction of the time to time-sensitive applications through real-time scheduling. In this way, real-time priorities can be securely used even by non-privileged users (it is interesting to note how Linux developers arrived to similar conclusions through a completely different path [12, 11]).

The main advantage of this technique is the ability to provide *temporal protection* between tasks, meaning that the temporal behaviour of each task is isolated from the rest of the system, and it is possible to guarantee the real-time performance of a task by considering it in isolation. Such property is particularly useful when mixing hard and soft real-time applications on the same system.

A Resource Reservation RSV_i on a resource $\mathcal{R}$ for a task τ_i is described by the tuple (Q_i, T_i), meaning that the task is *reserved* the resource $\mathcal{R}$ for a time Q_i in a period T_i. Q_i is also called *maximum budget* of the reservation, and T_i is called *reservation period*.

The first example of a Resource Reservation algorithm was the CPU Capacity Reserve proposed by Mercer and Tokuda [14]. According to this algorithm, each task is assigned a *budget* q_i that is decreased during task execution. The CPU is scheduled according to Fixed Priority and a task is allowed to execute only when its budget is greater than 0. The budget is periodically recharged to Q_i every T_i time units. This is basically the behaviour of the Deferrable Server (DS) algorithm [18], where each task is served by a dedicated server. As a result, aperiodic activations and deactivations can break the reservation behaviour, as shown in [4].

The CBS algorithm [2] addresses this problem by using dynamic priorities: each task τ_i is scheduled through an abstract entity S_i called *server*, which is responsible for assigning τ_i a *scheduling deadline*, denoted by d_i^s, according to the algorithm rules. Then, the algorithm orders all active servers in an Earliest Deadline First (EDF) queue by their scheduling deadlines. The server with the earliest scheduling deadline is selected for execution. Like all the reservation-based algorithms, the CBS is characterised by three mechanisms: *accounting*, *enforcement*, and *replenishment*. Accounting is based on the server *budget* q_i, which measures the consumed CPU time according to the following rule:

---

[2]We believe that the problem of selecting the server parameters is a higher level problem that depends on the characteristics of the application. Moreover, a good scheduler with certain good properties forms a strong base for building solid feedback control schemes and heuristics.

- when the task served by S_i executes for a time δt, q_i is decreased by δt

Enforcement and replenishment are performed according to the following rule:

- when the budget arrives to 0 and the task has not yet completed its execution, the budget is *immediately* recharged, the server scheduling deadline is postponed to $d_i^s = d_i^s + T_i$, and the EDF queue is reordered (hence, a preemption *may* occur)

If, after postponing the server deadline, the server is still the earliest deadline server, the task continues to execute; note that this behaviour implements *soft reservations*, as defined in [15]. A more formal description of the CBS algorithm is provided in Appendix A.

A CBS is characterised by the following property:

Property 1 *If a task served by a server $S_i = (Q_i, T_i)$ is activated for the first time at time t_0, and has an execution requirement of C_i, it will receive at least $\max(C_i, kQ_i)$ units of execution within $t_0 + kT_i$, for all $k = 1, \ldots, \left\lceil \frac{C_i}{Q_i} \right\rceil$.*

Since the CBS can properly cope with aperiodic activations, in this paper it will be used as a baseline reservation-based algorithm.

2.4 Algorithms based on CBS

Over the past years, some other reservation algorithms which use basic CBS techniques to cope with dynamic activations and deactivations have been proposed. They can be classified according to the following properties:

Definition 3 *An algorithm implements* hard reservations *if, when the server budget is depleted, the task is suspended until the next replenishment time. This rule is the opposite of the* soft reservation *rule used by the original CBS*

Definition 4 *An algorithm implements* reclaiming *if it is able to reclaim the excess bandwidth from other servers executing less than their allocation, and/or by other unallocated bandwidth in the system.*

The IRIS algorithm (Idle-time Reclaiming Improved Server, [13]) adds the hard reservation property to CBS and uses such a property to do reclaiming. The idea is that, when the processor becomes idle, then it is possible to anticipate the recharging time of all servers. The algorithm is very simple and effective, but it might suffer from an excessive number of context switches, as explained in the original IRIS paper.

The GRUB (Greedy Reclamation of Unused Bandwidth, [9]) algorithm adds reclaiming to the CBS algorithm. It introduces the concept of *state* of a server, and keeps track at each instant of the sum of the bandwidths of the *active* servers to efficiently reclaim all unused bandwidth in the system, both from unallocated system bandwidth and from servers executing less than allocated. GRUB can efficiently handle both periodic and aperiodic tasks, and can be used in open systems.

3 Non Real-Time Tasks

Resource Reservation algorithms have been designed to allow scheduling mixtures of hard real-time, soft real-time and non real-time tasks so that the real-time tasks are not penalised by the other tasks running in the system (temporal protection).

However, when using an algorithm like the CBS in a GPOS it is possible to notice some limitations that make it inadequate for open systems. First of all, it could be difficult to correctly dimension the reservations for non real-time tasks, because such (batch or interactive) tasks are often characterised by an intrinsically non periodic behaviour. Moreover, batch tasks are often CPU hungry (so, their execution time will generally be much larger than the reserved budget). In this situation, a *soft reservation rule* allows the served task to execute for more time than reserved (if such time is not used by other reservations), but in doing so it postpones the scheduling deadline, creating some scheduling anomalies.

3.1 Issues with the Original CBS

Consider two batch tasks τ_1 and τ_2 served by two servers with the same period. If the two tasks are activated simultaneously, the CBS guarantees that each of the two tasks gets the proper share of the processor in every interval of time. However, if τ_1 is activated at time t_1 and τ_2 is activated at time $t_2 >> t_1$, according to the CBS rules τ_1 is scheduled without interruptions in the interval $[t_1, t_2]$. As a consequence, the scheduling deadline d_1^s is postponed many times and can soon assume a large value. This phenomenon is called *"deadline aging"*.
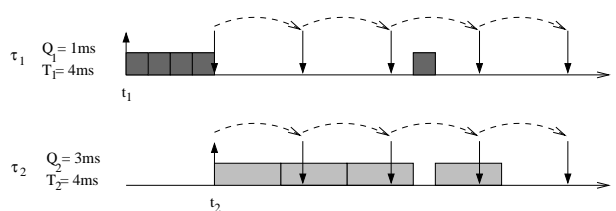

FIGURE 1: *The "Greedy Task" problem.*

The effects of deadline aging are visible in Figure 1, showing two tasks τ_1 and τ_2 served by two CBSs S_1 and S_2 with parameters $Q_1 = 1ms$, $Q_2 = 3ms$, and $T_1 = T_2 = 4ms$, when τ_2 is activated $4ms$ after τ_1. By looking at the figure, it is possible to see that from time t_1 to $t_1 + 4ms$, τ_1 is the only active task in the system, so it is allowed to execute and its deadline is postponed every time the budget is exhausted (at time $t_1 + 1ms$, $t_1 + 2ms$, $t_1 + 3ms$, and $t_1 + 4ms$). As a result, when τ_2 is activated at time $t_2 = t_1 + 4ms$ its scheduling deadline $d_2^s = t_1 + 8ms$ is much smaller than $d_1^s = t_1 + 16ms$, and τ_2 is continuously scheduled until $t_1 + 13ms$ when the two deadlines d_1^s and d_2^s are comparable. As a result, τ_1 is not executed for a large interval of time. After that, the schedule continues as expected and the CPU is fairly shared between the two tasks. In the following, we will refer to this phenomenon as the *"Greedy Task"* problem.

Intuitively, the greedy task problem is related to the fact that τ_1 (which being a batch task is CPU-intensive) starts to consume its *future* reserved time, because there is no other task ready for execution. When τ_2 is activated, τ_1 has consumed much of its future reserved time, therefore it cannot use the processor until the task τ_2 "catches up".

A second problem presented by a CBS when serving non real-time tasks occurs in presence of servers with different periods. In this case, the server period can be thought as a measure of the "granularity" of the resource allocation — i.e., how often a task (or an application) is allocated the reserved budget. So, a naive user might tend to use "short period" reservations to serve batch or interactive tasks that need to execute often. However, when the difference between reservations' periods is too large it often happens that "short period" servers are not scheduled often enough, causing unexpected execution patterns. Again, this problem is related to the *soft reservation* rule of the CBS. When a "short period" reservation is depleted, the scheduling deadline is postponed but it might still remain the shortest one. Hence, the served task continues to execute for several server instances, as if it was served by a reservation with a longer period. In the following, we will refer to this problem as the *"Short Period"* problem. As an example of the short period problem, consider two batch tasks τ_1 and τ_2 served by two servers with $Q_2 = 10.5Q_1$ and $T_2 = 10.5T_1$. The schedule produced by the CBS algorithm is shown in Figure 2 (note that τ_1 does *not* execute every T_1 time units).
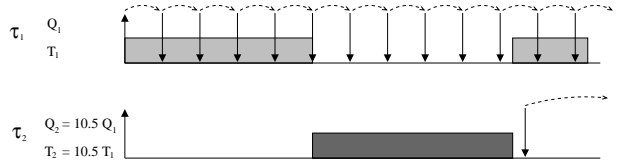

FIGURE 2: *The "Short Period" problem.*

The issues shown above are particularly critical for the global scheduler in hierarchical scheduling systems [8], consisting of two levels of schedulers. In such systems, a *global scheduler* can select a second level scheduler (also known as local scheduler), which in turn selects the task to execute. When a reservation-based scheduler is used as a global scheduler, it is expected to provide temporal isolation between groups of tasks (so that a group of tasks can be guaranteed independently from the other ones). However, if a server allows the served tasks to consume their reserved time in advance, this property is broken.

For example, consider a server used as a global scheduler for an application with two real-time tasks τ_1^a and τ_1^b, the first one with a short deadline and the second one with a large deadline (and large execution requirement). In order to ensure that τ_1^a meets all deadlines, the global scheduler is based on a reservation with a short period (less than the task deadline). However, if the reservation mechanism allows τ_1^b to consume the reserved time in advance, it can postpone the scheduling deadline by a large amount before τ_1^a arrival. So, when τ_1^a activates the server deadline is very large and the server may not allow its tasks to execute for a long interval of time, causing a potential deadline miss for τ_1^a.

All these examples show that even if the CBS guarantee (see Property 1) is respected, the schedule is not "fair enough" (if τ is activated at time t_0 and is continuously active in $[t, kT_i]$, it is not guaranteed that in $[(k-1)T_i, kT_i]$ the task executes for a time Q_i). This issue is similar to the problem of the fairness of the WFQ algorithm, highlighted in [7, 6] — similarly to WFQ, whose primary concern are the virtual finishing times, the original CBS algorithm is only concerned with deadlines, but allows the task to execute "too early". As noted by Bennet and Zhang [7], there are cases (for example, hierarchical scheduling) in which a better fairness is needed. Such an issue was addressed by the WF^{2}Q algorithm by introducing the *worst-case fairness* property [6]. Similarly, we introduce the worst-case fairness property for reservations.

Property 2 *A reservation-based scheduling algorithm is said to be* worst-case fair *if a server S_i with*

budget q_i at time t is guaranteed to consume its budget before $t + T_i$ (that is, if the server budget at time t is q_i then the served task is guaranteed the possibility of executing for at least q_i before $t + T_i$).

It can be shown that Property 2 is equivalent to worst-case fairness as defined by Bennet and Zhang: informally speaking, if a server is guaranteed to consume its budget in a time T_i, then the served task can execute for Q_i time units every T_i, and this permits to bound the job finishing time.

The "Greedy Task" problem shows that the original CBS algorithm is not worst-case fair because, for example, server S_1 in Figure 1 has $q_1 > 0$ at time $t_2 = t_1 + 4ms$, but can consume it only at time $t_1 + 13ms$ (similarly, in Figure 2 server S_1 has budget $q_1 = Q_1$ at time $5T_1$, but cannot consume it before $5T_1 + Q_2$).

To avoid the two problems presented above, the Property 2 (worst-case fairness) is required (in addition to Property 1) when serving non real-time tasks.

It is important to point out that the original CBS algorithm has been designed to serve real-time tasks, which are characterised by the model defined in Section 2.1. In this case, it is possible to compute the server parameters Q_i and T_i so that the task can receive a predictable QoS [3], and the "Greedy Task" and the "Short Period" problems are not a concern. This is the reason why such problems were observed only when the CBS has been implemented in the Linux kernel and their relevance was only understood when the CBS algorithm has been used to schedule generic non real-time (interactive and batch) tasks, which can easily act as greedy.

3.2 HGRUB

It is possible to note that worst-case fairness can be easily provided by depleting a reservation when its budget arrives to 0, and postponing the replenishment to a later time - generally corresponding with the scheduling deadline. Since this is the essence of the hard reservation behaviour, it might be natural to think that a generic algorithm based on hard reservations is adequate for serving non real-time tasks. However, a straightforward implementation of the hard reservation behaviour can make the scheduling algorithm *non-work-conserving* (more specifically, it may happen that all active tasks are suspended waiting for recharging their server budgets, and the CPU is idle). Therefore this approach has an adverse effect both on the throughput of batch tasks and the responsiveness of interactive tasks failing to comply with requirement 4.

The solution to this efficiency issue can be found by looking at the "Greedy Task" problem, which

seems to indicate that some kind of reclaiming mechanism can avoid deadline aging (helping to cope with tasks that consume their "future" reserved CPU time). That is to say, an algorithm performing CPU reclaiming is very efficient (it never leaves the CPU unused when at least one task is ready for execution) and solves the "Greedy Task" problem. However, it is not worst-case fair. In fact, the "Short Period" problem shows that a reclaiming mechanism alone is not enough; for example, a soft reservation behaviour results in the possibility to consume execution time "in advance" breaks the worst-case fairness even in presence of CPU reclaiming.

Since CPU reclaiming and the hard/soft reservation behaviour are orthogonal properties, they can be combined in a scheduling algorithm; in particular, if GRUB reclaiming is combined with a hard reservation behaviour then the resulting algorithm (called HGRUB) is both efficient and worst-case fair. For this reason in the rest of the paper we will focus on HGRUB.

4 Formal HGRUB Description

This section formally describes the HGRUB Algorithm, which is similar to the GRUB algorithm [9] (which in turns is a variation of the CBS algorithm [2]). Basically, HGRUB adds reclamation and a hard reservation behaviour to the CBS, ensuring at the same time that the algorithm remains work-conserving. The HGRUB algorithm is described based on the original CBS algorithm (see Appendix A), by showing how the original accounting mechanism is modified by GRUB, and how the enforcement mechanism is modified to implement a hard reservation behaviour.

4.1 GRUB Reclaiming

GRUB reclaiming is implemented by introducing a global variable U_{act} representing is the sum of the bandwidth $U_i = \frac{Q_i}{T_i}$ of all servers that are not inactive and is updated as follows:

- At system startup, U_{act} is initialised to 0;

- When τ_i activates at time t, if $q_i < (d_i^s - t)U_i$ (the current scheduling deadline is used) U_{act} is unchanged, otherwise (if a new scheduling deadline is generated and the budget is recharged) $U_{act} = U_{act} + U_i$;

- When τ_i stops (a job finishes) at time t, if $q_i \geq (d_i^s - t)U_i$ then $U_{act} = U_{act} - U_i$. Otherwise, U_{act} will be decreased by U_i later, at time $t = d_i^s - \frac{q_i}{U_i}$

Then, the accounting rule is modified in this way:

- While task τ_i executes, the server budget is decreased as $dq_i = -U_{act}dt$ (**GRUB accounting rule**)

Note that in the original GRUB paper different server states were used to update U_{act}.

4.2 Hard Reservations and GRUB

The last CBS rule (enforcement rule) indicates that the original CBS implements a soft reservation behaviour. If a hard reservation behaviour is desired, the enforcement rule must be modified as follows:

- When the budget is exhausted ($q_i = 0$), the server is said to be *depleted*, and the served task is not allowed to be scheduled until $t = d_i^s$. At such time, the scheduling deadline is postponed ($d_i^s = d_i^s + T_i$) and the budget is recharged to Q_i (**hard enforcement rule**)

The HGRUB algorithm combines the hard enforcement rule and the GRUB accounting rule to obtain the worst-case fairness property. Since this combination risks to make the algorithm non-work-conserving, a second global variable R, indicating the *residual budget* has been added. R is updated as follows:

- When τ_i stops (a job finishes) at time t, if $q_i \geq (d_i^s - t)U_i$ then $R = q_i - \frac{(d_i^s - t)}{U_i}$

This residual budget R is given to the next executing server, let it be S_j: $q_j = q_j + R$. If no server is eligible for execution (that is, all the servers are depleted or inactive), then:

- If there is at least a depleted server (a server with $q_i = 0$ that has not been replenished yet), then it is safe to give R to the depleted server having the earliest scheduling deadline (so, its budget is replenished earlier than expected);

- If there are no depleted servers, then it is safe to reset U_i to 0 discarding R.

5 Experimental Results

The effectiveness of the HGRUB algorithm, has been tested by modifying an implementation of the CBS scheduler in the Linux kernel [4, 17].

To minimise the amount of modifications needed to the kernel, the CBS scheduler has been implemented as a kernel module. In this way, the scheduling module can be inserted into (and removed from)

the kernel at runtime, and most of the code is independent from the kernel version (for example, porting the scheduler from the 2.4 kernel to 2.6 has been quite simple). The scheduling module works by intercepting job arrivals (i.e., task unblockings) and job terminations (i.e., task blockings), and is notified whenever a task is created or destroyed.

Some experiments showing the overhead introduced by the scheduler and the efficiency of its implementation are already presented in [4]; this paper only presents experiments showing the differences between the schedules produced by various CBS-based algorithms. To do this, the CBS scheduler has been extended to implement hard reservations, GRUB reclaiming and the HGRUB algorithm.

5.1 Schedule Correctness

First of all, a set of experiments has been run to verify the correctness of the HGRUB algorithm. In particular, this set of experiments has reproduced the "Greedy Task" and the "Short Period" problems, to see how the HGRUB algorithm copes with them. These experiments have been performed by using a measurement technique similar to the one used by Hourglass [16].
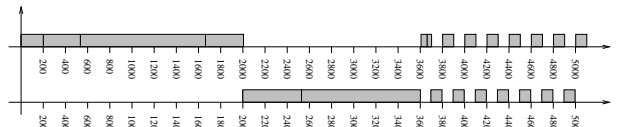

FIGURE 3: *Pathological schedule produced by CBS with the "Greedy Task" problem.*

The "Greedy Task" problem, shown in Figure 1, can be simply avoided by using the GRUB reclaiming mechanism. This has been shown by using two batch tasks τ_1 and τ_2 served by two identical CBSs with parameters ($100ms, 500ms$), and by activating τ_2 2 seconds after τ_1. The schedule produced by the CBS scheduler is shown in Figure 3, whereas Figure 4 shows the schedule generated by GRUB. By comparing the two figures it is possible to see that GRUB avoids the problem by allocating to τ_1 the amount of CPU bandwidth not used and not reserved during the first two seconds. In this way, when τ_1 is the only active task, its scheduling deadline is not postponed too much, and when τ_2 activates the two scheduling deadlines are comparable.
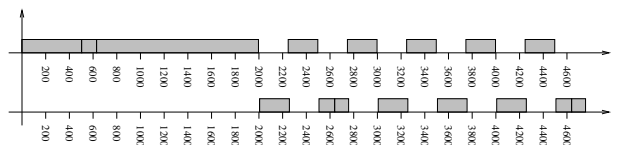

FIGURE 4: *CPU reclaiming performed by GRUB on the "Greedy Task" problem.*

Also the use of a hard reservation technique could alleviate the "Greedy Task" problem, but in this case τ_1 would execute for less time (see Figure 5).
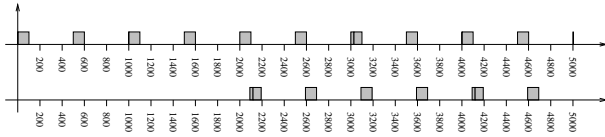

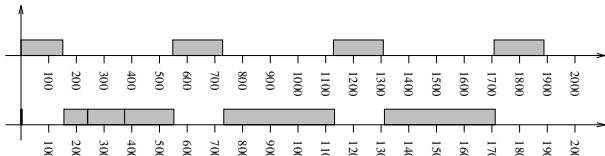**FIGURE 5:** *CPU reclaiming performed by hard CBS on the "Greedy Task" problem.*

FIGURE 6: *The "Short Period" problem with CBS.*

The simple usage of a reclaiming mechanism, however, does not solve the "Short Period" problem, as shown by running two batch tasks τ_1 and τ_2 served by two servers with parameters $(30ms, 150ms)$ and $(400ms, 900ms)$. The schedule produced by the CBS is shown in Figure 6, whereas Figure 7 shows the schedule produced by GRUB. Note that using both algorithms the distances between continuous "blocks of execution" for task τ_1 (about $400ms$ for the CBS and $600ms$ for GRUB) is much bigger than the period of server S_1 (i.e., $150ms$).
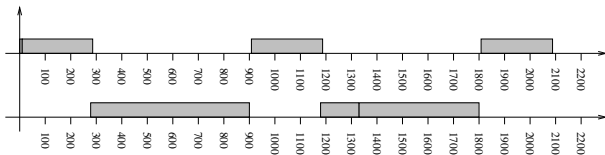

FIGURE 7: *The "Short Period" problem with GRUB.*

A great improvement can be achieved by using a hard reservation behaviour, as shown in Figure 8. Again, the hard algorithm is less efficient, and the tasks execute for less time than in the previous cases.
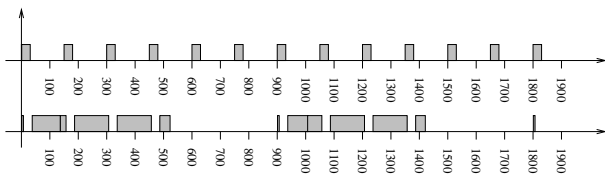

FIGURE 8: *The "Short Period" problem with hard CBS.*

The best solution, thus, is to combine the GRUB reclaiming with the hard reservation behaviour, as done by HGRUB. The schedule produced by HGRUB is shown in Figure 9.
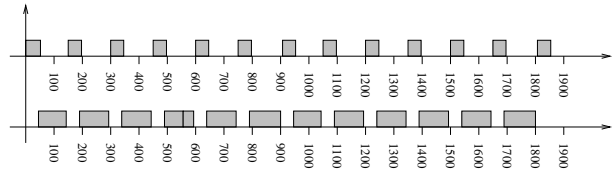

FIGURE 9: *The "Short Period" problem with HGRUB.*

5.2 Real-Time Performance

In a second set of experiments, the various scheduling algorithms have been evaluated in terms of soft and hard real-time performance. This goal has been achieved by running a task set composed by some soft and hard real-time tasks, and by measuring the Cumulative Distribution Function (CDF) $C(x) = P\{f_i < x\}$ of the finishing times f_i, representing the probability to finish a job within a specified time. All the tasks were characterised by randomly varying execution times and periods, and the reservation parameters were assigned to guarantee the respect of hard deadlines (so, for hard tasks $Q_i > C_i$ and $T_i < \min\{r_{i,j+1} - r_{i,j}\}$). As a first check, it has been verified that all the hard tasks respected all their deadlines: Figure 10 shows the CDF of the finishing times for an hard task with relative deadline $D_i = 300ms$, and it shows that for every considered scheduling algorithm $P\{f_i < D_i\} = 1$ (as expected).
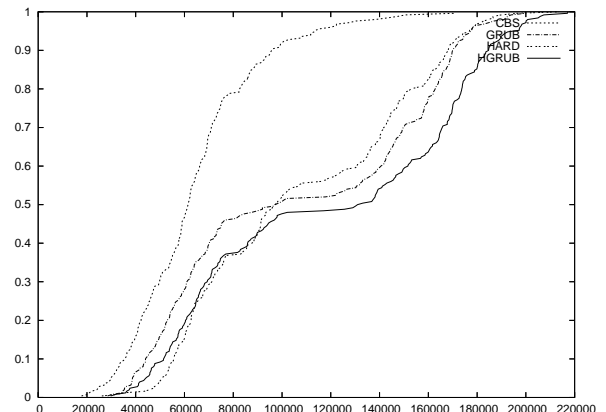

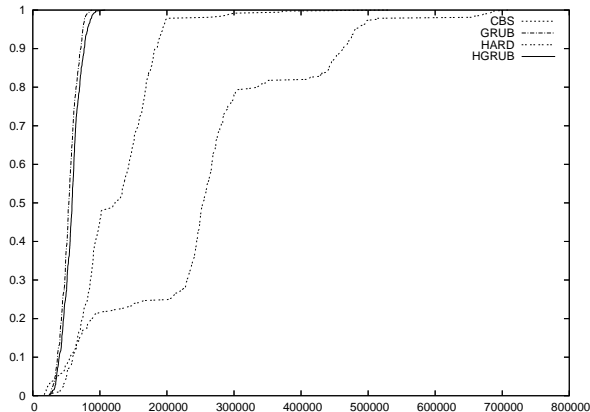**FIGURE 10:** *CDF of the finishing times for a hard real-time task.*

FIGURE 11: *CDF of the finishing times for a soft real-time task.*

Then, the finishing times CDF for a soft task has been measured, to show the effectiveness of the reclaiming mechanism. As can be noticed from Figure 11, the probability of missing the soft deadline $D_i = 200$ is 0 when GRUB or HGRUB is used, is about 0.025 when the CBS is used, and grows to more than 0.7 for the hard CBS. This result clearly shows that hard CBS is not a viable solution from the efficiency point of view.

5.3 Fairness and Throughput

The previous experiments show that the hard and soft real-time performance of GRUB and HGRUB are comparable; in fact the advantages of HGRUB are only visible when serving non-real-time tasks. Such advantages are highlighted in a third set of experiments, in which a non-real-time task has been added to the previous task set (which was composed by real-time tasks only).
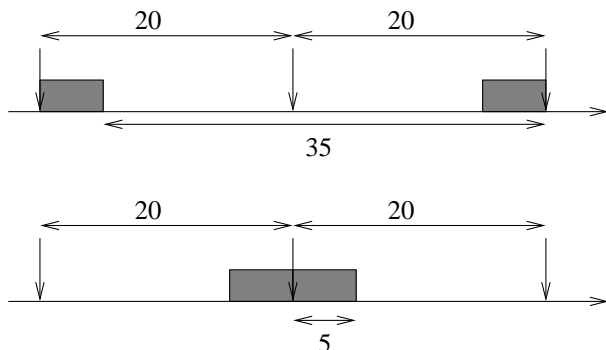

FIGURE 12: *Maximum and minimum theoretical time needed do serve a 5ms with a hard $(5, 20)$ reservation.*

In particular, the task is cyclically measuring the time needed for executing for $5ms$, and is attached to a reservation $(Q = 5ms, T = 20ms)$. When using a CBS (even with GRUB reclaiming) to serve the non real-time task, execution time can be consumed in advance, so the time needed to execute $5ms$ ranges from $5ms$ to a very large time (due to deadline aging, or to the "Short Period" problem). On the other hand, when using a hard reservation behaviour the time needed to execute $5ms$ can range from $5ms$ to $35ms$ (see Figure 12).
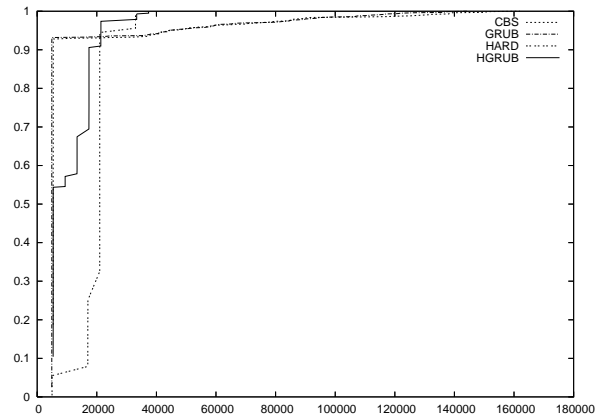

FIGURE 13: *CDF of the time needed to serve 5ms in a non real-time task served by a $(5, 20)$ reservation.*

The CDF of the time needed for executing $5ms$ in the non real-time task has been measured, and is shown in Figure 13. Notice that (as expected) the CDF for CBS and GRUB arrives to 1 after a much longer time.

6 Conclusions and future work

The paper described some problems encountered when using CPU reservations in the Linux kernel, that have been solved by adding a reclaiming mechanism and hard reservation behaviour to the scheduling algorithm. The new algorithm (called HGRUB) proved to be very effective in a wide range of situations, like serving both real-time and non-real-time tasks and hierarchical applications.

The HGRUB algorithm has been implemented and tested in the Linux kernel, and a prototype is freely available[3]. As future work, we plan to extend the current implementation to support resource sharing through the BWI protocol, to better integrate with the preemtp-RT patches [1] (the scheduler has been adapted to 2.6.21-rc5-rt4, and this configuration is currently under test), and to compare the

---

[3]distributed under the GPL; contact Luca Abeni for obtaining a copy of the scheduler.

reservation approach with the new CFS scheduler provided by Linux.

References

[1] Ingo Molnar's RT Tree. `http://people.redhat.com/mingo/realtime-preempt`.

[2] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[3] L. Abeni and G. Buttazzo. QoS Guarantee Using Probabilistic Deadlines. In *Proceedings of the IEEE Euromicro Conference on Real-Time Systems*, York, England, June 1999.

[4] L. Abeni and G. Lipari. Implementing resource reservations in linux. In *Real-Time Linux Workshop*, Boston (MA), Dec. 2002.

[5] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole. Analysis of a Reservation Feedback Scheduler. In *Proc. 23rd IEEE Real Time Systems Symposium*, 2002.

[6] J. Bennett and H. Zhang. WF2Q: Worst-case fair weighted fair queueing. In *Proc. of INFOCOM'96*, March 1996.

[7] J. Bennett and H. Zhang. Why WFQ is not good enough for integrated services networks. In *Proceedings of NOSSDAV'96*, April 1996.

[8] X. Feng and A. K. Mok. A model of hierarchical real-time virtual resources. In *Proceedings of the 23$^{\mathrm{rd}}$ IEEE Real-Time Systems Symposium*, pages 26–35, Austin, TX USA, Dec. 2002.

[9] G.Lipari and S. Baruah. Greedy reclamation of unused bandwidth in constant bandwidth servers. In *IEEE Proceedings of the 12th Euromicro Conference on Real-Time Systems*, Stokholm, Sweden, June 2000.

[10] Timesys linux. http://www.timesys.com.

[11] C. Kolivas. Isochronous class for unprivileged soft RT scheduling. `http://ck.kolivas.org/patches/SCHED_ISO`.

[12] D. Libenzi. SCHED_SOFTRR linux scheduler policy. `http://xmailserver.org/linux-patches/softrr.html`.

[13] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo. Iris: A new reclaiming algorithm for server-based real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 211–218, 2004.

[14] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems. Technical Report CMU-CS-93-157, Carnegie Mellon University, Pittsburgh, May 1993.

[15] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proc. of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

[16] J. Regehr. Inferring scheduling behavior with hourglass. In *Proc. of the USENIX Annual Technical Conf. FREENIX Track*, Monterey, CA, June 2002.

[17] C. Scordino and G. Lipari. Energy saving scheduling for embedded real-time linux applications. In 5$^{\mathrm{th}}$ *Real-Time Linux Workshop*, Valencia, Spain, Nov. 2003.

[18] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard-real-time environments. *IEEE Transactions on Computers*, 4(1), January 1995.

A The Basic CBS

As already explained in Section 2.3, every CBS-based algorithm works by maintaining two variables for every server S_i: the server remaining budget q_i (used for accounting) and the current scheduling deadline d_i^s (used for assigning a priority to the scheduled task and for enforcement). Such variables are updated as follows:

- When a server is created, q_i and d_i^s are initialised to 0;

- When τ_i activates (job arrival) at time t, the scheduler checks if the current scheduling deadline can be used (if $q_i < (d_i^s - t)U_i$), otherwise a new scheduling deadline $d_i^s = t + T_i$ is generated and q_i is recharged to Q_i;

- While task τ_i executes, the server budget is decreased as $dq_i = -dt$ (**accounting rule**);

- When the budget is exhausted ($q_i = 0$), it is recharged to Q_i and the scheduling deadline is postponed ($d_i^s = d_i^s + T_i$) (**enforcement rule**)

The algorithm can handle an arbitrary number of servers, given that $\sum_i U_i \leq 1$.