Resource Reservations for General Purpose Applications

Luca Abeni, Luigi Palopoli, Claudio Scordino, and Giuseppe Lipari

Abstract—Resource reservations are an effective technique to support hard and soft real-time applications in open systems. However, they generally focus on providing guarantees to real-time applications, without paying too much attention to the performance of non-real-time activities. In this paper, the main limitations encountered when using a conventional reservation-based scheduler for serving non-real-time tasks are described and formally analyzed. Then, a novel algorithm that overcomes these problems (called HGRUB) is proposed, and both theoretical and experimental evidence of its effectiveness is provided.

Index Terms—Integration of real-time and non-real-time applications, quality of service, real-time scheduling.

I. INTRODUCTION

A REAL-TIME computing system is required to produce the results of its computations within predefined timing constraints. This simple requirement raises issues of challenging complexity when the system permits the execution of concurrent tasks. In this case, a critical role is played the operating system, which has to guarantee a predictable execution to the different tasks. The traditional view is that such design goals can only be fulfilled using special purpose operating systems (RTOSs), which are built from the ground up placing a major emphasis on temporal predictability.

Recently, a new trend is making inroad: modifying General Purpose Operating Systems (GPOSs) to improve the real-time performance. Obvious advantages are the availability of a large set of legacy applications and development tools, the possibility of executing both real-time and non-real-time applications at once and the support for a wide range of hardware platforms. However, the task can be far from trivial and require a huge development effort tapping different aspects of the system design. As an example, a very tough problem is represented by the maximum latency introduced by the kernel, which can be unacceptably large in a GPOS. Substantial reductions in the latencies can only be obtained by modifying the internal structure of the kernel. A notable example is a recent patch for the Linux Kernel called preempt-rt [1], which reduces the maximum latency down to a few dozens of microseconds.

Another crucial aspect is the scheduling algorithm. To this regard, the scheduling heuristics typically used in GPOSs for non-real-time task introduce unpredictable delays. On the other
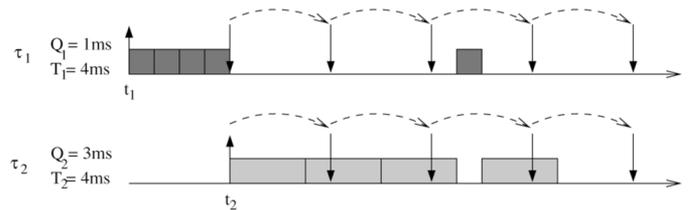
Fig. 1. "Greedy Task" problem.

hand, real-time scheduling solutions based on fixed or on dynamic priorities [20] (traditionally developed for hard real-time control applications) have evident shortcomings when applied to dynamic soft real-time task sets, whose computation times are hardly known and/or experience large variations. In recent years, such scheduling algorithms as the Resource Reservations [22] have emerged as an effective technique to support a wide range of real-time applications (both soft and hard) on GPOSs. By using this technique, each real-time task is reserved a fixed amount of time (maximum budget) in every reservation period, regardless of the behavior of the other tasks. The scheduling guarantees offered by this solution can be combined with effective design procedures [4], [16] for the choice of the scheduling parameters in order to offer predictable levels of quality-of-service (QoS) to real-time applications. Among the many implementations of this paradigm available in the literature, the Constant Bandwidth Server (CBS) [2] is particularly effective for managing general activation patterns.

Unfortunately, the attractive features of the CBS in scheduling real-time applications are not always matched by a good performance for non-real-time tasks, thus reducing its applicability for heterogeneous task sets. Indeed, an application that is activated frequently and for a short amount of time may be not scheduled as frequently as expected. This is the "Short Period" problem described in detail in Section III-A and illustrated in Fig. 2. It can also happen that an application executes for a long time when the CPU is idle and later is not scheduled for a long time. This is the "Greedy Task" problem, which again is described in Section III-A and illustrated in Fig. 1. The consequences of these anomalies can be severe in several respects. For instance, interactive applications may not produce a "fluid" response to the user inputs. The purpose of this paper is to show an alternative scheduling technique allowing one to overcome these limitations retaining the good real-time features of the CBS.

The problems described above have a common root: to reduce the time needed to react to external events, the CBS allows a task to consume its future allocated time when the CPU is
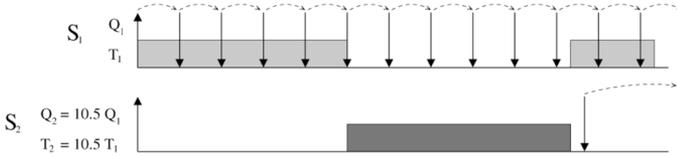
Fig. 2. "Short Period" problem.

idle. A possible solution strategy can be to adopt *hard reservations* [23], which prevent a task from receiving more than its allocated budget. The price to pay is that the algorithm is not work-conserving (i.e., it allows the CPU to remain idle in presence of computation requests) and its efficiency in terms of throughput can be drastically reduced. In spite of this limitation, hard reservations represented a good source of inspiration for the scheduling solution called hard GRUB (HGRUB) proposed in this paper. The idea behind HGRUB is to combine hard reservations with a reclaiming mechanism. By *reclaiming*, we mean the ability of the system to redistribute the CPU time unused by a task to the other ones [14].

As shown in this paper, the coexistence of the two techniques bears considerable advantages. First, as it happens for the standard CBS, it is possible to provide QoS guarantees to both periodic and aperiodic real-time applications. Second, the application of the hard reservations paradigm enables HGRUB to offer service guarantees in terms of the BLAH property (formalized in Section III) to non-real-time tasks. Third, the application of the reclaiming technique improves the efficiency of HGRUB, which is work-conserving and introduces a moderate number of preemptions. Finally, HGRUB can be used as a basic mechanism for hierarchical scheduling solutions that respect the timing properties of the different applications.

The first contribution of this paper is to formalize the requirements of non-real-time tasks and hierarchical schedulers by introducing the notion of BLAH, and identify the main problems of the CBS that make it unfit to achieve this property. The second contribution of the paper is the study of the integration of hard reservations with a reclaiming mechanism, leading to the definition of the HGRUB scheduling algorithm. The third contribution is the theoretical analysis of the HGRUB algorithm, showing that it overcomes the limitations of the CBS scheduler. We offer an experimental validation of these results using a Linux based implementation of HGRUB.

The rest of the paper is organized as follows. Section II introduces the scheduling model that will be used throughout the paper, and briefly recalls the basic Resource Reservation concepts. Section III presents the problems addressed in this paper, and introduces the HGRUB algorithm as an effective solution. A formal description of such algorithm is contained in Section IV, whereas Section V presents some experimental results. Section VI compares our algorithm to other algorithms presented in the literature. Finally, Section VII presents the conclusions, open issues and future work.

II. BACKGROUND

In this section, we will formally introduce the task model and the notation used throughout the paper and offer a quick summary of the CBS algorithm, which will be used as a basis for this work.

A. Definitions

A *task* is a schedulable entity that can either be a thread (sharing its address space with other threads) or a process (owning a private address space). In the considered scenario, tasks compete for a shared CPU. In this paper, we deal with both real-time and non-real-time tasks.

A *real-time task* τ_i is a stream of jobs (or instances) $J_{i,1}, J_{i,2}, J_{i,3}, \ldots$, where $J_{i,j}$ becomes ready for execution (arrives) at time $r_{i,j}$, requires a computation time $c_{i,j}$, and finishes at time $f_{i,j}$. Each job $J_{i,j}$ is also characterized 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 $r_{i,j+1}$.

Non-real-time tasks can be batch or interactive. A task is said to be *batch* if it is (almost) continuously active, and it is never (or very rarely) blocked. Scientific computation programs are the typical example of such tasks. A task is *interactive* if it is often blocked waiting for some external event. When the event occurs, the task is activated and it executes for some (usually short) time producing visible results before being blocked again.

B. Resource Reservations

The basic idea behind Resource Reservations is to *reserve* a fraction of the time to time-sensitive applications through real-time scheduling. 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*.

Although there are several scheduling algorithms that claim adherence to this paradigm, we will particularly focus on the Constant Bandwidth Server (CBS) [2]. In the CBS algorithm 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. The algorithm then orders all active servers in an Earliest Deadline First (EDF) queue by their scheduling deadlines, hence the server with the earliest scheduling deadline is selected for execution. The CBS is based on three different mechanisms: *accounting*, *enforcement*, and *replenishment*. The **CBS accounting rule** is used to keep track of the time used by a task in each server period. This mechanism is implemented using a variable called server *budget* q_i, which measures the execution time of the task served by S_i: when the task executes for a time $\delta t, q_i$ is decreased by δt.

Enforcement and replenishment are performed as follows: when the budget q_i becomes 0 (the server is said to be *depleted*) and the task has not yet completed its execution, the budget is *immediately* recharged to the value Q_i (**replenishment rule**), the server scheduling deadline is postponed to $d_i^s = d_i^s + T_i$, and the EDF queue is reordered (this is the **CBS enforcement rule**).

If, after postponing the server deadline, the server still holds the earliest scheduling deadline, the task continues to execute. Otherwise, a preemption occurs. The scheduling deadline d_i^s is only an artifact used by the algorithm and it is not necessarily coincident with the possible deadlines of the task (which are a design decision).

Depending on the replenishment rule, it is possible to distinguish between hard and soft reservations. A scheduling algorithm is said to implement *hard reservations* if, when the server is depleted (i.e., when the budget becomes 0), the served task is suspended until the next *replenishment time*. This is the opposite of the *soft reservation* behavior used by the CBS (the budget is replenished to Q_i as soon as it arrives to 0).

A more formal description of the CBS algorithm is provided in the original paper [2].

III. SCHEDULING NON-REAL-TIME TASKS

When the CBS is used as a base scheduling mechanism in a GPOS, it exhibits important limitations. In the first place, the choice of the scheduling parameters for tasks characterized by a highly dynamic behavior is far from obvious. A second limitation applies to tasks requiring a very large computation time (much larger than the budget) in a single shot of execution and to tasks associated with short reservation periods. In this case, the *soft reservation* behavior descending from the enforcement/replenishment rule can generate scheduling anomalies, which we will refer to as the *"Greedy Task"* problem and the *"Short Period"* problem. In the rest of the section, we will first expose these anomalies and then formalize a property that constitutes a basic design goal that a scheduler needs to comply with to solve these problems.

A. Issues With the Original CBS

Both the "Greedy Task" problem and the "Short Period" problem can be seen as consequences of the *"deadline aging"* effect, which happens when the scheduling deadline d^s of a server is postponed many times and soon takes a large value compared to the scheduling deadlines of the other servers (hence the name "deadline aging").

To illustrate the "Greedy Task" problem, consider two tasks τ_1 and τ_2 scheduled by two servers S_1 and S_2 with parameters $Q_1 = 1$ m, $Q_2 = 3$ ms, and $T_1 = T_2 = 4$ ms, as depicted in Fig. 1. Moreover, τ_1 is activated at time t_1, whereas τ_2 is activated at time $t_2 = t_1 + 4$ ms, hence at the beginning τ_1 is the only active task and it is scheduled to execute.

According to the CBS replenishment and enforcement rules, every time the budget q_1 becomes 0, the scheduling deadline is postponed to $d_1^s = d_1^s + T_1$, and the budget is immediately recharged to Q_1 (at time $t_1 + 1$ ms, $t_1 + 2$ ms, $t_1 + 3$ ms, and $t_1 + 4$ ms). Thus, the deadline d_1^s is moved forward taking a large value. In presence of other tasks, the EDF mechanism would determine a preemption of the task. However, since the task is the only one active between t_1 and t_2, it continues to execute pushing the deadline even further.

When τ_2 is activated at time $t_2 = t_1 + 4$ ms its scheduling deadline $d_2^s = t_1 + 8$ ms is much smaller than $d_1^s = t_1 + 16$ ms. Thus, τ_2 takes hold of the CPU until time $t_1 + 13$ ms, 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. The user, therefore, perceives an annoying "stop-and-go" effect. Intuitively, we can explain this problem by saying that between t_1 and t_2 task τ_1 is consuming its future budget and is then forced to stop until τ_2 catches up. When the anomaly terminates, the two tasks start to progress at the same speed.

The "Short Period" problem, instead, happens when a task served by a server with a short period ends up being executed as if it was served by a server with a much larger period. The server period can be thought as a measure of the *granularity* of the reservation—i.e., how often a task is allocated the reserved budget. From this, a user may incorrectly assume that a task served by a reservation (Q_i, T_i) will be scheduled for Q_i time units every T_i. As a result, she/he may be tempted to use a very small reservation period for task τ_i to ensure a very fine grained allocation of its CPU time.

In Fig. 2, for instance, two tasks with large computation times τ_1 and τ_2 are served by two servers S_1 and S_2 with parameters $Q_2 = 10.5Q_1$ and $T_2 = 10.5T_1$. Unfortunately, the *reservation* of Q_i units every T_i does not necessarily imply that for every scheduling period the task will actually *execute* for Q_i. In our example, due to the difference between T_1 and T_2, task τ_1 executes without interruption for the first five reservation periods consuming its future allocated budget, while it will not receive any fraction of the CPU time for the five subsequent periods. Hence, contrary to the expectations of the user, in this example τ_1 does *not* execute every T_1 time units. Indeed, it executes as it were in a server with different parameters.

B. Consequences of the Scheduling Anomalies

The original CBS algorithm has been designed to serve real-time tasks, whose model is defined in Section II-A. 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 [4], and the "Greedy Task" and the "Short Period" problems are not a concern. Only when the CBS is implemented in a GPOS [6], [25] and is used to schedule generic non-real-time (interactive and batch) tasks, which can easily act as a greedy task, do the anomalies become apparent. Their consequences are particularly severe in two cases: interactive applications and hierarchical scheduling.

For interactive applications, the user perceives annoying "stop-and-go" effect: while in some intervals he/she does not experience any reaction to his/her inputs, in others the system response accelerates dramatically.

Another problem occurs when a CBS is used as *global scheduler* in a hierarchical scheduling scheme [13], [19], [27]. In a hierarchical scheduling system, the so-called *global scheduler* assigns chunks of the CPU time to "local schedulers," which in turn distribute their time to different tasks. This technique is useful in many situations, e.g., to schedule multi-threaded applications, to do OS level virtualization or simply to isolate the behavior of tasks belonging to different components. Clearly, when this scheme is used for real-time applications, it is very important to produce composite guarantees composing the local guarantees provided by the local schedulers with the guarantees provided by the global one. In particular, the global scheduler must guarantee a minimum amount of *service* (i.e., allocation of CPU time) to each local scheduler in every interval of time, so
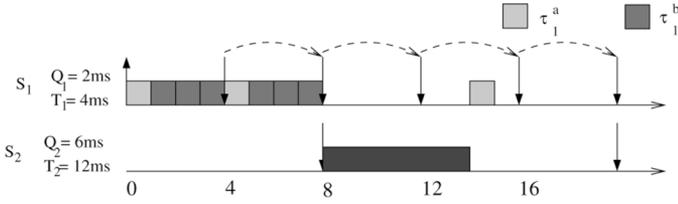
Fig. 3.   Problems using the CBS in a hierarchical scheduling scheme.

that the local scheduler can provide guarantees that its real-time tasks meet their deadlines. While such guarantees can be provided by a hard reservation mechanism, this is not possible using the CBS server due to the "Short Period problem."

For example, consider a server S_1 (with parameters $Q_1 = 2$ ms, $T_1 = 4$ ms) used to serve a group of two periodic real-time tasks τ_1^a and τ_1^b, activated simultaneously. Let the periods of τ_1^a and τ_1^b be, respectively, equal to $P_1^a = 4$ ms and $P_1^b = 24$ ms, and the computation times be $C_1^a = 1$ ms, $C_1^b = 6$ ms. In this example, the relative deadlines are equal to the periods and that rate monotonic (RM) is used as the local scheduler. Now, suppose that at time 8 a new server S_2 (serving a third task τ_2^a) is activated with $Q_2 = 6$ ms and $T_2 = 12$ ms. The resulting schedule is shown in Fig. 3. In the first 8 ms, server S_1 schedules two jobs of τ_1^a and one of τ_1^b. By doing so, it consumes part of its future reserved budget (due to the "short period problem") and when S_2 is activated its scheduling deadline is the earliest. Therefore S_1 fails to schedule correctly the third job of τ_1^a that misses its deadline. However, a simulation of the schedule on the hyperperiod (24 ms) shows that, using hard reservations as a global scheduler, it is possible to guarantee that the two tasks can be scheduled with a (2 ms, 4 ms) reservation.[1]

C. Fairness Property

All these examples show that even if the real time guarantees are correctly provided by the CBS, the schedule may be not "fair enough." Indeed, if τ is activated at time t_0 and is continuously active in $[t_0, t_0 + kT_i]$, it is not guaranteed that in $[t_0 + (k - 1)T_i, t_0 + kT_i]$ the task executes for a time Q_i. This problem is shown in Fig. 2 (task τ_1 is not scheduled between the sixth and the tenth server period) and Fig. 1 (task τ_1 is not scheduled in the second, in the third, and in the fifth server periods).

This issue is similar to the problem of the fairness of the Weighted Fair Queueing algorithm (WFQ) [9], [8]. Informally speaking, the WFQ algorithm computes for each task a virtual finishing time, which is the time the task would end if it executed on a dedicated slower CPU. However, the algorithm only guarantees that the task finishes *before* its virtual finishing time. Therefore, under certain conditions, the task is allowed to execute "too early" compromising the system fairness. Likewise, the CBS algorithm only guarantees that the task finishes *before* the deadline allowing it to consume its future budget with similar consequences.

Such an issue can be addresses by requiring the worst $-$ casefairness property for reservations (inspired by Bennett and Zhang's *worst-case fairness* [9], [8]).

[1]This is due to the fact that τ_1^a and τ_1^b have harmonic periods and start simultaneously.

Definition 1: A reservation-based scheduling algorithm is said to be worst $-$ casefair if a server S_i with budget q_i at time t can 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 $t + T_i$).

Note that 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 $-$ casefair because, for example, server S_1 in Fig. 1 has $q_1 > 0$ at time $t_2 = t_1 + 4$ ms, but can consume it only at time $t_1 + 13$ ms (similarly, in Fig. 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, we need a modified version of the scheduler combining the worst $-$ casefairness property for non-real-time tasks with the same level of real-time guarantees offered by the original CBS.

IV. HGRUB ALGORITHM

In this section, we will shortly recall the basics of the GRUB algorithm [14] and then show how it can be combined with a hard reservation behavior. Then we will analyze the theoretical properties of the resulting HGRUB algorithm [6], [25].

A. GRUB Reclaiming

GRUB reclaiming is implemented in the CBS algorithm by modifying the accounting rule. This is done by introducing a global variable U^{act}, which keeps track of the fraction of CPU time currently used by the tasks present in the system. The evolution of U^{act} is as follows.

- At system startup, U^{act} is initialized to 0, since no task is currently using the CPU.
- When a task τ_i is activated at time t, the server checks if the previously used scheduling deadline d_s^i and budget q_i can still be used or not (this step is identical to the one performed in the CBS algorithm). In the first case, U^{act} is left unchanged because it already accounts for the fraction of CPU currently used by the task. In the second case, a new deadline has to be generated using the rule $d_s^i = t + T_i$, q_i is recharged to Q_i and U^{act} has to be updated by the rule $U^{\text{act}} = U^{\text{act}} + U_i$ (with $U_i = Q_i/T_i$) to take into account the additional fraction of CPU time used by the task. The condition to check if a new deadline has to be generated is: $q_i > (d_i^s - t)U_i$.
- When τ_i is blocked (e.g., at the termination of a real-time job) at time t, if $q_i < (d_i^s - t)U_i$ then d_i^s and q_i can be used by future jobs of τ_i (as explained by the previous rule). Therefore, U^{act} is left unchanged, and in this case it will not be changed by the previous rule upon the activation of the new job. If, on the contrary, $q_i \geq (d_i^s - t)U_i$, then $U^{\text{act}} = U^{\text{act}} - U_i$.
- If U^{act} is not updated by the previous rule upon a job termination, then it will be decreased by U_i at time $t = d_i^s - q_i/U_i$. Indeed, beyond this instant, the server will have to generate a new scheduling deadline on the arrival of a new job.

Then, the CBS accounting rule is modified in this way: while task τ_i executes, the server budget is decreased as

$dq_i = -U^{\mathrm{act}}dt$ (**GRUB accounting rule**). In this way, in presence of a low CPU workload, the time accounted to the task will be lower than the one it actually consumed and the fraction of CPU time given by $U^{\mathrm{lub}} - U^{\mathrm{act}}$ is redistributed among the executing tasks (this is the reclaiming mechanism). A more formal description of the GRUB algorithm is provided in the original paper [14].

B. Hard Reservations and GRUB

The CBS algorithm can be transformed into a hard reservation algorithm by modifying the enforcement rule: when the budget is exhausted $(q_i = 0)$, the server is said to be *depleted*, and the served task is not scheduled until time $t = d_i^s$. When this time instant arrives, 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 − casefairness property. Since this combination could make the algorithm non-work-conserving, the following additional **fairness preserving** rule is needed: if task τ_i is blocked when all the servers associated to the other tasks are depleted, server S_i is not immediately deactivated, but it can be used to schedule the task having the earliest scheduling deadline. The server will become inactive at time $t = d_i^s - (q_i)/(Q_i)T_i$.

C. Algorithm Analysis

We start the theoretical analysis of HGRUB by proving the following *schedulability* property.

Lemma 1: Using HGRUB, if $\sum_i (Q_i/T_i) \le 1$, then every server S_i never misses its scheduling deadlines. In other words, at any instant t, the scheduling deadline d_i^s is always greater than t.

  Proof: See Appendix A.   ■

A first important consequence of this lemma is that the *hard schedulability property* of the original CBS is also valid for HGRUB.

Theorem 2: A hard real-time task τ_i, with worst-case execution time C_i and minimum interarrival time P_i, served by a reservation (Q_i, T_i) with $T_i < P_i$ and $Q_i > C_i$ is guaranteed to meet all its deadlines.

  Proof: See [2].   ■

Note that thanks to Lemma 1, also probabilistic guarantees [4] can be provided by HGRUB. Indeed, Lemma 1 states that each task always terminates before its scheduling deadlines. Since in the cited paper a stochastic model is constructed for the evolution of the scheduling deadlines, this model provides also information on the stochastic evolution of the finishing times.

Another important point is the following relation between d_i^s and t.

Lemma 3: For any algorithm implementing hard reservations, if $\sum_i (Q_i/T_i) \le 1$, then at any instant $t, d_i^s - T_i \le t$.

  Proof: Let $t_{i,h}$ represent the time in which the scheduling deadline d_i^s is increased for the hth time. This can be a consequence of a task arrival or of a replenishment. In either case we have: $d_i^s = t_{i,h} + T_i$, which proves the claim for $t = t_{i,h}$. For all $t \in [t_{i,h}, t_{i,h+1}]$, we have: $t \ge t_{i,h} \ge d_i^s - T_i$.   ■

As a consequence of Lemma 3, HGRUB is endowed with worst − casefairness property.

Corollary 4: Any algorithm implementing hard reservations (and in particular HGRUB) is worst − casefair.

  Proof: The inequality $t \le d_i^s$ (Lemma 1) implies that a server S_i is always able to consume its current budget before the current scheduling deadline, and inequality $t \ge d_i^s - T_i$ (Lemma 3) implies $t + T_i \ge d_i^s$. Hence, the server is guaranteed to consume its current budget before $t + T_i$, which leads to the claim.   ■

The property shown above is of the greatest relevance: it proves that HGRUB can be used as global scheduler in a hierarchical scheduling system and for scheduling both interactive and real-time applications.

Another important point to clarify is the efficiency of the algorithm. To this regard, the algorithm can be proven to be work-conserving. Instrumental to this goal is the following.

Lemma 5: If *Active* is the set of tasks whose servers contribute to U^{act} (active tasks, or inactive tasks served by CBSs with $q_i \le ((d_i^s - t)/T_i)Q_i)$, then for algorithm HGRUB the following invariant holds:

$$\forall t \quad \sum_{\tau_i \in \mathrm{Active}} (d_i^s - t)U_i - q_i = 0. \tag{1}$$

  Proof: See Appendix A.

Based on Lemma 5, it is possible to prove that HGRUB is a work-conserving algorithm.

Theorem 6: Algorithm HGRUB is work-conserving.

  By Contradiction: If the algorithm is not work-conserving, then there must be an instant of time in which all the servers serving active tasks are depleted (i.e., no active task can be scheduled). Based on Theorem 1, $\forall S_j, t < d_j^s$. When a server is depleted, based on the rules of the algorithm, we also have $q_j = 0$. Therefore, if all servers are depleted we get: $\sum_{\tau_i \in \mathrm{Active}} (d_i^s - t)U_i - q_i = \sum_{\tau_i \in \mathrm{Active}} (d_i^s - t)U_i > 0$. This is in contradiction with the invariant condition 1 of Lemma 5, proving our claim.■

The intuition behind the proof above is that requiring the algorithm to be work-conserving is tantamount to requiring that it is impossible to reach a situation in which all servers associated to tasks in the Active set are depleted. In the proof of Lemma 5 it is shown that this situation cannot occur.

V. EXPERIMENTAL RESULTS

The effectiveness of the HGRUB algorithm has been tested by modifying an implementation of the CBS scheduler in the Linux kernel [5], [6], [25], [26]. Since some experiments showing the overhead introduced by the scheduler and the efficiency of its implementation are already presented in [5], this section focuses more on algorithmic aspects exposing the differences between different scheduling algorithms [6].

A. Comparison With Other Scheduling Algorithms

As explained in the previous sections, the HGRUB algorithm has been designed to correctly address the "Greedy Task" and the "Short Period" problems, and this result is achieved by providing worst − casefairness. Although there are many proportional share and pfair algorithms [8], [15], [28], that provide such
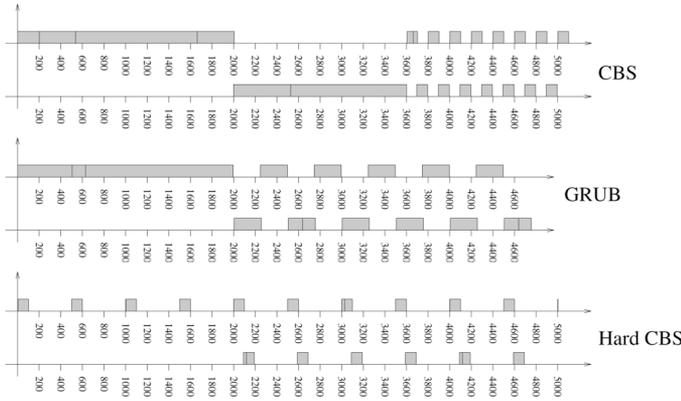
Fig. 4. "Greedy Task" problem with CBS, GRUB, and Hard CBS (times are in ms).

Fig. 5. "Short Period" problem (times are in ms).

a property, this paper focuses on reservation-based schedulers, because (as discussed in Section VI) they offer a better support for real-time scheduling [3].

The $worst-case fairness$ property is certainly provided by hard resource reservations. For this reason we will consider them in this comparison (in particular we will consider a CBS modified to exhibit a hard reservation behavior). As shown in the experiments, this choice corresponds to a remarkable performance loss for non-real-time tasks.

To the authors' best knowledge, the only previous reservation-based algorithm that tries to address the "Greedy Task" and the "Short Period" problems without penalising the performance is IRIS [21]. However, in [7] it has been shown that IRIS has a poor performance (in terms of reclaiming) than BEBS [7]. In its turn, BEBS has reportedly a performance in line with the one of CASH [17] (a more detailed discussion of the algorithms mentioned above is presented in Section VI). Moreover, an extensive set of experiments [12] has shown that the performance of CASH in reclaiming CPU is lower than that of GRUB. Hence, in the following experiments GRUB will be used as a reference. This is also the main reason why GRUB has been selected as a starting point for developing HGRUB.

B. Schedule Correctness

As a first thing, the main property of HGRUB (the ability to cope with the "Greedy Task" and the "Short Period" problems) has been verified through a set of experiments performed by using a measurement technique similar to the one used by Hourglass [28].

The "Greedy Task" problem, shown in Fig. 1, can be reproduced by using two batch tasks τ_1 and τ_2 served by two identical CBSs with parameters (100 ms, 500 ms), and by activating τ_2 2 s after τ_1. The schedule produced by CBS, GRUB, and a CBS with hard enforcement (Hard CBS) is shown in Fig. 4. By looking at the figure it is possible to see that the GRUB reclaiming mechanism is able to avoid the problem by allocating to τ_1 the fraction of CPU time not used and not reserved during the first two seconds. Hard reservation techniques could be used to alleviate the "Greedy Task" problem, but in this case τ_1 would execute for a smaller time.
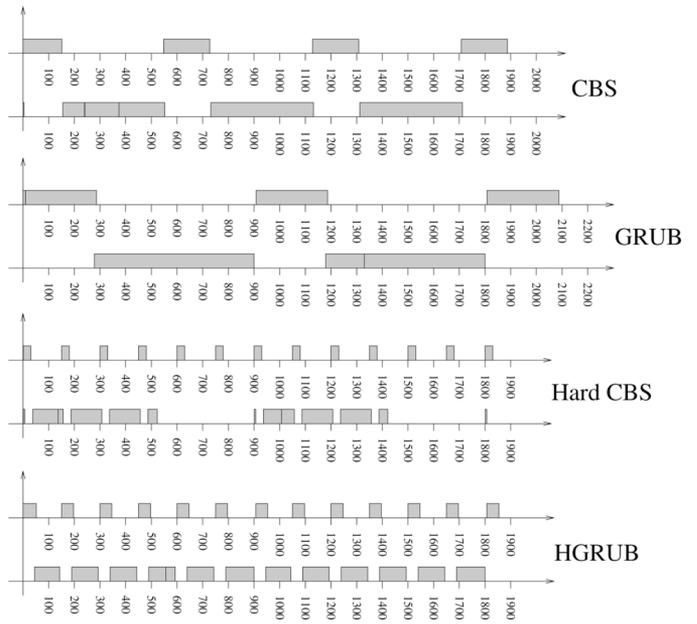
CPU reclaiming alone, however, does not solve the "Short Period" problem, reproduced by running two batch tasks τ_1 and τ_2 served by two servers with parameters (30 ms, 150 ms) and (400 ms, 900 ms), as shown in Fig. 5. When using CBS or GRUB, the distances between continuous "blocks of execution" for task τ_1 (about 400 ms for the CBS and 600 ms for GRUB) is much bigger than the period of server S_1 (i.e., 150 ms). The Hard CBS introduces a good improvement but is again less efficient, (tasks execute for a smaller time than in the CBS or GRUB cases). The best solution, thus, is to combine the GRUB reclaiming with the hard reservation behavior, as done by HGRUB.

C. Response Times

The soft and hard real-time performance of HGRUB have been evaluated, by running a task set composed by five hard real-time tasks and three soft real-time tasks, and by measuring the cumulative distribution function (CDF) $C(t) = P\{f_i < t\}$ of the finishing times f_i (representing the probability of finishing a job within a specified time). Task periods were randomly selected with a uniform distribution inside the interval [100 ms, 500 ms] and 500 ms, and the execution times were uniformly distributed in randomly generated intervals so that the average system load was about 0.7. The reservation parameters were assigned to guarantee the respect of hard deadlines (so, for hard tasks $Q_i > C_i$ and $T_i < \min\{r_{i,j+1} - r_{i,j}\}$). Confirming the theoretical expectations, the hard tasks respected all their deadlines: in fact, it resulted that $P\{f_i < D_i\} = 1$.

The effectiveness of the reclaiming mechanism is evident by looking at the CDF for one of the soft tasks, displayed in Fig. 6 (this particular task was characterized by a soft relative deadline $D_i = 200$ ms). The probability of missing the soft deadline 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
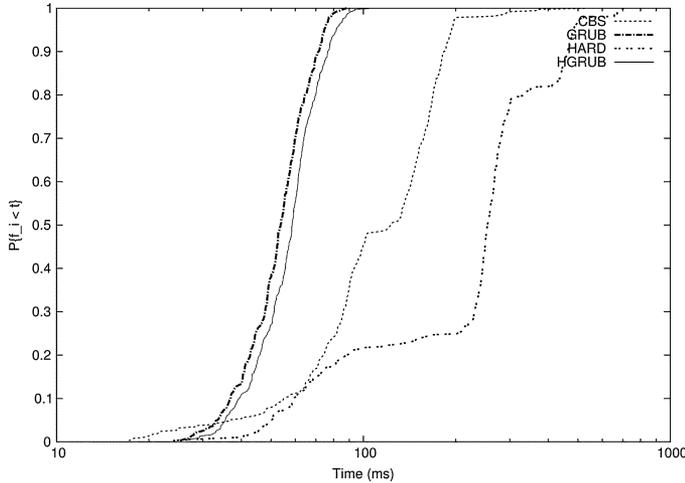
Fig. 6.   CDF of the finishing times for a soft real-time task.

Fig. 7. CDF of the time needed to serve 5 ms in a non-real-time task served by a (5, 20) reservation.

clearly shows that hard CBS is not a viable solution from the efficiency point of view.

As expected, the hard and soft real-time performance of GRUB and HGRUB looks comparable; in fact the advantages of HGRUB are only visible when serving non-real-time tasks. Hence, to cast some light on the differences between HGRUB and GRUB, a non-real-time task has been added to the task set. Such a task cyclically measures the time needed for completing a function f() (which we define *service time*), which has an execution time of 5 ms and has been attached to a reservation ($Q = 5\,\mathrm{ms}, T = 20\,\mathrm{ms}$).

The CDF of the service time of f() is shown in Fig. 7. Since CBS and GRUB allow the task to consume execution time in advance, f() can be served in a time ranging from 5 ms to a very large value (due to deadline aging, or to the "Short Period" problem). As a result, the CDF for CBS and GRUB reach the value 1 for a large values of the service time. When a hard reservation is used, the time needed for the server to execute a task for 5 ms can range from 5 to 35 ms, as explained in Fig. 8. As shown in the Figure, the CDF for HGRUB represents a good compromise between the CDF of the other algorithms. Indeed, the reclaiming mechanism allows it to have a higher probability of serving f() in a small time than hard CBS (although smaller than soft CBS or GRUB) but the maximum time (i.e., the time required for the CDF to reach 1) is 35 ms as for hard CBS. Therefore, the CDF does not exhibit the long "tails" (i.e., the occasionally high value of the service time) of CBS and GRUB.

VI. RELATED WORK

In the past decades, there has been an intense research activity in real-time scheduling policies for soft real-time applications. A first prong of research activities has been focused on a class of algorithms including *proportional share* algorithms (EEVDF [28], WF^{2}Q [8], etc.) or p-fair [15]. The idea is to mimic as closely as possible a fluid flow allocation of the CPU (generalized processor sharing—GPS). Each task is associated to a *weight* w_i and receives a percentage of service proportional to $(w_i/\sum_{j \in \mathrm{Active}} w_j)$. The maximum deviation (*lag*) from the progress that the task would mark on with a fluid partitioning of
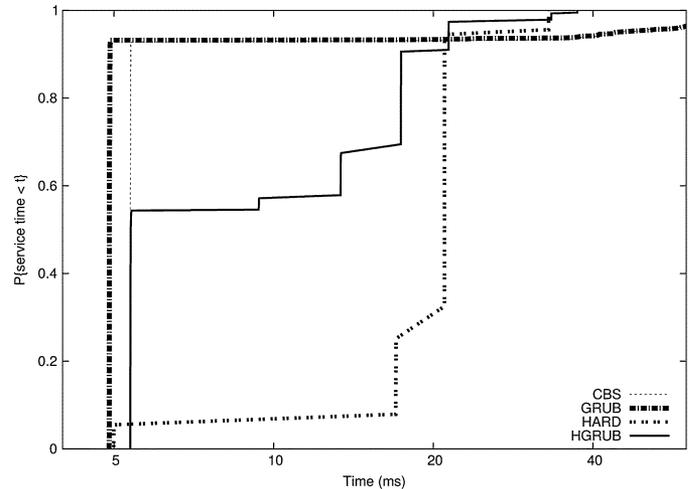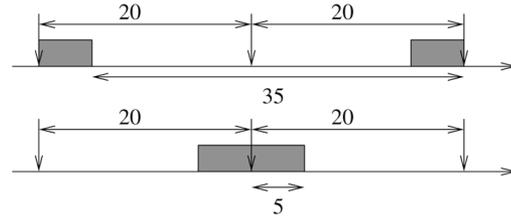

Fig. 8.   Maximum and minimum service time with a hard (5, 20) reservation (times are in ms).

the CPU is bound and is a system-wide parameter. The result is a fair distribution of the CPU time, in which each task receives a fraction proportional to its weight. Because the granularity of the allocation is a system wide parameter, it is not easy to accommodate deadline constraints specified with different timing scales for the different tasks [3].

The principal design goal of the resource reservations algorithms [22], [23] instead, was exactly the enforcement of both soft and hard real-time constraints. To this end, the designers associated to each task both a budget Q_i and a reservation period T_i. The ratio Q_i/T_i allows one to choose the fraction of the CPU time allocated to the task, while the period T_i allows one to decide the granularity of the allocation on a per-task basis. Since one of the objectives of this paper is to enforce real-time constraints, the resource reservations are a natural starting point.

Traditional implementations of the resource reservations [22], [23] for the CPU (CPU reservations) use a Deferrable Server for each task receiving a reservation. However, this strategy has evident shortcomings in presence of a-periodic arrivals of the tasks [29]. To solve this problem, the Constant Bandwidth Server (CBS) class of algorithms [2] uses dynamic priorities, whereby both periodic and aperiodic tasks are effectively managed. This generality was the main motivation leading to our choice of the CBS as a basic scheduling mechanism for this paper,

Several proposals have been done to enhance the CBS algorithm with a reclaiming mechanism. In this class, we can

cite the CApacity SHaring algorithm (CASH) [11], which uses a queue of budgets to store the unused computation capacity and to redistribute it to the tasks requiring it. Although an effective mechanism, CASH can only be applied to periodic tasks whereas the goal of this paper is to address generic task sets. A different proposal is the GRUB algorithm [14], [18], which introduces a new state to identify *active* reservations. At each point in time, the algorithm keeps track of the CPU fraction (bandwidth) allocated to *active* reservations. Whenever the sum of all allocated bandwidths is below the total availability, the computation time in excess can be reclaimed. Contrary to CASH, the GRUB algorithm does not make restrictions on the task sets and for this reason it is the second cornerstone (along with the hard reservation) of the HGRUB algorithm presented in this paper.

Strongly related to HGRUB are the Idle-time Reclaiming Improved Server (IRIS) algorithm [21], the Best Effort Bandwidth Server (BEBS) [7], and the Rate-Based Earliest Deadline (RBED) scheduler [10]. Similarly to HGRUB, the IRIS algorithm uses a combination of hard reservations with reclaiming. However, as recognised by the same authors, IRIS generates a very large number of preemptions introducing a large overhead [25]. In the case of BEBS or RBED, the authors manipulate the scheduling parameters to accommodate non-real-time applications. In contrast, the idea of HGRUB is to maintain a clear separation between the scheduling mechanism and the policy whereby scheduling parameters are chosen. This separation of concerns endows the designer with a large flexibility in pursuing different goals using the same scheduling mechanism.

Some other works propose to use explicit *slack reclaiming* techniques [17]. However, such works still focus on (soft) real-time performance, and propose to schedule all the non-real-time tasks in a single server, making it difficult (for example) to provide worst − case fairness. Moreover, HGRUB allows to reclaim unused execution time in a simpler way, by just modifying the CBS accounting and enforcement mechanisms.

VII. CONCLUSIONS AND FUTURE WORK

This paper described HGRUB, a reservation-based scheduling algorithm that has been developed to address some problems encountered when using CPU reservations in a general-purpose operating system. The new algorithm adds a reclaiming mechanism and hard reservation behavior to the original CBS scheduling algorithm. Theoretical evidence has been offered of the effectiveness of HGRUB in a wide range of situations: from serving hard and soft real-time tasks to non-real-time tasks and hierarchies of schedulers.

This good performance is the result of some important properties: HGRUB is *work-conserving*, and it provides *hard schedulability* and worst − case fairness. These properties have been introduced in previous papers, but have been formally proved only in this paper. The effectiveness of the proposed algorithm has also been proven through extensive experiments performed on the Linux implementation, and the most significant of such experiments have been reported in the paper.

As a future work, HGRUB will be extended to support resource sharing, and the current implementation will be better integrated with the preempt-RT patches [1]. This will allow us to use HGRUB for scheduling interrupt threads (some preliminary tests with the patched version of the Linux kernel named `2.6.21-rc5-rt4` are being currently performed).

APPENDIX

The theorem is proved by contradiction, based on the schedulability of GRUB. The basic idea is to show that if HGRUB is not able to respect a scheduling deadline, then GRUB is not able to respect such a deadline either. Since the schedulability property has been already proved for GRUB, this is not possible.

First of all, note that the **fairness preserving** rule does not affect the assignment of scheduling deadlines, or the accounting of the budget. Hence, such a rule does not affect HGRUB schedulability, and it is sufficient to prove that algorithm HGRUB without the **fairness preserving** rule is schedulable.

Thus, let us consider HGRUB without the **fairness preserving** rule, and let us assume that it is not schedulable. Hence, it is possible to find a task set $\Gamma = \{\tau_i\}$ for which a task τ_j misses a scheduling deadline. Based on Γ, it is possible to construct a second task set $\Gamma' = \{\tau_i'\}$ such that every job $J_{i,j}'$ of a task $\tau_i' \in \Gamma'$:

- arrives when a job of task $\tau_i \in \Gamma$ arrives, or when server S_i is replenished;
- finishes when a job of task $\tau_i \in \Gamma$ finishes, or when server S_i is depleted.

By definition of Γ', the schedule of Γ' is identical to the schedule of Γ, and the scheduling deadlines assigned to tasks τ_i are the same as those assigned to tasks τ_i'. The only difference between the two task sets is that when serving Γ' no server is ever depleted (because the tasks τ_i' is blocked when the server would be depleted, and unblocked when the server would be recharged).

Note that since no server is depleted, the schedule of Γ' generated by HGRUB (without the **fairness preserving** rule) is identical to the schedule generated by the original GRUB algorithm. Hence, since τ_j misses a scheduling deadline when scheduled by HGRUB, τ_j' misses a deadline too, and τ_j' also misses a deadline when scheduled by GRUB. However, it is well known that when using GRUB if $\sum_i (Q_i)/(T_i)$ then no task misses its scheduling deadline. Hence the contradiction.

As a result, when using HGRUB no scheduling deadline is missed.

The lemma is evidently valid when the `Active` set is empty. To prove the property, it is sufficient to prove that any possible change in the system state cannot invalidate the invariant. To this regard, we can have several cases.

The first case, is when server S_i executes[2] in $[t, t + \Delta t]$ without depleting and without any changes in U^{act}. In this case, if the invariant is respected at time t, then the invariant is

---

[2]The expression "server S_i executes" here is used to indicate that task τ_i served by S_i is executed.

respected at time $t + \Delta t$ as follows:

$$\sum_{\tau_j \in \text{Active}} \left(d_j^s(t + \Delta t) - (t + \Delta t)\right) U_j - q_j(t + \Delta t)$$

$$= \sum_{\tau_j \in \text{Active}, j \neq i} \left(d_j^s(t + \Delta t) - (t + \Delta t)\right) U_j - q_j(t + \Delta t)$$
$$+ \left(d_i^s(t + \Delta t) - (t + \Delta t)\right) U_i - q_i(t + \Delta t)$$

$$= \sum_{\tau_j \in \text{Active}, j \neq i} \left(d_j^s(t) - t\right) U_j - \Delta t U_j - q_j(t)$$
$$+ \left(d_i^s(t) - t\right) U_i - \Delta t U_i - q_i(t) + \Delta t U^{\text{act}}$$

$$= \sum_{\tau_j \in \text{Active}} \left(d_j^s(t) - t\right) U_j - q_j(t) + \Delta t U^{\text{act}}$$
$$- \sum_{\tau_j \in \text{Active}} \Delta t U_j = 0 + \Delta t U^{\text{act}} - \Delta t$$
$$\times \sum_{\tau_j \in \text{Active}} U_j = 0$$

where $q_i(t)$ and $d_i^s(t)$, respectively, denote the value of the budget at time t.

Note that $q_i(t + \Delta t) = 0$ (server S_i is depleted at time $t + \Delta t$), then

$$\sum_{\tau_j \in \text{Active}} \left(d_j^s(t + \Delta t) - t - \Delta t\right) U_j - q_j(t + \Delta t) = 0$$

$$\Rightarrow (d_i^s(t + \Delta t) - t - \Delta t)) U_i - q_i(t + \Delta t)$$
$$= - \sum_{\tau_j \in \text{Active}, j \neq i} \left(d_j^s(t + \Delta t) - t - \Delta t\right) U_j - q_j(t + \Delta t)$$
$$\Rightarrow (d_i^s(t + \Delta t) - t - \Delta t) U_i$$
$$= - \sum_{\tau_j \in \text{Active}, j \neq i} \left(d_j^s(t + \Delta t) - t - \Delta t\right) U_j - q_j(t + \Delta t).$$

Since

$$d_i^s(t + \Delta t) \geq t + \Delta t \Rightarrow (d_i^s(t + \Delta t) - t - \Delta t) U_i \geq 0$$

we can conclude

$$\sum_{\tau_j \in \text{Active}, j \neq i} \left(d_j^s(t + \Delta t) - t - \Delta t\right) U_j - q_j(t + \Delta t) \leq 0.$$

But $d_j^s(t + \Delta t) \geq t + \Delta t$, hence there is at least a task $\tau_j \in$ Active, $j \neq i$ with $q_j(t + \Delta t) > 0$.

The second case is when the deadline of server S_i is postponed at time t. Since the deadline is only postponed when the budget $q_i(t)$ is 0, the contribution of S_i to the sum before postponing the deadline is $(d_i^s(t) - t)U_i - 0 = (d_i^s(t) - t)U_i$. After the deadline has been postponed, the contribution of S_i to the sum is

$$(d_i^s(t) + T_i - t)U_i - Q_i = (d_i^s(t) - t)U_i + T_i U_i - Q_i$$
$$= (d_i^s(t) - t)U_i + Q_i - Q_i$$
$$= (d_i^s(t) - t)U_i$$

hence the invariant is not affected.

The third case, is when task τ_i is activated at time t. There are two possible situations:

1) $t \leq d_i^s(t) - (q_i(t)/Q_i)T_i$: in this case, τ_i is already in the *Active* set. Since $U^{\text{act}}, d_i^s(t)$, and $q_i(t)$ are not changed, the invariant is not affected by τ_i's arrival;

2) $t > d_i^s(t) - (q_i(t)/Q_i)T_i$: in this case, $d_i^s(t^+) = t + T_i$, and $q_i(t^+) = Q_i$. Hence, S_i contributes to the invariant with a term $(d_i^s(t^+) - t)U_i - q_i(t^+) = (t + T_i - t)U_i - Q_i = Q_i - Q_i = 0$. So, the invariant is not affected again.

The fourth case is when task τ_i is blocked at time t. Even in this case, there are two possible situations:

1) $t \leq d_i^s(t) - (q_i(t)/Q_i)T_i$: in this case, τ_i remains in the active set until time $d_i^s(t) - (q_i(t)/Q_i)T_i$. Since $U^{\text{act}}, d_i^s(\text{t})$, and $q_i(\text{t})$ are not changed, the invariant is not affected by τ_i's blocking. Note that $t \leq d_i^s(t) - (q_i(t)/Q_i)T_i \Rightarrow ((d_i^s(t) - t)/T_i)Q_i - q_i \geq 0$ and $\sum_{\tau_j \in \text{Active}}(d_j^s(t) - t)U_j - q_j(t) = 0 \Rightarrow (d_i^s(t) - t)U_i - q_i(t) = -\sum_{\tau_j \in \text{Active}, j \neq i}(d_j^s(t) - t)U_j - q_j(t)$, hence $\sum_{\tau_j \in \text{Active}, j \neq i}(d_j^s(t) - t)U_j - q_j(t) \leq 0$. Since $d_j^s \geq t$, this means that there is at least a task $\tau_j \in$ Active, $j \neq i$ with $q_j(t) > 0$;

2) $t > d_i^s(t) - (q_i(t)/Q_i)T_i$: according to **fairness preserving** rule, τ_i blocks, but server S_i is used to serve a different task and does not deactivate. Hence, the invariant is not affected (server S_i will be deactivated only when $t = d_i^s(t) - (q_i(t)/Q_i)T_i$).

The last case is when a blocked task τ_i exits the Active set at time t, this means that $t = d_i^s(t) - (q_i(t)/Q_i)T_i \Rightarrow (d_i^s(t) - t)U_i - q_i(t) = 0$. Hence, the invariant is not affected.

Finally, note that the case in which all servers serving tasks $\tau_i \in$ Active are depleted cannot happen, as previously noted.

REFERENCES

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

[2] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proc. IEEE Real-Time Systems Symp.*, Madrid, Spain, Dec. 1998.

[3] L. Abeni and G. Buttazzo, "Constant bandwidth vs. proportional share resource allocation," in *Proc. IEEE Int. Conf. Mutimedia Computing and Systems*, Florence, Italy, Jun. 1999.

[4] L. Abeni and G. Buttazzo, "QoS guarantee using probabilistic deadlines," in *Proc. IEEE Euromicro Conf. Real-Time Systems*, York, U.K., Jun. 1999.

[5] L. Abeni and G. Lipari, "Implementing resource reservations in Linux," in *Proc. Real-Time Linux Workshop*, Boston, MA, Dec. 2002.

[6] L. Abeni, C. Scordino, G. Lipari, and L. Palopoli, "Serving non real-time tasks in a reservation environment," in *Proc. 9th Real-Time Linux Workshop*, Linz, Austria, Nov. 2007.

[7] S. A. Banachowski, T. Bisson, and S. A. Brandt, "Integrating best-effort scheduling into a real-time system," in *Proc. RTSS*, 2004, pp. 139–150.

[8] J. Bennett and H. Zhang, "WF2Q: Worst-case fair weighted fair queueing," in *Proc. INFOCOM'96*, Mar. 1996.

[9] J. Bennett and H. Zhang, "Why WFQ is not good enough for integrated services networks," in *Proc. NOSSDAV'96*, Apr. 1996.

[10] S. A. Brandt, S. Banachowsky, C. Lin, and T. Bisson, "Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes," in *Proc. Real-Time Systems Symp.*, Cancun, Mexico, Dec. 2003.

[11] M. Caccamo, G. C. Buttazzo, and L. Sha, "Handling execution overruns in hard real-time control systems," *IEEE Trans. Comput.*, vol. 51, no. 7, pp. 835–849, Jul. 2002.

[12] M. Caccamo, G. C. Buttazzo, and D. C. Thomas, "Efficient reclaiming in reservation-based real-time systems with variable execution times," *IEEE Trans. Comput.*, vol. 54, no. 2, pp. 198–213, Feb. 2005.

[13] X. Feng and A. K. Mok, "A model of hierarchical real-time virtual resources," in *Proc. 23rd IEEE Real-Time Systems Symp.*, Austin, TX, Dec. 2002, pp. 26–35.

[14] G. Lipari and S. Baruah, "Greedy reclamation of unused bandwidth in constant bandwidth servers," in *IEEE Proc. 12th Euromicro Conf. Real-Time Systems*, Stokholm, Sweden, Jun. 2000.

[15] P. Holman and J. H. Anderson, "Implementing pfairness on a symmetric multiprocessor," in *Proc. IEEE Real-Time and Embedded Technology and Applications Symp.*, 2004.

[16] D.-I. Kang, R. Gerber, and M. Saksena, "Parametric design synthesis of distributed embedded systems," *IEEE Trans. Comput.*, vol. 49, no. 11, pp. 1555–1169, Nov. 2000.

[17] C. Lin and S. A. Brandt, "Improving soft real-time performance through better slack reclaiming," in *Proc. Real-Time Systems Symp.*, Miami, FL, Dec. 2005.

[18] G. Lipari and S. K. Baruah, "A hierarchical extension to the constant bandwidth server framework," in *Proc. IEEE Real-Time Technology and Applications Symp.*, Taipei, Taiwan, May 2001, IEEE Comput. Soc. Press.

[19] G. Lipari and E. Bini, "A methodology for designing hierarchical scheduling systems," *J. Embed. Comput.*, vol. 1, no. 2, pp. 257–269, 2004.

[20] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. Assoc. Comput. Mach.*, vol. 20, no. 1, pp. 46–61, 1973.

[21] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo, "Iris: A new reclaiming algorithm for server-based real-time systems," in *Proc. IEEE Real-Time and Embedded Technology and Applications Symp.*, 2004, pp. 211–218.

[22] C. W. Mercer, S. Savage, and H. Tokuda, Processor Capacity Reserves for Multimedia Operating Systems, Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-CS-93-157, May 1993.

[23] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource kernels: A resource-centric approach to real-time and multimedia systems," in *Proc. SPIE/ACM Conf. Multimedia Computing and Networking*, Jan. 1998.

[24] J. Regehr, "Inferring scheduling behavior with hourglass," in *Proc. USENIX Annual Technical Conf. FREENIX Track*, Monterey, CA, Jun. 2002.

[25] C. Scordino, "Dynamic voltage scaling for energy-constrained real-time systems," Ph.D. dissertation, Comput. Sci. Dept., Univ. Pisa, Pisa, Italy, Dec. 2007.

[26] C. Scordino and G. Lipari, "Using resource reservation techniques for power-aware scheduling," in *Proc. 4th ACM Int. Conf. Embedded Software (EMSOFT)*, Pisa, Italy, Sep. 2004, pp. 16–25.

[27] I. Shih and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Proc. 24th Real-Time Systems Symp.*, Cancun, Mexico, Dec. 2003, pp. 2–13.

[28] 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 *Proc. IEEE Real-Time Systems Symp.*, Dec. 1996.

[29] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard-real-time environments," *IEEE Trans. Comput.*, vol. 44, no. 1, pp. 73–91, Jan. 1995.

Luca Abeni received the computer engineering degree from the University of Pisa, Pisa, Iraly, in 1998 and the Ph.D. degree from "Scuola Superiore Sant'Anna, Pisa" in 2002.

He is an assistant professor at the "Dipartimento di Ingegneria e Scienza dell'Informazione—DISI", University of Trento, Trento, Italy. From 2003 to 2006, he worked in Broadsat S.R.L., developing IPTV applications and audio/video streaming solutions over wired and satellite (DVB-MPE) networks. His main research interests are in real-time operating systems, scheduling algorithms, quality-of-service management, multimedia applications, and audio/video streaming.
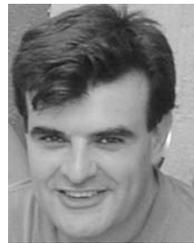
Luigi Palopoli received the computer engineering degree from the University of Pisa, Pisa, Italy, in 1992 and the Ph.D. degree in computer engineering from "Scuola Superiore Sant'Anna, Pisa" in 2002.

He is an Assistant Professor of computer engineering at the University of Trento, Trento, Italy. His main research activities are in embedded system design with a particular focus on resource—aware control design and adaptive mechanisms for quality-of-service management. He has served on the program committee of different conferences in the area of real-time and control systems.

Claudio Scordino received the computer engineering and Ph.D. degrees from the University of Pisa, Pisa, Italy, in 2003 and 2007, respectively.

During 2005, he was a visiting student at the University of Pittsburgh, Pittsburgh, PA, collaborating with Prof. D. Mossè on energy-aware real-time scheduling. He has been a teaching assistant at the University of Pisa for two years. His research activities includes operating systems, real-time scheduling, energy saving, and embedded devices.

Giuseppe Lipari received the computer engineering degree from the University of Pisa, Pisa, Italy, in 1996 and the Ph.D. degree in computer engineering from "Scuola Superiore Sant'Anna, Pisa" in 2000.

He is an Associate Professor of operating systems with Scuola Superiore Sant'Anna. His main research activities are in real-time scheduling theory and its application to real-time operating systems, soft real-time systems for multimedia applications, and component-based real-time systems. He has been a member of the program committees of many conferences in the field.

Dr. Lipari is currently an Associate Editor for IEEE TRANSACTIONS ON COMPUTERS.