A Resource Reservation Algorithm for Power-Aware Scheduling of Periodic and Aperiodic Real-Time Tasks

Claudio Scordino, *Student Member*, *IEEE*, and Giuseppe Lipari, *Member*, *IEEE*

Abstract—Power consumption is an important issue in the design of real-time embedded systems. As many embedded systems are powered by batteries, the goal is to extend the autonomy of the system as much as possible. To reduce power consumption, modern processors can change their voltage and frequency at runtime. A power-aware scheduling algorithm can exploit this capability to reduce power consumption while preserving the timing constraints of real-time tasks. In this paper, we present GRUB-PA, a novel power-aware scheduling algorithm based on a resource reservation technique. In addition to providing temporal isolation and time guarantees and, unlike most of the power-aware algorithms proposed in the literature, GRUB-PA can efficiently handle systems consisting of both hard and soft, aperiodic, sporadic, and periodic tasks. We compared our algorithm with existing power-aware scheduling algorithms on an extensive set of simulation experiments on synthetic task sets. The results show that the performance of our algorithm is in line with the state-of-the-art power-aware algorithms. We also present the implementation of our algorithm in the Linux operating system and discuss practical implementation issues like switching overhead and power models. Finally, we show the results of experiments performed on a real testbed application.

Index Terms—DVS, real-time, resource-reservation, scheduling, power-aware.

1 Introduction

The problem of reducing energy consumption is becoming very important in the design and implementation of embedded real-time systems. Many of these systems are powered by rechargeable batteries and the goal is to extend the autonomy of the system as much as possible. Battery technology is improving rather slowly and cannot keep up with the demands of modern digital systems. A wide range of battery operated systems, like notebook computers, smart phones, autonomous robots, sensor nodes, and PDAs (Personal Digital Assistants), can only operate on a limited battery supply. Battery life is one of the most important parameters for such devices, directly influencing the system size and weight.

Problems related to energy consumption can also be found in high-end processors, like the ones used in normal workstation PCs and in servers [1], [2]. The increases in performance are obtained by increasing the clock frequency and reducing the size of the transistors. The net effect is that the power consumed by these processors is increasing due to increases in the static leakage current and in the dynamic switching frequency. As a side effect, dissipating the heat generated becomes more difficult. Conventional computers are currently air cooled and manufacturers are facing the problem of building powerful systems without introducing additional techniques such as liquid cooling. It has been shown [1] that the fans driving the cooling system can consume up to 50 percent of the total system power in small commercial servers. To better understand the impact of techniques for reducing power consumption in a high-end server, consider the cost savings that can be obtained by reducing the energy consumed in large Web server farms in terms of air conditioning and cooling systems. However, this reduction in power consumption must not affect the performance of the applications.

One possible approach to reducing power consumption is to selectively adjust the processor voltage. This technique is called *Dynamic Voltage Scaling* (DVS) [3]. Many modern processors can dynamically lower the voltage to reduce the power consumption [4], [5], [6], [7]. However, by reducing the voltage, the gate delay increases. Thus, in most cases, it is necessary to lower the operating frequency and the processor speed. As a consequence, all applications will take more time to execute.

In real-time systems, a task may be assigned timing constraints like *deadlines*. A real-time task must complete before its deadline; otherwise, the results could be produced too late to be useful. In safety critical applications, a deadline miss could result in serious consequences for the system. A *hard real-time task* is a critical activity whose deadline can never be missed; otherwise, a critical system failure can compromise the functionality of the system. This kind of task is typically used to control or monitor some physical device and a missed deadline may cause catastrophic consequences. Hard real-time tasks are needed in a number of application domains, including air-traffic, industrial, chemical, nuclear, safety-critical, and military controls. Examples of hard real-time systems operated by

- *C. Scordino is with the Computer Science Department, University of Pisa, Largo B. Pontecorvo, 3-56127 Pisa, Italy. E-mail: scordino@di.unipi.it.*
- *G. Lipari is with the Retis Lab, Scuola Superiore Sant'Anna, Piazza Martiri della Libertà, 33-56127 Pisa, Italy. E-mail: lipari@sssup.it.*

Manuscript received 1 July 2005; revised 1 Dec. 2005; accepted 21 Mar. 2006; published online 20 Oct. 2006.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0219-0705.

batteries or by solar cells are autonomous robots operating in hazardous environments (e.g., the robots sent by NASA for exploring the surface of Mars).

Soft real-time tasks are less critical. For these tasks, the timing constraints are important but not critical and the system tolerates some occasional deadline misses. Typically, the number of missed deadlines is related to the Quality of Service (QoS) provided by the application. A deadline miss does not compromise the correctness of the system, but its QoS degrades. Typical examples of soft real-time are multimedia and telecommunication applications.

In practice, many systems consist of a mixture of hard and soft real-time tasks. The objective is to guarantee that all hard tasks will always complete before their deadlines and, at the same time, to maximize the quality of service provided by soft real-time tasks.

When applying DVS techniques to a real-time embedded system, it is necessary to identify the conditions under which we can safely slow down the processor without missing any deadline (for hard real-time systems) or missing a limited number of deadlines (for soft real-time tasks). If the frequency change is not done properly, the timing requirements of the application cannot be respected.

A *power-aware scheduling algorithm* exploits DVS by selecting, at each instant, both the task to be scheduled and the processor's operating frequency. The problem becomes more difficult in systems with a combination of hard and soft, periodic and aperiodic real-time tasks. Recently, many power-aware algorithms have been proposed in the literature. We discuss the previous work on DVS in Section 2.

The problem of mixing hard and soft real-time tasks can be efficiently solved by using the *resource reservation framework* [8]. In such a framework, each task is assigned a *server* characterized by a budget Q and a period P, the interpretation being that the task is allowed to execute for at least Q units of time every P. Many server algorithms have been presented in the literature, for both fixed priority and dynamic priority schedulers [9], [10], [11], [12]. If the tasks execute less than expected, the remaining *slack time* can be used to reduce the response time of soft aperiodic tasks. Techniques for using this slack time are usually referred to as *reclamation techniques*. Examples of such reclamation techniques for resource reservation under dynamic scheduling have been proposed recently [13], [14].

Intuitively, the problem of reclaiming the spare bandwidth is similar to the problem of power-aware scheduling. We can divide both problems into two parts. The first part consists of identifying the spare bandwidth (or the slack time) in the system, whereas the second part consists of deciding how to use the spare bandwidth. The first part of the problem is common to both the bandwidth reclamation and the power-aware scheduling problems. The second part, instead, differs radically: In the reclamation problem, the goal is to use the spare time to anticipate the execution time of aperiodic tasks, whereas, in the power-aware scheduling problem, the goal is to lower the processor frequency as much as possible. We believe that many reclamation algorithms can be used as power-aware schedulers by modifying their "second" part.

In this paper, we present the GRUB-PA (*Greedy Reclamation of Unused Bandwidth—Power Aware*) algorithm [15] that follows the previous idea. It is based on the GRUB algorithm, proposed by Lipari and Baruah [13], [16], which in turn is based on the resource reservation framework. Therefore, our algorithm can support both hard and soft real-time tasks. Moreover, unlike many power-aware algorithms, GRUB-PA is able to deal with periodic, sporadic, and even aperiodic tasks.

The paper is organized as follows: In Section 2, we present related work in the area of resource reservation and power-aware scheduling. Section 3 introduces the models and the notation that will be used throughout the paper. In Sections 4 and 5, we present our algorithm and prove its correctness. In Section 6, we present simulation experiments that compare GRUB-PA with three state-of-the-art algorithms: the DRA algorithm, proposed by Aydin et al. [17], the RTDVS algorithm, proposed by Pillai and Shin [18], and the DVSST algorithm, proposed by Qadi et al. [19].

Finally, in Section 7, we present the implementation of our algorithm in the Linux operating system and some experiments on a real test-bed system. In Section 8, we state our conclusions.

2 Related Work

Power-aware scheduling techniques can be divided into static (offline) and dynamic (online) techniques. Static techniques are typically applied to periodic tasks, and make use of offline parameters (such as periods and worst-case execution times) to select the appropriate processor voltage/frequency to be used. These techniques can be further divided into two classes: fixed system voltage and fixed task voltage. In the former case, a single optimal speed is computed and assigned to all tasks, without any overhead for voltage switching at runtime. Pillai and Shin [18] derived an optimal algorithm for computing the minimal speed that can make a task set schedulable under the Earliest Deadline First (EDF) scheduler [20], [21] and proposed a near-optimal method under Rate Monotonic. Saewong and Rajkumar [22] provided an algorithm to find the optimal speed value for fixed priority assignments.

In the second class of static methods, the processor voltage is not fixed but *statically* assigned before system execution, based upon a task's parameters [22], [23], [24]. In other words, given a set of periodic tasks, the algorithm assigns a possibly different voltage to each individual task. The assignments are still computed offline and are fixed until the task set changes. Shin and Kim [25] proposed a static algorithm for real-time systems with both periodic and aperiodic tasks. Aperiodic tasks are handled using a dedicated server. Saewong and Rajkumar [26] proposed a voltage-scaling algorithm (called PM-Clock) for hard real-time systems using fixed-priority (i.e., Rate Monotonic or Deadline Monotonic) schedulers.

Dynamic techniques have been the topic of much recent research. In most applications, the probability of a task taking an amount of runtime equal to its worst-case execution time (WCET) is very low [27]. Hence, dynamic techniques can exploit the slack time for reducing energy consumption when tasks have a variable execution time.

Pillai and Shin [18] proposed the RTDVS-Cycle Conserving and RTDVS-Look Ahead algorithms to take into account the slack time. Aydin et al. [17] proposed the DRA algorithms based on EDF for reclaiming the spare time. A power-aware algorithm for EDF scheduling has been proposed also by Zhu and Mueller [28]. Similar techniques have been proposed by Saewong and Rajkumar [22] in the context of fixed priority scheduling. All these techniques assume hard real-time periodic task sets.

Some work has been done in the context of soft real-time tasks. For example, Pouwelse et al. [29], [3] presented a study of power consumption and power-aware scheduling applied to multimedia streaming. Lorch and Smith addressed variable voltage scheduling of tasks with soft deadlines in [30]. Kumar and Srivastava [31] proposed a prediction mechanism for fixed-priority scheduling of soft periodic tasks. However, these techniques are based on heuristics and cannot provide guarantees to hard real-time tasks.

Recently, Qadi et al. [19] presented the DVSST algorithm that schedules sporadic hard real-time tasks, reclaiming the unused bandwidth to lower the processor frequency. The basic idea is to keep track of the total bandwidth used by all active sporadic tasks with a variable U: When a sporadic task is activated, U is increased by U_i (the task's utilization, $U_i = \frac{C_i}{T_i}$) and, at task's deadline, the bandwidth is decreased by U_i. The processor frequency is changed depending on the value of U. This approach resembles our algorithm GRUB-PA. However, the DVSST algorithm is not able to reclaim the spare bandwidth that is due to tasks with variable execution time. Indeed, in the case of periodic tasks, DVSST maintains a constant U. As we will see in the remainder of the paper, our algorithm, GRUB-PA, instead, explicitly reclaims the spare bandwidth of tasks that execute less than the worst case and, therefore, is able to reclaim spare time in the case of both periodic and sporadic tasks.

Shin and Kim [25] proposed dynamic algorithms for power-aware scheduling, using both fixed priority or EDF policies and a dedicated server (i.e., Deferrable Server [32] or Total Bandwidth Server [11]) to handle aperiodic tasks. Using some existing DVS algorithms (including a modified version of DRA [17]), slack time is reclaimed for both periodic and aperiodic tasks.

Our approach takes a more abstract view and, thus, is more general than the approach by Shin and Kim. Our algorithm is based on the resource reservation framework [8]. All resource reservation algorithms provide the *temporal isolation property*: The temporal behavior of one task (i.e., its ability to meet its deadlines) is not affected by the behavior of the other tasks. Thanks to the temporal isolation property, each task executes as if it were on a slower dedicated processor. Therefore, it is possible to provide guarantees on a per-task basis.

3 System Model and Notation

In this section, we introduce the models and the notation that will be used throughout the paper.

3.1 Task Model

Typically, a real-time system is implemented as a set of concurrent tasks that are executed on a real-time operating system (RTOS). The objective of an RTOS is to manage and control the assignment of some resources (e.g., the processor) to the tasks that need them in order to meet predefined timing constraints. In this paper, we consider the processor as the only resource shared by a set of real-time tasks, reducing the scheduling problem to the choice of a possible assignment of the processor to the tasks.

A real-time task can be modeled as a sequential stream of jobs. The *job* is the unit of work, scheduled and executed by the operating system. Each task τ_i generates a sequence of jobs $J_i^1, J_i^2, J_i^3, \ldots$, where J_i^j becomes ready for execution (arrives) at time a_i^j ($a_i^j \leq a_i^{j+1}\ \forall i, j$) and requires a computation time of c_i^j. Jobs of the same task must be executed sequentially (it is not possible to parallelize two jobs of the same task) and are executed in FIFO order—i.e., J_i^j has to finish before J_i^{j+1} can start executing. Moreover, each job is assigned an absolute deadline d_i^j, which is the time by which the job *must* complete (in the case of a hard real-time task) or *should* complete (in the case of a soft real-time task).

Periodic tasks release their jobs at regular intervals of time: $a_i^j = a_i^{j-1} + T_i$, where T_i is the task period. Sporadic tasks have a minimum interarrival time between consecutive jobs: $a_i^j \geq a_i^{j-1} + T_i$, where T_i denotes the minimum interarrival time. Usually, the job deadline d_i^j is computed based on the *task relative deadline* d_i: $d_i^j = a_i^j + d_i$.

The scheduling algorithm presented in this paper is very general and does not assume knowledge of tasks periodicity. More formally, the algorithm makes the following assumptions:

- The arrival times of the jobs (the a_i^js) are not a priori known, but are only revealed online during system execution. Hence, our scheduling strategy cannot require knowledge of future arrival times. Notice that many power-aware algorithms for periodic task sets (like DRA [17] or RTDVS [18]) exploit knowledge of future arrival times to simplify the solution.
- The exact execution requirements, c_i^j, are also not known beforehand: They can only be determined by actually executing J_i^j to completion.

Our algorithm can handle any kind of task—periodic, sporadic, and aperiodic tasks. Of course, to be able to do schedulability analysis, the designer must know the minimum interarrival times and the worst-case execution times of the tasks. However, since our algorithm provides temporal protection, each task can be analyzed and guaranteed *in isolation*—i.e., without making any assumption on the other tasks in the system.

3.2 Processor Model

The tasks are executed on a single processor with a variable operating frequency. Many power-aware algorithms make the assumption of continuous frequency scaling, even though no existing processor can vary its frequency with continuity. In fact, all processors that support DVS provide a set of operating modes, each one characterized by a value of frequency and voltage [4], [5], [6], [7].

We assume that the processor can provide M frequencies, $\phi_1, \ldots, \phi_M$, in increasing order. A supply voltage $V_{DD-1}, \ldots, V_{DD-M}$ and a normalized processor "speed"

$\overline{U}_1, \ldots, \overline{U}_M$ are associated to each frequency, again in increasing order, with $\overline{U}_M = 1$. The computation times of the tasks are relative to the maximum operating speed, $\overline{U}_M = 1$, and vary linearly with the processor speed: Therefore, if a job executes for e_i^j units of time when the processor speed is 1, it executes for $e_i^j/\overline{U}_k$ when the processor speed is set equal to $\overline{U}_k$ (in Section 7, this assumption will be validated experimentally on a real embedded system).

3.3 Power Consumption

Today, most digital devices are implemented using Complementary Metal Oxide Semiconductor (CMOS) circuits. The power consumption of this kind of circuit can be modeled accurately with simple equations [33], [29], [34].

CMOS circuits have both static and dynamic power consumption. In the ideal case, they do not dissipate static power since, in steady state, there is no open path from source to ground. In reality, bias and leakage currents through the MOS transistors cause a static power consumption which is a (usually) small portion of the total power consumed by the circuit. Although the static power today is about two orders of magnitude smaller than the total power, as integration technology advances, it is expected that the leakage power will significantly affect, if not dominate, the overall energy consumption in integrated circuits [34].

The dynamic power consumption in CMOS microprocessors is dissipated during the transient behavior (i.e., during switches between logic levels). Every transition of a digital circuit consumes power because every charge or discharge of the digital circuit's capacitance drains power.

If we assume that the dynamic component is the most dominant one, we can associate a power consumption

$$P_k \propto \phi_k * V_{DD-k}^2 \tag{1}$$

to the frequency ϕ_k, as done in [34], [3], [33]. Notice that the power consumption scales linearly with the frequency and quadratically with the voltage—i.e., reducing frequency and voltage together reduces energy per operation quadratically, but only decreases performance linearly.

3.4 Overhead

One issue that must be taken into careful consideration is the overhead of changing frequency. Changing frequency is not "for free" as the processor needs some transitory time to adjust to the new frequency. The duration of this transitory is variable and varies a lot from processor to processor. For example, on the Intel PXA250, it can go up to 500 μsec. Even though, in many soft real-time applications, this can be considered negligible, it should not be ignored. We will show how to account for this delay in the GRUB-PA algorithm in Section 5.4.

The presence of an energy overhead at every frequency switch is also undeniable. This overhead depends on the particular kind of processor the algorithm is running on and it is quite difficult to estimate and measure. In this paper, we do not explicitly take into account this energy overhead. However, we devise a technique to limit the number of switches in an interval of time, therefore limiting the maximum amount of energy spent for switching frequency.

4 Algorithm GRUB

We are interested in integrating our scheduling methodology with traditional real-time scheduling—in particular, we wish to design a scheduler that is a minor variant of the classical Earliest Deadline First (EDF) scheduling algorithm [21].

Since the GRUB-PA algorithm proposed in this paper is based on the GRUB algorithm (Greedy Reclamation of Unused Bandwidth) [13], [16], in this section we briefly describe the original algorithm. The interested reader can refer to the original paper for a more detailed presentation of the algorithm.

In this section, we assume that the processor speed is set to the maximum and is not changed. In Section 5, we will show how it is possible to extend the GRUB algorithm to exploit DVS.

GRUB is an algorithm belonging to the class of *aperiodic servers with dynamic priorities*. This class of techniques consists of creating an abstract entity for each task called *server*. Several server-based schedulers (e.g., CBS [35]) can offer performance guarantees somewhat similar to the one made by algorithm GRUB. However, algorithm GRUB has an added feature that is not to be found in many of the other schedulers—an ability to *reclaim* unused processor capacity ("bandwidth") that is not used because some of the servers may have no outstanding jobs awaiting execution.

4.1 Description of the GRUB Algorithm

4.1.1 Server Model

Each server is characterized by two parameters (U_i, P_i), where U_i is the server bandwidth (or fraction of the processor utilization) and P_i is the period.

We consider a system comprised of n servers, $S_1, S_2, \ldots, S_n$, with each server S_i characterized by the parameters U_i and P_i as described above. We require the sum of the processor shares of all the servers to be no more than one, i.e.,

$$\left(\sum_{i=1}^{n} U_i\right) \leq 1.$$

4.1.2 Algorithm Variables

For each server S_i in the system, algorithm GRUB maintains two variables: a *deadline*, D_i, and a *virtual time*, V_i. Initially, these variables are both initialized to 0. The server deadline, D_i, is used to select which server is executing on the processor—GRUB essentially implements the EDF algorithm among all active servers. The virtual time, V_i, is a measure of how much bandwidth the server has consumed. The meaning of these variables will be clearer later.

At any instant in time during runtime, each server S_i is in one of three states: Inactive, Active Contending, or Active Non Contending. The initial state of each server is Inactive. Intuitively, at time t_o, a server is in the Active Contending state if it has some jobs awaiting execution at that time, in the Active Non Contending state if it has completed all the jobs that arrived prior to t_o, but in doing so has "used up"

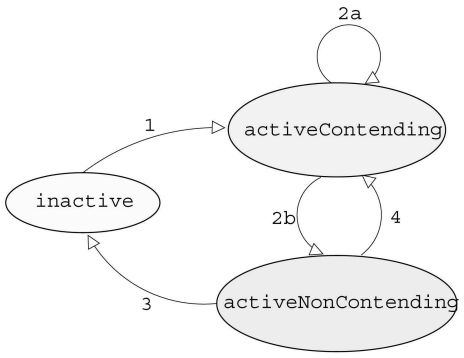


Fig. 1. State transition diagram.

its share of the processor until beyond t_o (i.e., its virtual time is greater than t_o), and in the Inactive state if it has no jobs awaiting execution at time t_o and it has *not* used up its processor share beyond t_o.

At each instant in time, from among all servers that are in the Active Contending state, algorithm GRUB chooses for execution (the next job of) the server S_i whose deadline parameter D_i is the smallest.

While (a job of) S_i is executing, its virtual time V_i increases (the exact rate of this increase will be specified later); while S_i is not executing, V_i does not change. If, at any time, this virtual time becomes equal to the deadline ($V_i == D_i$), then the deadline parameter is incremented by P_i ($D_i \leftarrow D_i + P_i$). Notice that this may cause S_i to no longer be the earliest-deadline active server, in which case, it may yield control of the processor to an earlier-deadline server.

4.1.3 State Transitions

Certain (external and internal) events cause a server to change its state (see Fig. 1).

1. If server S_i is in the Inactive state and a job J_i^j arrives (at time-instant a_i^j), then the following code is executed:

$$V_i \leftarrow a_i^j$$
$$D_i \leftarrow V_i + P_i$$

   and server S_i enters the Active Contending state.
2. When a job J_i^{j-1} of S_i completes (notice that S_i must then be in the Active Contending state), the action taken depends upon whether the next job J_i^j of S_i has already arrived.
   a. If so, then the deadline parameter D_i is updated as follows:

$$D_i \longleftarrow V_i + P_i$$

      and the server remains in the Active Contending state.
   b. If there is no job of S_i awaiting execution, then server S_i changes state and enters the Active Non Contending state.
3. For a server S_i in the Active Non Contending state, it is required that $V_i > t$ at any instant t. If this is not so (either immediately upon transiting into this state or because time has elapsed but V_i does not change for servers in the Active Non Contending state), then the server enters the Inactive state.
4. If a new job J_i^j arrives while server S_i is in the Active Non Contending state, then the deadline parameter D_i is updated as follows:

$$D_i \longleftarrow V_i + P_i$$

   and server S_i returns to the Active Contending state.
5. There is one additional possible state change—if the processor is ever idle, then *all* servers in the system return to their Inactive state.

Algorithm GRUB maintains a global variable *total system utilization* that, at every instant, is equal to

$$U = \sum_{\substack{i=1 \\ S_i \neq \text{Inactive}}}^{n} U_i,$$

where n is the number of servers in the system. This variable is initialized to 0 and it is updated every time a server enters in or exits from state Inactive. In particular, when S_i exits from state Inactive, U is increased to U_i, whereas, when S_i enters state Inactive, it is decreased by U_i.

The rule for updating the virtual time of every server is as follows:

$$\frac{d}{dt}V_i = \begin{cases} \frac{U}{U_i} & \text{if (a job of) } S_i \text{ is executing,} \\ 0 & \text{otherwise.} \end{cases} \quad (2)$$

The rate of increase of the virtual time is proportional to the current total bandwidth of the active servers and is automatically adjusted depending on the current system load.

Observations. The virtual time is an important variable as it gives a measure of the progress that the server task has done. Let us make a simple example to explain the way the algorithm updates the virtual time.

Consider a server S_1 with bandwidth $U_1 = 0.25$ and period $P_1 = 20$ msec. If the system is fully utilized (i.e., the total system bandwidth U is equal to 1), then (2) tells us that the virtual time is increased at a rate of $1/0.25 = 4$. By looking at the algorithm rules, we see that the server executes for $P_1/4 = 5$ msec before the server deadline is postponed.

In general, the bandwidth U_1 can be computed using some rule of thumb or by performing a careful analysis of the application code. For our purposes, in this example, we assume that 5 msec are enough to complete task's jobs in most cases.

However, suppose that, for some interval of time, the total system utilization U goes down to 0.75. Then, server S_1 can execute more than 5 msec every period because we can reclaim the spare bandwidth. According to (2), the virtual time is increased at a rate of $0.75/0.25 = 3$. Therefore, if U is equal to 0.75 for the entire duration of the period P_1, server S_1 can execute for up to $P_1/3 = 6.66$ msec within the period.

Thus, if our task sometimes requires more than 5 msec to complete, it can take advantage of the reclaimed bandwidth and still execute inside the period boundary. This property can help us in setting the server bandwidth U_1 to a lower

value. For example, we can decide to set U_1 equal to the average bandwidth required by the task and try to exploit the reclamation property of GRUB to dynamically get more bandwidth.

4.2 Performance Guarantees

The following theorems formally state the performance guarantee that can be made by algorithm GRUB vis à vis the behavior of each server when executing on a dedicated processor. For proofs of the following theorems, see [13], [16].

Theorem 1. *Given a set of servers $S_1, \ldots, S_n$, with $\sum_{i=1}^{n} U_i \leq 1$, then all servers execute within their deadlines, regardless of the served tasks. More formally, at each instant t, $\forall i = 1, \ldots, n$ $D_i \geq t$.*

Theorem 2. *Suppose that job J_i^j would begin execution at time-instant A_i^j, if all jobs of server S_i were executed on a dedicated processor of capacity U_i. In such a dedicated processor, J_i^j would complete at time instant $F_i^j \stackrel{def}{=} A_i^j + (e_i^j/U_i)$, where e_i^j denotes the execution requirement of J_i^j. If J_i^j completes execution by time-instant f_i^j when our global scheduler is used, then it is guaranteed that*

$$f_i^j \leq A_i^j + \left\lceil \frac{(e_i^j/U_i)}{P_i} \right\rceil \cdot P_i. \tag{3}$$

From the previous inequality, it follows that $f_i^k < F_i^k + P_i$. Thus, the period P_i represents the *granularity* of the time from the point of view of the server: By using algorithm GRUB, every job finishes at most P_i time units later than the completion time on a dedicated slower processor.

Moreover, the GRUB algorithm is able to serve hard real-time periodic tasks without any deadline miss, as stated by the following theorem:

Theorem 3 ([16]). *Let τ_i be a hard real-time periodic task with worst-case execution time C_i and period T_i. If task τ_i is assigned a server S_i with bandwidth $U_i \geq \frac{C_i}{T_i}$ and period $P_i = T_i$, then no deadline of τ_i will be missed.*

5 Power-Aware Scheduling

We now modify GRUB for power-aware scheduling. The new resulting algorithm is called GRUB-PA (Power-Aware). As the first step, let us assume that the processor speed can be varied continuously, from a maximum speed factor of 1 (i.e., the processor works at its maximum speed) to a minimum of 0 (i.e., processor halted), and that the time to change speed is negligible. We will relax these simplifying assumptions in Sections 5.3 and 5.4.

As explained previously, GRUB maintains a global variable U that is the sum of the bandwidths of all servers that are not in the Inactive state. The key idea is that, if we set the speed factor of the processor to be equal to U, no server will miss its deadline. This idea is similar to the one on which the DVSST algorithm [19] is based. However, GRUB-PA updates the variable U in a more effective way, allowing additional power saving also in the case of periodic tasks, as shown in Section 5.1.

It is important to note that we are implicitly assuming that the execution time of a task varies linearly with the processor frequency. In Section 7, we will validate this assumption.

The original GRUB algorithm can be divided into two different parts: a set of rules for identifying the spare bandwidth $(1 - U)$ and a set of rules for reassigning the spare bandwidth. The second part can be adapted for power-aware scheduling. In practice, if the processor is not fully utilized $(U < 1)$, the exceeding bandwidth $(1 - U)$ can be used in two ways:

1. To execute the active servers for a longer time so that they can execute faster and finish earlier. This is the "reclamation" property and it is the original goal for which the GRUB algorithm was designed.
2. To slow down the processor. Each active server will execute for a longer time, but at a slower speed. The net effect is that its performance is not degraded.

The reclamation rule in GRUB is given by (2), thus the increment in the virtual time depends on the amount of bandwidth actually used in the system. This rule can also be used in the power-aware part to automatically adapt the server bandwidth to the new frequency. Moreover, we need an additional rule that sets the processor speed equal to U whenever a server goes in (or leaves) the Inactive state.

Hence, in the new GRUB-PA algorithm, state transitions 1 and 3 (see Fig. 1) are modified as follows:

1. When a job J_i^j arrives at time instant a_i^j, update the following variables:

$$V_i \leftarrow a_i^j,$$
$$D_i \leftarrow V_i + P_i,$$
$$U \leftarrow U + U_i.$$

   Moreover, the processor speed is set equal to U.
2. When a server is in the Active Non Contending state and $V_i = t$, then the server goes into the Inactive state and the system utilization is updated as follows:

$$U \leftarrow U - U_i.$$

   Moreover, the processor speed is set equal to U.

5.1 Example

In this section, we present a complete example showing how the GRUB-PA algorithm updates the processor speed depending on the bandwidth of the active servers. Consider a system consisting of two tasks. Task τ_1 is a sporadic task with minimum interarrival time $T_1 = 8$ and computation time C_1 varying between 2 and 4. This task is assigned a server with $U_1 = 0.5$ and $P_1 = 8$. The second task, τ_2, is a periodic task with period $T_2 = 10$ and constant execution time $C_2 = 5$. τ_2 is assigned a server with $U_2 = 0.5$ and $P_2 = 10$.

Suppose that the first job of task τ_1 arrives at time $t = 0$ requesting two units of computation time; the second job of τ_1 arrives at time $t = 12$ with computation time equal to 3. The resulting schedule is shown in Fig. 2. The upward arrows denote an arrival time, while the downward arrows denote a deadline. The plot under the schedule reports the variations of variable U during system evolution. In this case, we assume that deadline ties are broken in favor of the task with a lower index. However, in general, ties can be broken arbitrarily.

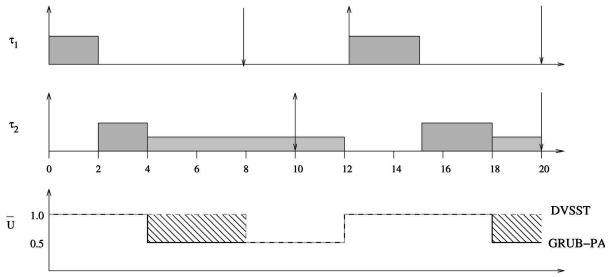


Fig. 2. Example of schedule produced by GRUB-PA.

Initially, all servers are active, so $U = 1$ and the processor speed $\overline{U}$ is set equal to 1. At time $t = 0$, task τ_1 is selected to execute since the deadline of the server $D_1 = 8$ is the earliest server deadline. The task executes until $t = 2$, when it completes. At this time, the virtual time is $V_1 = 2/U_1 = 4$, so the server goes into the Active Non Contending state. Then, task τ_2 starts executing and it executes for two time units until $t = 4$. At this time, the first server changes state from Active Non Contending to Inactive: The total bandwidth of all active servers is decreased to $U = U - U_1 = 0.5$, so the processor speed can be slowed down to $\overline{U} = 0.5$. Then, task τ_2 can continue executing at half the speed. However, its virtual time, V_2, is also increased at half the speed: For each unit of execution, the virtual time will now increase at a rate of $dV_2 = dt\frac{U}{U_2} = dt$. Therefore, task τ_2 can now execute for six units of time, which corresponds to three more units of the execution time at maximum speed, and complete just by the deadline at 10. However, at time $t = 10$, another job of task τ_2 arrives, so the second server remains in the Active Contending state and τ_2 resumes execution at half the speed.

At time $t = 12$, the second job of τ_1 is activated. The server becomes Active Contending and $U = U + U_1 = 1$. Therefore, the processor speed is again raised to $\overline{U} = 1$ and task τ_1 can start executing (as it is the one with the earliest server's deadline).

Notice that the mechanism used by the GRUB-PA algorithm is very similar to the one used by the DVSST algorithm [19]: They both use variable U to set the processor speed. However, there is a difference in the instant *when* the variable is updated. The DVSST algorithm does not keep track of the actual execution time of the tasks. Therefore, it can only subtract the bandwidth of a completed task *at the task's deadline*. In the example above, even if task τ_1 completes by time $t = 2$, the DVSST algorithm must wait until time $t = 8$ to lower the processor speed. Instead, algorithm GRUB-PA can anticipate this time at $t = 4$ as it explicitly takes into account the fact that task τ_1 has executed less than expected. The difference between the speeds set by the two algorithms is shown in Fig. 2. The GRUB-PA algorithm always anticipates this time with respect to algorithm DVSST, resulting in a larger amount of saved energy.

5.2 Properties of GRUB-PA

In this section, we formally prove that Theorem 1 is valid for the GRUB-PA algorithm as well. First, we define an ideal algorithm GPS-PA (Generalized Processor Sharing-Power Aware) that allocates the processor in proportion to the bandwidths U_i of the active tasks.

Definition 1. *Algorithm GPS-PA is a fluid algorithm that adjusts the processor frequency and allocates the processor to tasks according to the following rules:*

- *The processor speed is set equal to the sum of the utilization of all active tasks.*
- *For every interval Δt, the processor is allocated to all active tasks in proportion to their utilization.*

Clearly, GPS-PA is an ideal algorithm and cannot be implemented in practice since it is impossible to allocate any infinitesimally small interval Δt to different tasks in proportion to their utilizations. GPS-PA will be used only as a reference algorithm for GRUB-PA.

GPS-PA has the following interesting properties:

Lemma 4. *Under algorithm GPS-PA, under the constraint that the sum of the utilization of all tasks is upper bounded by 1, all jobs will complete exactly at time $F_i^j = max(A_i^j, F_i^{j-1}) + \frac{e_i^j}{U_i}$.*

Proof. At all times, GPS-PA sets the processor speed equal to the sum of the bandwidths of all active tasks U. Thus, the processor is allocated to each task in proportion to its bandwidth. The rate of execution of task τ_i is constant and equal to

$$R_i = \frac{U \cdot U_i}{U} = U_i.$$

Therefore, the finishing time F_i^j of job J_i^j does not depend on the presence of other tasks in the system and each task executes as if it were on a slower dedicated processor of constant speed U_i. The finishing time of the first job of task τ_i is $F_i^0 = \frac{e_i^0}{U_i}$. Any successive job of τ_i starts at the latest time between the finishing time of the previous job and its arrival time. Hence, the lemma is proved. □

Now, we divide every job in one or more subjobs, each one of maximum length $U_i \cdot P_i$.

- A job J_i^j with $e_i^j \leq U_i \cdot P_i$ is transformed into a subjob $J_i^j(1)$ with the same execution time, the same arrival time, and deadline $D_i^j(1) = a_i^j + P_i$.
- A job J_i^j with $e_i^j > U_i \cdot P_i$ is divided into $K = \lceil \frac{e_i^j}{U_i P_i} \rceil$ subjobs $J_i^j(1), \ldots, J_i^j(K)$. All subjobs have execution time $e_i^j(k) = U_i \cdot P_i$, except the last one, which can be shorter. Each one of these subjobs is assigned an arrival time and a deadline: The first subjob is assigned an arrival time equal to the arrival time of the original job and a deadline $D_i^j(1) = a_i^j + P_i$. The following ones are assigned arrival times and deadlines as follows:

$$a_i^j(k) = D_i^j(k-1) \qquad D_i^j(k) = a_i^j(k) + P_i.$$

Corollary 5. *For each subjob, $F_i^j(k) \leq D_i^j(k)$.*

Proof. This splitting operation does not influence the behavior of algorithm GPS-PA. Therefore, the corollary trivially follows from Lemma 4. □

At this point, we will show that the schedule generated by algorithm GPS-PA can be "transformed" into the

schedule generated by GRUB-PA maintaining certain important properties. The transformation is done by following a well-known technique described by Coffman and Denning [36, Chapter 3]. First we transform the schedule generated by GPS-PA into a nonfluid schedule. Then, we transform this second schedule into the schedule generated by GRUB-PA.

Definition 2 (Job Transformation). *Let $\sigma_f(t)$ be the schedule generated by GPS-PA. It is a function with multiple values: For every time t, $\sigma_f(t)$ is the set of executing subjobs that coincides with the set of active subjobs.*

Now, we generate a function $\sigma_I(t)$ in the following way: For every t, let $[t_1, t_2]$ be a maximal interval containing t in which $\sigma_f(t)$ is constant and no subjob completes in (t_1, t_2).[1] It follows that either a subjob completes in t_2 or a subjob is activated in t_2. Let x be the number of jobs active in (t_1, t_2).

Then, we divide interval $[t_1, t_2]$ into x subintervals, one for each active subjob $J_i^j(k)$, each one of length $(t_2 - t_1) \cdot U_i$. Then, function $\sigma_I(t)$ assumes value $J_i^j(k)$ in the corresponding subinterval.

Moreover, in the new schedule, the processor frequency is changed at the same instants as in schedule $\sigma_f(t)$.

By construction, the finishing times of any subjob in $\sigma_I(t)$ is not greater than the finishing times of the same subjob in $\sigma_f(t)$. Therefore, the following corollary is trivially proven.

Corollary 6. *No subjob misses its deadline in schedule $\sigma_I(t)$.*

Lemma 7. *GRUB-PA changes frequency at the same instants of time as GPS-PA.*

Proof. In GPS-PA, frequency is updated at the arrival times of the jobs A_i^j or at the finishing times F_i^j. At each instant t, the virtual time $V_i(t)$ in GRUB-PA corresponds to the instant of time in the GPS-PA in which the task has received the same amount of service as in the GRUB-PA.

GRUB-PA changes the U (and possibly the frequency) when the task arrives (i.e., $V(t) = t$) or when the task goes into the Inactive state (again, $V(t) = t$). In the first case, we have $V(t) = A_i^j = t$. In the second case, we have $V(t) = t = F_i^j$. Hence, the lemma is proven. □

Finally, the last step of our demonstration is to transform the schedule $\sigma_I(t)$ into the schedule generated by GRUB-PA.

Theorem 8. *Server S_i never misses its deadline. In other words, at any instant t, the server deadline is always greater than t.*

Proof. We use a well-known technique by Dertouzos [20], originally used for proving the optimality of EDF. Given a feasible schedule $\sigma_I(t)$ as obtained by the technique described in Definition 2, by the optimality of EDF, with an exchange procedure, we can obtain a feasible schedule $\sigma(t)$ in which the subjobs are scheduled in EDF order. Notice that the deadlines of the subjobs are equal to the deadlines of the servers as assigned by GRUB-PA. Therefore, the schedule is the same as obtained by GRUB-PA as GRUB essentially performs EDF on the subjobs.

Since $\sigma(t)$ is feasible, the theorem follows. □

1. As usual, symbols [] denote a closed time interval and symbols () denote an open time interval.

5.3 Processor Model

No existing processor can vary its frequency with continuity. All processors that support DVS provide a discrete set of frequencies [4], [5], [6], [7]. Correspondingly, we can set some "thresholds" on the values of the total system bandwidth. Suppose that the processor supports M different frequencies $\phi_1, \ldots, \phi_M$. We can compute $\overline{U}_1, \ldots, \overline{U}_M$ different values of the bandwidth. If $U(t)$ is comprised in $(\overline{U}_k, \overline{U}_{k+1}]$ for some k, then the processor frequency is set equal to ϕ_{k+1}.

It is easy to see that, by using this simple approach, the properties of the GRUB-PA algorithm continue to hold. In fact, the actual speed of the processor is always set to a value $\overline{U}_{k+1}$ greater than or equal to the theoretical desired bandwidth U.

Unfortunately, the net effect of this approach is that some energy is wasted as the desired frequency is always approximated by a higher frequency. One possibility would be to alternate the two frequencies, ϕ_k and ϕ_{k+1}, so that the average utilization is equal to the desired utilization. This idea has been recently proposed by Bini et al. [37] in the context of static DVS. Their methodology consists of computing the minimum theoretical processor speed that, if constantly applied to the system, makes the task set schedulable. Then, if the corresponding frequency is not available in the set of processor frequencies, the methodology selects two available frequencies that will be alternated in a *duty cycle*.

Applying such a methodology to our GRUB-PA algorithm is not trivial. In GRUB-PA, the system dynamically varies the value of U at instants of times that cannot be predicted a priori. In particular, it is not possible to know how long the system will maintain a certain value of U. Therefore, only a clairvoyant algorithm can find the *optimal* way of alternating the two frequencies ϕ_k and ϕ_{k+1}. We are currently investigating the possibility of finding a suboptimal algorithm for the above problem.

From a practical point of view, observe that the waste of energy is less evident as the number of available processor frequencies increases. Modern processors provide a large number of combinations voltage/frequencies and, therefore, the difference between desired frequency and actual frequency is often very little.

As a final consideration, GRUB-PA maintains the ability to reclaim spare capacity for soft real-time tasks. In fact, the difference $\overline{U}_{k+1} - U$ is automatically accounted for by the algorithm as spare bandwidth and reclaimed for the tasks that need to execute more than their assigned bandwidth.

5.4 Overhead

Since every processor needs some transitory time to adjust to a new frequency, it is important to avoid limit situations in which the processor keeps changing its frequency up and down because this would completely trash the system.

For example, suppose that a task with a very low bandwidth is activated and deactivated very often. If the total utilization is close to one threshold value, $\overline{U}_k$, every activation would cause an increase of the frequency and every deactivation would cause a decrease in the frequency.

To avoid these situations, when the total system bandwidth U goes over one of the thresholds $\overline{U}_k$, we

TABLE 1
Operating Parameters for the Intel PXA250 Processor

Frequency (MHz)	Voltage (V)	Normalized speed	Normalized power consumption
100 MHz	0.85	25%	11%
200 MHz	1	50%	30%
300 MHz	1.1	75%	54%
400 MHz	1.3	100%	100%

TABLE 2
Operating Parameters for the Transmeta TM5800 Processor

Frequency (MHz)	Voltage (V)	Normalized speed	Normalized power consumption
300 MHz	0.8	30%	11%
433 MHz	0.875	43%	20%
533 MHz	0.95	53%	28%
667 MHz	1.05	67%	44%
800 MHz	1.15	80%	63%
900 MHz	1.25	90%	83%
1000 MHz	1.3	100%	100%

immediately increase the processor frequency because we do not want to risk a hard task missing its deadline. When a decrease of the total system bandwidth U goes below one of the thresholds $\overline{U}_k$, instead, we do not change the frequency immediately, but we set a timer. If the timer expires and U is still below the threshold, we lower the frequency. If U goes above the threshold again, we cancel the timer. In this way, we limit the number of frequency switches.

We now explain how it is possible to account for the delay of frequency/voltage switching through a proper tuning of the system's parameters. Let δ be the maximum time it takes to switch frequency and let Δ be the timer expiration interval. We can have a maximum of two frequency switches every Δ, one to go down and another one to go up. In the worst case, this accounts for a bandwidth reduction of $\frac{2\delta}{\Delta}$. Therefore, we can admit new servers up to a total bandwidth of $1-\frac{2\delta}{\Delta}$ and set the processor speed to $U+\frac{2\delta}{\Delta}$.

As anticipated in the previous sections, we decided to not consider the energy spent during a frequency switch. In particular, we do not account for this energy overhead in the simulation model presented in the next section. Instead, the presence of this overhead has been automatically accounted for in our experimental results (see Section 7). In fact, the total energy consumed by our testbed also comprises the energy due to frequency changes.

6 Evaluation of the Algorithm

We evaluated our algorithm through comparisons with different power-aware algorithms proposed in the literature. We chose to compare our algorithm with the DRA algorithm, proposed by Aydin et al. [17], with the EDF version of the RTDVS algorithms, proposed by Pillai and Shin [18], and with the DVSST algorithm, proposed by Qadi et al. [19].

To compare the algorithms, we used a simulation environment called *RTSim* (which stands for "Real-Time system SIMulator") [38], [39]. It is a collection of programming libraries written in C++ for simulating and analyzing real-time control systems. In this tool, a simulation is a C++ program that must be linked to an appropriate library of components that includes schedulers, task models, etc. RTSim started as an academic project, and it has been used primarily for experimenting with new scheduling algorithms and solutions. For this reason, it contains, already implemented, many scheduling algorithms proposed in the literature. The tool is released as Open Source (under the GNU General Public License (GPL)) to give researchers a common simulation platform for comparing the performance of new scheduling algorithms. For our purposes, we extended the processor components of RTSim to include models of processors with varying speed. Moreover, we implemented the new power-aware schedulers.

We modeled the power consumption of both an Intel PXA250 [5] processor, using four different operating frequencies, and a Transmeta Crusoe TM5800 [4] processor, using seven operating frequencies. Tables 1 and 2 show the operating parameters for these models of processors. The values of power consumption have been obtained according to (1).

The power consumption model chosen in the simulation is very simple but effective. In fact, we are not interested in accurate simulations of the real consumed power, but, rather, in a comparative analysis among different algorithms.

6.1 Comparison with DRA and RTDVS

The DRA algorithm permits us to schedule periodic tasks in a hard real-time environment, reducing the energy consumption without missing any deadline. In particular, the DRA scheme consists of a basic algorithm and of two extensions.

The basic algorithm (DRA-Standard) uses a queue of tasks (called α-queue) ordered by earliest deadline. The queue is used to compute the earliness of tasks when they are dispatched. At any time, it contains information about tasks that would be active (i.e., running or ready) at that time in the canonical schedule S^{can}, which is the static optimal schedule in which every instance presents its worst-case workload and the processor runs at the constant speed $\overline{S} = max\{S_{min}, U_{tot}\}$. At time t, this queue contains information about all instances T_i^j such that $r_i^j \leq t \leq d_i^j$ and whose remaining execution time is greater than 0. At dispatch time, the algorithm computes the earliness of tasks and adjusts the processor speed according to this value.

The *"One Task"* extension (DRA-OTE) further slows down the processor speed when there is only one task in the ready queue and its worst-case execution time (under the current speed) does not extend beyond the next event.

The *"Aggressive"* extension (DRA-AGGR) speculatively assumes that current and future instances of tasks will most probably present a computational demand lower than the worst case. Hence, it tries to reduce the speed of the running task to a level even lower than the one suggested by DRA-OTE. However, when the worst-case scenario happens, this

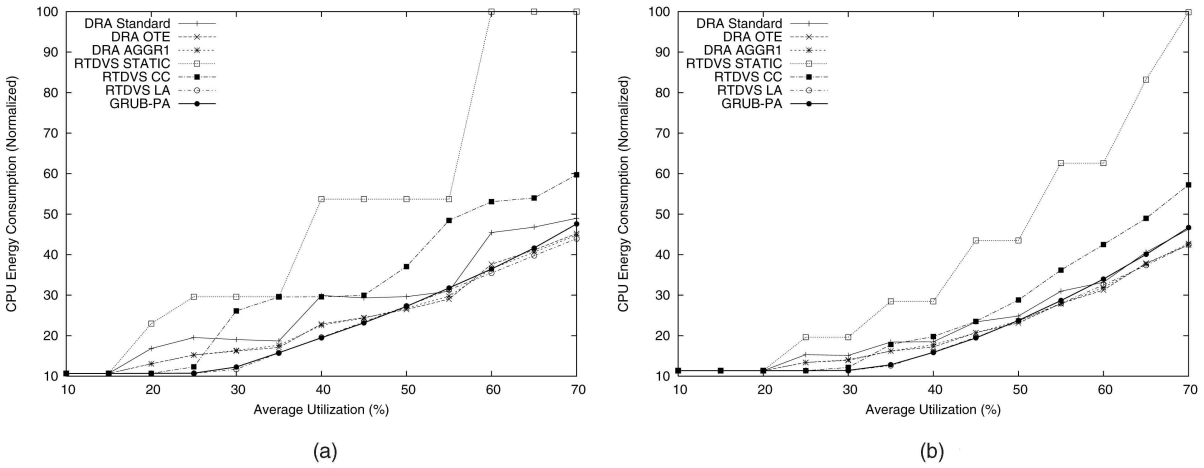


Fig. 3. Energy consumption with WCET/BCET ratio equal to 2 (a) on a PXA250 and (b) on a TM5800.

algorithm has to increase the processor speed later to guarantee the feasibility of future tasks.

We compared GRUB-PA with the DRA algorithm and with both its extensions (for the aggressive one, we chose the AGGR1 policy described in [17]).

We also compared our algorithm with the algorithms proposed by Pillai and Shin [18]. They proposed three different algorithms for hard periodic real-time tasks. The first one is a static (offline) algorithm which selects the lowest possible operating frequency that allows us to meet all the deadlines for the given task set. In the rest of the paper, we will refer to this static algorithm using the name RTDVS-Static. The second algorithm (RTDVS-Cycle Conserving) assumes the worst case at release time and executes at a high frequency until the task completes and only then reduces operating frequency and voltage. This algorithm may need to dynamically reduce frequency on each task completion and increase frequency on each task release. The last algorithm (RTDVS-Look Ahead) tries to defer as much work as possible and sets the operating frequency to meet the minimum work that must be done now to ensure that all future deadlines will be met. This may require running at higher frequencies later to complete all the deferred work in time. However, if tasks tend to use much less than their worst-case execution times, the peak execution rates for deferred work may never be needed. These algorithms have been proposed for both Rate Monotonic and Earliest Deadline First schedulers. For the comparison, we chose the EDF versions since they are more similar to the GRUB-PA algorithm.

To compare the algorithms, we performed two different kinds of simulations. We followed the same methodology as Aydin et al. [17]. Let WCET and BCET indicate the worst-case and the best-case execution times, respectively. In the simulations with GRUB-PA, we generated a server for each task, with P_i equal to the task period and U_i equal to the ratio WCET/period so that, according to Theorem 3, no task ever misses its deadline.

In the first set of simulations, we fixed a constant WCET/BCET ratio for each task while using different values for the average workload. For each value of the workload, we simulated 100 different task sets, each one consisting of 15 different periodic tasks with randomly generated periods. The results are shown in Fig. 3 for a WCET/BCET ratio equal to 2 and in Fig. 4 for a ratio equal to 4. The simulations have been performed for both the PXA250 (Fig. 3a and 4a) and the TM5800 (Fig. 3b and 4b) processors.

The second test measured the amount of power consumption with a constant average workload (50 percent) and a variable WCET/BCET ratio. For each value of the WCET/BCET ratio, we ran 100 simulations using different task sets, again with each set consisting of 15 tasks. Since the convexity of the power/speed curve suggests using a uniform speed to obtain a lower power consumption [33], we expected to see a greater energy saving using a WCET/BCET ratio close to 1 (that is, a small variation in the execution times). Our results (see Fig. 5a and Fig. 5b) confirm this assumption.

The confidence intervals obtained during the simulations have not been shown since they were very small (in all simulations, all algorithms presented a 99 percent confidence interval less than one unit of the normalized value of the energy consumption).

From all simulations, it is possible to conclude that GRUB-PA shows better performance than most of the algorithms. Among all algorithms, RTDVS-Look Ahead, proposed by Pillai and Shin [18], presented the lowest average power consumption. However, GRUB-PA has very similar performance compared to RTDVS-Look Ahead. Moreover, it is important to point out that, unlike DRA and RTDVS, GRUB-PA does not assume a hard real-time periodic task model, and it can be applied to both hard and soft, periodic, sporadic, or even aperiodic tasks.

6.2 Comparison with DVSST

We also compared GRUB-PA against the DVSST algorithm proposed by Qadi et al. [19] since it is an algorithm that assumes a sporadic task model.

In each simulation run, we generated eight sporadic tasks with minimum interarrival times T_i randomly chosen

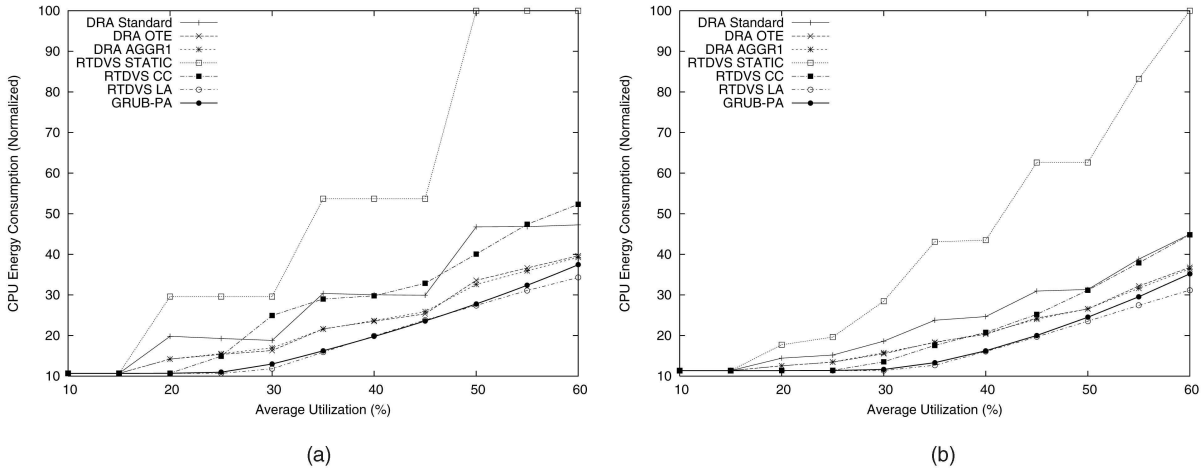


Fig. 4. Energy consumption with WCET/BCET ratio equal to 4 (a) on a PXA250 and (b) on a TM5800.

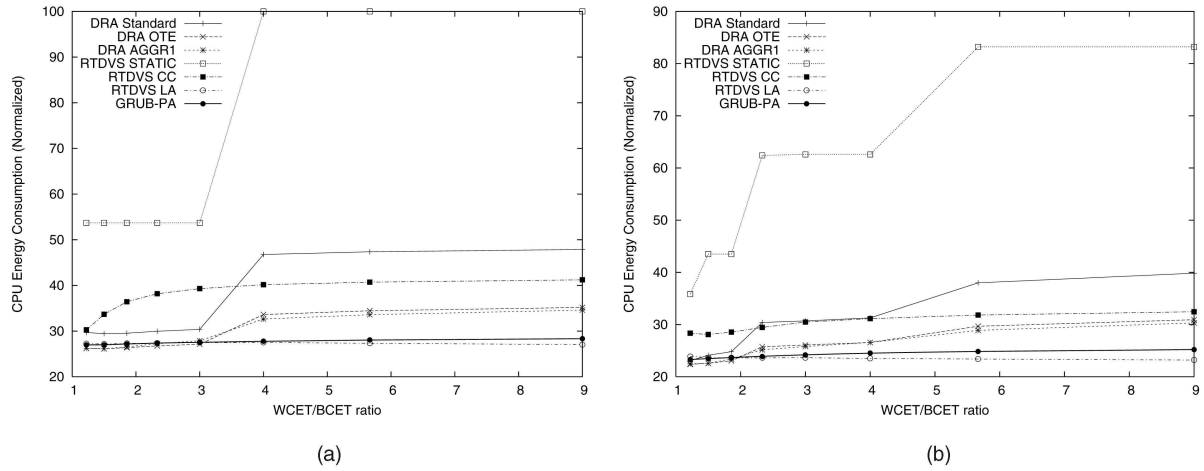


Fig. 5. Energy consumption with constant average workload (a) on a PXA250 and (b) on a TM5800.

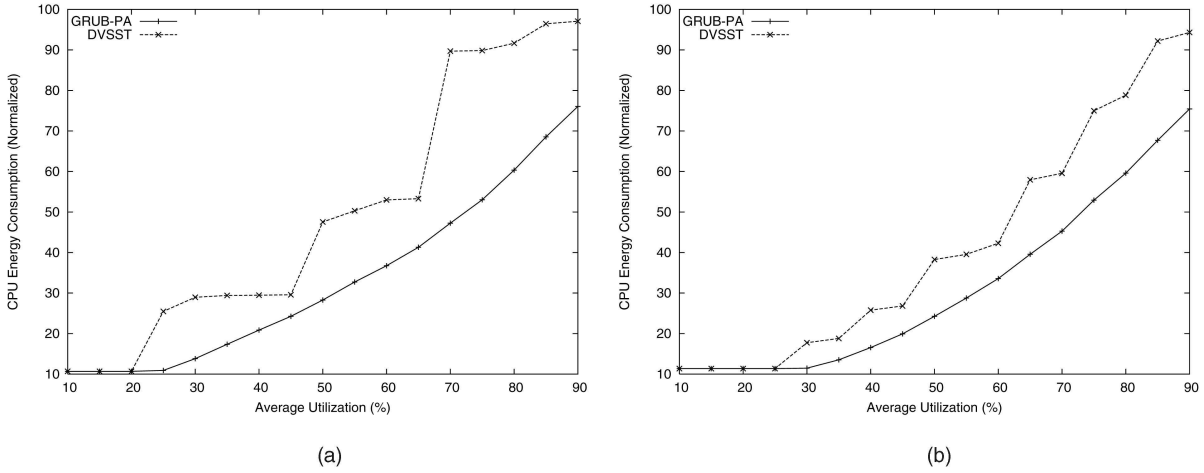


Fig. 6. Energy consumption with constant average workload varying between 0.1 and 0.9 (a) on a PXA250 and (b) on a TM5800.

between 1,000 and 10,000 and with actual interarrival time uniformly distributed between T_i and $T_i * 1.1$. Each task has a variable computation time, with a 20 percent of variation over the central value. In each experiment, the sum U_{max} of the maximum bandwidth requested by all tasks is constant. Finally, U_{max} is varied between 10 percent and 90 percent. The results for the PXA250 and the TM5800 processors are shown in Fig. 6.

As is possible to see, the DVSST algorithm is much more sensitive to the discretization of frequencies with respect to GRUB-PA due to the lack of reclamation of early tasks' completions. The irregularity of the pattern for DVSST decreases as the number of available discrete frequencies increases as can be noticed by comparing Fig. 6a with Fig. 6b. As expected, GRUB-PA presents an improvement up to 40 percent with respect to DVSST.

7 Implementation and Experimental Results

The implementation of the GRUB-PA algorithm in the Linux operating system has been done in the context of the OCERA project (IST-35102), funded by the European Commission in the fifth framework program. The main objective of this project was the design and implementation of a library of free software components for embedded real-time systems. These components have been used to create flexible (supporting a wide variety of applications), configurable (scalable from a small to a fully featured system), robust (fault-tolerant and with high performance), and portable (adaptable to several hardware and software configurations) systems.

We modified the scheduling policy of Linux 2.4.18. Since we wanted to limit, as much as possible, the modifications to the standard Linux scheduler, we decided to apply a small patch (called *Generic Scheduler Patch*) that exports the necessary kernel events. Then, we implemented our scheduler as a loadable kernel module.

Our scheduler needs to "intercept" the job arrival (i.e., tasks that are unblocked) and the job finishing (i.e., tasks that are blocked). Moreover, the scheduler must know when tasks are created and when tasks terminate.

We decided to export an interface to the scheduler through the standard `sched_setscheduler()` system call, adding a new scheduling policy, and extending the structure `sched_param`.

Moreover, the Generic Scheduler Patch exports the following *hooks* that can be used to intercept the interesting scheduling events:

- *block_hook* is invoked when a task is blocked such that the scheduler understands that the current job has finished.
- *unblock_hook* is invoked when a task is unblocked such that the scheduler is informed of the arrival of a new job.
- *fork_hook* is invoked when a new task is created by a `fork()` and a pointer to the task is passed as parameter.
- *cleanup_hook* is invoked when a task is terminated, such that the scheduler can free the internal resources.
- *setsched_hook* is invoked when the system calls `sched_setscheduler()` or `sched_setparam()` are called by the user.

All the hooks, except `setsched_hook`, have a parameter that is a pointer to the structure `task_struct` of the corresponding task.

The patch inserts a new field called `private_data` in the `task_struct` of type `void*`. It is a pointer used by our scheduler to access the private real-time data of every task. In our case, it is a pointer to the server that handles the task. If necessary, the scheduler must set this field to the appropriate data structure during the `fork_hook`. When the module is removed, it must ensure that all tasks have their `private_data` set to `NULL`.

Our dynamically loadable scheduler modifies the task priority, raising the selected task to the maximum priority, and then calls the standard Linux scheduler. Based on the information received by the hooks, our scheduler selects which task has to be executed and sets its policy to `SCHED_FIFO` or `SCHED_RR` and the `rt_priority` to the maximum real-time priority + 1. Then, it invokes the Linux scheduler. In practice, the Linux scheduler acts as a dispatcher for our scheduler. Thus, the modifications to the standard Linux scheduler are minimal.

Notice that, in this implementation, the scheduling algorithm does not assume any periodic behavior of the task. As a matter of fact, the scheduler only intercepts the blocking/unblocking events of a task and it is the task's responsibility to implement a periodic behavior, if required. Thus, our scheduler is able to serve any kind of task, from nonperiodic legacy Linux processes to periodic soft real-time tasks. An in-depth description of the implementation can be found in [40].

We tested GRUB-PA on an Intrinsyc CerfCube 250 architecture. It consists of 32 MB Flash ROM, 64 MB SDRAM, and a Ethernet 10/100 Mbps. The processor is an Intel PXA250 [5]. It is a superpipelined 32 bits RISC processor based on the Intel *Xscale* microarchitecture. This processor permits an on-the-fly switch of the clock frequency and a sophisticated power consumption management. We configured the system to support three different frequencies—i.e., 100 MHz, 200 MHz, and 400 MHz. By using these three levels, we were able to use the minimum possible frequency (100 MHz) and the maximum one (400 MHz). Therefore, we had two thresholds, $U_{th1} = 1/4$ and $U_{th2} = 1/2$.

Our study was particularly focused on multimedia applications. Therefore, we decided to evaluate the performance of our system using a multimedia application. However, our approach can be used for a large range of different applications because it is completely transparent to the application characteristics. Unfortunately, our testbed system, the Intrinsyc CerfCube, does not present a video output. Hence, we decided to focus our attention on an audio decoder.

One may argue that varying the processor frequency only, without touching the peripherals frequencies (like memory, for example) does not bring appreciable advantages. We performed some experiments showing that, in the considered testbed, this is not the case. To execute the first test, we decompressed a set of audio streams at 44100 Hz and two channels, measuring the time necessary to decompress every stream under different fixed clock frequencies.

From the obtained values, we extracted how much the speed of decompression is related to the speed of the processor. The result is shown in Fig. 7, where we show, on the x-axis, the frequency of the processor and, on the y-axis,

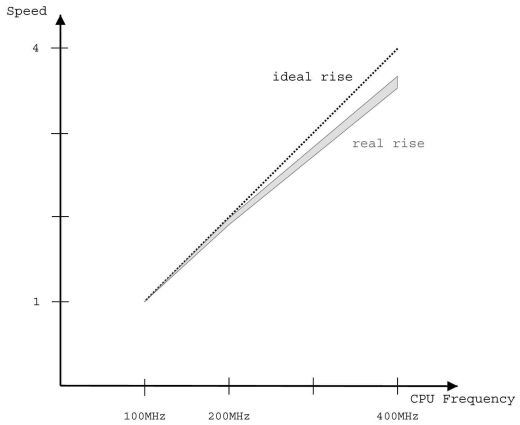


Fig. 7. Decompression speed related to processor speed (99 percent confidence interval).

the decompression speed. As the reader can see, the relationship is almost linear. This justifies our assumption that, by doubling the processor frequency, the computation time of one task's job halves. In Fig. 7, we also show the 99 percent confidence interval.

Then, we evaluated the power consumed by our system under different conditions, with and without GRUB-PA. We inserted a dedicated electronic circuit between the CerfCube board and the power supply to measure the input current to the board. The circuit is powered by a separate 9V battery: It puts a very small resistor in series with the CerfCube board and measures the voltage at the ends of the resistor. The resulting data are sampled and sent through a serial link to a PC that collects the data.

By using our algorithm, we measured the temporal evolution of the current under different workloads. We computed the average values of the input current, reported in Table 3.

We did many experiments using the multimedia application and we observed that, using our frequency scaling mechanism, we saved up to 38.4 percent of the total power consumed by the system. Notice that this is the percentage of energy saved with respect to the total energy consumed by the board, although our algorithm acts only on the processor voltage and frequency.

TABLE 3
Average Values of the Input Current

Processor frequency	Current
100 MHz	446.0 mA
200 MHz	508.5 mA
400 MHz	579.9 mA

8 Conclusions

In this paper, we presented the GRUB-PA algorithm, a novel power-aware scheduling algorithm suitable for systems consisting of hard periodic and soft aperiodic real-time tasks. The algorithm is based on the resource reservation framework, so it does not make any restrictive assumption on the characteristics of the tasks.

Our simulations show that GRUB-PA, besides giving guarantees about the temporal execution of tasks, presents performance similar to those provided by other power-aware scheduling algorithms presented in the literature. However, GRUB-PA can also be applied to hard and soft, and periodic, sporadic, or even aperiodic tasks.

Moreover, we presented an implementation of the GRUB-PA in the Linux operating system. The experimental results on a real testbed system show that, by using GRUB-PA, we save up to 38.4 percent of the total power consumed by the system with respect to the unmodified one.

Acknowledgments

The authors would like to thank Franco Zaccone for his precious help in setting up the experimental system and for building the electronic circuit for measuring the input current. Special thanks to Hakan Aydin for helping the authors in the development of the code for the DRA algorithm in the RTSim environment and to Luca Abeni for his precious work on the Linux scheduler. This work was supported in part by the European Commission under the OCERA IST project (IST-35102).

References

[1] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T.W. Keller, "Energy Management for Commercial Servers," *Computer,* vol. 36, no. 12, pp. 39-48, Dec. 2003.

[2] P. Bohrer, E.N. Elnozahy, T. Keller, M. Kistler, C. Lefurgy, C. McDowell, and R. Rajamony, *The Case for Power Management in Web Servers.* Kluwer Academic, 2002.

[3] J. Pouwelse, K. Langendoen, and H. Sips, "Dynamic Voltage Scaling on a Low-Power Microprocessor," *Proc. Seventh ACM Int'l Conf. Mobile Computing and Networking (Mobicom),* 2001.

[4] *Crusoe Processor Model TM5800 Version 2.1 Data Book Revision 2.01,* Transmeta Corp., http://www.transmeta.com, June 2003.

[5] *Intel PXA250 and PXA210 Application Processors Developer's Manual,* Intel Corp., Feb. 2002.

[6] *Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor,* Intel Corp., Mar. 2004.

[7] *Intel PXA27x Processor Family Power Requirements,* Intel Corp., 2004.

[8] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems," *Proc. SPIE/ACM Conf. Multimedia Computing and Networking,* Jan. 1998.

[9] J. Lehoczky, L. Sha, and J. Strosnider, "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments," *Proc. IEEE Real-Time Systems Symp.,* Dec. 1987.

[10] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic Task Scheduling for Hard-Real-Time Systems," *J. Real-Time Systems,* vol. 1, July 1989.

[11] M. Spuri and G. Buttazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems," *J. Real-Time Systems,* vol. 10, no. 2, 1996.

[12] L. Abeni and G. Buttazzo, "Integrating Multimedia Applications in Hard Real-Time Systems," *Proc. 19th IEEE Real-Time Systems Symp.,* Dec. 1998.

[13] G. Lipari and S. Baruah, "Greedy Reclamation of Unused Bandwidth in Constant Bandwidth Servers," *Proc. IEEE 12th Euromicro Conf. Real-Time Systems,* June 2000.

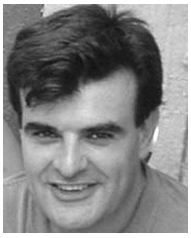
[14] M. Caccamo, G. Buttazzo, and L. Sha, "Capacity Sharing for Overrun Control," *Proc. IEEE Real-Time Systems Symp.,* Dec. 2000.

[15] C. Scordino and G. Lipari, "Using Resource Reservation Techniques for Power-Aware Scheduling," *Proc. Fourth ACM Int'l Conf. Embedded Software (EMSOFT),* pp. 16-25, Sept. 2004.

[16] G. Lipari, "Resource Reservation in Real-Time Systems," PhD dissertation, Scuola Superiore S.Anna, 2000.

[17] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez, "Power-Aware Scheduling for Periodic Real-Time Tasks," *IEEE Trans. Computers,* vol. 53, no. 5, pp. 584-600, May 2004.
[18] P. Pillai and K.G. Shin, "Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems," *Proc. 18th ACM Symp. Operating Systems Principles,* 2001.
[19] A. Qadi, S. Goddard, and S. Farritor, "A Dynamic Voltage Scaling Algorithm for Sporadic Tasks," *Proc. 24th Real-Time Systems Symp.,* pp. 52-62, 2003.
[20] M.L. Dertouzos, "Control Robotics: The Procedural Control of Physical Processes," *Information Processing,* 1974.
[21] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM,* vol. 20, no. 1, 1973.
[22] S. Saewong and R. Rajkumar, "Practical Voltage-Scaling for Fixed-Priority RT-Systems," *Proc. Ninth IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS),* May 2003.
[23] F. Yao, A. Demers, and S. Shenker, "A Scheduling Model for Reduced CPU Energy," *Proc. IEEE Ann. Foundations of Computer Science,* pp. 374-382, 1995.
[24] Y. Liu and A.K. Mok, "An Integrated Approach for Applying Dynamic Voltage Scaling to Hard Real-Time Systems," *Proc. Ninth IEEE Real-Time and Embedded Technology and Applications Symp.,* pp. 116-123, May 2003.
[25] D. Shin and J. Kim, "Dynamic Voltage Scaling of Periodic and Aperiodic Tasks in Priority-Driven Systems," *Proc. Asia and South Pacific Design Automation Conf. (ASP-DAC '04),* pp. 653-658, Jan. 2004.
[26] S. Saewong and R. Rajkumar, "Optimal Static Voltage-Scaling for Real-Time Systems," technical report, Real-Time and Multimedia Systems Laboratory, Carnegie Mellon Univ., Pittsburgh, Penn., 2002.
[27] C. Scordino and E. Bini, "Optimal Speed Assignment for Probabilistic Execution Times," *Proc. Second Workshop Power-Aware Real-Time Computing (PARC '05),* Sept. 2005.
[28] Y. Zhu and F. Mueller, "Feedback EDF Scheduling Exploiting Dynamic Voltage Scaling," *Proc. 10th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS '04),* May 2004.
[29] J. Pouwelse, K. Langendoen, and H. Sips, "Energy Priority Scheduling for Variable Voltage Processors," *Proc. Int'l Symp. Low Power Electronics and Design (ISLPED),* 2001.
[30] J.R. Lorch and A.J. Smith, "Improving Dynamic Voltage Scaling Algorithms with Pace," *Proc. ACM SIGMETRICS 2001 Conf.,* June 2001.
[31] P. Kumar and M. Srivastava, "Predictive Strategies for Low-Power RTOS Scheduling," *Proc. IEEE Int'l Conf. Computer Design: VLSI In Computers & Processors (ICCD '00),* Sept. 2000.
[32] J.K. Strosnider, J.P. Lehoczky, and L. Sha, "The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard-Real-Time Environments," *IEEE Trans. Computers,* vol. 4, no. 1, Jan. 1995.
[33] T. Ishihara and H. Yasuura, "Voltage Scheduling Problem for Dynamically Variable Voltage Processors," *Proc. Int'l Symp. Low Power Electronics and Design,* pp. 197-202, Aug. 1998.
[34] F. Gruian, "Energy-Centric Scheduling for Real-Time Systems," PhD dissertation, Dept. of Computer Science, Lund Inst. of Technology, Lund, Sweden, Nov. 2002.
[35] L. Abeni, "Server Mechanisms for Multimedia Applications," Technical Report RETIS TR98-01, Scuola Superiore S. Anna, 1998.
[36] J.E. Coffman and P.J. Denning, *Operating Systems Theory.* Prentice-Hall, 1973.
[37] E. Bini, G. Buttazzo, and G. Lipari, "Speed Modulation in Energy-Aware Real-Time Systems," *Proc. 17th Euromicro Conf. Real-Time Systems,* July 2005.
[38] L. Palopoli, G. Lipari, G. Lamastra, L. Abeni, B. Gabriele, and P. Ancilotti, "An Object Oriented Tool for Simulating Distributed Real-Time Control Systems," *Software: Practice and Experience,* 2002.
[39] Rtsim (real-time system simulator), available on Internet under the GNU General Public License (GPL), http://rtsim.sourceforge.net, 2006.
[40] C. Scordino and G. Lipari, "Energy Saving Scheduling for Embedded Real-Time Linux Applications," *Proc. Fifth Real-Time Linux Workshop,* 2003.

Claudio Scordino received the master's degree in computer engineering from the University of Pisa in 2003. During 2005, he was a visiting student at the University of Pittsburgh, collaborating with Professor Daniel Mossé on energy-aware real-time scheduling. He is currently a PhD student and teaching assistant at the University of Pisa and he collaborates with the Scuola Superiore Sant'Anna. His research activities include operating systems, real-time scheduling, energy saving, and embedded devices. He is a student member of the IEEE.



Giuseppe Lipari graduated in computer engineering from the University of Pisa in 1996 and received the PhD degree in computer engineering from the Scuola Superiore Sant'Anna in 2000. Currently, he is an associate professor of operating systems with the 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 committes of many conferences in the field. He is currently an associate editor of the *IEEE Transactions on Computers*. He is a member of the IEEE.