Using Resource Reservation Techniques for Power-Aware Scheduling*

Claudio Scordino
Computer Science Department
University of Pisa, Italy
scordino@di.unipi.it

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

ABSTRACT

Minimizing energy consumption is an important issue in the design of real-time embedded systems. As many embedded systems are powered by rechargeable batteries, the goal is to extend, as much as possible, the autonomy of the system.

Recently, many scheduling algorithms have been proposed in the literature to exploit the capability of some processor to dynamically change its operating voltage and frequency. The goal of the scheduling algorithm is to select not only the task to be scheduled, but also the operating frequency, so minimizing the energy consumed without jeopardizing the schedulability of the real-time tasks.

In this paper we present GRUB-PA, a new scheduling algorithm for power-aware systems. The algorithm can efficiently handle systems consisting of hard and soft real-time tasks. In addition, tasks can be periodic, sporadic or aperiodic. The algorithm reclaims the spare bandwidth caused by periodic tasks that execute less than expected or by sporadic tasks that arrive less frequently, and use this information to lower the processor frequency. We show the effectiveness of the GRUB-PA algorithm in scheduling hard and soft real-time tasks with a set of simulations. Finally, we present the implementation of GRUB-PA in the Linux OS.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-based systems**]: Real-time and embedded systems; D.4.1 [**Operating Systems**]: Process Management—*Scheduling*

General Terms

Algorithms

Keywords

DVS, real-time, resource-reservation, scheduling, power-aware

*This work has been supported in part by the European Commission under the OCERA IST project (IST-35102)

EMSOFT'04, September 27–29, 2004, Pisa, Italy.

1. INTRODUCTION

The problem of reducing the 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 as much as it is possible the autonomy of the system. Similar problems can be found in normal workstation PCs. As processors become more and more powerful, their power consumption increases correspondingly, and it becomes a problem to dissipate the heat produced by the processor.

To reduce energy consumption, one possible approach is to selectively change the processor voltage. This technique is called *dynamic voltage scheduling* (DVS). Many modern processors can dynamically lower the voltage to reduce the power consumption. However, by reducing the voltage, the gate delay increases. Thus, the processor must also lower the operating frequency and the processor speed. As a consequence, all tasks will take more time to be executed. In real-time systems, if this frequency change is not done properly, some important task may miss its deadline. Thus, it is necessary to identify the conditions under which we can safely slow down the processor without missing any deadline.

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

The problem of mixing hard and soft real-time tasks can be efficiently solved by using the *resource reservation framework* [19]. In such 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, both for fixed priority and dynamic priority schedulers [10, 24, 25, 2]. 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 as *reclamation techniques* [5, 3].

Intuitively, the problem of reclaiming the spare bandwidth is similar to the problem of power-aware scheduling. We can divide both problems in two parts. A first part consists in identifying the spare bandwidth (or the slack time) in the system. A second part consists in deciding how to use the spare bandwidth. The first part of the problem is common to both the bandwidth reclamation problem and the power-

aware scheduling problem. The second part, instead, differs radically: in the reclamation problem, the goal is to use the spare time to anticipate the execution time of the aperiodic tasks, whereas in the power-aware scheduling problem the goal is to lower the processor frequency as much as it is 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, that is based on the GRUB algorithm, proposed by Lipari and Baruah [5, 11]. In turn, the GRUB algorithm is based on the resource reservation framework, so it can support both hard and soft real-time tasks. Tasks can also be periodic, sporadic or even aperiodic.

The paper is organized as follows. After recalling the rules of the algorithm and proving its correctness, we present simulation experiments that compare GRUB-PA with the DRA and DRA-OTE algorithms, proposed by Aydin et al. [6] and with the DVSST algorithm proposed by Qadi et al [18]. Finally, we present the implementation of the algorithm in the Linux operating system and some experiments on a real test-bed system.

2. RELATED WORK

Power-aware scheduling techniques can be divided into static off-line and dynamic on-line techniques. In static techniques, the goal is to find the minimum constant processor frequency so that no task misses its deadline. Yao et al. [28] provided a static off-line scheduling algorithm, assuming periodic tasks and worst-case execution times. Non-preemptive power aware scheduling is investigated in [7].

Dynamic voltage scheduling (DVS) has been the topic of much recent research. When tasks have a variable execution time, DVS can exploit the slack time for reducing the energy consumption. Pillai and Shin [15] proposed different techniques to take into account the slack time. Aydin et. al. [6] proposed three algorithms based on EDF for reclaiming the spare time. Similar techniques have been proposed by Saewong and Rajkumar [20] 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. [17, 16] presented a study of power consumption and power-aware scheduling applied to multimedia streaming. Lorch and Smith addressed the variable voltage scheduling of tasks with soft deadlines in [13]. However, these techniques are based on heuristics.

Recently, Qadi et. al. [18] 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 utilisation, $U_i = \frac{C_i}{T_i}$), and at the 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 remaining of the paper, our algorithm GRUB-PA, instead, explicitly reclaims the spare bandwidth of tasks that execute less than expected, and therefore is able to reclaim spare time even in the case of periodic tasks.

Shin and Kim [23] proposed dynamic algorithms for power-aware scheduling, using a fixed priority or earliest deadline first policy and a dedicated server (i.e. Deferrable Server [26] or Total Bandwidth Server [25]) to handle aperiodic tasks. Using some existing DVS algorithms (including a modified version of the DRA [6]) they reclaim the slack time for both periodic and aperiodic tasks.

We extended the latter approach by taking a more abstract view. Our algorithm is based on the resource reservation framework [19], so it provides the temporal isolation property. Therefore, it is possible to provide real-time guarantees on a per-task basis.

3. ALGORITHM GRUB

In this section, we describe the GRUB (Greedy Reclamation of Unused Bandwidth) algorithm, proposed by Lipari and Baruah [5, 11]. The interested reader can refer to the original paper for a more detailed presentation of the algorithm.

GRUB is an algorithm belonging to the class of *aperiodic servers with dynamic priorities*. This class of techniques consists in creating an abstract entity for each task — *the server*. Each server is assigned a budget Q_i and a period P_i. The algorithm guarantees that the served task will execute at least Q_i units of time every P_i.

Almost all servers provide the *temporal isolation property*: the temporal behaviour of one task (i.e. its ability to meet its deadlines) is not affected by the behaviours of the other tasks. If a task misbehaves and requires a large execution time, it cannot monopolize the processor. Thanks to temporal isolation property, each task executes as it were on a slower dedicated processor. Therefore, it is possible to provide guarantees on a per-task basis.

Several server-based global schedulers (e.g., CBS [1]), 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.

3.1 System model

In our model, each server is characterised by two parameters, (U_i, P_i), where U_i is the server bandwidth (or fraction of the processor utilisation) and P_i is the period.

Each task τ_i executing on server S_i generates a sequence of jobs J_i^1, J_i^2, J_i^3, ..., 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. We assume that, inside each server, these jobs are executed in FIFO order, i.e. J_i^j has to finish before J_i^{j+1} can start executing.

We make the following requirements of our scheduling discipline:

- The arrival times of the jobs (the a_i^j's) are not *a priori* known, but are only revealed on line during system execution. Hence, our scheduling strategy cannot require knowledge of future arrival times.
- 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. (Nor do we require an *a priori* upper bound (a "worst-case execution time") on the value of c_i^j.)

- 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* scheduling algorithm (EDF) [12].

In this paper, we will 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 that the sum of the processor shares of all the servers sum to no more than one; i.e.,

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

The tasks will be executed on a single processor with a variable operating frequency. We assume that the processor can provide M frequencies, $\phi_1, \ldots, \phi_M$, in increasing order. Each frequency is associated a normalized processor "speed" $\overline{U}_1, \ldots, \overline{U}_M$, 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 processor speed: therefore, if a job executes e_i^j units of time when the processor speed is 1, it will execute for $e_i^j/\overline{U}_k$ when the processor speed is set equal to $\overline{U}_k$ (in Section 6, this assumption will be validated experimentally).

In the following section, we will assume that the processor speed is set to the maximum and is not changed. In Section 4 we will show how it is possible to extend the GRUB algorithm to exploit variable frequency.

3.2 Description of the GRUB algorithm

3.2.1 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.

At any instant in time during run-time, 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 jobs that arrived prior to t_o, but in doing so has "used up" 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 the servers that are in their Active Contending state, Algorithm GRUB chooses for execution (the next job needing execution 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 surrender control of the processor to an earlier-deadline server.

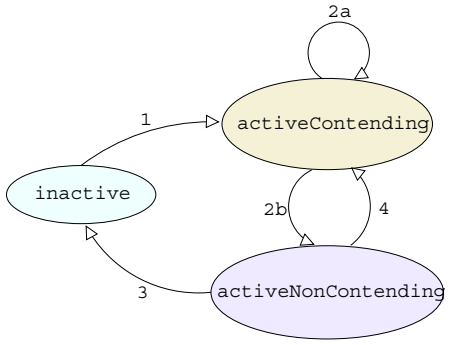

Figure 1: State transition diagram.

3.2.2 State Transitions.

Certain (external and internal) events cause a server to change its state (see Figure 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 its 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:

$$D_i \longleftarrow V_i + P_i \ ;$$

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 server S_i to be in the Active Non Contending state at any instant t, it is required that $V_i > t$. 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 utilisation* that, at every instant, is equal to

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

where n is the number of servers in the system. This variable is initialised 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 of U_i, whereas when S_i enters state Inactive it is decreased of U_i.

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

$$\frac{d}{dt}V_i = \begin{cases} \frac{U}{U_i} & \text{if } S_i \text{ is executing} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

The rate of increase of the virtual time is proportional to the current total bandwidth of the active servers. In other words, the rate of update of the virtual time is automatically adjusted depending on the current system load.

Let us make an example to understand the way the algorithm works. Consider a server S_1 with bandwidth $U_1 = 0.25$ and period $P_1 = 20$msec that serves a MPEG player that needs to visualise 25 frames per second. If the system is fully utilised (i.e. the total system bandwidth U is equal to 1), then Equation 1 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 approximately $P_1/4 = 5$msec every period P_1.

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 in the worst case 5msec are enough to visualise a frame in most cases.

However, suppose that at some point the total system utilisation U is equal to 0.75. Then, server S_1 can execute more than 5msec every period, because we can reclaim the spare bandwidth. According to Equation 1, the virtual time is increased at a rate of $0.75/0.25 = 3$. This means that our server will be able to execute for $P_1/3 = 6.66$msec every period.

Thus, if our application sometimes requires more than 5msec to display a frame, 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 application. Algorithm GRUB ensures that our application will take advantage of the spare bandwidth and execute more than $U_1 P_1$ in most cases. This property of GRUB is called "reclamation", because we are giving the spare bandwidth to the needing servers.

3.3 Performance guarantees

The following theorem formally states the performance guarantee that can be made by Algorithm GRUB *vis a vis* the behaviour of each server when executing on a dedicated processor. For proofs of the following theorems, see [5, 11].

THEOREM 1. *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*

$$f_i^j \leq A_i^j + \left\lceil \frac{(e_i^j/U_i)}{P_i} \right\rceil \cdot P_i \ . \quad (2)$$

For 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 2 ([11]). *Let τ_i be a hard real-time periodic task with worst case execution time C_i and period P_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.*

4. POWER-AWARE SCHEDULING

We now modify GRUB for power-aware scheduling. The new resulting algorithm is called GRUB-PA (Power-Aware). As 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 is halted). 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 [18] is based. However, GRUB-PA updates variable U in a more effective way, allowing additional power saving also in the case of periodic tasks, as shown in 4.1.

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 6 we will validate this assumption.

The GRUB algorithm can be divided in two different parts: a set of rules for identifying the spare bandwidth $(1 - U)$; and a set of rules for re-assigning the spare bandwidth.

The second part can be adapted for power-aware scheduling. In practice, if the processor is not fully utilised ($U < 1$) 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 the GRUB algorithm was designed for.

2. To slow down the processor. Each active server will execute for a longer time, but they will execute at a slower speed. The net effect is that their performance is not degraded.

The reclamation rule in GRUB is given by Equation (1). The increment in the virtual time depends on the amount of used bandwidth in the system. This rule can also be used in the power-aware part to automatically adapt the server bandwidth to the new frequency. In addition, we need an additional rule that sets the processor's speed equal to U whenever a server goes in the idle state or leaves the idle state.

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

1 When a job J_i^j arrives at time instant a_i^j, update the

following variables:

$$\begin{aligned} V_i &\leftarrow a_i^j \\ D_i &\leftarrow V_i + P_i \\ U &\leftarrow U + U_i \end{aligned}$$

Moreover, the processor speed is set equal to U.

3 When a server is in the Active Non Contending state and $V_i = t$, then the server goes in the Inactive state and the system utilisation is updated:

$$U \leftarrow U - U_i$$

Moreover, the processor speed is set equal to U.

4.1 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 instance of task τ_1 arrives at time $t = 0$ requesting 2 units of computation time; the second instance arrives at time $t = 12$ with computation time equal to 3. The resulting schedule is shown in Figure 2. The upward arrows denote an arrival time, while the downward arrows denote a deadline.

Initially, all servers are active, so $U = 1$ and the processor speed 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 Active Non Contending state. Then, task τ_2 starts executing, and it executes for 2 time units until $t = 4$. At this time, the first server changes states 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 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 units of execution, the virtual time will now increase at a rate of $dV_2 = dt\frac{U_2}{U} = dt$. Therefore, task τ_2 can now execute for 6 units of time, which correspond to 3 more units of execution time at maximum speed, and complete just by the deadline at 10. However, at time $t = 10$ another instance 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 instance of τ_1 is activated. The server becomes active 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 [18]: 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. In general, the GRUB-PA algorithm always anticipates this time with respect to algorithm DVSST, resulting in a larger amount of saved energy.

4.2 Properties of GRUB-PA

We now demonstrate that Theorem 1 is valid for the GRUB-PA algorithm. The complete proof is very similar to the one for Theorem 1 and it can be found in [22]. The following proof sketch is meant to ease the understanding of "why" the GRUB-PA algorithm works. Again, the basic idea is to show that every task behaves approximately as it were executing alone on a processor of speed U_i. First, a preliminary lemma.

LEMMA 3. *Under the GRUB-PA algorithm, when a job J_i^j completes, the completion time f_i^j is always less than or equal to the current server deadline D_i.*

Proof sketch The basic idea is to compare the schedule generated by GRUB-PA with the schedule generated by a fluid algorithm. We first define an algorithm GPS-PA (Generalized Processor Sharing - Power Aware) that allocates the processor in proportion to the bandwidths U_i of the active jobs. Moreover, GPS-PA adjusts the processor speed to the sum of the bandwidths of all active jobs $U(t)$. In the fluid schedule $\sigma_f(t)$ generated by GPS-PA, it can be proved that all jobs complete no later than $D_i^j = A_i^j + \left\lceil \frac{e_i^j}{P_i} \right\rceil P_i$. Then we transform schedule $\sigma_f(t)$ into a non-fluid schedule $\sigma_{nf}(t)$, by following a technique proposed by Coffman and Denning [4, Chapter 3]. The new schedule maintains the properties on the finishing time of the jobs. Finally, by the optimality of EDF, we can easily prove that GRUB-PA has the same property. □

Now we report the proof sketch of Theorem 1 for the GRUB-PA algorithm.

Proof sketch.

We will prove the theorem by induction on j. Let VA_i^j denote the virtual time of server S_i when the job J_i^j starts executing and let VF_i^j denote the virtual time when the job completes. In addition, let E_i^j denote the set of time intervals in which job J_i^j is scheduled by the GRUB-PA algorithm. The server is allowed to execute with a deadline D_i until the virtual time is less than D_i. Therefore, the cumulative length of the intervals in E_i^j is

$$LE_i^j = \int_{VA_i^j}^{VF_i^j} \frac{U_i}{U} dV_i$$

If the processor speed is constantly equal to 1, $LE_i^j \equiv e_i^j$. The slower the processor speed, the largest is LE_i^j.

From Equation (1):

$$VF_i^j = VA_i^j + \int_{E_i^j} dV_i(t) = VA_i^j + \int_{E_i^j} \frac{U(t)}{U_i} dt.$$

Now consider the base induction step ($j = 1$). The fol-

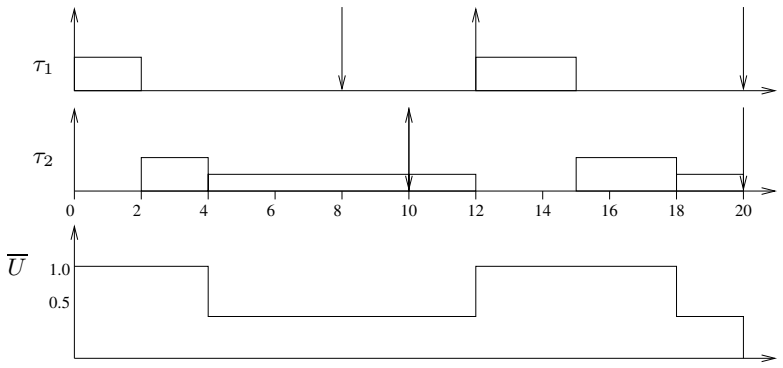

Figure 2: Example of schedule produced by GRUB-PA

lowing inequalities hold:

$$\begin{aligned} VA_i^0 &\equiv A_i^0 \\ VF_i^0 &= VA_i^0 + \int_{E_i^0} \frac{U(t)}{U_i} dt = A_i^0 + \frac{e_j^0}{U_i} \equiv F_i^0 \end{aligned}$$

The last equation holds because, even if the job executes with speed $U(t)$, LE_i^0 is such that VF_i^0 does not change. Then, the finishing time can be obtained as:

$$f_i^0 \leq VA_i^0 + \left\lceil \frac{VF_i^0 - VA_i^0}{P_i} \right\rceil P_i \leq A_i^0 + \left\lceil \frac{e_j^0}{U_i} \right\rceil P_i$$

Now, we develop the inductive step. Suppose that the following inequalities hold for job J_i^{j-1}:

$$\begin{aligned} VA_i^{j-1} &\leq A_i^{j-1} \\ VF_i^{j-1} &\leq F_i^{j-1} \end{aligned}$$

From Equations (1):

$$\begin{aligned} VA_i^j &= \max(a_i^j, VF_i^{j-1}) \leq \max(a_i^j, F_i^{j-1}) = A_i^j \quad (3) \\ VF_i^j &= VA_i^j + \int_{E_i^j} \frac{U(t)}{U_i} dt = VA_i^j + \frac{e_i^j}{U_i} \equiv F_i^j \end{aligned}$$

Finally, from Lemma 3, the finishing time can be bounded as follows:

$$f_i^j \leq D_i = VA_i^j + \left\lceil \frac{VF_i^j - VA_i^j}{P_i} \right\rceil P_i = A_i^j + \left\lceil \frac{e_i^j}{U_i} \right\rceil P_i$$

and the theorem is proved. □.

4.3 Discrete frequencies

Of course, no existing processor can vary its frequency with continuity. All processors that support DVS provide a discrete set of frequencies. 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}. By using this approach, the properties of the GRUB-PA algorithm continue to hold.

It is also possible to apply a technique similar to the one proposed by Ishihara et al. [9]. It consists in alternating between the two frequencies ϕ_k and ϕ_{k+1}, in order to further reduce the energy consumption by approximating the desired frequency $U(t)$. If the requested bandwidth is constant over an interval $[t_1, t_2]$, it is possible to compute an instant $t_1 < t' < t_2$ so that in interval $[t_1, t']$ the processor frequency is set equal to ϕ_{k+1} and in interval $[t', t_2]$ the processor frequency is set equal to ϕ_k. The original technique was devised as an off-line algorithm for periodic tasks with constant execution time. In our model, tasks may be periodic or aperiodic. Therefore, it is difficult to precisely know in which intervals the system bandwidth is constant. It is possible to further approximate in a conservative way the previous technique, for example by dividing the time line into intervals of arbitrary length and applying the previous technique. Since we always start with the highest frequency, we are guaranteed that in case the bandwidth changes, no deadline will be missed. The smallest are such intervals, the more precise is the approximation. However, the overheads in terms of time and energy of additional frequency switches need to be carefully considered. We leave the analysis of the potential trade-offs of such technique on our algorithm as a future work.

It is important to note that the GRUB-PA algorithm, like GRUB, is able to reclaim the spare bandwidth for soft aperiodic tasks. Suppose that the processor speed is set equal to $\overline{U}_k$, while the actual bandwidth is $U(t) < \overline{U}_k$. Then the GRUB-PA algorithm automatically reclaims the spare bandwidth $\overline{U}_k - U(t)$ for soft real-time aperiodic tasks. The spare bandwidth is assigned entirely to the currently executing server (GRUB-PA is a *greedy* algorithm).

5. EVALUATION OF THE ALGORITHM

We decided to compare our algorithm against the DRA algorithm proposed by Aydin et al. [6] and against the DVSST algorithm proposed by Qadi et al. [18].

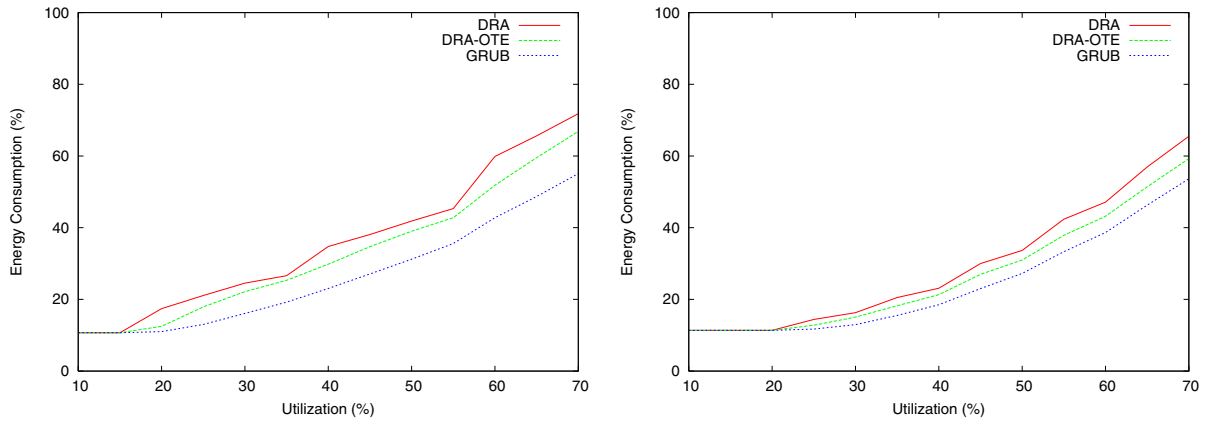

Figure 3: Energy consumption with constant WCET/BCET ratio on the a) PXA250 and on the b) TM5800

5.1 Comparison with DRA

The DRA algorithm permits to schedule periodic tasks in a hard real-time environment, reducing the energy consumption without missing any deadline. In particular, this solution consists of two parts:

- A static (off-line) solution that computes the optimal speed assuming the WCET for each instance;
- An on-line speed adjustment mechanism to dynamically reclaim slack time not used by tasks that complete without consuming their worst-case workload.

We compared GRUB-PA with the dynamic reclaiming algorithm (called *DRA*) and its '*One Task*' extension (called *DRA-OTE*). Both algorithms use a queue of tasks ordered by earliest deadline, called α-queue. 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 on 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 informations about all instances T_{ij} such that $r_{ij} \leq t \leq d_{ij}$, 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.

To compare the algorithms, we used a software tool called *RTSim* [1] [14]. 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. 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.

In particular, we simulated the power consumption on both a Intel PXA250([8]) processor, using 4 different operating frequencies, and a Transmeta Crusoe TM5800([27]) processor, using 7 operating frequencies. We associated a power consumption $P_k \propto \phi_k * V^2_{DD-k}$ to each frequency. 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.

To compare the algorithms we performed two different kinds of simulations. Each simulation consists of 5 different periodic tasks with randomly generated periods. We followed the same methodology of Aydin et al. [6]. Let WCET and BCET indicate the worst case and the best case execution time, 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 2, no task ever misses its deadline.

In the first set of simulations we fixed the WCET/BCET ratio of each task equal to 2, while the average workload varies from 0.1 to 0.7. For each value of the average workload, we simulated 100 different task sets. The results are shown in Figure 3.a for the PXA250 processor and on Figure 3.b for the TM5800 processor. In both cases, the GRUB-PA performs better than the DRA and DRA-OTE algorithms.

The second test measured the amount of power consumption with a constant average workload (50%) and a variable WCET/BCET ratio. For each value of the WCET/BCET ratio we run 100 simulations using different task sets.

Since the convexity of the power/speed curve suggests to use an uniform speed to obtain a lower power consumption, 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 Figure 4.a and Figure 4.b) confirm this assumption, and show that GRUB-PA is more clever than DRA and DRA-OTE at maintaining a constant CPU speed even when the task execution times are very different from the worst case ones.

Aydin et al. also proposed a third version of the DRA algorithm, called *DRA-aggressive*. It is an on-line speculative ("aggressive") speed adjustment mechanism to anticipate and compensate probable early completions by using the average-case workload information. According to the simulations results [6], DRA-aggressive performs 10% better than DRA-OTE, and its performance is approximately similar to the one presented by GRUB-PA.

However, GRUB-PA can also be applied to hard and soft, and periodic, aperiodic and sporadic tasks.

[1]The software is distributed under the GPL and it is freely downloadable from http://rtsim.sssup.it.

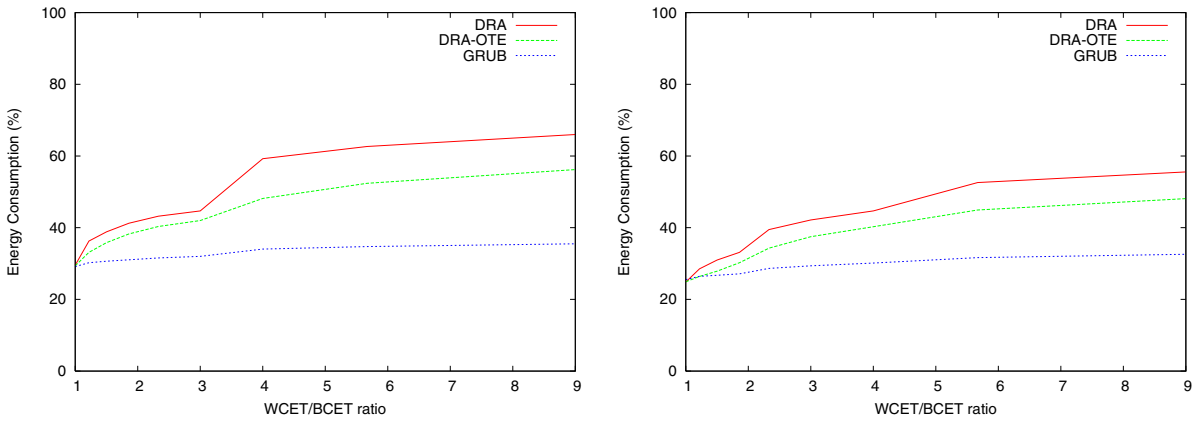

Figure 4: Energy consumption with constant average workload on a a) PXA250 and on a b) TM5800

5.2 Comparison with DVSST

We also compared the GRUB-PA algorithm against the DVSST algorithm [18]. In each simulation run, we generated 8 sporadic tasks with minimum interarrival times T_i randomly chosen 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% 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 0.1 and 0.9. The results for the PXA250 and the TM5800 processors are shown in Figure 5.

As it is possible to see, the DVSST algorithm is much more sensitive to the discretization of the 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 it can be noticed by comparing Figure 5.a with Figure 5.b. As expected, GRUB-PA presents an improvement up to 40% with respect to DVSST.

5.3 Overhead

One detail 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 $500\mu sec$. In many soft real-time applications, this can be considered negligible, however, it should not be ignored. Moreover, we want 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 de-activated very often. If the system is close to one threshold value $\overline{U}_k$, every activation would cause an increase of the frequency, and every de-activation would cause a decrease in the frequency.

To avoid these situations, when an increase of the total system bandwidth U goes over one of the thresholds $\overline{U}_k$, we immediately increase the processor frequency because we do not want to risk a hard task to miss its deadline. When a decrease of the total system bandwidth U goes below one of the threshold $\overline{U}_k$, 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 reduce the number of unnecessary frequency switches.

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 2 frequency switches every Δ, one to go down and another one to go up. Therefore, 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 $1 - \frac{2\delta}{\Delta}$.

It is also undeniable the presence of an energy overhead at every frequency switch. This overhead depends on the particular kind of CPU the algorithm is running on, and it is quite difficult to estimate and measure. Thus, we decided not to deal with it at simulation time. Instead, the presence of this overhead has been automatically accounted for in our experimental results (see the next section). In fact, the total energy consumed by our test-bed also comprises the energy due to frequency changes.

6. IMPLEMENTATION AND EXPERIMENTAL RESULTS

The implementation of the GRUB-PA algorithm has been done in the context of the OCERA project (IST-35102). The main objective of this project is the design and implementation of a library of free software components for embedded real-time systems. These components will be used to create flexible (new scheduling will support a wide variety of applications), configurable (scalable from a small to a fully featured system), robust (fault-tolerant and high performant) and portable (adaptable to several hw/sw configurations) systems.

We implemented the GRUB-PA on the Linux operating system, running on an Intrinsyc CerfCube 250 system. The CerfCube system consists of 32 MB Flash ROM, 64 MB SDRAM, and a Ethernet 10/100 Mbps. The processor is an Intel PXA250. It is a superpipelined 32 bits RISC processor based on the Intel *Xscale* micro-architecture. This architecture permits a on-the-fly switch of the clock frequency and a sophisticated power consumption management.

We configured the system to support three different frequencies, 100Mhz, 200Mhz and 400 Mhz. The modified OS is Linux 2.4.18. A description of the implementation can be

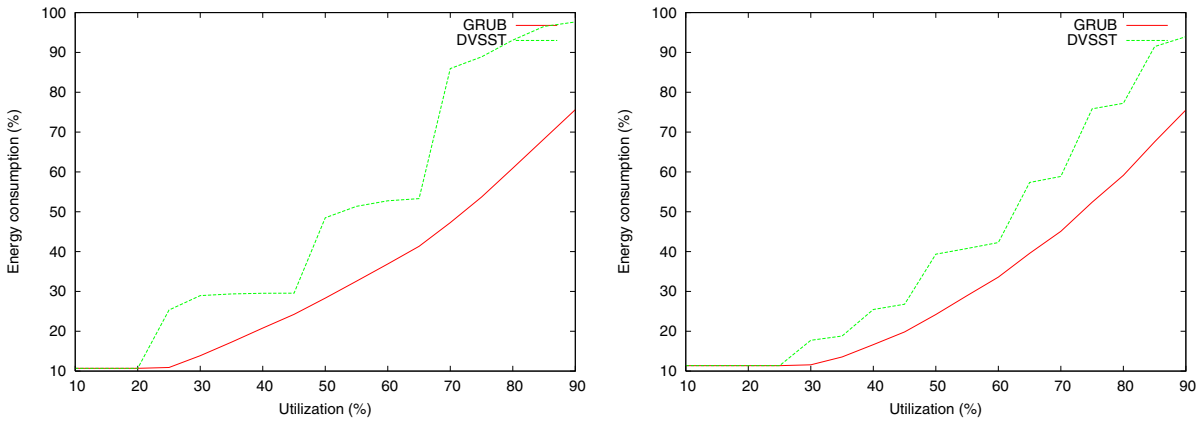

Figure 5: Energy consumption with constant average workload varying between 0.1 and 0.9 a) PXA250 and on a b) TM5800

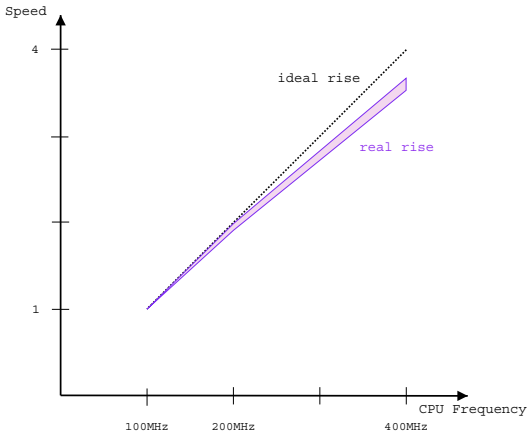

Figure 6: Decompression speed related to CPU speed (99% confidence interval).

found in [21]. Our study is 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. So, we decided to focus our attention on a audio decoder.

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 the different fixed clock frequencies.

From the obtained values, we extracted how much the speed of decompression is related to the speed of the CPU. The result is shown in Figure 6, where we show on the x-axis the frequency of the processor, and on the y-axis the decompression speed. As you 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 the Figure we also show the 99% confidence interval.

Then, we evaluated the power consumed by our system

Table 1: Average values of the input current.

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

under different conditions, with or 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 loads. We computed the average values of the input current, reported in the table 1.

We did many experiments using the multimedia application, and we observed that, using our frequency scaling mechanism, we saved up to 38.4% of the total power consumed by the system. Notice that this is the energy saved in the whole system, not only by the processor.

7. CONCLUSIONS

In this paper we presented the GRUB-PA algorithm, a novel power-aware scheduling algorithm suitable to systems consisting of hard periodic and soft aperiodic real-time tasks. The GRUB 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 a significant improvement over other power-aware scheduling algorithms presented in the literature. Moreover, we presented an implementation of the GRUB-PA in the Linux operating system. The experimental results show that using GRUB-PA we saved up to 38.4% of the total power consumed by the system with respect to the unmodified one.

8. ACKNOWLEDGEMENTS

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.

9. REFERENCES

[1] L. Abeni. Server mechanisms for multimedia applications. Technical Report RETIS TR98-01, Scuola Superiore S. Anna, 1998.

[2] L. Abeni and G.Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, december 1998. IEEE.

[3] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proceedings of the IEEE Real-Time Systems Symposium*, Orlando, Florida, December 2000.

[4] J. E. Coffman and P. J. Denning. *Operating Systems Theory*. Prentice-Hall, Englewood Cliff, NJ, 1973.

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

[6] H.Aydin, R.Melhem, D.Mossé, and P.Mejia-Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proceedings of Real-Time System Symposium*, pages 95–105., London, UK, December 2001.

[7] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. Srivastava. Power optimization of variable voltage core-based systems. In *In Proceedings of the 35th Design Automation Conference*, 1998.

[8] Intel corporation. *Intel PXA250 and PXA210 Application Processors Developer's Manual*, February 2002.

[9] H. Ishihara, T.and Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 197 – 202, Aug. 1998.

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

[11] G. Lipari. *Resource Reservation in Real-Time Systems*. PhD thesis, Scuola Superiore S.Anna, 2000.

[12] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1), 1973.

[13] J. R. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with pace. In *In Proceedings of the ACM SIGMETRICS 2001 Conference*, Cambridge, MA, June 2001.

[14] 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.

[15] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceeding of the 18th ACM Symposium on Operating Systems Principles*, 2001.

[16] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *7th ACM Int. Conf. on Mobile Computing and Networking (Mobicom)*, 2001.

[17] J. Pouwelse, K. Langendoen, and H. Sips. Energy priority scheduling for variable voltage processors. In *Int. Symposium on Low Power Electronics and Design (ISLPED)*, 2001.

[18] A. Qadi, S. Goddard, and S. Farritor. A dynamic voltage scaling algorithm for sporadic tasks. In *Proceedings of the 24th Real-Time Systems Symposium*, pages 52 – 62, Cancun, Mexico, 2003.

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

[20] S. Saewong and R. Rajkumar. Practical voltage-scaling for fixed-priority rt-systems. In *Proceedings of the ninth IEEE Real-Time and Embedded Technology and Applications Symposiuam (RTAS)*, May 2003.

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

[22] C. Scordino and G. Lipari. Using resource reservation techniques in power-aware scheduling: The grub-pa algorithm. Technical report, Scuola Superiore Sant'Anna, 2003.

[23] D. Shin and J. Kim. Dynamic voltage scaling of periodic and aperiodic in priority-driven systems. In *proceedings of ASP-DAC '04*, pages 653–658, January 2004.

[24] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Journal of Real-Time Systems*, 1, July 1989.

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

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

[27] Transmeta Corporation, http://www.transmeta.com. *Crusoe^{TM} Processor Model TM5800 Version 2.1 Data Book Revision 2.01*, June 2003.

[28] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. *IEEE Annual foundations of Computer Science*, pages 374–382, 1995.