An Energy-Aware Algorithm Exploiting Limited Preemptive Scheduling under Fixed Priorities

Mario Bambagini[1], Marko Bertogna[2], Mauro Marinoni[1] and Giorgio Buttazzo[1]

[1]Scuola Superiore Sant'Anna, Pisa, Italy
[2]University of Modena and Reggio Emilia, Italy

***Abstract*—This paper presents a new energy-aware algorithm that integrates Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Power Management (DPM) techniques to further reduce energy consumption in embedded systems. It consists of an off-line DVFS-stage, for computing the speed that minimizes energy consumption during active intervals while guaranteeing timing constraints, and an online DPM-stage, for prolonging sleep intervals by postponing task execution. Moreover, limited preemptive scheduling is exploited to reduce preemption costs and further extend sleep intervals under fixed-priority systems, with respect to fully preemptive schedulers. The online algorithm has a constant complexity and preemption costs are taken into account in the analysis. A set of simulation experiments are reported to illustrate the behavior of the proposed approach as a function of different parameters and compare its performance with the state-of-art methods available in the literature.**

I. Introduction

Energy saving became a crucial goal in modern embedded systems, especially for battery operated devices, such as autonomous mobile robots and wearing devices, for which the minimization of energy consumption leads to a longer lifetime, which in turn allows saving money and curbing environmental pollution.

Two widely used techniques to save energy in the actual technology are *Dynamic Voltage and Frequency Scaling* (DVFS) and *Dynamic Power Management* (DPM). The DVFS approach trades energy with performance by decreasing the voltage and/or frequency of the processor to reduce the overall energy consumption. Since a frequency reduction increases the execution times of the computational activities, the objective of this technique is to find the slowest processor speed that still guarantees real-time constraints. On the other hand, DPM techniques aim at switching the processor in a low-power inactive state for the longest possible time, thus postponing the tasks execution as long as possible, still guaranteeing the task real-time constraints.

In CMOS technology, which is the actual leader, power consumption is due to both dynamic and static components, which are ascribable to the system activity and static dissipation, respectively. Unless the system is off, the static contribution is always present, independently of the actual performance level. Thus, DVFS approaches that modify the clock frequency are more suitable for reducing the dynamic power, whereas DPM solutions are best suited for decreasing the impact of the static component. These techniques can also be integrated to exploit their complementary features for saving more energy.

During the last decades, many algorithms have been proposed for scheduling real-time tasks and a considerable effort has been dedicated to those taking energy management into account. Most of them consider either a fully preemptive or non-preemptive model, according to how task arrivals are handled. In a fully preemptive system, if a newly activated task has a priority higher than the running task, a preemption occurs to move the running task in the ready queue and assign the processor to the new one. In a non-preemptive model, in the same scenario, the execution of the new task would have been postponed at the end of the running task. Although non-preemptive scheduling can save a lot of runtime overhead and make the execution time more predictable, it may introduce large blocking times during the execution of high priority tasks, affecting the schedulability of a real-time system.

Limited preemptive scheduling has recently been exploited as a hybrid technique to take advantage of both types of scheduling modes, trying to mitigate their drawbacks. As shown in the literature [1], limited preemptive scheduling is able to increase the schedulability of both fully and non-preemptive scheduling models, even when the preemption overhead is neglected. The improvement is even more significant when considering the preemption cost, which generally includes the context switch time for suspending the running task and dispatching the new one, the time taken to flush the pipeline, and the cache-related preemption delay due to cache misses. Moreover, limited preemptive scheduling allows an implicit mutual exclusion management (when critical sections are encapsulated inside non preemptive regions) and permits reducing the minimum stack memory requirements.

Contribution of the paper:

In this paper, limited preemptive scheduling is exploited for further reducing energy consumption with respect to fully preemptive and non-preemptive models.

A two-stage algorithm (consisting of an off-line DVFS phase followed by an online DPM phase) is presented for fixed priority systems consisting of periodic real-time tasks. In the first stage, an off-line algorithm selects the speed that

This work has been partially supported by the 7th Framework Programme Juniper (FP7-ICT-2011.4.4) project, founded by the European Community under grant agreement n318763.

978-1-4799-0658-1/13/$31.00

minimizes energy consumption during active intervals while guaranteeing the feasibility of the task set under limited preemptive scheduling. For those architectures taking advantage of speed scaling techniques for reducing energy consumption (DVFS-sensitive architectures), the speed computed by the proposed off-line algorithm is shown to be lower than the one selected by existing DVFS algorithms under fully preemptive or non-preemptive models. Conversely, for those architectures in which the use of low-power states is more convenient (DPM-sensitive architectures), the off-line stage returns the highest available speed. In the second stage, a DPM technique is applied online to prolong idle intervals as long as possible to take advantage of low-power sleep states. It is shown that such a technique is able to significantly decrease energy consumption with a negligible runtime overhead. This is possible thanks to the off-line phase, used to compute the longest delay that can be added after an idle interval to keep the processor in sleep mode, so avoiding complex online computations. In this way, the proposed technique exhibits a performance similar to that of the best existing DPM algorithms, but with a much smaller runtime overhead. For this reason, it is very suited for embedded systems with limited resources.

The power model adopted in this paper is general enough to represent different consumption profiles and features, such as low power sleep states (and relative overheads) and discrete speeds. This allows the presented results to be applicable to different kinds of architectures and power models.

Finally, a set of experimental results are reported to illustrate the benefits of the presented techniques under different conditions. The performance of the algorithm is tested by varying different architectural parameters, including the break-even time and the preemption overhead. Although the presented algorithm shows the best performance even when the preemption overhead is neglected, the improvement with respect to existing preemptive and non-preemptive techniques increases when preemption costs are considered. Simulation experiments show that the proposed algorithm outperforms other well-known algorithms, reducing energy consumption down to 13%.

Organization of the paper:

Section II introduces the system model in terms of computational workload and energy consumption. The motivation example in Section III aims at addressing the problem and showing the ample room of improvement given by the limited preemptive task model. Section IV reports the schedulability analysis for the limited preemptive task model that is adopted in this paper. Section V presents the proposed solution and an implementation of the algorithm, while Section VI reports the experimental results obtained by exhaustive simulations. The state of the art concerning energy saving is reported in Section VII. Section VIII ends the paper with the final remarks, pointing out ideas for future improvements.

II. System Model

We consider a set Γ of n fixed priority periodic tasks, $\tau_1, \tau_2, \ldots, \tau_n$, executing upon a single processor platform with preemption support. Without loss of generality, we assume that tasks are indexed in decreasing priority order (i.e., if $0 < i < j \leq n$, then τ_i has higher priority than τ_j). The processor can vary the clock frequency f by selecting one of the available frequencies in a discrete set $\{f_1, \ldots, f_m\}$, ordered by ascending values. In the following, the normalized speed s, defined as $s = f/f_m$, will be used as a more convenient parameter ($s_m = 1$ denotes the maximum speed).

Each task τ_i is characterized by a worst-case execution time (WCET) $C_i^{NP}(s)$, which is a function of the speed, a relative deadline D_i and a period T_i. The WCET of τ_i is computed as $C_i^{NP}(s) = \alpha_i C_i^{NP} + (1-\alpha_i) C_i^{NP}/s$, where C_i^{NP} denotes the time to execute τ_i in a non-preemptive mode at the maximum speed ($C_i^{NP} = C_i^{NP}(s_m)$) and α_i represents the portion of execution time that does not scale with the speed (e.g. I/O operations). Moreover, the symbol $C_i(s)$ denotes the worst-case execution time of task τ_i in limited preemptive mode, including the preemption overhead. Relative deadlines can be smaller than, equal to, or greater than periods. All parameters are assumed to be in $\mathbb{N}^+$. Each task generates an infinite sequence of jobs, with the first job arriving at time zero and subsequent arrivals separated by T_i units of time.

Each task τ_i consists of a sequence of non-preemptive chunks and can be preempted only at the end of a chunk. For the sake of the analysis, the duration of the longest chunk (at the current speed s) is denoted as $q_i^{max}(s)$, and the one of the last chunk is denoted as $q_i^{last}(s)$. Note that the last non preemptive chunk of a task is crucial for decreasing its response time, because it reduces the interference from higher priority tasks. For this reason, it is convenient to make the last chuck as long as possible. However, q_i^{max}(s) can not be arbitrarily large to limit the blocking time imposed to higher priority tasks. Note that, under such a model, tasks do not need to be independent, but can interact through shared resources, provided that critical sections are entirely contained within a non-preemptive chunk.

Finally, preemption cost is taken into account by considering a penalty ξ, which includes the context switch overhead, the pipeline invalidation delay, and the cache-related preemption delay. Since only the context switch overhead depends on the actual speed, the preemption cost ξ is assumed to be constant and speed independent.

A. Power model

To characterize the power model of the considered systems, we adopt the following relation derived by Martin et al. [2], which represents a general expression of the power consumption of an active processor as a function of the speed:

$$P(s) = K_3 s^3 + K_2 s^2 + K_1 s + K_0. \quad (1)$$

The K_3 term is the coefficient related to the consumption of components varying with both voltage and frequency. The second order term (K_2) describes the non linearity of DC-DC regulators in the range of the output voltage. The K_1 coefficient is related to the hardware components that can only vary the clock frequency, whereas K_0 represents the power

consumed by the components that are not affected by the processor speed.

Switching between two frequencies or operating modes takes a different amount of time and consumes a different amount of energy, depending on the specific transition. Such an overhead is usually non-negligible.

Note that the energy needed to execute a job is the product of the power and the execution time at the selected speed; moreover, a higher speed reduces the execution time, but increases the power consumption. Hence, the quantity that it is important to minimize is the energy consumption of each clock cycle $E_{clk}(s) = \alpha P(s) + (1-\alpha)P(s)/s$ (where α is the overall fraction of computation time which does not scale with the speed). Considering the shape of $P(s)$ as a function of s, as reported in Equation 1, a critical speed s^* that minimizes $E_{clk}(s)$ can be found [3].

As an example, consider a processor with ten speeds uniformly distributed from 0.1 to 1.0, and in which the power consumption is modeled as $P(s) = 0.9s^3 + 0.1$ (describing a DVFS-sensitive architecture). Assuming $\alpha = 0.2$, the curves representing $P(s)$ and $E_{clk}(s)$ are illustrated in Figure 1 and the speed that minimizes $E_{clk}(s)$ is $s^* = 0.4$. In the case in which $P(s) = 0.3s + 0.7$, as reported in Figure 2, the speed that minimizes the energy during active intervals is $s^* = 1.0$. In other words, in the second example (describing a DPM-sensitive architecture), speed scaling is not energy convenient.

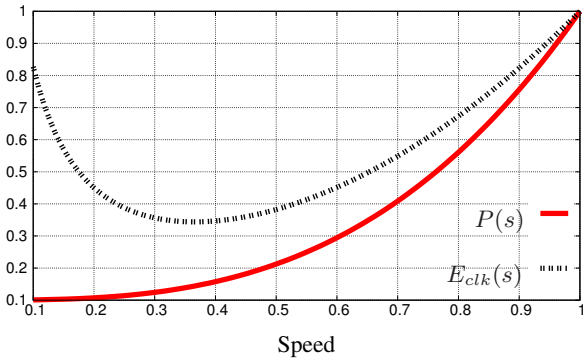

Fig. 1. $P(s)$ and $E_{clk}(s)$ of a DVFS-sensitive architecture.

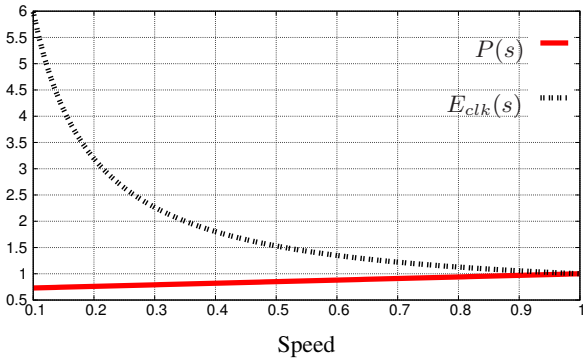

Fig. 2. $P(s)$ and $E_{clk}(s)$ of a DPM-sensitive architecture.

An additional feature provided by almost all the current processors is to switch to low-power states, suspending the code execution. For the sake of simplicity, in the rest of the paper only a single low-power state, called *sleep* state, is assumed. However, the proposed approach can easily be extended to consider more low-power states with different characteristics. In particular, the power consumption in the sleep state is denoted as P_σ and the overhead times to enter and exit the sleep state are $\delta_{s\to\sigma}$ and $\delta_{\sigma\to s}$, respectively. They are reported in Figure 3. The sum of such switching times, referred to as *break-even time* (δ), determines the shortest idle interval that must be available in the schedule to exploit the sleep state. Such an overhead is assumed independent from the actual running speed. The energy consumed during a transition from active to sleep and viceversa is denoted by E_δ.

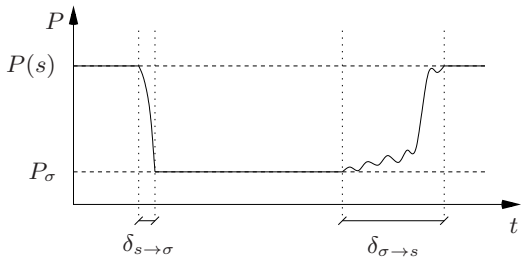

Fig. 3. Overhead due to switching from active to sleep state and viceversa.

III. Motivational Example

To illustrate the benefit of limited preemption scheduling to save energy, let us consider a processor with two speeds, $s_1 = 0.5$ and $s_2 = 1$, without low-power states (the processor is always on), executing two tasks, τ_1 and τ_2, with the following parameters: $C_1 = 30$, $T_1 = D_1 = 80$, $C_2 = 25$ and $T_2 = D_2 = 200$ (computation times are referred to speed s_2). Tasks are scheduled using Rate Monotonic and, for the sake of simplicity, preemption costs are considered negligible and $\forall \tau_i : \alpha_i = 0$. The processor utilization factor at speed s_2 is $U = 0.5$ and the task set results feasible under fully-preemptive, non-preemptive, and limited preemptive scheduling. Switching to s_1, however, computation times become $C_1 = 60$ and $C_2 = 50$, making the task set unfeasible under both fully-preemptive and non-preemptive modes. Nevertheless, a feasible schedule can be found under the limited preemptive model by splitting task τ_2 into three chunks of length 10, 20, and 20 units of time, respectively, under speed s_1. The schedules produced by the Rate Monotonic under the three different preemption modes are shown in Figure 4.

This example shows that, using the limited preemption model, the processor can run with a speed lower than that allowed by fully-preemptive and non-preemptive models, hence saving more energy.

IV. Background on Limited Preemption

The limited preemption scheduling model has been introduced to limit the preemption overhead of fully preemptive schedulers, without incurring in the blocking overhead of non-preemptive solutions. According to this model, each task

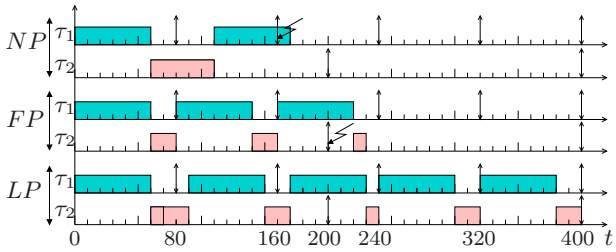

Fig. 4. Schedules produced by Rate Monotonic at speed $s = 0.5$ under Non-Preemptive (NP), Fully-Preemptive (FP), and Limited Preemptive (LP) scheduling.

is divided into a set of non-preemptive regions, so that preemptions can take place only at chunk's boundaries. Two different sub-models have been defined in the literature. In the *Floating* Non-Preemptive Region model [4], [5], the location of each non-preemptive region is not known a priori, but might vary during task execution. Instead, in the *Fixed* Non-Preemptive Region model [6], [7], a set of fixed preemption points is statically defined for each task, so that a task might be preempted only at these well-determined points. This last model has been shown in [8] to dominate all other techniques, since it is able to schedule a strictly larger number of task sets than the fully preemptive, the non-preemptive and the floating non-preemptive region models. This model, which will be adopted throughout the remainder of the paper, has several benefits, including:

- A bounded number of preemptions, strictly smaller than the number of chunks;
- A simpler and tighter evaluation of the preemption overhead, as a task can be preempted only at a small number of deterministic locations;
- A smaller preemption cost due to a smaller cache-related preemption delay, by a reduced number of cache misses;
- A smaller worst-case memory stack, which is used to store the contexts of the running and suspended tasks [9];
- A simplified management of the mutual exclusions, as critical sections contained in a non-preemptive chunk do not need any shared resource protocol.

An exact scheduling analysis for such a model was provided by Bertogna et al. [8]. Here the main results are reported, adapting them to the task model considered in this paper (in particular, considering the dependence of the execution parameters to the processing speed).

Note that the number of chunks into which a task is divided depends on the selected speed and is denoted as $p_i(s)$. The duration of the k-th chunk of task τ_i at speed s is denoted as $q_{i,k}(s)$. Hence, the task computation time with limited preemption (at speed s) can be also expressed as the sum of the chunk durations: $C_i(s) = \sum_{k=1}^{p_i(s)} q_{i,k}(s)$. Note that the preemption overhead is included within each chunk length $q_{i,k}(s)$, and the worst-case execution time which considers preemption overhead is

$$C_i(s) = C_i^{NP}(s) + \xi \cdot (p_i(s) - 1). \tag{2}$$

The maximum length $q_i^{max}(s)$ of a chunk is computed for a specific speed, as explained in Section V-A. Then, the last chunk is assigned the maximum length, setting it to $q_i^{last}(s) = q_i^{max}(s)$, in order to reduce as much as possible the interference on the considered task. The remaining chunks are all assigned the maximum length $q_i^{max}(s)$, except for the first one, which takes the remaining computation time*. Therefore,

$$p_i(s) \stackrel{\text{def}}{=} \left\lceil \frac{C_i^{NP}(s) - q_i^{max}(s)}{q_i^{max}(s) - \xi} \right\rceil + 1. \tag{3}$$

$$\begin{cases} q_{i,1}(s) = C_i(s) - (p_i(s) - 1) q_i^{max}(s) \\ q_{i,j}(s) = q_i^{max}(s) \qquad \forall j \in [2, p_i(s)]. \end{cases} \tag{4}$$

The maximum blocking time $B_i(s)$ actually experienced by a generic task τ_i can then be computed as

$$B_i(s) = \max_{i<j\leq n} \left\{ q_j^{max}(s) - 1 \right\}. \tag{5}$$

Another important parameter useful for checking the feasibility of a task τ_i is the maximum amount of blocking that τ_i can tolerate from lower priority tasks without violating any of its deadlines. Such a time is referred to as *blocking tolerance* and is denoted as $\beta_i(s)$. As shown in [8], the blocking tolerance of task τ_i can be computed as the minimum blocking tolerance among all its jobs arriving in the largest level-i active period L_i:

$$\beta_i(s) = \min_{k\in[1,K_i]} \beta_{i,k}(s), \tag{6}$$

where K_i is the number of jobs in L_i: $K_i = \left\lceil \frac{L_i}{T_i} \right\rceil$, and L_i is the largest level-i active period, computed recursively as

$$\begin{cases} L_i^{(0)}(s) = B_i(s) + C_i(s) \\ L_i^{(l)}(s) = B_i(s) + \sum_{j=1}^{i} \left\lceil \frac{L_i^{(l-1)}(s)}{T_j} \right\rceil C_j(s), \end{cases} \tag{7}$$

until $L_i^{(l)}(s) = L_i^{(l-1)}(s)$†.

Finally, as shown in [8], the blocking tolerance of job $\tau_{i,k}$ can be computed as

$$\beta_{i,k}(s) = \max_{t\in\Pi_{i,k}} \left\{ t - kC_i(s) + q_i^{last}(s) - W_i(t,s) \right\}, \tag{8}$$

where $\Pi_{i,k}(s)$ is the set of arrivals of jobs interfering with $\tau_{i,k}$:

$$\begin{aligned} \Pi_{i,k}(s) \stackrel{\text{def}}{=} & \left[(k-1)T_i, (k-1)T_i + D_i - q_i^{last}(s) \right] \bigcap \\ & \{ hT_j - 1, \forall h \in \mathbb{N}, j \leq i \} \bigcup \\ & \left\{ (k-1)T_i + D_i - q_i^{last}(s) \right\}, \end{aligned} \tag{9}$$

*Actual chunk sizes might be slightly smaller in order to accommodate potential critical sections within a non-preemptive region. In this way, there is no need of any shared resource protocols.

†As shown in [8], $\beta_{i,1}(s)$ can be used as an upper bound of $B_i(s)$, whenever the latter value is not known.

while $W_i(t,s)$ represents the cumulative execution request of all tasks with priority greater than τ_i over any interval $[a,b]$ of length t. It is computed as

$$W_i(t,s) \stackrel{\text{def}}{=} \sum_{j=1}^{i-1} \text{RBF}_j(t,s). \quad (10)$$

For any task τ_i and any non-negative number $t \in \mathbb{N}^+$, the **request bound function** $\text{RBF}_i(t,s)$ denotes the maximum sum of the execution requests at the specific speed s that could be generated by jobs of τ_i arriving within a contiguous time-interval $[a,b]$ of length t, considering the preemption costs. It has been shown [10] that the request bound function for a task τ_i is:

$$\text{RBF}_i(t,s) \stackrel{\text{def}}{=} \left(\left\lfloor \frac{t}{T_i} \right\rfloor + 1 \right) C_i(s). \quad (11)$$

As a result of the provided analysis, a task set is considered feasibile at speed s under the limited preemptive task model if and only if all the task blocking tolerances are not negative, i.e. $\forall \tau_i \in \Gamma : \beta_i(s) \geq 0$.

V. Proposed Approach

The proposed approach consists of a two-stage algorithm: the first step is executed off-line and computes the slowest available speed that guarantees the task set feasibility (also considering preemption costs and energy model). The computed speed is set at the system start and is never changed during execution. The second part of the algorithm is executed at run-time and aims at prolonging the idle intervals as long as possible for exploiting the sleep state, as short idle intervals might not be usable due to the break-even time.

The first stage of the algorithm (exploiting DVFS techniques) is analyzed in Section V-A, while the second stage (exploiting DPM techniques) is presented in Section V-B.

A. DVFS Algorithm

The off-line stage of the method, reported in Algorithm 1, consists of finding the slowest speed that guarantees the task set feasibility, considering the worst-case preemption costs and the particular power function $P(s)$. The speed found by this procedure is never changed at run time.

The algorithm receives as input the task set Γ and the worst-case preemption cost ξ. The first line of the code computes the critical speed s^*, as described in Section II-A. Then, the speed subset S' is created by sorting the speeds in ascending order and discarding those that are smaller than s^*, since they are not convenient from an energy point of view. Speeds greater than s^* might instead be needed when the task set is not feasible at s^*. For DPM-sensitive architectures, S' will contain only $s^* = 1.0$, even though lower speeds might be feasible. For each speed in S' (cycle at line 4), all parameters introduced in Section IV are computed to find a feasible solution. When a speed that leads to a feasible solution is found, the procedure provides the feasible speed and the corresponding minimum blocking tolerance, denoted as β_{min}.

Algorithm 1 DVFS algorithm

```
1:  function DVFS_ALGORITHM (Γ, ξ)
2:      s* ← compute_critical_speed ()
3:      S' ← {s ∈ S | s ≥ s*}
4:      for each s ∈ S' do
5:          β_min ← ∞
6:          for i ∈ [1, n] do
7:              q_i^max ← min (C_i(s), β_min + 1)
8:              β_i ← compute_task_tolerance (i, s, ξ)
9:              β_min ← min (β_i, β_min)
10:             if β_min < 0 then
11:                 break
12:             end if
13:         end for
14:         if β_min ≥ 0 then
15:             return [s, β_min]
16:         end if
17:     end for
18:     return No speed found
19: end function
```

The cycle at line 6 checks, from the highest to the lowest priority task, whether the considered task can be executed at such a speed without missing deadlines. The feasibility test is done at line 10 by checking whether β_{min} (the minimum blocking tolerance among the tasks analyzed so far) is negative. In fact, $\beta_{min} < 0$ means that a task may be blocked by a lower priority task for a time longer than its slack. The value of β_{min} is first initialized for each speed at line 5 and then updated at line 9 based on the blocking tolerance of the current task (computed at line 8 according to Equation 6).

Once all the tasks have been considered at a specific speed, if the minimum blocking tolerance is not negative, then the algorithm completes successfully as the slowest speed that guarantees the feasibility has been found. At this point, the length of each chunk is easily computed by Equation 4.

Note that Algorithm 1 increases the complexity of the preemption point placement procedure (which is pseudo-polynomial) by a factor of m (the number of available speeds).

B. DPM Algorithm

Once the slowest feasible speed has been found for scheduling the task set under limited preemption, a further power reduction can be obtained by exploiting low-power sleep states. Indeed, whenever the processor can be left idle for an amount of time larger than δ (the break-even time) without missing any deadline, it is convenient to switch to the sleep state to save more energy. The longer the processor can remain in sleep mode, the smaller the energy consumption.

In this section we present a new on-line DPM algorithm that exploits limited preemptive scheduling to extend idle intervals as much as possible. The idea behind the proposed algorithm is that, whenever a new job arrives and the processor is idle, this job can be delayed by at least the minimum blocking tolerance β_{min}. In this way, the processor can safely

remain in sleep mode for a longer time. An advantage of the presented method is that it does not require any particular on-line computation of the slack times of the incoming jobs. All meaningful parameters are statically computed before run-time, and no external hardware is needed to compute the length of the idle times and to enforce the sleep states.

The larger blocking tolerances allowed under limited preemptive scheduling [11] can extend the sleep time to save a significant amount of power, comparable to the power saved by more aggressive on-line DPM algorithms that require a much higher run-time computational effort. As an example, the algorithm presented in [3] requires building the schedule after each idle instant until the earliest deadline, computing the idle time of each job in the considered window, and postponing the arrival of each job by the corresponding idle time. Everything needs to be done on-line, whereas the algorithm presented here is able to obtain a similar performance without any of the above on-line operations.

The pseudo code of the DPM procedure is reported in Algorithm 2 and is invoked every time a job ends and the ready queue is empty. The algorithm takes as input the parameters of the task set (Γ) and as a result it prolongs the idle intervals as much as possible by postponing the execution of the jobs that may arrive in between, still preserving their deadlines.

Algorithm 2 DPM algorithm

```
1: function DPM_ALGORITHM (Γ)
2:     t ← current_time ()
3:     t_act ← next_arrival ()
4:     t_up ← t_act + β_min
5:     if (t_up − t) ≥ δ then
6:         sleep_for (t_up − t − δ_σ→s)
7:         wake_up ()
8:     end if
9: end function
```

After reading the current time t, the algorithm invokes a function that returns the arrival time of the next job. Then, the minimum blocking tolerance computed by Algorithm 1 is added to the arrival time in order to find when the system can be awaken without missing any deadline.

In case the interval spendable in sleep mode is longer than the break-even time δ, the routine invoked at line 6 handles all the operations to switch into sleep mode. More precisely, this function sets the job arrival interrupt mask, sets an external timer to send an external waking up interrupt at time $t_{up} - \delta_{\sigma \to s}$ and physically switches the system off.

As soon as the external waking up interrupt arrives, the code execution is recovered from line 7, which unmasks job arrival interrupts and handles pending job activations.

The particular way in which the DPM algorithm is implemented allows extending the sleep state and taking advantage of different slack sources:

- unused processor bandwidth related to task set utilizations smaller than one (at the critical speed) and idle times due to the use of a speed higher than the optimal one, since only a discrete set of speeds is available;
- spare capacities associated to early task terminations are automatically reclaimed by the work-conserving nature of the scheduler and collected in the first idle interval.

Note that no external hardware controller is needed to implement the DPM algorithm, except for a simple timer (available in most of the processors) that is programmed by the processor itself before entering the low-power sleep state.

Assuming that task activations are retrieved in constant time, the complexity of the presented DPM algorithm is $O(1)$, making it suitable even for the simplest microprocessors.

C. Algorithm example

To better explain the proposed algorithm, let us consider a system with four speeds $s_1 = 0.3$, $s_2 = 0.6$, $s_3 = 0.7$ and $s_4 = s_m = 1.0$, power function $P(s) = 0.9s^3 + 0.1$ and break-even time $\delta = 10$. The task set consists of two periodic tasks, τ_1 and τ_2 characterized by $C_1 = 18$, $C_2 = 42$ (at $s = 1.0$), $T_1 = 60$, $T_2 = 150$ and hyperperiod of 300. For the sake of simplicity, preemption costs are considered negligible and $\forall \tau_i : \alpha_i = 0$.

The DVFS algorithm starts computing the critical speed $s^* = 0.4$, which lets us discard s_1 from the analysis. The first speed taken into account is s_2, at which computation times become $C_1(s_2) = 30$ and $C_2(s_2) = 70$, causing a negative blocking tolerance. Thus, the procedure considers next speed s_3, at which computation times become $C_1(s_3) = 26$ and $C_2(s_3) = 60$, the blocking tolerance is $\beta_{min} = 34$, and the task set is feasible. Thus, the algorithm stops. According to this configuration, τ_1 runs in a non-preemptive way for all its execution and τ_2 is split into two chunks of 26 and 34 units of time, respectively.

The task set execution at the slowest feasible speed without using the DPM algorithm is reported in Figure 5. Many idle intervals shorter than δ are present in the schedule, leading to a waste of energy.

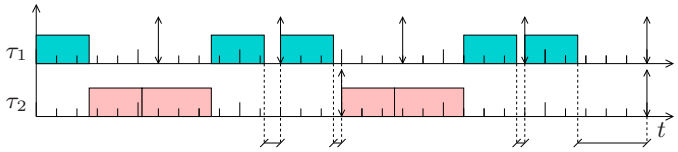

Fig. 5. Task execution at speed s_3 without using the DPM algorithm on a DVFS-sensitive architecture.

The advantage of introducing the DPM algorithm is shown in Figure 6. The algorithm is invoked for the first time at the end of the second job of τ_1 and all the small idle times are collected into a single longer interval, postponing the execution of the third job of τ_1 and the second job of τ_2.

A new instance of the DPM algorithm is launched before the end of the hyperperiod. Since β_{min} is longer than the break-even time δ, the processor switches to the sleep state.

For DPM-sensitive architectures, the only speed taken into account is the maximum one ($s = s^* = 1.0$) which leads to a

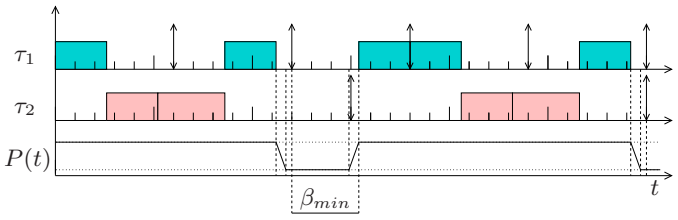

Fig. 6. Task execution at speed s_3 using the DPM algorithm on a DVFS-sensitive architecture.

feasible schedule. The two tasks contain only a single chunk each, meaning that they are executed in a non-preemptive way, with $\beta_{min} = 42$. As shown in Figure 7, scheduling the task set without using the DPM algorithm generates several idle intervals, which are compacted when the DPM algorithm is enabled (as depicted in Figure 8).

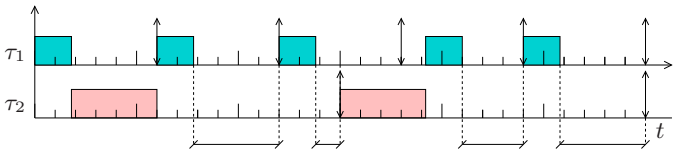

Fig. 7. Task execution at speed s_m without using the DPM algorithm on a DPM-sensitive architecture.

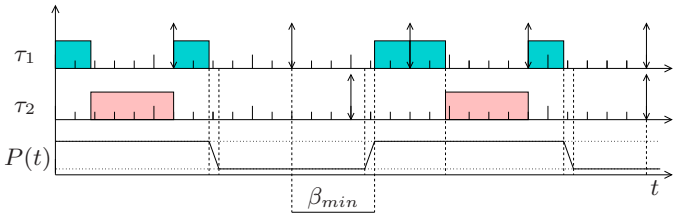

Fig. 8. Task execution at speed s_m using the DPM algorithm on a DPM-sensitive architecture.

VI. Experimental Results

This section presents a set of simulation experiments carried out for evaluating the proposed approach under different scenarios and with respect to different system features, such as preemption costs and power consumption models. Another set of experiments is aimed at comparing the proposed approach with the VOSS algorithm presented by Chen and Kuo [3].

The synthetic task sets used in the tests are composed of 10 periodic tasks randomly generated using the UUniFast algorithm [12], where the total utilization U is varied in a given range and each computation time $C_i^{NP}(s_m)$ is uniformly distributed in [100,500]. For the sake of simplicity, $\alpha_i = 0.2$ is set for each task. In these tests, relative deadlines are set equal to periods, which are derived once computation times and task utilizations have been generated. Tasks are scheduled under fixed priorities assigned with the Rate Monotonic algorithm and each simulation run is performed until the hyperperiod.

A processor with 19 discrete speeds has been considered, varying in the range of $[0.1, 1]$ with step 0.05.

Two power models have been considered in the experiments: $P^{(1)}(s) = 0.9s^3 + 0.1$ and $P^{(2)}(s) = 0.278s + 0.722$. The first one is often used in literature to model DVFS-sensitive architectures and is characterized by a critical speed $s^* = 0.4$. The second one represents the power consumption of a NXP LPC1768 (Arm Cortex M3), modeling a DPM-sensitive architecture with $s^* = s_m$ (making speed scaling not energy-convenient). The power consumed in the sleep state and the energy required for a complete state transition (from active to sleep and then back to active) are $P_\sigma^{(1)} = 0.05$, $P_\sigma^{(2)} = 0.4$, $E_\delta^{(1)} = 0.051 \cdot \delta$ and $E_\delta^{(2)} = 0.45 \cdot \delta$.

The experiments are divided in two parts: the first set shows the performance of the two phases of the proposed method under several scenarios, whereas the second set aims at comparing the proposed approach with the VOSS algorithm presented by Chen and Kuo [3].

A. Performance of the proposed approach

The first experiment aims at testing the impact of the DVFS algorithm under different scheduling approaches. In particular, the average lowest speed achieved by the limited preemptive scheduler is compared with the ones obtained by the fully preemptive and non-preemptive schedulers, for different task set utilizations and under different preemption costs. For space reasons, only two preemption costs are shown: $\xi = 0$ and $\xi = 10$. The second value represents a system with a preemption cost equal to one tenth and one fiftieth of the shortest and longest possible task execution (as $C_i^{NP} \in [100, 500]$), respectively. For each utilization, the average lowest speed was computed over 700 feasible task sets.

Figure 9 reports the resulting average lowest feasible speed as a function of the utilization factor. Note that, although the non-preemptive scheduler does not suffer from preemption overhead, it always requires the highest speed. Moreover, the speed found under fully-preemptive scheduling is always higher than or equal to the one under limited preemption, even with zero preemption cost. Such a benefit comes from the capability of limited preemptive schedulers of increasing the number of feasible task sets. Note that the introduction of a preemption cost $\xi = 10$ leads to a significant speed increase for the fully preemptive model, while its impact on the limited preemptive scheduler is almost negligible. Also observe that, for utilizations higher that $U = 0.9$, only the limited preemptive approach can achieve a feasible schedule for a significant number of generated task sets (at least one out of two). Finally, the plateau observed for $U > 0.92$ under the limited preemptive model is due to the fact that almost all the feasible task sets require a speed of one, while most generated task sets are discarded since their feasibility could be guaranteed only with a speed greater than the maximum one. This leads to an average value near to $s = 1$ for several utilization points.

The second experiment aims at evaluating the energy saved by the two-step algorithm with respect to the case in which the processor is kept always active at $s = 1$. The power model considered in this test is $P^{(1)}(s)$, representing a DVFS-sensitive architecture, since in DPM-sensitive architectures the

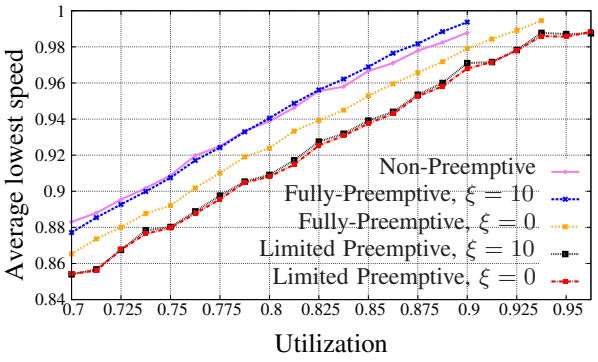

Fig. 9. Average lowest speed for different schedulers and preemption costs.

DVFS stage would not introduce any contribution, always returning the maximum speed.

Results are reported in Figure 10 for $\xi = 10$ and two values of δ (0 and 500 time units). The *Pure DVFS* curve represents the energy consumption obtained by using only the DVFS step. The introduction of the DPM stage allows a further energy reduction, which is about 8%, in the best case. Note that high break-even times push the curve closer to the one of *pure DVFS*, as the algorithm is not able to switch the processor off during all idle intervals. For the sake of completeness, the figure also shows the behavior of the algorithm under a *pure DPM* approach, where only the second stage is considered at the maximum speed. Since the power model is intrinsically speed scaling-convenient, DPM consumes more than the others, even assuming a null break-even time.

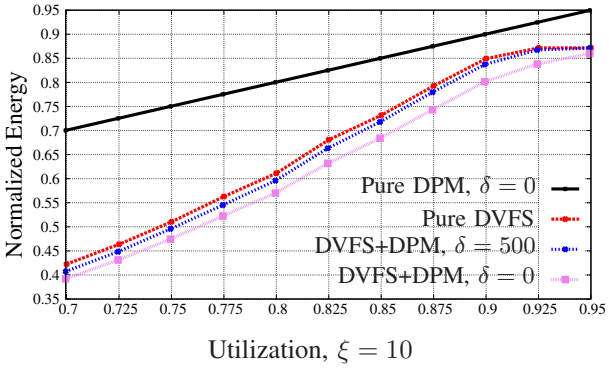

Fig. 10. Energy saved by our approach with respect to running always at the maximum speed $s = 1$.

B. Comparison with VOSS

A second set of experiments was carried out to compare the proposed approach against existing solutions available in the literature. Among the existing works that deal with fixed priority systems and that do not require additional hardware controllers, the algorithm that showed the best performance in terms of energy saving is VOSS, presented in [3]. We therefore decided to compare our algorithm only against VOSS, as the improvements over other existing solutions would be even greater. However, it is worth noticing that VOSS is an on-line algorithm with a complexity of $O(n \cdot \log(n))$ to be paid at each idle interval, whereas our online algorithm is $O(1)$. Finally, an improved version of VOSS is adopted that computes the feasible speed using the tighter Response Time Analysis [13] (including preemption costs) instead of Liu and Layland's bound [14].

Figure 11 shows the energy percentage saved by the proposed approach with respect to VOSS as a function of the total utilization, using the $P^{(1)}(s)$ model and for different preemption costs ($\xi \in \{0, 10\}$) and break-even times ($\delta \in \{250, 500, 750\}$).

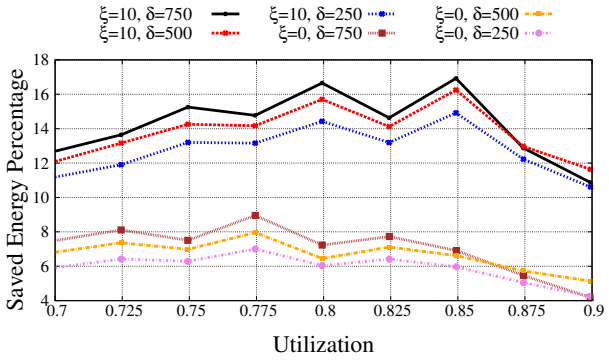

Fig. 11. Energy percentage saved by the proposed approach with respect to VOSS, for a DVFS-sensitive architecture.

As shown in the graph, the energy improvement of our approach (considering both DVFS and DPM algorithms), when the preemption cost is neglected, is around 7%. Using more realistic values for the preemption overhead (one tenth of the shortest task), the improvement increases up to 16%. This is a consequence of the reduced number of preemptions of our approach, which makes it even more competitive for higher preemption costs. The impact of the break-even time is smaller, although the longer δ, the higher the gain. The reason is that our method exploits longer blocking tolerances than VOSS, overcoming the break-even time limit more easily. When δ is either too long or too short, the performance of the DPM algorithms are equivalent. At high utilizations, the margin of improvement is barely usable, due to the reduced number of feasible task sets and the short idle time, so the behavior of the two algorithms is similar. Note that, for $\xi = 10$, the analysis ends at $U = 0.9$ as there are no feasible task sets under fully preemptive scheduling.

The two algorithms have also been compared under the second power model $P^{(2)}$, typical of DPM-sensitive architectures, under which the speed returned by our off-line DVFS algorithm is always equal to the maximum available (never exploiting the speed scaling feature), and the whole energy improvement is due to the DPM algorithm.

Figure 12 reports the improvements achieved by the proposed approach with respect to VOSS as a function of the total utilization, with $\delta \in \{250, 500, 750\}$ and $\xi \in \{0, 10\}$.

Note that the proposed approach always outperforms VOSS. Also, increasing the preemption cost leads to a higher improve-

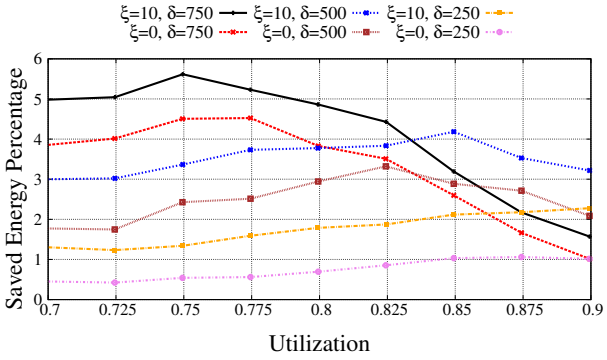

Fig. 12. Energy percentage saved by the proposed approach with respect to VOSS, for a DPM-sensitive architecture.

ment, as the context switch overhead of the fully-preemptive model becomes higher. For the same value of ξ, the higher δ, the higher the energy gain, as the blocking tolerance used to delay task execution under the limited preemption model is longer than that in fully preemptive mode. This happens up to a certain utilization, after which no algorithm is able to postpone task execution for a time longer than δ. When $U = 0.75$, our algorithm consumes almost 6% less than VOSS. The improvement would be even bigger when accounting for the additional number of on-line operations required by VOSS at every idle time ($O(n \log(n))$ instead of $O(1)$).

An additional experiment (not reported for space limitations) showed that by increasing α_i, the DVFS algorithm is able to reach slower speeds (as a more significant fraction of the task code is not affected by speed), so leading to higher savings than those reported in Figure 11. For DMP-sensitive platforms, the results shown in Figure 12 are not influenced by α_i as speeds lower than s_m are not taken into account.

VII. Related work

This section introduces the state of art concerning energy saving. DVFS algorithms are discussed first, followed by DPM approaches.

One of the first papers about power management exploiting frequency scaling was due to Yao et al. [15]. The authors proposed an off-line algorithm that, given a task set, computes the minimum energy schedule under the Earliest Deadline First scheduling (EDF) algorithm [14]. Then, they use an on line method to scale the speed according to the actual workload at every scheduling event. The analysis compares the efficiency of the algorithm with respect to different power models, but without taking switching overheads into account.

Aydin et al. [16] proposed three algorithms with growing complexity. The first one computes the lowest CPU speed such that the task set is schedulable under the assumption that all tasks execute for their WCET. The second algorithm (DRA) keeps track of the times at which a task is going to be dispatched. At runtime, if a task is dispatched earlier, the CPU is slowed down to prolong the execution until the original finishing time. The third algorithm (AGR) estimates the tasks completion times based on past instances and computes the lowest CPU speed to keep the task set feasible assuming that tasks execute for such estimates. However, since the estimations can be optimistic, the algorithm may speed the CPU up to recover from a task overrun.

The problem of obtaining an optimal frequency from a discrete frequency range was discussed by Bini et al. [17]. The authors provided a method for computing the optimal speed off-line (that could be unavailable in a specific architecture) and introduced a speed modulation technique to achieve the required speed using two discrete values. The analysis selects the pair of frequencies that minimizes energy consumption also considering switching overheads. Despite its innovative contribution, such an off-line approach does not take advantage of tasks early terminations to further reduce consumption.

Some authors [18], [19] reported that online DVFS techniques that frequently scale the execution speed may lead to transient faults. This problem was also addressed by Zhao et al. [20], who proposed a recovery allowance and an additional recovery task to run in case of fault. Another side effect of DVFS techniques was emphasized by Kim et al. [21], who noticed that such algorithms increase the number of preemptions, leading to a higher system utilization and, therefore, a higher energy consumption. To mitigate such a problem, they proposed two preemption control DVFS techniques.

The raising impact of leakage power in modern architectures, highlighted by Kim et al. [22] and empirically tested by Bambagini et al. [23], is driving the research on power management toward DPM techniques.

Lee et al. [24] proposed two leakage control algorithms for procrastinating tasks execution as long as possible, both under dynamic (LC-EDF) and fixed (LC-DP) priority scheduling. Using a dual priority scheme [25], LC-DP computes the longest delay (*promotion time*) each task can suffer still satisfying its deadline. The main difference with respect to the method proposed in this paper is that the critical speed is not taken into account (i.e., the system runs at the maximum speed) and the overhead introduced at runtime is much higher, due to the higher complexity of the online analysis.

Jejurikar et al. [26] proposed an approach (CS-DVS-P) based on critical speed analysis and task procrastination working for periodic tasks under EDF. First, an off-line DPM algorithm computes the maximum amount of time each task can spend in the sleep state within its period. Then, at run-time, sleep management is delegated to an external controller that switches the system off for the corresponding pre-computed time. Jejurikar and Gupta [27] extended the previous method to consider early terminations and fixed priority scheduling [28]. Our approach differs from the previous one in that sleep management does not require an external controller (we only assume the presence of an external interrupt handler), but it is launched when there are no tasks to execute. This allows automatically exploiting early terminations.

Chen and Kuo [3] showed that the DPM part of the algorithm in [28] may lead to deadline misses, and proposed some solutions to avoid such a problem. The first method simulates the execution of periodic tasks to compute the idle

time available until the next deadline. Then, such a time is used to postpone the task activations and switch the system into the sleep state. The other algorithms introduce the *virtual blocking*, which is the maximum blocking that tasks can suffer, but neglect preemption cost.

Awan and Petters [29] proposed to accumulate task execution slack to switch the processor off during such intervals under EDF, considering tasks with different criticality and several low-power states and different break-even times. However, tasks are always executed at the maximum speed.

VIII. Conclusions

This paper presented a new energy-aware algorithm that integrates DVFS and DPM techniques to further reduce energy consumption in embedded systems. It consists of an off-line DVFS analysis, for computing the most suitable speed (depending on architecture features and task set feasibility) and an online DPM algorithm, for prolonging sleep intervals by postponing task execution until the first job arrival plus the minimum blocking tolerance β_{min}. Moreover, the adoption of a limited preemptive scheduler in fixed priority systems allows extending the value of β_{min} and reducing the preemption cost with respect to fully preemptive schedulers.

With respect to other proposed algorithms [3] our method delays task executions rather than job arrivals, allowing a further reduction of the number of preemptions. Since the algorithm is invoked when the processor becomes idle, spare times due to early terminations are automatically reclaimed.

No extra hardware is required, except for a timer (active also when the processor is in sleep mode) needed to handle the wake-up events. Such timers are provided by most of the modern platforms, especially in the embedded system domain.

Concerning complexity, even though the off-line analysis is pseudo-polynomial, at run-time the algorithm has a constant complexity, $O(1)$, as it exploits only parameters obtained during the off-line stage. With respect to VOSS, whose complexity is $O(n \log(n))$ at every idle time, our algorithm requires a negligible runtime overhead, although such an overhead has not been considered in our simulation experiments.

As a future work, we plan to improve the DPM algorithm to consider dynamic parameters (such as variable procrastination delays) to further reduce energy consumption, while keeping low its complexity. Moreover, we aim at supporting also sporadic tasks which are common in embedded systems with event driven activations, such as sensors nodes.

References

[1] G. C. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems: a survey," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, Febrary 2013.

[2] T. Martin and D. Siewiorek, "Non-ideal battery and main memory effects on cpu speed-setting for low power," *IEEE Transactions on VLSI Systems*, vol. 9, no. 1, pp. 29–34, 2001.

[3] J.-J. Chen and T.-W. Kuo, "Procrastination for leakage-aware rate-monotonic scheduling on a dynamic voltage scaling processor," *SIGPLAN Notices*, vol. 41, no. 7, pp. 153–162, Jun. 2006.

[4] M. Bertogna and S. Baruah, "Limited preemption EDF scheduling of sporadic task systems," *IEEE Transactions on Industrial Informatics*, 2010.

[5] G. Yao, G. Buttazzo, and M. Bertogna, "Bounding the maximum length of non-preemptive regions under fixed priority scheduling," in *Proc. of the 15th IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA 2009)*, Beijing, China, 2009.

[6] A. Burns, "Preemptive priority based scheduling: An appropriate engineering approach," *S. Son, editor, Advances in Real-Time Systems*, 1994.

[7] R. Bril, J. Lukkien, and W. Verhaegh, "Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption," *Real-Time Systems: The International Journal of Time-Critical Computing*, vol. 42, no. 1-3, pp. 63–119, 2009.

[8] M. Bertogna, G. Buttazzo, and G. Yao, "Improving feasibility of fixed priority tasks using non-preemptive regions," in *Proceedings of the 2011 IEEE 32nd Real-Time Systems Symposium*, ser. RTSS '11, 2011.

[9] M. Bertogna, G. Buttazzo, M. Marinoni, G. Yao, F. Esposito, and M. Caccamo, "Preemption points placement for sporadic task sets," in *Proceedings of the 2010 22nd Euromicro Conference on Real-Time Systems*, ser. ECRTS '10, 2010.

[10] J. P. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," in *Proceedings of the Real-Time Systems Symposium (RTSS'89)*, Santa Monica, California, USA, 1989.

[11] M. Bertogna, G. C. Buttazzo, and G. Yao, "Improving feasibility of fixed priority tasks using non-preemptive regions," in *RTSS*, 2011.

[12] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Syst.*, vol. 30, no. 1-2, pp. 129–154, May 2005.

[13] S. Altmeyer, R. Davis, and C. Maiza, "Pre-emption cost aware response time analysis for fixed priority pre-emptive systems," University of York, techreport YCS-2011-464, May 2011.

[14] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the Association for Computing Machinery*, vol. 20, no. 1, pp. 46–61, January 1973.

[15] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced cpu energy," in *Proc. of the 36th Annual Symp. on Foundations of Computer Science*, ser. FOCS '95. IEEE Computer Society, 1995, pp. 374–382.

[16] H. Aydin, P. Mejía-Alvarez, D. Mossé, and R. Melhem, "Dynamic and aggressive scheduling techniques for power-aware real-time systems," in *Proc. of the 22nd IEEE Real-Time Systems Symp.*, ser. RTSS '01, 2001.

[17] E. Bini, G. Buttazzo, and G. Lipari, "Minimizing cpu energy in real-time systems with discrete speed management," *ACM Trans. Embed. Comput. Syst.*, vol. 8, no. 4, pp. 31:1–31:23, Jul. 2009.

[18] Y. Zhang and K. Chakrabarty, "Energy-aware adaptive checkpointing in embedded real-time systems," in *Proceedings of the conference on Design, Automation and Test in Europe*, ser. DATE '03, 2003.

[19] D. Zhu, R. Melhem, and D. Mosse, "The effects of energy management on reliability in real-time embedded systems," in *Proceedings of the International conference on Computer-aided design*, 2004.

[20] B. Zhao, H. Aydin, and D. Zhu, "Energy management under general task-level reliability constraints," in *Proceedings of the 2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, ser. RTAS '12, 2012.

[21] W. Kim, J. Kim, and S. L. Min, "Preemption-aware dynamic voltage scaling in hard real-time systems," in *Proceedings of the 2004 international symposium on Low power electronics and design*, 2004.

[22] Kim, T. Austin, J. S. Hu, and M. Jane, "Leakage current: Moore's law meets static power," *Computer*, 2003.

[23] M. Bambagini, F. Prosperi, M. Marinoni, and G. Buttazzo, "Energy management for tiny real-time kernels," in *Energy Aware Computing (ICEAC), 2011 International Conference on*. IEEE, 2011, pp. 1–6.

[24] Y.-H. Lee, K. P. Reddy, and C. M. Krishna, "Scheduling techniques for reducing leakage power in hard real-time systems," in *ECRTS'03*, 2003.

[25] R. Davis and A. Welling, "Dual priority scheduling," in *Proceedings Real Time Systems Symposium*, ser. RTSS '05, 1995.

[26] R. Jejurikar, C. Pereira, and R. K. Gupta, "Leakage aware dynamic voltage scaling for real time embedded systems," in *In Proc. of the Design Automation Conference*, 2004, pp. 275–280.

[27] R. Jejurikar and R. Gupta, "Dynamic slack reclamation with procrastination scheduling in real-time embedded systems," in *Proceedings of the 42nd annual Design Automation Conference*, ser. DAC '05, 2005.

[28] ——, "Procrastination scheduling in fixed priority real-time systems," in *In Proceedings of the Language Compilers and Tools for Embedded Systems*, 2004.

[29] M. A. Awan and S. M. Petters, "Enhanced race-to-halt: A leakage-aware energy management approach for dynamic priority systems," in *Proceedings of Euromicro Conference on Real-Time Systems*, 2011.