Power-Aware Real-Time Scheduling upon Identical Multiprocessor Platforms

Vincent Nélis*, Joël Goossens, Raymond Devillers, Dragomir Milojevic
Université Libre de Bruxelles (U.L.B.)
CP 212, 50 Av. F. D. Roosevelt,
1050 Brussels, Belgium
{vnelis, joel.goossens, rdevil, dragomir.milojevic}@ulb.ac.be

Nicolas Navet
LORIA - Equipe TRIO
Campus Scientifique - B.P. 239
54506 Vandoeuvre-lès-Nancy, France
nicolas.navet@loria.fr

Abstract

In this paper, we address the power-aware scheduling of sporadic constrained-deadline hard real-time tasks using dynamic voltage scaling upon multiprocessor platforms. We propose two distinct algorithms. Our first algorithm is an off-line speed determination mechanism which provides an identical speed for each processor. That speed guarantees that all deadlines are met if the jobs are scheduled using EDF. The second algorithm is an on-line and adaptive speed adjustment mechanism which reduces the energy consumption while the system is running.

1 Introduction

1.1 Context of the study

Some important applications impose temporal constraints on the response time while running on systems with limited power resource (such as real-time communication in satellites). As a result, the research community has investigated during the past 15 years the low-power system design. Actually, the dynamic voltage scheduling (DVS) framework became a major concern for power-aware computer systems. This framework consists in minimizing the system energy consumption by adjusting the working voltage and frequency of the CPU. For real-time systems, this DVS framework focuses on minimizing the energy consumption while respecting all the timing constraints.

Many power-constrained embedded systems are built upon multiprocessor platforms because of high-computational requirements and because multiprocessing often significantly simplifies the design. As pointed out in [4], another advantage is that multiprocessor systems are more energy efficient than equally powerful uniprocessor platforms, because raising the frequency of a single processor results in a *multiplicative* increase of the consumption while adding processors leads to an *additive* increase.

1.2 Problem definition

In the following, we consider the problem of minimizing the energy consumption needed for executing a set of sporadic constrained-deadline real-time tasks scheduled upon a fixed number of *identical* processors. The scheduling is preemptive and uses the global EDF policy [15]. "Global" scheduling algorithms, on the contrary to partitioned algorithms, allow different instances of the same task (also called jobs or processes) to be executed upon different processors. Each process can start its execution on any processor and may migrate at run-time from one processor to another if it gets meanwhile preempted by smaller-deadline processes.

We first tackle the problem of choosing the smallest (or so) processor frequency for the set of CPUs, such that all deadlines will be met. The procedure is performed off-line (i.e., before the system starts its execution) and provides a static result in the sense that the computed speed does not change over time. Such a static solution is sufficient to significantly reduce the energy consumption; however, due to the discrepancy between Worst-Case Execution Time (WCET) and Actual-Case Execution Time (ACET) [11], it

[1]Supported by the Belgian National Science Foundation (FNRS) under a FRIA grant.

usually leads to pessimistic results. In a second step, we thus propose an on-line scheme that takes advantage of unused CPU slots to further reduce the energy consumption.

1.3 Previous work

There is a large number of researches about *uni*processor energy-aware scheduling but much less for the multiprocessor case, where low-power scheduling problems are often NP-hard when the actual applicative constraints are taken into account (see [7] for a starting point). Among the most interesting studies, one can cite [14] where the authors provide power-aware scheduling algorithms for bag-of-tasks applications with deadline constraints on DVS-enabled cluster systems. A study particularly relevant to the DVS framework is [6] which targets energy-efficient scheduling of periodic real-time tasks over multiple DVS processors with the considerations of power consumption due to leakage current (i.e. the static part of the energy dissipation). In [8], the authors propose a set of multiprocessor energy-efficient task scheduling algorithms with different task remapping and slack reclaiming schemes, where tasks have the same arrival time and share a common deadline. A large number of such "slack reclaiming" approaches have been developed over the years for the *uni*processor case. Among those, some strategies dynamically collect the unused computation times at the end of each job and share it among the remaining active jobs. Examples of algorithms following this "reclaiming" approach, include the ones proposed in [19, 16, 21, 3]. Some reclaiming algorithms even anticipate the early completion of tasks for further reducing the CPU speed [16, 3], some having different levels of "aggressiveness" [3].

1.4 Contribution of the paper

Unlike the work considered in [4], we study the case where the number of processors is already fixed. This constraint can be imposed by the availability of hardware components, by design considerations not related to power-consumption. Notice that in practical situations, the task characteristics are unknown at (hardware) design time.

The first contribution of this paper, is based on [13], and provides a technique which determines the minimum off-line processor speed for the fixed and identical multiprocessor platform using EDF.

The second, and the main contribution of this document, is a slack reclaiming algorithm which is, to the best of our knowledge, the first of its kind for the global preemptive scheduling problem of distinct-deadlines tasks on multiprocessor platforms. This contribution can be considered as an extension to the multiprocessor case of a previous proposal of Shin and Shoi in [19], which is usually referred to as "One Task Extension" (OTE). We proved that our on-line proposal does not jeopardize the system feasibility.

Organization of the paper. The document is organized as follows: in Section 2, we introduce our model of computation, in particular our task model; in Section 3, we present our off-line processor speed determination; in Section 4, we present our on-line speed reduction technique; in Section 5, we present our experimental results; in Section 6, we consider our future works and in Section 7, we conclude.

2 Model of computation

2.1 Application model

We consider in this paper the scheduling of *sporadic constrained-deadline tasks*, i.e., systems where each task $\tau_i = (C_i, D_i, T_i)$ is characterized by three parameters – a worst-case execution requirement (WCET) denoted C_i, a minimal inter-arrival delay T_i and a deadline $D_i \leq T_i$ – with the interpretation that the task generates successive *jobs* $\tau_{i,j}$ (with $j = 1, 2, \ldots, \infty$) arriving at times $e_{i,j}$ such that $e_{i,j+1} - e_{i,j} \geq T_i$, each such job has an execution requirement of at most C_i execution units, and must be completed by its deadline noted $D_{i,j} = e_{i,j} + D_i$. We therefore assume that the worst-case execution time is always lower than the deadline, i.e. $C_i \leq D_i$. We assume that preemption is allowed – an executing job may be interrupted, and its execution resumed later (may be upon another processor), with no loss or penalty. Let $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ denotes a sporadic task system. For each task τ_i, we define its *density* λ_i as the ratio of its execution requirement to its deadline: $\lambda_i \overset{\text{def}}{=} C_i/D_i$. Since $C_i \leq D_i$ we have that $\lambda_i \leq 1$. We also define the *total density* $\lambda_{\text{sum}}(\tau)$ of sporadic task system τ as $\lambda_{\text{sum}}(\tau) \overset{\text{def}}{=} \sum_{i=1}^{n} \lambda_i$, and its *maximal density* as $\lambda_{\max}(\tau) \overset{\text{def}}{=} \max_{\tau_i \in \tau} \lambda_i$. Without loss of generality, we assume in the remainder of the paper that $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$, and consequently $\lambda_{\max}(\tau) = \lambda_1$.

2.2 Platform model

In our platform model, a processor can dynamically adapt its working frequency in some continuous range $[f_{\min}, f_{\max}]$. The case where the number of frequencies is finite can be addressed as in [12]. In the remainder of this paper, we denote by $s(t)$ the processor speed at any time-instant t. The processor speed $s(t)$ is defined as the ratio of its current functioning frequency (say $f(t)$) over the maximal frequency $f_{\max}$, i.e.: $s(t) \overset{\text{def}}{=} \frac{f(t)}{f_{\max}}$, with $f_{\min} \leq f(t) \leq f_{\max}$. Notice that the processor speed always lies between $\frac{f_{\min}}{f_{\max}}$ and 1, whatever the values of $f_{\min}$

and $f_{\max}$, and to each speed corresponds exactly one frequency.

We consider in this document multiprocessor platforms composed of a known and fixed number m of identical processors $\{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_m\}$ upon which a set of real-time tasks is scheduled. The working power of each processor may be characterized by its speed (or computing capacity) s – with the interpretation that a job that executes on a processor of speed s for R time units completes $s \times R$ units of execution. The minimal and maximal admissible speed of all processors are identical and are denoted by $s_{\min} \stackrel{\text{def}}{=} \frac{f_{\min}}{f_{\max}} > 0$ and $s_{\max} \stackrel{\text{def}}{=} \frac{f_{\max}}{f_{\max}} = 1$, respectively. Since we assume that the range of available frequencies is continuous between $f_{\min}$ and $f_{\max}$, the speed of the processors can take any real value between $s_{\min}$ and $s_{\max}$ at every instant. Notice that the task computing requirements (C_i's) are defined for the maximal speed $s_{\max}$.

In Section 3 we assume that all the processors share a common speed which is fixed before the system starts its execution. This speed does not change during the scheduling and thus, *we will use the notation s instead of $s(t)$ to simplify the presentation.* Then, we study the case in Section 4 where each processor may run at a different speed and may change it at any time during the scheduling. In our work, speed assignments are determined at job-level: voltage/speed changes only occur at job dispatching instants. That is, once a job is assigned to a CPU, the CPU speed is fixed until the job is preempted or completed.

3 Off-line speed determination

3.1 Introduction

Off-line processor speed determination is the process of determining, during the design of the real-time application, the lowest processor speed s in order to schedule the sporadic task set τ upon an identical multiprocessor platform with m processors running at speed s. In this Section, we consider the case where, at any instant, all processors must be running at the same speed noted s. We shall use the following result:

Theorem 1 (Bertogna, Cirinei and Lipari [5]). *Any sporadic constrained-deadline task system τ satisfying*

$$\lambda_{\text{sum}}(\tau) \leq m - (m-1) \cdot \lambda_{\max}(\tau)$$

is schedulable by the EDF algorithm upon a platform with m identical processors.

Then, we get the following sufficient feasibility condition:

Corollary 1. *A sporadic constrained-deadline task system τ is EDF-schedulable upon an identical multiprocessor platform with m processors running at speed s if:*

$$s \geq \lambda_{\max}(\tau) + \frac{\lambda_{\text{sum}}(\tau) - \lambda_{\max}(\tau)}{m} \qquad (1)$$

Notice that, from the expression (1) (which is a sufficient condition), s is always greater or equal to $\lambda_{\max}(\tau)$, which is a necessarily condition to ensure the system schedulability, whatever the scheduling algorithm.

3.2 Algorithm EDF$^{(k)}$

Following an idea from [13], but adapted to our offline speed determination where the number of processors is *fixed*, we shall present an improvement on the speed needed in order to schedule sporadic task sets.

Algorithm EDF$^{(k)}$ (Goossens, Funk and Baruah [13]): Assuming that the task indexes are sorted by non-increasing order of task densities and $1 \leq k \leq m$, EDF$^{(k)}$ assigns priorities to jobs of tasks in τ according to the following rules:

For all $i < k$, tau_i jobs are assigned the highest priority (ties are broken arbitrarily).

For all $i \geq k$, τ_i jobs are assigned priorities according to EDF (ties are again broken arbitrarily).

That is, Algorithm EDF$^{(k)}$ assigns the highest priority to jobs generated by the $(k-1)$ tasks in τ that have highest densities, and assigns priorities according to deadlines to jobs generated by all other tasks in τ (thus, "pure" EDF is EDF$^{(1)}$). We show in the following that we get another lower-bound for the speed s when using EDF$^{(k)}$ instead of EDF, and this bound is always lower than (or equal to) the one provided by Expression (1). But first, we introduce the notation $\tau^{(i)}$ to refer to the task system composed of the $(n-i+1)$ minimum-density tasks in τ: $\tau^{(i)} \stackrel{\text{def}}{=} \{\tau_i, \tau_{i+1}, \ldots, \tau_n\}$; (according to this notation, $\tau \equiv \tau^{(1)}$).

Theorem 2. *Any sporadic constrained-deadline task system τ is EDF$^{(k)}$-schedulable upon an identical multiprocessor platform with m processors at speed s_k if $s_k \geq \max\{\lambda_1, \lambda_k + \frac{\lambda_{\text{sum}}(\tau^{(k+1)})}{m-k+1}\}$*

Corollary 2. *A sporadic constrained-deadline task system τ is schedulable upon m processors at speed s_{ol} by EDF$^{(\ell)}$, with*

$$s_{\text{ol}} \stackrel{\text{def}}{=} \max\{\lambda_1, \min_{k=1}^{m}\{\lambda_k + \frac{\lambda_{\text{sum}}(\tau^{(k+1)})}{m-k+1}\}\} \qquad (2)$$

and ℓ is the parameter minimizing the speed s_{ol} of s_k.

Proof. The proof is a direct consequence of Theorem 2. □

It may be seen that this expression always yields a better bound than Inequality (1).

3.3 Implementation

A more detailed description of our off-line speed determination mechanism is given by Algorithm 1. Let s_{ol} denote the returned speed, defined by Expression (2). Before applying this algorithm, we assume that the number of processors is sufficient to schedule the system τ at the maximal speed. Consequently, the speed s_{ol} is initially set to $s_{\max}$ (line 3). Then, the algorithm searches the minimal speed by sweeping the value of k between 1 and m (line 4 to line 13). Finally, in order that $\mathsf{EDF}^{(k)}$ assigns the highest priorities to the $(k-1)$ tasks that have highest densities, we set the deadline of these tasks to $-\infty$ (line 14).

Algorithm 1: Off-line speed determination

```
    Input: $\tau$, $m$, $s_{\max}$, $s_{\min}$
    Output: $s_{\mathrm{ol}}$
 1  begin
 2      $k_{\mathrm{opt}} := 1$;
 3      $s_{\mathrm{ol}} := s_{\max}$ ;
 4      $s_{\mathrm{limit}} := \max\{s_{\min}, \lambda_1\}$ ;
 5      for ($k := 1$ ; $k \le m$ and $s_{\mathrm{ol}} > s_{\mathrm{limit}}$ ; $k := k+1$) do
 6          $s := \max\{\lambda_1, \lambda_k + \frac{\lambda_{\mathrm{sum}}(\tau^{(k+1)})}{m-k+1}\}$ ;
 7          if ($s < s_{\mathrm{ol}}$) then
 8              $s_{\mathrm{ol}} := s$ ;
 9              $k_{\mathrm{opt}} := k$ ;
10              if ($s_{\mathrm{ol}} < s_{\mathrm{limit}}$) then $s_{\mathrm{ol}} := s_{\mathrm{limit}}$ ;
11      foreach $\tau_i \in \left\{\tau_1, ..., \tau_{k_{\mathrm{opt}}-1}\right\}$ do $D_i := -\infty$ ;
12      return ($s_{\mathrm{ol}}$) ;
13  end
```

4 Multiprocessor One Task Extension

4.1 Introduction

In this section, we consider the case where processors still share the same minimal and maximal speeds $s_{\min}$ and $s_{\max}$, but each one may run at its own execution speed during the scheduling. We assume that, when a processor is idle, its execution speed is always fixed to the minimal common speed $s_{\min}$. We propose a low-complexity on-line algorithm that aims to further reduce the speeds of the CPUs by performing "local" adjustments, when it is safe to reduce the speed below s_{ol} defined by Equation (2).

We term our technique MOTE for Multiprocessor One Task Extension, since it is a *multiprocessor* version of the technique proposed in [19] and usually referred to as OTE. The idea is the following: the speed of a CPU can safely be reduced below the speed s_{ol} during the execution of a job if the reduced speed does not change anything with respect to the schedule of the subsequent jobs scheduled on that CPU. More precisely, subsequent jobs will not be delayed by more (nor less) higher-priority workload than with s_{ol}.

4.2 Notations

We denote by t the *current time* in the schedule and by $B_i(t)$ the *last release time* of τ_i before or at time t, with $B_i(0)$ initially set to $-T_i$ (see Equation 3 to understand this initialization). During the scheduling, $B_i(t)$ is updated at each time t a job is released by τ_i. The ready queue, denoted by *ready-Q*, holds all the pending jobs (i.e. ready to be executed but waiting for a CPU) sorted according to the $\mathsf{EDF}^{(k)}$ rule, where ties are broken according to an arbitrary rule; recall that using $\mathsf{EDF}^{(k)}$, the priorities of the jobs are *constant*. In the following, s_i denotes the processor speed for the job $\tau_{i,j}$ at time t. We shall use the following functions.

The function $\mathcal{A}_i(t,t')$ indicates *if the sporadic task τ_i may generate a job at time $t' \geq t$*. Since T_i denotes the minimal inter-arrival delay between job releases of the sporadic task τ_i, we get:

$$\mathcal{A}_i(t,t') \stackrel{\mathrm{def}}{=} \begin{cases} 1 & \text{if } t' \geq B_i(t) + T_i \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Notice that $B_i(0)$ is initially set to $-T_i$ in order to have $\mathcal{A}_i(0,0) = 1$ since our task model considers that each task may release its first job at time $t = 0$.

Then, the function $\mathrm{PotAct}_i(t,t')$ (for **Pot**entially **Act**ive at time t') indicates *if τ_i has an active job at time t which may still be active at time t'*. This function returns 1 only if τ_i is active at time t *and* if t' is not larger than the deadline of this job:

$$\mathrm{PotAct}_i(t,t') \stackrel{\mathrm{def}}{=} \begin{cases} 1 & \text{if } \omega_i^{s_i}(t) > 0 \text{ and} \\ & \quad t \leq t' < B_i(t) + D_i \\ 0 & \text{otherwise} \end{cases}$$

where $\omega_i^{s_i}(t)$ denotes the *remaining worst-case execution requirement* of the last released job of τ_i if executed at speed s_i (if a job is done, its ω is set to zero, even if the WCET is not exhausted).

Theorem 3. *The function*

$$\Pi(\tau_{u,v}, t, t') \stackrel{\mathrm{def}}{=} m - \sum_{\tau_i \in \tau \setminus \{\tau_u\}} \mathrm{PotAct}_i(t,t') - \sum_{\tau_i \in \tau} \mathcal{A}_i(t,t')$$

if non-negative, provides a lower bound of the number of available CPUs at time $t' \geq t$, when ignoring the schedule of the current job of τ_u (if any).

Corollary 3. *At each time t where a job $\tau_{u,v}$ is allocated to CPU $\mathcal{P}_\ell$, the earliest future time instant in the schedule such that $\mathcal{P}_\ell$ may be required by another job (possibly from the same task) is given by:*

$$t_{\mathrm{next}} = \begin{cases} \min\{t' \geq t \mid \Pi(\tau_{u,v}, t, t') \leq 0\} & \textit{if } m \leq n \\ +\infty & \textit{otherwise} \end{cases}$$

4.3 MOTE scheme

EDF$^{(k)}$ is a job-level fixed-priority consequently a job executed on a CPU can only be preempted upon its completion or the release of a (higher priority) job. In our scheme, the speed reduction of a job is decided when the job is allocated to a CPU, for the first time or when it resumes after being preempted. Upon its release, a job is inserted into the *ready-Q* if it cannot receive a processor (i.e. all processors are used and the job is of lower priority). We do not make any assumptions on the CPU allocation rule when several CPUs are available for a single job. For instance, free CPUs can be granted according to the rule "smaller CPU index first."

Since we consider multiprocessor platforms, we know that we have to be very careful to any change in the original schedule because of scheduling anomalies. We say that a scheduling algorithm suffers from anomalies if a change which is intuitively positive in a schedulable system can turn it unschedulable. An "intuitively positive change" is a change which seems to help the scheduling, like reducing the density of a task (by increasing its period or reducing its execution requirement) or advancing the start-time of a job; this can also be an increase of the number of processors on the platform. Unfortunately, multiprocessor platforms are subject to scheduling anomalies [2]. For that reason, our on-line low-power mechanism only focuses on the last allocated-job and avoids to change the schedule of the other jobs.

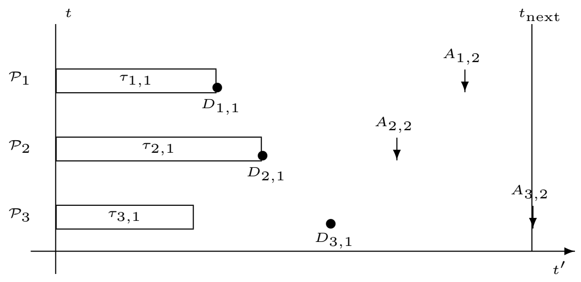

Figure 1. Illustration of a 3-task system.

Figure 1 illustrates the main idea of our on-line algorithm when 3 tasks are scheduled upon 3 processors at speed s_{ol}. This example shows a schedule where t is the current time, $\tau_{1,1}$, $\tau_{2,1}$ and $\tau_{3,1}$ are the active jobs at time t (the ready-queue is empty since there are only three tasks in the system) and plain circles and vertical arrows represent the deadlines and the (earliest) arrival times (since tasks are sporadic) of each task, respectively. Suppose that $\tau_{1,1}$ and $\tau_{1,2}$ are allocated to $\mathcal{P}_1$ and $\mathcal{P}_2$. Before allocating $\tau_{3,1}$ to the processor $\mathcal{P}_3$, we see that $\mathcal{P}_3$ cannot be required by another job than $\tau_{3,1}$ until time t_{next}. Indeed, $\tau_{1,2}$ and $\tau_{2,2}$ could be assigned (if they arrive at time $A_{1,2}$ and $A_{2,2}$) to the CPUs $\mathcal{P}_1$ and $\mathcal{P}_2$ since the system feasibility ensures that $\tau_{1,1}$ and $\tau_{2,1}$ will be completed by their deadline. Consequently, when ignoring the schedule of $\tau_{3,1}$, we see that t_{next} is the earliest time instant (after the time t) such that all processors may be required. Indeed, t_{next} is the earliest time instant after time t such that $\Pi(\tau_{3,1}, t, t_{\text{next}}) = 3 - 0 - 3 = 0$.

Since t_{next} is *the earliest time instant* (after the current time t) such that $\mathcal{P}_3$ may be required by another job than $\tau_{3,1}$ (assuming that all the other active jobs are scheduled on other processors), one can conclude that $\mathcal{P}_3$ will only execute the job $\tau_{3,1}$ between time instants t and t_{next}. That is, we proved that $\mathcal{P}_3$ can modify its working speed in such a way that $\tau_{3,1}$ completes in the worst-case at time $\min\{D_{3,1}, t_{\text{next}}\}$ (or earlier if s_{min} imposes it).

Principle: Our on-line power-aware algorithm deals with a priority rule that assigns a constant priority to each job. In this work, these priorities are determined by the algorithm EDF$^{(k)}$. Our power-aware algorithm is only applied when a job $\tau_{i,j}$ is to be allocated to a CPU $\mathcal{P}_\ell$ at time t during the scheduling, which corresponds to its arrival or to the completion of a higher priority job. At this time, our method determines the earliest time instant t_{next} such that $\mathcal{P}_\ell$ may be needed by another job. The function $\Pi(\tau_{i,j}, t, t')$ (based on the deadlines of the jobs currently executing) is used to sweep the task set (with a running time linear in the number of tasks). Notice that the function $\Pi(\tau_{i,j}, t, t')$ could be evaluated *only* at the deadline-times of the jobs currently under execution and at the next (possible) arrival-time of every task (since between these instants, the function $\Pi(\tau_{i,j}, t, t')$ is constant). It follows from Corollary 3 that $\mathcal{P}_\ell$ will not execute another job than $\tau_{i,j}$ until the time instant t_{next}. The speed for $\tau_{i,j}$ can be safely reduced in such a way that it completes at time $\min\{D_{i,j}, t_{\text{next}}\}$ (if the corresponding speed is lower than the current one). Obviously, the working speed of a processor can never be reduced under $s_{\min}$.

Algorithm 2: Determination of t_{next}

```
Input: $t$, $\tau_i$
Output: $t_{\text{next}}$
begin
    $n_a$ := number of active tasks at time $t$ ;
    $L$ := set of the next deadline and possible arrival-time of
    each task, sorted by increasing order of the occurring time ;
    $t_{\text{next}} := t$;
    $\Pi := m - (n_a - 1)$;
    while ($\Pi > 0$ and $L \neq \phi$) do
        $e \leftarrow$ L.top();
        $t_{\text{next}}$ := e.occurring_time ;
        if (e.task $\neq \tau_i$) and (e.type == deadline) then
        $\Pi := \Pi + 1$;
        else if (e.type == arrival) then $\Pi := \Pi - 1$;
        L.pop() ;
    return $t_{\text{next}}$;
end
```

Let s_i denote the processor speed of the active job $\tau_{i,j}$.

Algorithm 3: Speed-allocation to $\tau_{i,j}$ at time t

```
Input: $\tau_{i,j}$
Output: $\phi$
begin
    // Initialization step
    if ($\tau_{i,j}$ is allocated for the first time) then
        if ($i < k$) then $s_i := \lambda_i$;
        else $s_i := \lambda_k + \frac{\lambda_{\text{sum}}(\tau^{(k+1)})}{m-k+1}$ ;
    // MOTE step
    if ($m \leq n$) then $t_{\text{next}} :=$ Call Algorithm2($t$, $\tau_i$) ;
    else $t_{\text{next}} := \infty$ ;
    if ($t_{\text{next}} > t$) then
        $s_i := \min\{s_i, \frac{\omega_i^{s_i}(t) \cdot s_i}{\min\{D_{i,j}, t_{\text{next}}\} - t}\}$ ;
        if ($s_i < s_{\min}$) then $s_i := s_{\min}$ ;
        $\tau_{i,j}$ is allocated to any available CPUs ;
        The speed of the designated CPU is fixed to $s_i$ ;
    else No speed reduction can occur. The EDF$^{(k)}$ rule
    applies; $\tau_{i,j}$ either preempts the lowest priority job
    currently under execution or is allocated to any available
    CPU, and the processor speed is fixed to $s_i$. ;
end
```

This speed s_i is initialized when $\tau_{i,j}$ is released. In a simple version of the MOTE technique, the execution speed of every released job is initially set to s_{ol}, since we assume that the priorities are assigned by EDF$^{(k)}$ and we proved that the system feasibility is guarantee when it is scheduled by EDF$^{(k)}$ at speed s_{ol} (Theorem 2). However, we adopt here another initialization step in order to profit from the individual speed of each processor. In this "optimized" initialization step, two cases may arise at the arrival of the job $\tau_{i,j}$:

1. if $\tau_i \in (\tau \setminus \tau^{(k)})$ (the set of the $(k-1)$ tasks with highest densities), s_i is fixed to λ_i.
2. if $\tau_i \in \tau^{(k)}$, s_i is fixed to $\lambda_k + \frac{\lambda_{\text{sum}}(\tau^{(k+1)})}{m-k+1}$.

We proved that all deadlines are met when the system is scheduled while using this rule. Then, when the job $\tau_{i,j}$ is to be allocated to a CPU during the scheduling, we determine the earliest time instant t_{next} such that $\Pi(\tau_{i,j}, t, t_{\text{next}}) \leq 0$ and if $t_{\text{next}} > t$, one has:

$$s_i := \min\left\{s_i, \frac{\omega_i^{s_i}(t) \cdot s_i}{\min\{D_{i,j}, t_{\text{next}}\} - t}\right\} \tag{4}$$

We proved also that the system feasibility is not jeopardized by this speed modification.

4.4 Implementation

Before the system starts its execution, our algorithm computes the speed s_{ol} by determining the optimal value of k thanks to Equation (2) (see Algorithm 1). Then, while the system is running, there is only one kind of situation where the decision to reduce or not the CPU speed for a job $\tau_{i,j}$ is taken: when it is allocated to an available CPU (upon its release, or when it is waiting for an available processor at the head of the ready-Q and a job terminates its execution). A detailed description of the applied procedure at any allocation time is given in Algorithm 3. Algorithm 2 shows how to compute t_{next} with a *linearithmic* (also called *quasilinear*) worst-case computing complexity O($n \cdot \log(n)$), where n is the number of tasks.

It worth noting that the MOTE step (see Algorithm 3) is applied at most once to each job (and only if $i > k$); indeed, a job whose speed has been changed by this step will not be preempted in the future and thus will not be (re-)stored in the *ready-Q* before its end of execution. However, when the speed of a job (with a normal priority) is initialized but not modified by the MOTE step at its arrival, it can possibly be reduced by the MOTE step in the future, if the job is at the head of the *ready-Q* and another job completes its execution. Section 5 shows that the MOTE algorithm indeed *significantly* improves the energy consumption of a real-time sporadic system.

5 Experiments

5.1 Introduction

In our simulations, we have scheduled *periodic constrained-deadline systems* (i.e., T_i is here the exact inter-arrival delay for each task τ_i). The energy consumption of each generated system is computed by simulating the three methods described in this paper during one hyper-period (i.e. the least common multiple of the task periods); indeed, the authors of [9] show that, for the specific case of synchronous periodic task systems, the schedule repeats from the origin with a period equals to the hyper-period. The three methods are: the off-line speed reduction for EDF (Equation (1)), the off-line speed reduction for EDF$^{(k)}$ (Equation (2)) and the MOTE algorithm (combined with EDF$^{(k)}$). The energy consumptions generated by these three methods are compared with the consumption by the $S_{\max}$ method (i.e. all jobs are executed at the maximal processors speed $s_{\max}$), while using different processor models. During our simulations, about 5000 constrained-deadline systems were generated and simulated; with the number of tasks n in $[5, 40]$ (with density below 1 and $\lambda_{\text{sum}}(\tau)$ between 1 and 10). During each simulation, the ACET of each job was generated using a pseudo-random generator. We made many graphics from our results, but they are omitted here due to space limitation. To ensure that the number m of processors is sufficient to schedule the generated systems at speed $s_{\max}$, m is determined by the following Equation (from [13]):

$$m := \min \left\{ n, \left\lceil \frac{\lambda_{\text{sum}}(\tau) - \lambda_{\max}(\tau)}{1 - \lambda_{\max}(\tau)} \right\rceil \right\}$$

5.2 Processor models

In our experiments, we used two realistic processor models. These models, noted P1 and P2 in the following, are derived from the processor Crusoe TM5400 from Transmeta and the processor StrongARM SA-1100 from Intel, respectively. In these two processor models, the voltage can only vary in a limited range. Moreover, only a fixed number of functioning frequencies/voltages are available. For that reason, we use the available processor speed immediately above the desired one, if the latter is not available. Note that the use of the two adjacent frequencies to the requested frequency is more efficient from an energy point of view (see, for instance, [12]). Table 1 (adopted from [17] and [20]) summarizes the relationship between frequency, voltage, power consumption and the corresponding speed for the Transmeta TM5400 (P1) and the StrongARM SA-1100 (P2).

CPU	Freq. (MHz)	Volt. (V)	Power (%)	Speed
P1	700	1.65	100	1
	600	1.60	80.59	0.857
	500	1.50	59.03	0.714
	400	1.40	41.14	0.571
	300	1.25	24.60	0.429
	200	1.10	12.70	0.286
P2	206	1.50	100	1
	195	1.42	78.9	0.947
	180	1.30	63.2	0.874
	165	1.20	50.0	0.801
	150	1.15	39.9	0.728
	135	1.10	33.6	0.655
	120	1.08	33.0	0.583
	105	0.95	19.8	0.510
	90	0.90	15.0	0.437
	75	0.82	11.8	0.364
	60	0.80	9.44	0.291

Table 1. Processors characteristics.

Tables 2 provides the average consumption profit generated by each method (expressed in percent), compared to the consumption using the $S_{\max}$ method over the entire simulation.

5.3 Observations

We observe a large variation in the power saving of our algorithms when they are simulated upon the Crusoe processor and upon the StrongARM SA-1100. This variation is due to the difference in the shape of their consumption function: the consumption function of the StrongARM processor has a higher curvature than the Crusoe processor. That is, a speed reduction in the StrongARM implies a more significant reduction of the system energy consumption. This reduction is therefore *even more significant* when we use the standard dynamic consumption model where the power consumption function is modeled as a constant plus a cubic function (or at least a quadratic function) of the speed [22]. However, our results for this theoretical case are omitted due to the space limitation.

According to [18], the Crusoe processor performs a speed transition less than 20 μs. This time overhead is negligible for most real-time systems, since the order of magnitude of the task characteristics is about few milliseconds. With the Strong ARM SA-1100 processor, Pouwelse et al. [17] report that a voltage/speed change can be performed in less than 140 μs. If this may not be considered as negligible, since we have at most two speed transitions for each job (one initially and one for a MOTE step), the "voltage change overheads" can be incorporated into the worst-case execution requirement.

results with the StrongARM SA-1100 processor		
Method name	**Power saving over $S_{\max}$**	**Standard deviation**
offline EDF	4.33 %	3.34
offline EDF$^{(k)}$	27.12 %	10.24
MOTE	44.74 %	8.82

results with the Crusoe processor		
Method name	**Power saving over $S_{\max}$**	**Standard deviation**
offline EDF	0.62 %	0.76
offline EDF$^{(k)}$	5.91 %	4.38
MOTE	23.3 %	7.55

Table 2. Simulation results.

6 Future works

Currently this work addresses the impact of the proposed scheduling algorithms only on the dynamic power component of the overall microprocessor power dissipation. Proposed methods do not take into account the power dissipated to hold the circuit state and/or power dissipation due to the imperfections of the physical implementation (static power dissipation component). However it is a very well known fact that for integrated circuits manufactured with technologies below 130 nm, and especially with current 90 nm and 65 nm technologies, the static power dissipation component becomes very important and comparable to the dynamic power dissipation [10]. A significant research effort has been provided, and is still deployed on the static power dissipation reduction techniques. Proposed methods target not only low-level, hardware actions (such as clock gating) but also higher-level (operating system) actions forcing the processor to enter one of the multiple low-power dissipation modes for better trade-off between power saving and wake-up time (see [1] as an example).

The problem of the increased static power dissipation of the sub-micron technologies is the main motivation for our future work, in which we will extend the existing controllable parameters of our scheduling algorithms (voltage and frequency) with a processor switch-off parameter.

7 Conclusion

In this paper, we proposed two approaches which reduce the energy consumption for real-time systems implemented upon multiprocessor platforms. The first one is an adaptation of the first proposal "Global EDF", called $\mathsf{EDF}^{(k)}$, which allows a lower computing speed of the processors than EDF. The second proposal (called MOTE) is an online low-power algorithm which takes into account the "unused" CPU times to adjust the processor speeds while the system is running. We show in our experiments that this on-line technique can significantly improve the processors energy consumption (up to 45% for the Intel StrongARM SA-1100). Moreover, our MOTE technique can incorporate the speed/voltage change overheads by simply adding the speed transition time of the processors to the worst-case workload of each task. Our two methods address *sporadic constrained-deadline real-time systems*. This model includes the most popular one: the sporadic and implicit-deadline task systems. The complexity of each decision (at any job allocation-time) is linear in the number of ready jobs in the system. This low-complexity makes the MOTE strategy a very mighty technique.

References

[1] Intel® pxa27x processor family optimization guide.

[2] B. Andersson. *Static-priority scheduling on multiprocessors*. PhD thesis, Chalmers Univerosty of Technology, 2003.

[3] R. Aydin, R. Melhem, D. Moss, and P. Mejia-Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Transactions on Computers*, 53(5):584–600, 2004.

[4] S. Baruah and J. Anderson. Energy-aware implementation of hard-real-time systems upon multiprocessor platform. In *Proceedings of the ISCA 16th International Conference on Parallel and Distributed Computing Systems*, pages 430–435, August 2003.

[5] M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of EDF on multiprocessor platforms. In *ECRTS' 05: Proceedings of the 17th Euromicro Conference on Real-Time Systems*, 2005.

[6] J.-J. Chen, H.-R. Hsu, and T.-W. Kuo. Leakage-aware energy-efficient scheduling of real-time tasks in multiprocessor systems. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 408–417, 2006.

[7] J.-J. Chen and T.-W. Kuo. Energy-efficient scheduling for real-time systems on dynamic voltage scaling (DVS) platforms. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 28–38. IEEE Computer Society, August 2007.

[8] J.-J. Chen, C.-Y. Yang, and T.-W. Kuo. Slack reclamation for real-time task scheduling over dynamic voltage scaling multiprocessors. In *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC)*, Taichung, Taiwan, June 2006.

[9] L. Cucu and J. Goossens. Feasibility intervals for multiprocessor fixed-priority scheduling of arbitrary deadline periodic systems. In *Design Automation and Test in Europe*, pages 1635–1640. IEEE Computer Society, 2007.

[10] N. Ekekwe and R. Etienne-Cummings. Power dissipation sources and possible control techniques in ultra deep submicron cmos technologies. *Microelectronics Journal*, 37(9):851–860, September 2006.

[11] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. In *Proceedings of the IEEE/ACM international conference on Computer-aided design*, pages 598–604, California, United States, 1997. IEEE Computer Society.

[12] B. Gaujal, N. Navet, and C. Walsh. Shortest path algorithms for real-time scheduling of fifo tasks with optimal energy use. In *ACM Transactions on Embedded Computing Systems*, volume 4, pages 907–933, November 2005.

[13] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on uniform multiprocessors. *Real Time Systems*, 25:187–205, 2003.

[14] K. Kyong Hoon, B. Rajkumar, and K. Jong. Power aware scheduling of bag-of-tasks applications with deadline constraints on dvs-enabled clusters. In *Seventh IEEE International Symposium on Cluster Computing and the Grid, 2007. CCGRID 2007*, pages 541–548, May 2007.

[15] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in hard real-time environment. In *Journal of the ACM (JACM)*, pages 46–61, february 1973.

[16] P. Pillai and K. Shin. Real-time dynamic voltage scaling for low powered embedded systems. *Operating Systems Review*, 35:89–102, October 2001.

[17] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 251–259, 2001.

[18] G. Quan and H. Xiaobo. Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. In *Proceedings of the 38th conference on Design automation*, pages 828–833, 2001.

[19] Y. Shin and K. Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Design Automation Conference*, pages 134–139, 1999.

[20] A. Sinha and A. P. Chandrakasan. Jouletrack: a web based tool for software energy profiling. In *Proceedings of the 38th conference on Design automation*, pages 220–225, 2001.

[21] F. Zhang and S. Chanson. Processor voltage scheduling for real-time tasks with non-preemptible sections. In *23th Real-Time Systems Symposium*, pages 235–245, 2002.

[22] D. Zhu. Reliability-aware dynamic energy management in dependable embedded real-time systems. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, 2006.*, pages 397–407, April 2006.