MORA: an Energy-Aware Slack Reclamation Scheme for Scheduling Sporadic Real-Time Tasks upon Multiprocessor Platforms

Vincent Nelis
Department of Computer Science
Université Libre de Bruxelles (U.L.B.)
Brussels, Belgium
vnelis@ulb.ac.be

Joël Goossens
Department of Computer Science
Université Libre de Bruxelles (U.L.B.)
Brussels, Belgium
joel.goossens@ulb.ac.be

Abstract—In this paper, we address the global and preemptive energy-aware scheduling problem of sporadic constrained-deadline tasks on DVFS-identical multiprocessor platforms. We propose an online slack reclamation scheme which profits from the discrepancy between the worst- and actual-case execution time of the tasks by slowing down the speed of the processors in order to save energy. Our algorithm called MORA takes into account the application-specific consumption profile of the tasks. We demonstrate that MORA does not jeopardize the system schedulability and we show by performing simulations that it can save up to 32% of energy (in average) compared to execution without using any energy-aware algorithm.

Keywords—multiprocessor scheduling; real-time scheduling; energy-aware scheduling;

I. INTRODUCTION

A. Context of the study

Nowadays, many modern processors can operate at various supply voltages, where different supply voltages lead to different clock frequencies and to different processing speeds. Since the power consumption of a processor is usually a convex and increasing function of its speed, the slower its speed is, the less its consumption is [1]. Among the most recent and popular such processors, one can cite the Intel PXA27x processor family [2], used by many PDA devices [3]. Many computer systems, especially embedded systems, are now equipped with such voltage (speed) scaling processors and adopt various energy-efficient strategies for managing their applications intelligently. Moreover, many recent energy-constrained embedded systems are built upon multiprocessor platforms because of their high-computational requirements. Supported by this emerging technology, the Dynamic Voltage and Frequency Scaling (DVFS) [4] framework therefore becomes a major concern for multiprocessor energy-aware embedded systems. For real-time systems, this framework consists in reducing the system energy consumption by adjusting the working voltage and frequency of the processors, while respecting all the timing constraints.

1Supported by the Belgian National Science Foundation (F.N.R.S.) under a F.R.I.A. grant.

B. Previous work

There are a large number of researches about the uniprocessor energy-aware real-time scheduling problem and among those, many slack reclamation approaches have been developed over the years (see for instance [5], [6], [7], [8]). Such techniques dynamically collect the unused computation times at the end of each early task completion and share it among the remaining pending tasks. In [4], Kuo et al. propose a state-of-art about energy-aware algorithms in multiprocessor environment. As it is mentioned in this state-of-art, many studies consider the frame-based task model, i.e., all the tasks share a common deadline and this “frame” is indefinitely repeated. Targeting a sporadic task model, Anderson and Baruah [9] explored the trade-off between the total energy consumption of task executions and the number of required processors, where all the tasks run at the same common speed. In previous work [10], we provided a technique that determines the minimum common offline speed for every task under global-EDF policy [11], while considering identical multiprocessor platforms. Furthermore, we proposed in the same study an online algorithm called MOTE which was, to the best of our knowledge, the first to address the global and preemptive energy-aware scheduling problem of sporadic constrained-deadlines tasks on multiprocessors. This algorithm cannot be considered as a slack reclamation scheme since it does not directly take advantage from early tasks completion, but it can be combined with slack reclaiming techniques (and in particular with MORA) in order to improve the energy savings.

C. Contribution of the paper

In this paper, we propose a slack reclamation scheme called MORA for the global and preemptive energy-aware scheduling problem of sporadic constrained-deadline real-time tasks on a fixed number of DVFS-capable processors. According to [4] and to the best of our knowledge, this is the first work which addresses a slack reclamation scheme in this context.
D. Organization of the paper

The document is organized as follows: in Section II, we introduce our model of computation, in particular our task and platform model; in Section III, we present our online slack reclamation technique called MOR; in Section IV, we present our simulation results and in Section V, we conclude.

II. Model of computation

A. Platform model

We consider multiprocessor platforms composed of a known and fixed number $m$ of DVFS-identical processors $\{P_1, P_2, \ldots, P_m\}$. “DVFS-identical” means that (i) all the processors have the same profile (in term of consumption, computational capabilities, etc.) and are interchangeable, (ii) two processors running at a same frequency execute the same amount of execution units, and (iii) all the processors have the same minimal and maximal operating frequency denoted by $f_{\text{min}}$ and $f_{\text{max}}$, respectively. As in [12], [13], the processors are referred to as independent, with the interpretation that they can operate at different frequencies at the same time. Furthermore, we assume that each processor can dynamically adapt its operating frequency (and voltage) at any time during the system execution, independently from each other. The time and energy overheads on frequency (voltage) switching are assumed to be negligible.

We define the notion of speed $s$ of a processor as the ratio of its operating frequency $f$ over its maximal frequency, i.e.: $s \equiv \frac{f}{f_{\text{max}}}$. We also define the corresponding speed $s_k$ of the processor running at speed $s$ for $R$ time units completes $s \times R$ execution units. When only $K$ discrete frequencies are available to a processor, they are sorted in the increasing order of frequency and denoted by $f_1, \ldots, f_K$. For each frequency $f_k$ such that $1 \leq k \leq K$, we denote by $s_k$ the corresponding speed (i.e., $s_k \equiv \frac{f_k}{f_{\text{max}}}$) and by $P(s_k)$ the power consumption (energy consumption rate) per second while the processor is running at speed $s_k$. The available frequencies and the corresponding core voltages of the Intel XScale processor [14] that will be used in our experiments are outlined in [15] (Table 1, page 2). Notice that, from our definition of the processor speed, the maximal processor speed noted $s_{\text{max}}$ is equals to $f_{\text{max}}$. Moreover, due to the finite number of speeds that are available to any practical processor, any speed $s$ computed by any energy-aware algorithm must be translated into one of the available speeds. In our simulation in Section IV, this translation is performed by the function $S(s) \equiv \min\{s_i \mid s_i \geq s\}$.

B. Application model

A real-time system $\tau$ is a set of $n$ functionalities denoted by $\{\tau_1, \tau_2, \ldots, \tau_n\}$. Every functionality $\tau_i$ is modeled by a sporadic constrained-deadline task characterized by three parameters $(C_i, D_i, T_i)$ – a Worst-Case Execution Time (WCET) $C_i$ at maximal processors speed $s_{\text{max}}$ (expressed in milliseconds for instance), a minimal inter-arrival delay $T_i$ and a relative deadline $D_i \leq T_i$ – with the interpretation that the task $\tau_i$ generates successive jobs $\tau_{i,j}$ (with $j = 1, \ldots, \infty$) arriving at times $a_{i,j}$ such that $a_{i,j} \geq a_{i,j-1} + T_i$ (with $a_{i,1} \geq 0$), each such job has a worst-case execution time of at most $C_i$ time units (at maximal processors speed $s_{\text{max}}$), and must be completed at (or before) its absolute deadline noted $D_{i,j} \equiv a_{i,j} + D_i$. According to our definition of the processors speed, a job $\tau_{i,j}$ may take up to $C_i$ time units to complete on a processor running at speed $s_{\text{max}} = 1$ and, at a given speed $s$, its WCET is $C_i \cdot \frac{s}{s_{\text{max}}}$. Notice that, since $D_i \leq T_i$, successive jobs of any task $\tau_i$ do not interfere with each other.

At any time $t$ in any schedule $S$, a job $\tau_{i,j}$ is said to be active iff $a_{i,j} \leq t$ and it is not completed yet in $S$. Moreover, an active job is said to be running at time $t$ in $S$ if it is executing on a processor. Otherwise, the active job is pending in a ready-queue of the operating system and we say that it is waiting. Furthermore, a job is said to be dispatched at time $t$ in $S$ if it passes from the waiting state to the running state at time $t$.

Although certain benchmarks provide measured power consumption, we should not ignore that different applications may have different instruction sequences and require different function units in the processor, thus leading to different dynamic consumption profiles. As it was already done in [16], we hence introduce a measurable parameter $e_i$ for each task $\tau_i$ that reflects this application-specific power difference between the applications and the measured benchmark. Accordingly, the consumption of any task $\tau_i$ executed for 1 time unit at speed $s_k$ can be estimated by $e_i \cdot (P(s_k) - P_{\text{idle}}) + P_{\text{idle}}$ [16], where $P(s)$ is the consumption of the processor while running at speed $s$ and $P_{\text{idle}}$ is its idle consumption. Numerical values of $P(s)$ and $P_{\text{idle}}$ can be found in [15] (Table 1, page 2) for the Intel XScale processor used in our experiments. In the remainder of this paper, we denote by $E_i(R, s_k)$ the energy consumed by the task $\tau_i$ when executed for $R$ time units at speed $s_k$ and we define it as $E_i(R, s_k) \equiv R \cdot (e_i \cdot (P(s_k) - P_{\text{idle}}) + P_{\text{idle}})$.

C. Scheduling specifications

We consider in this study the global scheduling problem of sporadic constrained-deadlines tasks on multiprocessor platforms. “Global” scheduling algorithms, on the contrary to partitioned algorithms, allow different tasks and different jobs of the same task to be executed upon different processors. Furthermore, we consider preemptive scheduling and Fixed Job-level Priority assignment (FJP), with the following interpretations. In the preemptive global scheduling problem, every job can start its execution on any processor and
may migrate at run-time to any other processor if it gets meanwhile preempted by a higher-priority job. We assume in this paper that preemptions are carried out with no loss or penalty. Fixed Job-level Priority assignment means that the scheduler assigns a priority to jobs as soon as they arrive and every job keeps its priority constant until it completes. Global Deadline Monotonic and Global Earliest Deadline First [11] are just some examples of such scheduling algorithms.

III. THE MULTIPROCESSOR ONLINE RECLAIMING ALGORITHM (MORA)

Although most previous studies on multiprocessor energy-efficient scheduling assumed that every task consumes its WCET (see for instance [17], [18], [19]), this work is motivated by the scheduling of tasks in practice, where tasks might usually complete earlier than their WCET. The proposed algorithm MORA is an online scheme which exploits early task completions by reclaiming as much as possible the unused time in order to reduce the speed of the processors. Although it has been inspired from the uniprocessor “Dynamic Reclaiming Algorithm” (DRA) proposed in [5], the way in which it profits from the unused time is very different from the DRA since MORA targets multiprocessors environment and takes into account the application-specific consumption profile of the tasks.

A. Notations

During the system execution, every active job $\tau_{i,j}$ has two associated speeds noted $s_{i,j}$ and $s_{i,j}^\text{off}$. The speed $s_{i,j}$ denotes the speed that a processor adopts while executing $\tau_{i,j}$. We assume that these execution speeds $s_{i,j}$ can be modified at any time during the system execution, even during the execution of $\tau_{i,j}$, and it is instantaneously reflected on the processor speed. On the other hand, the speed $s_{i,j}^\text{off}$ is the offline precomputed execution speed of $\tau_{i,j}$, in the sense that the value of $s_{i,j}$ is always set to $s_{i,j}^\text{off}$ at $\tau_{i,j}$ arrival time. These offline speeds $s_{i,j}^\text{off}$ are determined before the system execution and remain always constant at run-time. They may be simply set to the maximal processors speed $s_{\text{max}}$, or they can be determined by an offline energy-aware strategy, such that the one proposed in [10] for instance. These offline speeds must ensure that all the deadlines are met when the set of tasks is scheduled upon the $m$ processors, even if every job of every task consumes its WCET. Notice that, since each task generates an infinity of jobs, the method proposed in [10] determines a common speed for every task and assumes that every job $\tau_{i,j}$ inherits from the offline speed of $\tau_i$ at run-time.

MORA is based on reducing online (i.e., while the system is running) the execution speed $s_{i,j}$ of the jobs in order to provide energy savings while still meeting all the deadlines. To achieve this goal, MORA detects whenever the speed $s_{i,j}$ of an active job $\tau_{i,j}$ can safely be reduced by performing comparison between the schedule which is actually produced (called the actual schedule hereafter) and the offline schedule defined below. We will see in the remainder of this section that our algorithm MORA always refers to this offline schedule in order to produce the actual one.

Definition 1 (The offline schedule): The offline schedule is the schedule produced by the considered scheduling algorithm on which every job of every task $\tau_i$ runs at its offline speed $s_{i,j}^\text{off}$ and consumes its WCET.

At any time $t$, we denote by $\text{rem}_{i,j}(t)$ and $\text{rem}_{i,j}^\text{off}(t)$ the worst-case remaining execution time of job $\tau_{i,j}$ at speed $s_{\text{max}}$ in the actual and offline schedule, respectively. We assume that these quantities are updated at run-time for every active job $\tau_{i,j}$. Notice that from our definition of a processor speed, the worst-case remaining execution time of job $\tau_{i,j}$ at time $t$ and at speed $s$ in the actual and offline schedule is $\text{rem}_{i,j}(t)$ and $\text{rem}_{i,j}^\text{off}(t)$, respectively. We denote by $\text{disp}_{i,j}(t)$ the earliest instant (after time $t$) at which $\tau_{i,j}$ is dispatched in the offline schedule, when only the active jobs at time $t$ in the offline schedule are considered. Finally, $\text{nextdisp}(P_r,t)$ denotes the earliest instant after time $t$ at which a job which is not completed in the actual schedule at time $t$ is dispatched to $P_r$, in the offline schedule. Again, only the set of active jobs at time $t$ in the offline schedule is considered to compute $\text{nextdisp}(P_r,t)$.

B. The $\alpha$-queue

The $\alpha$-queue is the data structure that contains all the information required by MORA about the offline schedule. However, since the jobs arrival times are unknown while considering the sporadic task model, computing and storing the entire offline schedule cannot be done before the system execution. Hence, our algorithm only stores and updates at run-time a sufficient part of the offline schedule. Notice that using a dynamic data structure for embodying a sufficient part of the offline schedule was previously proposed in [5] and, as in [5], we call this data structure $\alpha$-queue. Formally, at any time $t$ during the system execution, the $\alpha$-queue is a list which contains the worst-case remaining execution time $\text{rem}_{i,j}^\text{off}(t)$ of every active jobs $\tau_{i,j}$ in the offline schedule. This list is managed according to the following rules, which are widely inspired from [5].

$\alpha$-Rule 1: At any time, the $\alpha$-queue is sorted by decreasing order of the job priorities, with the $m$ highest priority jobs at the head of the queue.

$\alpha$-Rule 2: Initially the $\alpha$-queue is empty.

$\alpha$-Rule 3: Upon arrival of a job $\tau_{i,j}$ at time $t$, $\tau_{i,j}$ inserts its WCET $C_i$ into the $\alpha$-queue in the correct priority position. This happens only once for each arrival, no re-insertion at return from preemptions.
α-Rule 4: As time elapses, the \( m \) fields \( \text{rem}_{i,j}^{\text{off}}(t) \) (if any) at the head of the \( \alpha \)-queue are decreased with a rate proportional to the offline speeds \( s_{i,j}^{\text{off}} \). Whenever one field reaches zero, that element is removed and the update continues, still with the \( m \) first elements (if any). Obviously, no update is performed when the \( \alpha \)-queue is empty.

By consulting the \( \alpha \)-queue at any time \( t \), MORA is able to get the required information about any active jobs \( \tau_{i,j} \) in the offline schedule, i.e., its worst-case remaining execution time \( \text{rem}_{i,j}^{\text{off}}(t) \), its next dispatching time \( \text{nextdisp}(P_r,t) \) and the next job dispatching time \( \text{nextdisp}(P_r,t) \) on any processor \( P_r \). Due to the space limitation, we omitted here the implementation details about the procedures which compute \( \text{disp}_{i,j}(t) \) and \( \text{nextdisp}(P_r,t) \) based on the \( \alpha \)-queue.

Notice that, for the same reasons that those explained in [5], the dynamic reduction of \( \text{rem}_{i,j}^{\text{off}}(t) \) from α-Rule 4 does not need to be performed at every clock cycle. Instead, for efficiency, we perform the reduction only before MORA has to modify a speed, by taking into account the time elapsed since the last update. Indeed, it is necessary to have an accurate \( \alpha \)-queue only at these instants. Formally, if \( \Delta t \) time units elapsed, the \( m \) fields at the head of the \( \alpha \)-queue are updated as follows: \( \text{rem}_{i,j}^{\text{off}}(t + \Delta t) = \text{rem}_{i,j}^{\text{off}}(t) - s_{i,j}^{\text{off}} \cdot \Delta t \).

C. Principle of MORA

MORA produces the actual schedule according to two rules. The first one can be summarized as follows: at runtime, whenever a job is dispatched to a processor \( P_r \) in the offline schedule, MORA also dispatches it to \( P_r \) in the actual one. As we will see below, when a job completes in the actual schedule without consuming its WCET, MORA benefits from this unused time by executing another job earlier in the actual schedule than in the offline one (see Rule 2 below). As a result, when a job (say \( \tau_{k,\ell} \)) is dispatched at time \( t \) in the offline schedule (and thus also in the actual one according to the first rule), its worst-case remaining execution time \( \text{rem}_{k,\ell}(t) \) could be lower than \( \text{rem}_{k,\ell}^{\text{off}}(t) \) if it profited from an early completion in the actual schedule. To illustrate that, Figure 1 depicts a 5-tasks system executed upon 2 processors, where only the first job of each task is represented. The characteristics of the tasks are the following: \( \tau_1 = (6, 14, 30), \tau_2 = (6, 15, 35), \tau_3 = (8, 16, 40), \tau_4 = (2, 17, 45) \) and \( \tau_5 = (6, 18, 50) \). Assuming Global-EDF, we have the following priority order: \( \tau_{1,1} > \tau_{2,1} > \tau_{3,1} > \tau_{4,1} > \tau_{5,1} \). Furthermore, we assume in this example that the offline speed \( s_{i,j}^{\text{off}} \) of every job \( \tau_{i,j} \) is the maximal processors speed \( s_{\text{max}} = 1 \).

In Figure 1 at time \( t = 2 \), \( \tau_{2,1} \) completes in the actual schedule on processor \( P_2 \) and leaves 4 unused time units. These 4 time units are reclaimed by starting the execution of \( \tau_{5,1} \) (we will see below how MORA selects the job which profits from the slack time) and therefore, when \( \tau_{5,1} \) is dispatched to \( P_2 \) in the offline schedule at time \( t = 8 \), it is also dispatched to \( P_2 \) in the actual one and we have \( \text{rem}_{5,1}(8) < \text{rem}_{5,1}^{\text{off}}(8) \). The difference between these remaining execution times is called the earliness of the job and we denote it by \( \varepsilon_{k,\ell}(t) := \text{rem}_{k,\ell}(t) - \text{rem}_{k,\ell}^{\text{off}}(t) \). According to this earliness, whenever any job \( \tau_{k,\ell} \) is dispatched in the offline schedule (and thus also in the actual one), its execution speed \( s_{k,\ell} \) may safely be reduced to \( s_{k,\ell}^{\text{def}} \) so that \( \frac{\text{rem}_{k,\ell}^{\text{off}}}{s_{k,\ell}^{\text{def}}} = \frac{\text{rem}_{k,\ell} + \varepsilon_{k,\ell}(t)}{s_{k,\ell}^{\text{def}}} \). Indeed, under this speed \( s_{k,\ell}^{\text{def}} \), \( \tau_{k,\ell} \) would complete simultaneously in both schedules if it consumes its WCET. Rule 1 gives a formal description of the first rule of MORA.

Rule 1: Any job \( \tau_{i,j} \) which is dispatched to any processor \( P_r \) at time \( t \) in the offline schedule is also dispatched to \( P_r \) at time \( t \) in the actual one and its execution speed \( s_{i,j} \) is modified according to

\[
\hat{s}_{i,j} = \frac{\text{rem}_{i,j}(t) \cdot s_{i,j}^{\text{off}}}{\text{rem}_{i,j}^{\text{off}}(t)}
\]

Informally, the second rule of MORA can be summarized as follows. When any job completes in the actual schedule without consuming its WCET, the unused time may be reclaimed by starting the execution of any waiting job earlier; and since the selected waiting job receives additional time for its execution, it can thereby reduce its execution speed. Using this concept, Figure 1 depicts an example of how MORA takes advantage from an early job completion. When \( \tau_{2,1} \) completes at time \( t = 2 \) in the actual schedule, MORA selects a waiting job (here, \( \tau_{5,1} \)) and executes it during the 4 time units left by \( \tau_{2,1} \). Since \( \tau_{5,1} \) is granted to use 4 additional time units, MORA reduces its execution speed \( s_{5,1} \) so that its worst-case remaining execution time increases by 4 time units. The selected job is the one for which the resulting speed reduction leads to the highest energy saving. Formally, MORA selects a waiting job and decreases its execution speed as described by Rule 2.

Rule 2: Whenever any processor \( P_r \) is about to get idle at time

\[
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
\]

\[
\tau_{5,1}
\]

\[
\text{deadline of } \tau_{5,1}
\]

\[
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
\]

\[
\tau_{5,1}
\]

\[
\text{deadline of } \tau_{5,1}
\]
Step 1. Use the \( \alpha \)-queue to compute the next dispatching time \( \text{nextdisp}(P_r, t) \) on processor \( P_r \) and proceed to the steps 2–5 for every waiting job \( \tau_{i,j} \) at time \( t \) in the actual schedule.

Step 2. Compute the amount \( L_{i,j}(t) \) of additional time units that \( \tau_{i,j} \) could reclaim in the actual schedule if it was dispatched at time \( t \), i.e.,

\[
L_{i,j}(t) \overset{\text{def}}{=} \min(\text{nextdisp}(P_r, t), \text{disp}_{i,j}(t)) - t
\]

In Figure 1 we have \( \text{nextdisp}(P_2, 2) = 6, \text{disp}_{5,2}(2) = 8 \) and \( L_{i,j}(2) = 4 \forall \tau_{i,j} \).

Step 3. Compute what would be the resulting execution speed \( s'_{i,j} \) if \( \tau_{i,j} \) was granted to use both its earliness and these \( L_{i,j} \) additional time units, i.e., \( s'_{i,j} \) is computed so that

\[
\frac{\text{rem}_{i,j}(t)}{s'_{i,j}} = \frac{\text{rem}_{i,j}(t) + L_{i,j}(t) - \text{disp}_{i,j}(t)}{s'_{i,j}}
\]

Step 4. Estimate what would be the resulting execution speed \( s''_{i,j} \) if \( \tau_{i,j} \) will not be executed in the actual schedule until time \( t'' \), we will have \( \text{rem}_{i,j}^*(t'') = \text{rem}_{i,j}(t) \) and \( \text{rem}_{i,j}^*(t'') = \text{rem}_{i,j}^*(t) \), and from Expression 1

\[
s''_{i,j} = \hat{S} \left( \frac{\text{rem}_{i,j}(t) \cdot s''_{i,j}}{\text{rem}_{i,j}^*(t)} \right)
\]

Step 5. Compute the energy saving \( \Delta E_{i,j} \) between execution at speed \( s''_{i,j} \) and at speed \( s'_{i,j} \):

\[
\Delta E_{i,j} = E_{i} \left( \frac{\text{rem}_{i,j}(t)}{s'_{i,j}} \right) - E_{i} \left( \frac{\text{rem}_{i,j}(t)}{s''_{i,j}} \right)
\]

Step 6. Dispatch the job \( \tau_{k,l} \) with the largest \( \Delta E_{k,l} \) to processor \( P_r \). If \( \Delta E_{i,j} \leq 0 \) for all the waiting jobs, then dispatch the waiting job \( \tau_{k,l} \) (if any) with the highest priority in order to complete it earlier and to potentially increase the length of future slack time.

Step 7. If there is a selected job \( \tau_{k,l} \), set its execution speed \( s_{k,l} \) to the computed one \( s'_{i,j} \). Otherwise, turn the processor \( P_r \) into the idle mode.

Applying Rules 1 and 2 does not jeopardize the system schedulability. We omitted the proofs here due to the space limitation but the interested reader may refer to [15] for formal proofs and a complete description of the algorithm.

IV. Simulation results

In this section, we compare the effectiveness of MORA with other energy-aware algorithms. However, it is meaningful to only compare MORA with approaches that consider the same models of computation and the most related paper to ours is [10], where two methods with the same task and platform model are proposed. However, these two methods do not take into account the application-specific parameter \( e_i \) of task \( \tau_i \). The first method proposed in [10] (that we denote by OFF thereafter) is an offline speed determination technique for Global-EDF which determines an unique and constant speed \( s_{i,j}^{\text{off}} \) for all the processors such that all the job deadlines are met under this speed. In our simulations, this OFF method is used by MORA in order to provide the offline speed \( s_{i,j}^{\text{off}} \) of every job \( \tau_{i,j} \). The second method proposed in [10] is the MOTE algorithm. At run-time, it anticipates the coming idle instants in the schedule and adjusts the speed of the processors accordingly, i.e., it reduces the processors speed in order to minimize the proportion of time during which the system is idle. Since this algorithm is also based on the concept of the offline speeds, we consider that OFF is also used to provide it.

In our simulations, we schedule periodic implicit-deadline systems (i.e., \( \forall \tau_i, T_i \) is here the exact inter-arrival delay between successive jobs and \( D_i = T_i \)). The tasks generation process is fully described in [15] but is omitted here due to the space limitation. The energy consumption of each generated system is computed by simulating three methods: MOTE, MORA and MORAOTE, i.e., a combination of the MOTE and MORA (see [15] for details). As we will see in our simulation results, this combination always improves the provided energy savings. The consumptions provided by these three methods are compared with the consumption of the MAX method, where all the jobs are executed at the maximal processors speed \( s_{\text{max}} = 1 \). That is, we consider that the consumption by MAX is 100% and the consumptions of the other methods are normalized.

![Figure 2: Average consumptions of MOTE, MORA and MORAOTE compared to MAX.](image)

Due to the space limit, we only depict in Figure 2 the results provided by Global-EDF on Intel XScale processors (outlined in [15], Table 1, page 2). The Y-axis is the average energy consumption of every method compared with the MAX method (in %) and the X-axis is the maximum density of the tasks during the simulation process (the density of a task \( \tau_i \) is the ratio \( \frac{\text{disp}_{i,j}(t)}{T_i} \)). In [15], the simulation results are discussed and relevant observations are presented.

\[\text{Figure 2: Average consumptions of MOTE, MORA and MORAOTE compared to MAX.}\]

Notice that, if any processor \( P_r \) is about to get idle in the actual schedule exactly when a job is dispatched to \( P_r \) in the offline one, only Rule 1 is applied.
shows that MORA can save up to 32% of energy (in average) over the MAX method (for tasks density not larger than 0.1) and the algorithm MORAOTE provides important energy savings for various maximal densities. Furthermore, although other processor models and scheduling algorithms led to different average consumptions, the evolution of the consumption with respect to the maximal density remains similar than in Figure 2.

V. Conclusion

In this paper, we propose a slack reclamation scheme called MORA which reduces the energy consumption while scheduling a set of sporadic constrained-deadline tasks by a global, preemptive and FJP algorithm on a fixed number of DVFS-identical processors. According to [4] and to the best of our knowledge, we are the firsts to address such approach in this context. The proposed algorithm MORA exploits early job completions at run-time by starting the execution of the next waiting jobs at a lower speed. Compared with other reclaiming algorithms such that the DRA proposed in [5], MORA takes into account the application-specific consumption profile of the tasks in order to improve the energy saving that it provides. Moreover, we proved that using MORA does not jeopardize the system schedulability and we show in our simulations that it can save up to 32% of energy (in average) compared to execution without using any energy-aware algorithm.

References


