A Scheduling Algorithm to Reduce the Static Energy Consumption of Multiprocessor Real-Time Systems

Vincent Legout, Mathieu Jan
CEA, LIST
Embedded Real Time Systems Laboratory
F-91191 Gif-sur-Yvette, France
{vincent.legout,mathieu.jan}@cea.fr

Laurent Pautet
Institut Telecom
TELECOM ParisTech
LTCI - UMR 5141 PARIS, France
laurent.pautet@telecom-paristech.fr

ABSTRACT

Energy consumption of real-time embedded systems is a growing concern. It includes both static and dynamic consumption and is now dominated by static consumption as the semiconductor technology moves to deep sub-micron scale. In this paper, we propose a new approach to efficiently use the low-power states of multiprocessor embedded hard real-time systems in order to reduce their static consumption. In a low-power state, the processor is not active and the deeper the low-power state is, the lower is the energy consumption but the higher is the transition delay to come back to the active state. Our approach increases the duration of the idle periods to allow the activation of deeper low-power states. Offline, we use an additional task to model the idle time and we use mixed integer linear programming to reduce its number of preemptions. Online, we extend an existing scheduling algorithm to increase the length of the idle periods. To the best of our knowledge, this is the first optimal multiprocessor scheduling algorithm minimizing static consumption. Simulations show that the energy consumption while processors are idle is dramatically reduced with our solution compared to existing multiprocessor real-time scheduling algorithms.

1. INTRODUCTION

Real-time embedded systems tend to have a limited power supply usually provided by batteries. Therefore minimizing their energy consumption is an important concern, for instance, in the automotive and the energy distribution fields. Chips used for designing these systems have generally a small amount of RAM and Flash memories. Besides, they have a limited number of levels of cache, usually one or two, and of small size, typically in the range of kilobytes. Furthermore, these domain fields do not use graphical processing units or devices such as touchscreen or LCD display. Processors and their associated levels of cache therefore consume most of the total energy consumption of such embedded systems [32, 11]. Finally, safety-critical systems are subject to certification constraints and therefore concerned by the predictability of the execution time of hard real-time tasks. Industrials designers of such systems are therefore slowly considering multiprocessors [29] for their next generation of systems. Consequently, this work focuses solely on reducing the energy consumption of processors, a key challenge for the design of these hard-real time embedded multiprocessors systems of next generation.

The energy consumption of processors can be divided into two categories: dynamic and static consumption. Dynamic consumption depends on the activity of the processors. On the other hand, static consumption is mainly due to leakage current and is present even when no operations are performed by the system. Dynamic consumption used to dominate static consumption for micrometer-scale semiconductor technology. Therefore, most of the research works were dedicated to reduce dynamic consumption by using DVFS (Dynamic Voltage and Frequency Scaling) which decreases the frequency of processors ([7, 14]).

Within these works, only few consider static consumption and usually only once dynamic consumption cannot be further reduced. However, several research works from Austin [4], Lesueur [21] or Awan [6] show that while dynamic consumption used to be preponderant, static consumption is now responsible for the majority of the energy consumption. According to Buttazzo and al. [18], the leakage current already accounted for 50% of the total power dissipation in 90 nm technologies and this trend is only increasing as the VLSI technology is scaling down to deep sub-micron domain. Thus we believe static consumption should be considered before dynamic consumption. Another advantage of targeting static consumption is that, unlike DVFS, the low-power states of the other components of the system, such as devices, can be activated.

This paper focuses on reducing the static consumption using the low-power states of a processor during which no instruction is executed. A transition delay, and therefore an energy penalty, is then required to get the system back to the active state. Systems usually have several low-power states available. A more efficient low-power state means that more components are turned off, thus the transition delay required to reactivate the system increases. On hard real-time systems, time constraints must always be fulfilled. Thus, the system must take into consideration transition delays when choosing and activating a low-power state such that the processor has enough time to wake up and execute tasks in time to avoid deadline misses.

So far, existing works have addressed the management of

RTNS 2013 , October 16 - 18 2013, Sophia Antipolis, France
Copyright 2013 ACM 978-1-4503-2058-0/13/10 ...$15.00.
http://dx.doi.org/10.1145/2516821.2516839.

static consumption on hard real-time multiprocessor systems using only partitioned or global non-optimal solutions. The contribution of this paper is, to the best of our knowledge, the first global optimal multiprocessor scheduling algorithm reducing static consumption. Note that in this paper, optimal means that the scheduling algorithm is optimal with regard to the utilization of the processors, it does not refer to the reduction of the energy consumption.

Our algorithm is called LPDPM for Linear Programming DPM. We consider periodic tasks characterized by their Worst Case Execution Time (WCET) and their period. Mixed integer Linear programming is used offline to generate a schedule guaranteeing the schedulability of the task set and minimizing the static energy consumption. Then, the online algorithm schedules tasks inside intervals. It further extends the length of the idle periods, especially when tasks do not consume all their WCET. When tasks use their WCET, the energy consumption of LPDPM while processors are not executing tasks is up to 8 times smaller than with recently proposed optimal multiprocessor schedulers.

A previous work [24] was based on a similar approach but used a heuristic to reduce the static energy consumption. Furthermore, it only gave preliminary results on a specific processor without providing neither a detailed analysis of these results nor a complexity analysis of our solution. In this work, we present a complete solution improving both the offline algorithm and the online scheduler.

The remainder of this paper is as follows. Section 2 describes the main solutions found in the literature. Then section 3 defines the processor and task models. The general approach is presented in section 4 while the offline method and the online algorithm are detailed in sections 5 and 6. Section 7 presents the complexity of our solution, section 8 evaluates the efficiency of LPDPM and section 9 concludes.

2. RELATED WORK

This section covers existing solutions to minimize the static energy consumption on both uniprocessor and multiprocessor systems.

2.1 Uniprocessor systems

On uniprocessor systems, Lee et al. [22] were the first to propose a solution to increase the length of the idle periods. When the processor is idle, their idea is to delay future executions of tasks to keep the processor in a low-power state. When a task is released while the processor is sleeping, a procrastination interval (i.e. the time spent delaying a task execution) is computed such that no deadline miss can occur. The complexity of computing the procrastination interval is $O(n^2)$ for a task set with n tasks. Jejurikar et al. [20] and Niu et Quan [28] then proposed solutions to extend the procrastination interval. The main drawback of these uniprocessor solutions is complexity because computing the procrastination intervals requires anticipating executions of tasks. For example, the solution proposed by Niu et Quan. [28] is optimal in the sense that delaying execution of tasks any further will cause at least one deadline miss, but its complexity is $O(N^2)$, with N the number of jobs within the hyper-period.

Still on uniprocessor systems, Awan and Petters [5] use a different model with both soft and hard real-time tasks. Offline, they compute the maximal procrastination interval to reduce the overhead. Online, they use slack stealing to further increase the time spent in a low-power state if tasks do not use their worst case execution time.

Overall, the literature provides efficient solutions to reduce the static consumption of uniprocessor systems while multiprocessor systems have not been as much covered.

2.2 Multiprocessor systems

Multiprocessor scheduling algorithms can be classified in two categories: partitioned and global. Partitioned scheduling algorithms bind tasks to a specific processor offline and cannot be optimal, while global scheduling algorithms allow online task migration and can be optimal.

Partitioned scheduling.

Partitioned scheduling is used by Chen et al. [13] for periodic task systems. They first partition tasks and then use power-aware uniprocessor scheduling algorithms to decrease both static and dynamic consumptions. Seo et al. [31] and Haung et al. [17] also use partitioned scheduling. Offline, they partition tasks, then on-line allow task reallocation when a task releases a new job. Indeed, jobs can finish earlier than expected and the idea is to migrate them, but only once, in order to create larger idle periods. It reduces the static energy consumption but a substantial complexity is then required online to know which job should migrate.

Based on their uniprocessor work, Awan and Petters [6] propose a partitioned approach for heterogeneous multiprocessor systems for which each core differs on its execution frequency. Tasks partitioning is a two-step process. It first assigns tasks to processors according to the energy efficiency of each task on each processor. Then, it takes into consideration the deepest low-power state usable for each core to reassign tasks such that deeper low-power states can be used. Thus, this approach only focuses on reducing static consumption in the second step of the partitioning process.

However, solutions using partitioned scheduling can only schedule task sets with a limited global utilization. Indeed a partitioned scheduling algorithm cannot have a utilization bound larger than 50%. Also, it makes it more difficult to use the additional idle processor time when tasks finish earlier than expected because tasks cannot migrate. Indeed, if two tasks finish earlier on two different processors, it creates two idle periods that cannot be gathered because migration is not allowed.

Global scheduling.

Global scheduling is used by Bhatti et al. [9]. Their goal is to use as few processors as possible such that the other processors can be in a low-power state. When one or more tasks are ready to be scheduled, their algorithm always keeps the same processor busy and activates another one only when required. However, their scheduling algorithm is not optimal. Furthermore, like the approach proposed by Lee et al. for uniprocessor systems, computing the procrastination intervals is done online and is costly: the complexity is $O(n^3)$ at each scheduling event.

In conclusion, all these algorithms try to efficiently use the low-power states of processors to reduce the static consumption by increasing the size of the idle periods. However, these solutions are online and thus require a significant complexity to generate large idle periods. Furthermore, they only try to extend the length of the idle periods without considering the characteristics of each low-power state.

3. MODEL

This section introduces the model of processors and tasks used in the remainder of this paper. Let us stress that we do not consider in this work processors used in high-end servers: we focus on processors used by hard real-time systems. Safety critical embedded systems are slowly transitioning from uniprocessor systems to multiprocessors systems [29], due to certification constraints. Consequently, the majority of existing embedded real-time systems only have two cores. Algorithm design and evaluations can therefore be made assuming a limited number of cores, typically at most 4. Besides, such processors have L1 and L2 caches of limited size, i.e. up to some tens of kilobytes for the L1 caches and up to a few megabytes for the L2. They do not use energy consuming devices such as touchscreen or LCD displays. Finally, moving to multicore architectures also means moving to deep sub-micro VLSI technology, where the leakage current and therefore the static energy is becoming more important than the dynamic energy [18].

Existing works ([13, 6]) do not integrate the preemptions and migrations costs in their model. The number of preemptions and migrations then becomes a criterion to evaluate the efficiency of a proposed scheduling algorithm. This is what is done for instance for evaluating the efficiency of global optimal multiprocessor algorithms. Other works have empirically studied these costs in multiprocessor algorithms [8]. Results show that in a underloaded system, preemption and migration costs are similar and predictable when the working set size does not trash the L2 cache. Besides, these costs depend on the preemption length but not on the task set size. In this work, we also compute the number of preemptions and migrations generated by our solution and then compare their number with existing global optimal multiprocessor algorithms. Section 8 analyzes these results and further discusses this point.

3.1 Processors

We use a system with m identical processors and each processor has ns low-power states. In a low-power state, a processor cannot execute any instruction and its consumption is reduced. We further assume that each low-power state can be activated independently of the state of the other processors. This assumption prevents the use of a low-power state that deactivates a level of cache shared between a set of processors (typically the L2 cache or higher). We plan to remove this restriction in future work. However, such a deep low-power state is rarely available on embedded chips, since they have no or one level of cache. This statement is based on an analysis of the low-power states of the entire ARM Cortex processor family.

The most efficient low-power state has index 0. The consumption of state s is C_s. Coming back from a low-power state to the active state requires a transition delay and the processor cannot execute any instruction while waking up. Let Pen_s be the consumption penalty to come back from low-power state s. The more energy efficient a low-power state is, the more components are turned off and the more important is the consumption required to come back to the active state. Thus $C_0 < C_1 < ... < C_{ns}$ and $Pen_0 > Pen_1 > ... > Pen_{ns}$. To avoid having to deal with a particular case for the idle mode, a fake low-power state was added when no low-power state can be activated. It has index ns and with $C_{ns} = 1$ and $Pen_{ns} = 0$. The minimum idle period for which activating a given low-power state saves more energy than letting the processor idle is called the *Break-Even time* (BET). The BET of each low-power state s is BET_s and $BET_{ns} = 0$.

Embedded processors usually have between three and four low-power states ([3, 2, 1]). The first low-power state has a short transition delay because it typically only stops the processor while preserving the content of all caches and maintaining all clocks active. On the other hand, deeper low-power states gradually shutdown clocks and caches and thus require a longer transition delay to setup the clocks and repopulate the caches. The ARM Cortex-A9 MPCore processor [1] can have up to four cores. Each core and each cache has its own power level. Thus each core can be separately put in all available low-power states, except the deepest low-power state which allows powering off the L2 cache. It can therefore only be activated once all cores are idle. We intend to integrate this constraint in future work to be able to schedule clustered and heterogeneous systems.

3.2 Tasks

We consider a set Γ of n independent, synchronous, preemptible and periodic tasks. Tasks can migrate from one processor to another. Each task τ_i has a Worst Case Execution Time C_i (WCET) and a period T_i. Tasks have implicit deadlines, i.e. deadlines are equal to periods. The task set hyper-period is named H and is the least common multiple of all periods of tasks in Γ. A job j is an instance of a task and is characterized by its WCET $j.c$ and its deadline $j.d$. The job set J_Γ contains all jobs of Γ scheduled during the hyper-period H. Utilization u_i of task τ_i is the ratio $\frac{C_i}{T_i}$ and task set global utilization is the sum of all utilizations: $U = \sum_{i=0}^{n-1} u_i$.

The objective being to decrease the static energy consumption, we do not consider situations where $U \in \mathbb{N}^+$. Indeed, with this assumption, U processors would always be active while $m - U$ processors would always be in the deepest low-power state. Thus we assume global utilization U is such as $m - 1 < U < m$. And if $m - x - 1 < U < m - x$, x processors can be left in a low-power state such that the assumption $m - 1 < U < m$ holds.

As in Lemerre et al. [25], the hyper-period is divided in intervals, an interval being delimited by two task releases. I is the set of intervals and $|I_k|$ is the duration of the k^{th} interval. A job can be present on several intervals, and we note $w_{j,k}$ the weight of job j on interval k. The weight of a job on an interval is defined as the fraction of processor required to execute job j on interval k. J_k is the subset of J_Γ that contains all active jobs in interval k. E_j is the set of intervals on which job j can run. It must contain at least one interval. The example task set in figure 1 has two tasks and four jobs (jobs 1 to 3 from τ_1 and job 4 from τ_2). In this example, E_4 is $\{1, 2, 3\}$ and J_1 is $\{1, 4\}$.

4. APPROACH

We claim that the approach used by the existing solutions is not efficient to create large idle periods and reduce the static energy consumption. This section discusses this statement and introduces our approach.

4.1 Problems with existing solutions

The existing scheduling algorithms minimizing static consumption mostly use partitioned scheduling. Thus they can-

Figure 1: 2 tasks with periods of 4 and 12.

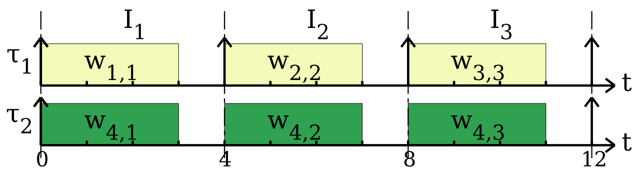
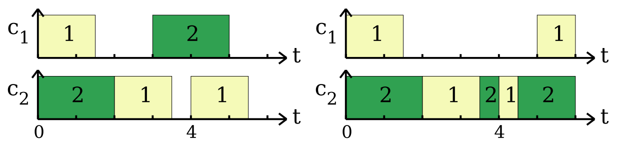


Figure 2: Example with 2 tasks τ_1 (1.5, 2) and τ_2 (2, 3).

(a) Without delayed execution (b) With delayed execution



not create large idle periods because it creates idle periods on each processor. Migration should be authorized in order to merge idle periods from different processors. This is the approach followed by Seo et al. [31] and Huang et al. [17]. These solutions are thus no longer exclusively using partitioned scheduling which do not authorize migrations. They are instead using global scheduling, as Bhatti et al. [9].

All these solutions are online and the online complexity prevents them from being usable (e.g. $O(n^3)$ for Bhatti with n being the number of tasks). This complexity is due to the computation required to decide if a task should migrate to another processor or if a task should be delayed to increase the length of an idle period. Deciding on whether a task can be delayed requires anticipating the execution of future jobs. The solution proposed by Niu et al. [28] on uniprocessor systems requires a computation over the next hyper-period to anticipate all future job executions.

This complexity required to create large idle periods comes from the fact that these solutions are based on scheduling algorithms like Global-EDF which are work-conserving. A work-conserving algorithm is a scheduling algorithm in which no processor can be left idle if a task is ready to be executed. For example, Bhatti et al. say that their solution can be used with both Global-EDF and Global-LLF.

This approach is not suitable for a scheduling algorithm which aims to create large idle periods. It should not be based on Global-EDF or Global-LLF. Scheduling decisions should instead be taken based on the knowledge of the current workload. Whether or not a task should be delayed does not depend on a parameter of the associated task model (i.e. deadline, WCET, period, ...), but on the current state of the system. For example, a task should be delayed if a processor is currently in a low-power state but should be executed if all processors are currently active.

To illustrate the above claim, let a task set with two tasks τ_1 and τ_2 whose WCET and period respectively are (1.5, 2) and (2, 3). The hyper-period is 6. This task set is scheduled on a two processors system (c_1 and c_2). As shown in figure 2a, a work-conserving multiprocessor scheduling algorithm like Global-EDF would schedule τ_1 on c_1 and τ_2 on c_2 when the system starts. The next two jobs of τ_1 would be scheduled on c_2 at $t = 2$ and $t = 4$ while the second job of τ_2 would be executed on c_1 at $t = 3$. To reduce the number of idle periods, one would want to be able to schedule the second job of τ_2 on c_2 as illustrated by figure 2b. This is obviously not possible at $t = 3$. Thus a computation is required to delay τ_2 to $t = 3.5$. Other computations are then necessary at $t = 4$ for τ_2 and at $t = 4.5$ for τ_1. This example shows that delaying task executions is required to create large idle period. This costly operation is performed 3 times on one hyper-period on this simple task set. This approach is therefore not usable with realistic task sets.

4.2 Offline approach

To avoid these issues, we adopt a different approach: we generate an energy efficient schedule offline. Being offline means that all scheduling decisions can be thought beforehand. These decisions are made such that the complete schedule is optimized over a hyper-period to reduce static consumption. Another advantage is the low complexity of the online algorithm because all scheduling decisions are taken offline. Online, the algorithm deals with situations where tasks do not consume all their WCET to extend the existing idle periods.

In contrast to classical multiprocessor scheduling algorithms using global scheduling, our solution is non work-conserving. That is processors can be idle while active tasks are awaiting execution.

The schedule computed offline minimizes the energy consumption when tasks used their WCET. Then, online, the energy consumption can only decrease when tasks finish their execution before reaching their WCET. This approach guarantees a minimal energy consumption, which can only be reduced online.

Offline, we use linear programming to express both the real-time constraints of the system and the objective of minimizing the static energy consumption. The computed scheduled therefore schedules and delays tasks when appropriate to activate the deep low-power states. The linear program computes a weight for each task on each interval. This computation is based on the WCET of tasks. Then, online, tasks are scheduled inside intervals given these weights.

This scheduling approach based on linear programming has been used in [25] to compute offline a partial but valid schedule. It builds global optimal real-time multiprocessor scheduling algorithms. Besides, the advantage of this approach is to be able to add specific scheduling objectives and custom constraints in the linear equation system. For example, Megel et al. [26] used it with the objective of decreasing the number of preemptions and migrations for optimal multiprocessor global scheduling. We also used it to reduce the energy consumption of mixed-criticality systems in [23].

Note that the linear program computes a schedule for one hyper-period. A computation on more hyper-periods is also possible and would be more efficient because it would cover transitions between hyper-periods. However, the linear program would be twice as big which would not be worth the very slight improvement in energy consumption savings.

The constraints and the objective function are respectively detailed in the next subsection and in section 5.

4.3 Real-time constraints

A set of constraints are first required to guarantee the schedulability of the task set, i.e. that no deadline is missed. These constraints are [25]:

$$\forall k, \sum_{j \in J_k} w_{j,k} \leq m \tag{1}$$

$$\forall k, \forall j, 0 \leq w_{j,k} \leq 1 \tag{2}$$

$$\forall j, \sum_{k \in E_j} w_{j,k} \times |I_k| = j.c \tag{3}$$

Equation 1 means that global utilization on an interval cannot exceed the number of processors. Thus any interval is schedulable. The second inequality forbids the duration of a job on an interval to be negative or to exceed the length of the interval. This is necessary so that a task can be scheduled in an interval. Finally, the third equality guarantees that all jobs are completely executed.

Solving the linear program made of equations (1), (2) and (3) gives a valid schedule. But it does not minimize the energy consumption. An objective function must be added.

5. OFFLINE LINEAR PROGRAM

Equations (1), (2) and (3) ensure that the temporal constraints of the system are satisfied in the generated schedule. Then, to minimize the static consumption, this section enhances the initial linear equation system by adding new constraints and an objective function.

5.1 Idle task τ'

To ease modeling the problem, we use an additional periodic task τ'. τ' accounts for the time where processors are expected to be idle. This additional task has a period of H and utilization of $m - U$, thus lower than 1 because we assume $m - 1 < U < m$. The task set now has $n + 1$ tasks and global utilization becomes equal to the number of processors m. Introducing τ' forbids two processors to be idling simultaneously because they would be both executing τ'. It should be noted that τ' does not represent the actual idle time when tasks are executed. Indeed, as tasks usually do not use all their WCET at run-time, processors can be idle while not executing τ' and multiple processors can be idle simultaneously. Modeling the expected idle time with τ' is just a way to help generating a schedule with guaranteed idle periods in the worst case scenario.

Inside an interval, the scheduling of τ' is trivial when the weight of τ' is either 0 or 1. However, when τ' does not occupy a full interval, the execution of τ' inside the interval plays a role in the reduction of the energy consumption. It should be executed either at the beginning or at the end of the interval. Indeed, when we merge this idle period with an idle period from a neighbor interval, we increase the opportunity to use deeper low-power states.

In our previous work [24], we used a heuristic to reduce the energy consumption. The linear program did not minimize the static energy consumption but the number of preemptions of τ'. Scheduling τ' at the beginning or at the end of the interval was decided online. In this paper, we refine this last step. We split τ' into two subtasks whose respective weights are computed in the linear program. The first subtask is executed at the beginning of the interval and the second at the end of the interval. This allows an offline computation of the best solution reducing the energy consumption based on the characteristics of each low-power state.

5.2 Scheduling τ' inside intervals

Dividing τ' into two subtasks executed at the boundaries of the intervals cannot increase the energy consumption inside an interval. The only situation which cannot be handled with this assumption is when τ' is executed in the middle of an interval, without any contact with the boundaries of the interval. However, this kind of schedule can never be more energy efficient than the same schedule with τ' being executed either at the beginning or at the end of the interval. Indeed the length of the idle period is not reduced and the idle period can be merged with another idle period from a neighbor interval. It should be noted that the two subtasks of τ' can never be preempted. This affirmation is discussed in section 6.

Let b_k and e_k be the weights of the two subtasks of τ' in interval k with $0 \leq b_k \leq 1$ and $0 \leq e_k \leq 1$. The weight of τ' in interval k is thus $b_k + e_k$, and an idle period lasts for the whole interval k if $b_k + e_k = 1$. To schedule intervals online, these two subtasks are respectively given the highest and lowest priorities, no matter their weight, so that they are executed at the beginning and at the end of the interval.

With τ', the three equations defined in section 4 are still valid. However, as the global utilization is now equal to the number of processors, the inequality in equation (1) becomes an equality.

5.3 Computing the length of the idle periods

Minimizing the static consumption requires computing the length of all idle periods to know which low-power state can be activated. In an interval, two distinct executions of τ' are possible. However, the execution of τ' at the end of interval k also includes the execution of τ' at the beginning of interval $k + 1$. Thus there is only one idle period per interval, with the exception of the first interval.

Let p_k be the length of the idle period of interval k. It includes both the execution of τ' at the end of interval k (i.e. e_k) and the execution of τ' at the beginning of interval $k+1$ (i.e. b_{k+1}). Note that it means the idle period of interval k can start in interval $k + 1$ if $e_k = 0$.

When the weight of τ' is 1 in interval $k + 1$, this means the idle period of interval k also includes the idle period of interval $k + 1$. Thus p_k is defined as:

$$p_k = \begin{cases} e_k \times |I_k| + b_{k+1} \times |I_{k+1}| + p_{k+1} & \text{if } b_{k+1} + e_{k+1} = 1 \\ e_k \times |I_k| + b_{k+1} \times |I_{k+1}| & \text{otherwise} \end{cases} \tag{4}$$

Thus the idle period starting at the end of interval k may include the idle period starting at the end of interval $k + 1$ (i.e. p_{k+1}) when $b_{k+1} + e_{k+1} = 1$ (i.e. τ' occupies the full interval $k + 1$).

Then, we remove the idle periods which are counted twice and introduce an additional variable q_k which gives the real length of each idle period:

$$q_k = \begin{cases} p_k & \text{if } b_k + e_k \neq 1 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

The idle period is kept only if τ' in the current interval k does not occupy the whole interval. Otherwise, the idle period is already included in q_{k-1} and is not kept. The length of each idle period is now known.

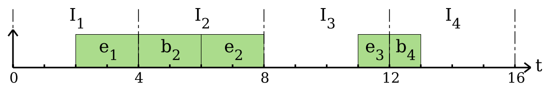
To illustrate these notations, figure 3 shows how to schedule task τ' on an example which $H = 16$. The WCET of τ'

is 8. There are 4 intervals and $\forall k, |I_k| = 4$. The values of all variables are shown in table 1. The column 0 represents the idle period starting with the execution of b_1. 2 idle periods of length 6 and 2 are created in the end of intervals 1 and 3. The variable q_k discards the idle period p_2 as the first idle period lasts across intervals 1 and 2.

Table 1: Example with 4 intervals, $\tau'.c = 8$.

k	0	1	2	3	4
$b_k * \lvert I_k \rvert$	0	0	2	0	1
$e_k * \lvert I_k \rvert$	0	2	2	1	0
p_k	0	6	2	2	0
q_k	0	6	0	2	0

Figure 3: Schedule of τ' from table 1.



5.4 Minimizing the static energy consumption

To know which low-power state is used on each idle period, let $LP_{s,k}$ be a binary variable. $LP_{s,k} = 1$ if the low-power state s is activated during the idle period starting in the end of interval k and $LP_{s,k} = 0$ otherwise. Each idle period with $q_k > 0$ must activate one and only one low-power state:

$$\forall k, \sum_s LP_{s,k} = \begin{cases} 0 & \text{if } q_k = 0 \\ 1 & \text{otherwise} \end{cases} \tag{6}$$

However, a low-power state cannot be activated if the idle period is smaller than the BET. Therefore this additional constraint is needed:

$$\forall k, \forall s, LP_{s,k} = 0 \text{ if } q_k \leq BET_s \tag{7}$$

The consumption while in low-power state s is C_s and the penalty consumption required to come back to the active state is Pen_s. Thus the consumption P_k of the idle period starting at the end of interval k is:

$$P_k = \sum_s LP_{s,k}(C_s \times q_k + Pen_s) \tag{8}$$

As we assume all tasks use their WCET, the consumption of the initial task set does not impact the overall energy consumption. Thus, minimizing the static energy consumption means minimizing on each interval the consumption of the idle task. Thus the final objective function of the mixed integer linear program (MILP) is:

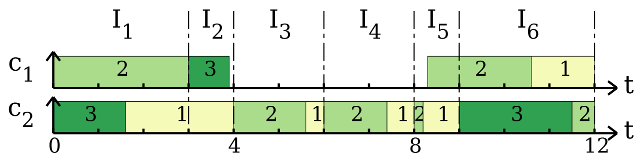
$$Minimize \sum_k P_k \tag{9}$$

Solving the linear program gives a weight for each task on each interval. The next section then details how tasks are scheduled inside intervals.

5.5 Example

To illustrate the effectiveness of our approach, figure 4 pictures the schedule using LPDPM of a task set composed of three tasks (with WCETs and periods of (1.4, 3), (3, 4) and (2.5, 6)) on a two processors system. Global utilization

Figure 4: LPDPM schedule.



is 1.63. The hyper-period is 12 and there are 6 intervals. LPDPM creates only one idle period with this specific task set, less than other schedulers like RUN or U-EDF. The weight of τ' is 1 on intervals 3 and 4 and 0 in intervals 1 and 6. In interval 2, the weight of τ' is only given to τ'_e and is 0.1. In interval 5, the weight of τ'_b is 0.3 to extend the current idle period.

6. ONLINE SCHEDULER

The objective of the online scheduler is to schedule tasks inside intervals according to the weights computed by the linear system. In this section, the idle task τ' includes two subtasks τ'_b and τ'_e. These two subtasks are never executed simultaneously.

This section first details our solution when tasks use their WCET. Then, idle periods are extended when tasks finish their execution before their WCET.

6.1 Tasks using their WCET

Scheduling tasks in an interval is similar to scheduling a set of tasks that share the same deadline. Several schedulers are suitable in such a situation like EDZL [33], FPZL [15] or IZL [26]. However, they have not been designed with the idea of gathering idle periods.

Our solution is based on FPZL (Fixed Priority until Zero Laxity) from Davis and Burn [15]. With FPZL, each job has a static priority and jobs are scheduled according to their priorities. The scheduler monitors pending jobs to schedule them when their laxity becomes zero. The laxity of a job is the difference between the remaining time in the interval and the remaining execution time of a job in the interval. It is enough for the scheduler to track the pending task with the lowest laxity in order to avoid deadline to be missed, as all tasks share the same deadline.

We chose FPZL because the two subtasks of τ' need to be handled differently than the other tasks. Jobs are assigned a priority based on their execution time in the interval, the longer the execution time the higher the priority. And to be executed at the beginning of the interval, τ'_b has the highest priority and τ'_e the lowest.

These two tasks can never be preempted. τ'_b has the highest priority and the only scenario to trigger a preemption of τ'_b would be when a task τ_i reaches a laxity of 0 and no other processor can be preempted. This scenario is impossible because it would mean all processors are running tasks with a laxity of 0 while τ'_b still has a remaining execution time. On the other hand, τ'_e has the lowest laxity and thus can only be scheduled when its laxity reaches 0, and thus cannot be preempted during its execution.

FPZL is not an optimal multiprocessor scheduling algorithm for periodic tasks but is optimal when tasks share a common period. To the best of our knowledge, this optimality has never been proved. Thus theorem 1:

Theorem 1. *FPZL is optimal for implicit-deadline tasks sharing a common period.*

Proof. Suppose a task τ does not finish its execution by the end of the interval, that is a deadline miss occurs. Task τ misses its deadline, thus τ has a negative laxity. This also means that τ was not scheduled when its laxity became zero, that is the scheduler was not able to preempt a task to schedule τ. This can only occur when all scheduled tasks have a zero laxity and cannot be preempted. But if all tasks have a zero laxity, it means all processors are going to be used until the end of the interval while a portion of τ would still have to be executed. And no idle period could have been created before because at least m tasks still have a positive execution time. It means global utilization is greater than m, this is a contradiction to the assumption that global utilization is no more than m. This proves theorem 1. $\square$

6.2 Tasks using their AET

A job may not consume all its WCET but instead an Actual Execution Time (AET), that is $0 < AET < WCET$. This subsection shows how the online algorithm takes advantage of the slack time or dynamic idle time released by tasks when $AET < WCET$. The slack time cannot be known before the execution.

The slack time, noted Δt, is released when a job finishes at a time t earlier than its WCET. Our goal is to increase the length of existing idle periods using the dynamic idle (or slack) time thus generated. By construction, the dynamic idle time is generated by a different processor than the one that executes τ' when τ' is being scheduled (either τ'_b or τ'_e).

How Δt can be redistributed to τ' depends on the state of τ' at time t. We note $\tau'_b.e$ the remaining execution time of τ'_b in the interval (respectively $\tau'_e.e$ for τ'_e). The following scenarios are possible at time t:

- τ'_b is being executed from the beginning of the interval, that is the m^{th} processor executing τ' is in a low-power state. The dynamic idle time can be yielded to τ'_b, i.e. $\tau'_b.e = \tau'_b.e + \Delta t$. The current execution of τ'_b is therefore extended and the processor executing τ'_b stays longer in a low-power state.

- τ'_b is not currently executing. The behavior of the scheduler depends on whether τ'_e is already being executed at time t.

  If τ'_e is not already being executed. The dynamic idle time can be given to τ'_e. Thus $\tau'_e.e = \tau'_e.e + \Delta t$. But τ'_e cannot have a negative laxity, thus portion of the generated idle time may be lost.

  Else it means τ'_e is already being executed. Thus the dynamic idle time cannot be given to τ'_e and is lost.

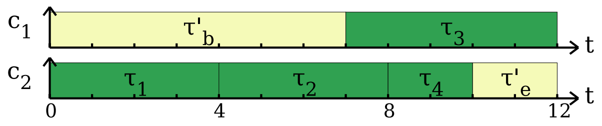
When we state that some idle time is lost, we mean that it cannot be used to extend the idle period represented by τ'_b or τ'_e. However, the scheduler could decide to set another processor in a low-power state if possible. If not, a DVFS strategy could be used in order to reduce the dynamic consumption of the system, as in [14]. As stated previously we currently do not include this DVFS possibility in our simulation in order to only evaluate the gain provided by DPM.

We illustrate how the algorithm behaves when AET is different from WCET in Figure 5. It pictures the schedule of the task set given in Table 2 (during a single interval). We

Table 2: Example task set schedule in Figure 5.

	τ_1	τ_2	τ_3	τ_4	τ'_b	τ'_e
WCET	6	5	5	3	5	0
AET	4	4	5	2		

Figure 5: Schedule of task set from table 2.



suppose the first processor is in a low-power state when the interval starts thus τ' is executed at the start of the interval.

When τ_1 ends at $t = 4$, 2 units of dynamic idle time are released. As τ'_b is currently executed, this dynamic idle time is yielded to τ'_b which now has a remaining execution time of 3. τ'_b then ends at $t = 7$ and the other tasks are scheduled. At $t = 8$, when τ_2 ends, a dynamic idle time of 1 is released and given to τ'_e. When τ_4 ends at $t = 10$, execution time of τ'_e is extended by 1 and τ'_e is executed. To know whether a low-power state can be activated, the scheduler looks at the weight of τ'_e in the following interval.

7. COMPLEXITY

This section discusses both the space and time complexities of LPDPM. These complexities depend on the number of tasks n and the number of intervals I in a hyper-period. I is maximal when no job share the same deadline, that is when periods are prime numbers between them. In this worst-case scenario, each job creates a new interval. However, the n jobs whose deadline is the hyper-period only create one interval. Thus I is equal to the number of jobs in a hyper-period minus $n-1$, the aforementioned jobs whose deadline is H. The number of jobs J in a hyper-period is:

$$J = \sum_{i=1}^{n} \frac{H}{T_i} \tag{10}$$

The number of intervals I is therefore bounded by:

$$I \leq J - n + 1 \tag{11}$$

7.1 Space complexity

For each task, the weights of jobs in each interval must be embedded in the memory of the system as they are used online by the scheduler. Alternatively, execution times of jobs in each interval can be stored (i.e. $w_{j,k} \times |I_k|$). That is $n \times I$ values. However, if this matrix is empty enough, it might be more efficient to store each non-empty value with its interval and task associated. Let e be the number of non-empty values in the aforementioned matrix, $e \times 3$ integers need to be stored instead.

Furthermore, the priorities of all tasks can also be computed offline on all intervals. That is another $n \times I$ values to store. The required space of our solution is thus:

$$min(n \times I, e \times 3) + n \times I \tag{12}$$

Therefore, compared to the size of flash memories used to store read-only data, between several tens of kilobytes and several megabytes, the space complexity is not an issue.

7.2 Time complexity

Our approach limits the online time complexity by performing most of the computations offline. The offline time complexity depends on the number of jobs in a hyper-period, leading to a theoretical combinatorial explosion. However, in practice, typical embedded hard real-time systems usually have a limited number of tasks, between ten and fifteen, and their periods often share common values. The number of jobs in a hyper-period is therefore reasonable. Besides, letting the computation running for several hours is not a problem compared to the time required to design and validate an industrial system. Thus, the time complexity of our linear system is not an issue.

Online, the scheduler chooses the task to schedule using priorities. In our implementation, the priorities of τ_b and τ_e are defined offline, while their weights are dynamically computed, as described in subsection 6. Then in the worst-case, choosing the tasks to run requires pulling the $m + 1$ tasks with the highest priority from the ordered task list because the weight of τ_b can be null. Thus the complexity of the online scheduler is $O(m)$ at each boundary of an interval. Within an interval, scheduling decisions are taken in $O(1)$.

8. EVALUATION

This section compares our solution LPDPM with two other optimal multiprocessor algorithms through simulations. Subsection 8.1 first details the simulation setup. The next two subsections then evaluate both the energy consumption and the number of context switches.

8.1 Simulation environment

A simulator is used to generate random task sets and schedule them. Each simulation is done on two hyper-periods to take into account the transitions between intervals even if we do not consider them in our solution. Each task set contains 10 tasks and is scheduled on 4 processors. For each task set, utilization of each task is computed randomly between 0.01 and 0.99 with a uniform distribution using the UUniFast-Discard algorithm from Davis and Burns [16], an extended version of the UUniFast algorithm from Bini and Buttazzo [10] targeting multiprocessor systems. The period of each task is also chosen randomly between $10ms$ and $100ms$ with a uniform distribution. The task sets with a hyper-period larger than $10s$ are rejected to remain in a realistic bound of typical industrial systems. For each task set, global utilization is set between 3 and 4.

For each utilization value, 500 random task sets are generated and scheduled by three optimal multiprocessor schedulers: RUN [30], U-EDF [27] and LPDPM. RUN and U-EDF are designed to reduce the total number of preemptions and migrations without regard to static consumption. Indeed, to the best of our knowledge, LPDPM is the first optimal multiprocessor algorithm minimizing static consumption. The implementation of LPDPM uses IBM ILOG CPLEX to solve the linear problems. To speed up our analysis, we arbitrarily set to 60 seconds the maximal time allowed to resolve a linear system. With this constraint, the solver does not compute the optimal solution but almost always gives a solution (less than 1% of the task sets were rejected).

Table 3 describes the characteristics of the low-power states we consider in our simulations. The definition of these three states as well as their energy consumption and transition delays is based on an analysis of the DPM hardware capabilities of the STM32L [3] (based on the ARM Cortex-M3 processor) and the FreeScale PowerQUICC microcontrollers (e.g. MPC551x [2]). These microcontrollers are typical in current safety-critical embedded systems within the automotive [12] and energy distribution fields for instance [19]. The energy consumption in the active state is normalized to 1. In the *Sleep* state, only the processor is stopped while the *Stop* state turns off all clocks including the PLL but retains the RAM and register contents. And in *Standby* state, the RAM and register contents are lost.

Table 3: Low-power states used for the simulation.

State	Energy consumption	Transition delay
Run	1	
Sleep	0.5	0.1 ms
Stop	0.1	2 ms
Standby	0.00001	10 ms

8.2 Overall idle energy consumption

Figure 6 pictures the overall energy consumption of RUN and U-EDF compared to LPDPM when processors are idle (i.e. while executing τ' in LPDPM). LPDPM is always the most energy efficient of these three algorithms. This figure shows that using LPDPM is between 5 and 9 times more energy efficient than using RUN and U-EDF when global utilization is equal to 3.1. When global utilization increases, the difference between all schedulers is reduced because the idle time available to use the low-power states is also reduced. And the energy consumption is almost identical when global utilization is close to 4.

The significant difference between all schedulers is due to the negligible energy consumption of LPDPM when global utilization is close to 3. Indeed, the energy consumption in Standby mode is almost zero and the energy consumption of LPDPM is almost only due to the transition delay required to come back to the Run mode. The overall energy consumption of the system could also be computed but it would include the energy consumption while tasks are being executed. However this energy consumption is a constant value and cannot be reduced.

Table 4 shows on average the number of activation of each low-power state for each scheduler and for all the global utilization between 3 and 4. LPDPM mostly uses the most efficient state, i.e. the *Standby* state, and almost never use the least efficient state. On the other hand, other schedulers, especially RUN, often use the *Stop* state and only a limited number of times the *Standby* state. This table is consistent with figure 6 where LPDPM always has the lowest energy consumption and RUN the highest.

The same simulations were also performed with a different number of processors (m set to 2 and 8) and a different number of tasks within the task sets (n set respectively to 5 and 20). The overall idle energy consumption of LPDPM is always lower with LPDPM than with RUN or U-EDF and the difference between all schedulers always is similar.

8.3 Number of preemptions

Figure 7 plots the mean number of preemptions for each

Figure 6: Overall idle energy consumption of RUN & U-EDF compared to LPDPM (energy consumption of LPDPM = 1).

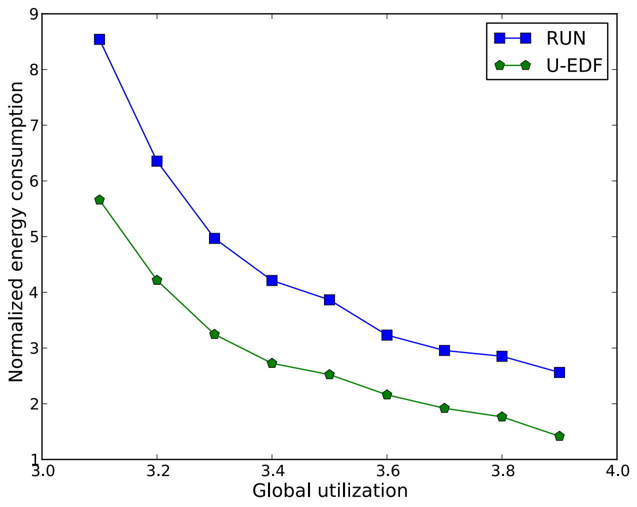


Table 4: Low-power states utilization

	Sleep	Stop	Standby
LPDPM	0.1	2.1	4.6
U-EDF	1.15	18.1	4.4
RUN	15.1	48.3	1.1

scheduler and for each global utilization. Results exhibit that LPDPM is at most 1.3 less efficient than RUN and always slightly better than U-EDF. Note that contrary to these algorithms, LPDPM does not yet try to minimize the total number of preemptions. This makes LPDPM viable, and improving the linear problem to reduce the number of preemptions of the regular tasks is part of our future work.

9. CONCLUSION

Static consumption due to leakage current is a major concern when scaling down semiconductor technology as it dominates dynamic consumption. However, power-aware scheduling algorithms have mainly focused on the reduction of dynamic consumption. They use the low-power states of processors only once the frequency cannot be further reduced. And existing algorithms aiming at reducing static consumption only consider uniprocessor systems or are using partitioned or global non-optimal approaches.

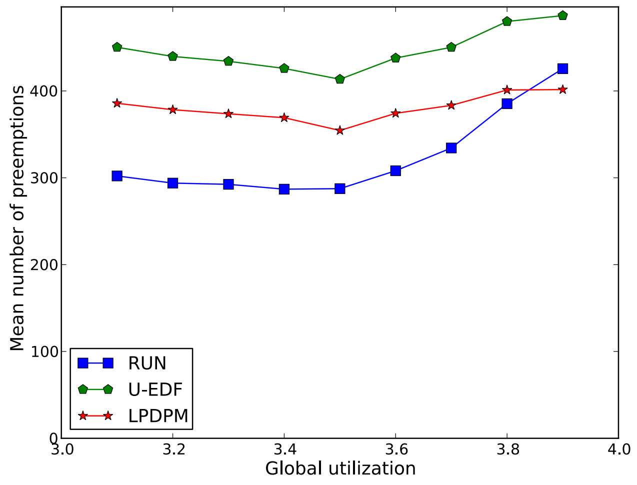
This paper focused on the problem of minimizing the static energy consumption of embedded multiprocessor hard real-time systems. Unlike existing solutions, our solution is offline and thus has a limited online complexity. The hyper-period is divided in intervals and linear programming is used to assign weights to tasks inside intervals. Using linear programming allows to express both the real-time constraints and the objective of minimizing the static consumption. Then, the online scheduler dynamically schedules tasks inside intervals. Online, the scheduler chooses the deepest low-power state that can be activated. The slack time produced by the system when tasks do not consume their worst-case execution time is used to enlarge existing idle periods.

Simulations show that our solution is more energy efficient than recently proposed optimal multiprocessor schedulers. When the idle time is maximal, the idle energy consumption of other schedulers is more than 8 times more important than LPDPM. Besides, LPDPM produces a similar number of preemptions and migrations than these optimal algorithms which have been specifically designed with the objective of minimizing the number of preemptions.

As future work, we plan to define online rules to avoid creating additional idle periods when a task finishes earlier than its WCET by anticipating the execution of active jobs present in the next intervals. Another plan is to study the robustness of our solution using the actual execution time of tasks. Another objective would be to relax the assumption $m - 1 < U < m$ to schedule systems with more idle time to exploit the low-power states of processors. This could be achieved by adding an idle task on each processor.

Finally, we would like to take into account the temperature factor as the energy consumption but also the reliability of multiprocessor systems depend on this parameter. The sleeping processor could for instance be changed to share the load and decrease the temperature of processors.

Figure 7: Mean number of preemptions.



10. REFERENCES

[1] *ARM Cortex-A9 MPCore Technical Reference Manual.*

[2] *FreeScale MPC5510 Microcontroller Family Reference Manual.*

[3] *ST Microelectronics STM32L151xx and STM32L152xx advanced ARM-based 32-bit MCUs Reference Manual.*

[4] T. Austin, D. Blaauw, S. Mahlke, T. Mudge, C. Chakrabarti, and W. Wolf. Mobile supercomputers. *Computer*, 37(5):81–83, May 2004.

[5] M. Awan and S. Petters. Enhanced race-to-halt: A leakage-aware energy management approach for dynamic priority systems. In *Proc. of the 23rd Euromicro Conf. on Real-Time Systems*, pages 92–101, 2011.

[6] M. A. Awan and S. M. Petters. Energy-aware partitioning of tasks onto a heterogeneous multi-core platform. In *Proc. of the 19th IEEE Real-Time & Embedded Technology & Applications Symp.*, pages 205–214, 2013.

[7] 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.*, pages 95–, 2001.

[8] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In *Proc. of the 31st Real-Time Systems Symp.*, pages 14–24, 2010.

[9] M. Bhatti, M. Farooq, C. Belleudy, and M. Auguin. Controlling energy profile of rt multiprocessor systems by anticipating workload at runtime. In *SYMPosium en Architectures nouvelles de machines*, 2009.

[10] E. Bini and G. C. Buttazzo. Biasing effects in schedulability measures. In *Proc. of the 16th Euromicro Conf. on Real-Time Systems*, pages 196–203, 2004.

[11] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *Proc. of the 2010 USENIX conference on USENIX annual technical conference*, pages 21–21, 2010.

[12] D. Chabrol, D. Roux, V. David, M. Jan, M. A. Hmid, P. Oudin, and G. Zeppa. Time- and angle-triggered real-time kernel for powertrain applications. In *Proc. of the Design, Automation Test in Europe Conference Exhibition*, pages 1060–1063, 2013.

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

[14] J.-J. Chen and C.-F. Kuo. Energy-efficient scheduling for real-time systems on dynamic voltage scaling (dvs) platforms. In *Proc. of the 13th IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications*, pages 28–38, 2007.

[15] R. Davis and A. Burns. Fpzl schedulability analysis. In *Proc. of the 17th IEEE Real-Time and Embedded Technology and Applications Symp.*, pages 245 –256, 2011.

[16] R. Davis and A. Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Syst.*, 47(1):1–40, Jan. 2011.

[17] H. Huang, F. Xia, J. Wang, S. Lei, and G. Wu. Leakage-aware reallocation for periodic real-time tasks on multicore processors. In *Proc. of the 5th Intl. Conf. on Frontier of Computer Science and Technology*, pages 85–91, 2010.

[18] K. Huang, L. Santinelli, J.-J. Chen, L. Thiele, and G. C. Buttazzo. Applying real-time interface and calculus for dynamic power management in hard real-time systems. *Real-Time Syst.*, 47, March 2011.

[19] M. Jan, V. David, J. Lalande, and M. Pitel. Usage of the safety-oriented real-time OASIS approach to build deterministic protection relays. In *Proc. of the 5^{th} Intl. Symp. on Industrial Embedded Systems*, pages 128–135, 2010.

[20] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proc. of the 41st annual Design Automation Conf.*, pages 275–280, 2004.

[21] E. Le Sueur and G. Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proc. of the Workshop on Power Aware Computing and Systems*, pages 1–8, 2010.

[22] Y.-H. Lee, K. Reddy, and C. Krishna. Scheduling techniques for reducing leakage power in hard real-time systems. In *Proc. of the 15th Euromicro Conf. on Real-Time Systems*, pages 105 – 112, 2003.

[23] V. Legout, M. Jan, and L. Pautet. Mixed-criticality multiprocessor real-time systems: Energy consumption vs deadline misses. In *1st workshop on Real-Time Mixed Criticality Systems*, 2013.

[24] V. Legout, M. Jan, and L. Pautet. An off-line multiprocessor real-time scheduling algorithm to reduce static energy consumption. In *First Workshop on Highly-Reliable Power-Efficient Embedded Designs*, 2013.

[25] M. Lemerre, V. David, C. Aussaguès, and G. Vidal-Naquet. Equivalence between schedule representations: Theory and applications. In *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symp.*, pages 237–247, 2008.

[26] T. Megel, R. Sirdey, and V. David. Minimizing task preemptions and migrations in multiprocessor optimal real-time schedules. In *Proc. of the 31st IEEE Real-Time Systems Symp.*, pages 37–46, 2010.

[27] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Reducing preemptions and migrations in real-time multiprocessor scheduling algorithms by releasing the fairness. In *Proc. of the 17th Intl. Conf. on Embedded and Real-Time Computing Systems and Applications*, pages 15 –24, 2011.

[28] L. Niu and G. Quan. Reducing both dynamic and leakage energy consumption for hard real-time systems. In *Proc. of the Intl. Conf. on compilers, architecture, and synthesis for embedded systems*, pages 140–148, 2004.

[29] P. Parkinson. Safety, security and multicore. In *Advances in Systems Safety*, pages 215–232. 2011.

[30] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt. Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In *Proc. of the IEEE 32nd Real-Time Systems Symp.*, pages 104–115, 2011.

[31] E. Seo, J. Jeong, S. Park, and J. Lee. Energy efficient scheduling of real-time tasks on multicore processors. *IEEE Trans. Parallel Distrib. Syst.*, 19(11):1540–1552, 2008.

[32] T. Šimunić, L. Benini, and G. De Micheli. Cycle-accurate simulation of energy consumption in embedded systems. In *Proc. of the 36th annual ACM/IEEE Design Automation Conference*, pages 867–872, 1999.

[33] H.-W. Wei, Y.-H. Chao, S.-S. Lin, K.-J. Lin, and W.-K. Shih. Current results on EDZL scheduling for multiprocessor real-time systems. In *Proc. of the 13th IEEE Intl. Conf. on Embedded and Real-Time Computing Systems and Applications*, pages 120–130, 2007.