Power-Aware Real-Time Scheduling on Identical Multiprocessor Platforms

Nicolas Navet, Joël Goossens, Olivier Zendra

Power-Aware Real-Time Scheduling on Identical Multiprocessor Platforms

November 3, 2005

1 Introduction

Context of the study. Energy consumption and battery lifetime are nowadays major constraints in the design of mobile embedded systems. Amongst all hardware and software techniques aimed at reducing energy consumption, supply voltage reduction, and hence reduction of CPU speed, is particularly effective. This is because CPU requires a large amount of energy (e.g., 30W at maximal frequency for an Intel P4 Mobile 1.8GHz [6]) and the energy consumption of the processor is usually at least quadratic in the speed of the processor (see §[6] for more details). The aim is thus to minimize the processor frequency as much as possible while satisfying the performance constraints of the system.

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 [11] and [3], another advantage is that multiprocessor systems are theoritically 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.

Problem definition. In the following, we consider the problem of minimizing the energy consumption needed for executing a set of real-time tasks scheduled on a fixed number of identical processors. The scheduling is preemptive and follows the global EDF policy. "Global" scheduling algorithms, on the contrary to partitioned algorithms, allows 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 preempted by smaller-deadline processes.

We first tackle the problem of choosing the smallest admissible processor frequency for the set of CPUs such that all deadlines will be met considering the worst-case workload. The procedure is performed off-line and provides a static result in the sense that the computed speed does not change over time. Such a static solution is necessary, however, due to the discrepancy between Worst-Case Execution Times (WCET) and Actual-Case Execution Times (ACET), it usually leads to very conservative results. In a second step, we thus propose an on-line "slack reclaiming" scheme that monitors task executions and take advantage of unused CPU time to further reduce frequency.

Existing work. There has been a large number of researches conducted on uniprocessor 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 [4] for a starting point). Among the most interesting studies, one can cite [3] where the authors address the central question of choosing the optimal number of processors[1]. In [4], the authors tackle the case where tasks share a common deadline and propose approximation algorithms with bounded worst-case performances. A study particularly relevant to the settings of this paper is due to Zhu et al. in [13] where the authors propose "slack reclaiming" strategies for the non-preemptive scheduling of a set of dependent/independent frame-based tasks (i.e. all tasks ready at time 0 and sharing a common deadline).

A large number of such "slack reclaiming" approaches have been developed over the years for the uniprocessor case. A first class of techniques [7, 9], known as "compiler-assisted scheduling", divides tasks into sections for which the WCET is known and the processor speed for the rest of the task is re-computed at the end of each section according to the difference between the WCET and the time that was actually needed to execute the task. These algorithms belong to the class of intra-task Dynamic Voltage Scaling (DVS) algorithm [9]. Other strategies dynamically collect the unused computation time at the end of each task and share it among the remaining active tasks (i.e. inter-task DVS). Examples of algorithms following this "reclaiming" approach, include the ones proposed in [10, 2, 8, 12, 1]. Some reclaiming algorithms even anticipate the early completion of tasks for further reducing the CPU speed [8, 2, 1], some having different levels of "aggressiveness" [2, 1].

Contribution of the paper. On the contrary to [3], 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, or it corresponds to the case, frequently encountered in practice, where the characteristic of the set of tasks is unknown at the time when the number of processors is chosen.

First Contribution : to be filled later

The second contribution of our paper is a slack reclaiming algorithm, which, to our best knowledge, is the first of its type for the global preemptive scheduling problem on multiprocessor platforms. This contribution can be considered as an extension to the multiprocessor case of the early proposal of Shin and Shoi in [10], which is usually refered to as "One Task Extension" (OTE). It is proven in the following that our on-line proposal does not jeopardize the feasibility of the system.

Organization of the paper.

2 System model

We assume a platform made of a known number of identical processors upon which a set of real-time tasks is scheduled. The tasks are recurrent sporadic: each task generates an infinite number of successive instances, or jobs, and there is a minimum interarrival time T_i between two successive instances of the same task τ_i. The very first instance a task τ_i can be released anywhere after the origin of time and the release time of this first instance is denoted by R_i. The WCET of each task is C_i and the real-time constraint is that each instance must be fully executed by its deadline, that is $\overline{D}_i$ units of time after the release of the instance.

[1] Ils doivent donc nécessairement trouver une vitesse et évaluer la conso - il faudra se positionner par rapport à ce papier..

3 Off-line speed determination

In the periodic model of hard real-time tasks, a **task** $\tau_i = (C_i, T_i)$ is characterized by two parameters – an execution requirement C_i and a period T_i – with the interpretation that the task generates a *job* at each integer multiple of T_i, and each such job has an execution requirement of C_i execution units, and must complete by a deadline equal to the next integer multiple of T_i. We assume that preemption is permitted – an executing job may be interrupted, and its execution resumed later, with no loss or penalty. A periodic task system consists of several independent such periodic tasks that are to execute on a specified preemptive processor architecture. Let $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ denote a periodic task system. For each task τ_i, define its *utilization* U_i to be the ratio of τ_i's execution requirement to its period: $U_i \stackrel{\text{def}}{=} C_i/T_i$. We define the *utilization* $U_{\mathsf{sum}}(\tau)$ of periodic task system τ to be the sum of the utilizations of all tasks in τ: $U_{\mathsf{sum}}(\tau) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} U_i$. Furthermore, we define the *maximum utilization* $U_{\mathsf{max}}(\tau)$ of periodic task system τ to be the largest utilization of any task in τ: $U_{\mathsf{max}}(\tau) \stackrel{\text{def}}{=} \max_{\tau_i \in \tau} U_i$.

We consider in this work identical multiprocessor platforms, multiprocessor machines in which all the processors are identical and have a speed (or a computing capacity) of s with the interpretation that a job that executes on a processor of speed s for t time units completes $s \times t$ units of execution. A processor of computing capacity s is called a s-processor in the following. Notice that the task's computing requirements ($C_i's$) are expressed by definition for a 1-processor.

Off-line processor speed determination is the process of determining, during the conception of the real-time application system, the lowest (if any) processor speed s in order to schedule the periodic task set τ upon a identical multiprocessor platform composed by m s-processors.

We shall use the following result from [5], which relates feasibility upon (non-identical) multiprocessor platforms to EDF-feasibility upon identical multiprocessors.

Theorem 1 (Theorem 5 from [5]) *Let I denote a hard-real time instance of jobs, which is feasible on a multiprocessor platform with total computing capacity S_{sum} in which the fastest processor has a computing capacity S_{max}. Instance I is scheduled to always meet all deadlines on m processors each of computing capacity s by EDF, provided*

$$S_{sum} \leq m \cdot s - (m-1)S_{max} \tag{1}$$

■

We consider in this work the scheduling of sporadic tasks, the instance I is generated by a sporadic task τ, moreover we know that τ is feasible upon a (non-identical) multiprocessor platform with the total computing capacity be at least U_{sum}, and the fastest processor be of speed at least U_{max}. Consequently, from Equation 1, we can derive an expression for the minimum speed in terms of the number of processors $m, U_{\mathsf{sum}} and U_{\mathsf{max}}$

$$s \geq U_{\mathsf{sum}} + (m-1)U_{\mathsf{max}} \tag{2}$$

3.1 Algorithm $\mathsf{EDF}^{(k)}$

Following the idea used in [5], but adapted for our off-line speed determination where the number of processors is *fixed*, we shall present an improvement on the speed needed in order to schedule a periodic task set.

Using pure EDF is not necessary mandatory nor the best scheduling rule in order to minimizes the static processor speed. In particular if $U_{\max}$ is large, based on Equation 2 the corresponding speed will be large as well. The idea is to adapt the scheduling algorithm, and instead of using pure EDF, using a semi-partioning approach. Indeed, we can often schedule the periodic task system τ on less powerful platform than the speed $U_{\text{sum}} + (m-1)U_{\max}$.

Algorithm EDF$^{(k)}$ assigns priorities to jobs of tasks in τ according to the following rule:

For all $i < k$, τ_i's jobs are assigned highest priority (ties broken arbitrarily) — this is trivially achieved within an EDF implementation by setting all deadlines of τ_i equal to $-\infty$.

For all $i \geq k$, τ_i's jobs are assigned priorities according to EDF.

That is, Algorithm EDF$^{(k)}$ assigns highest priority to jobs generated by the $k-1$ tasks in τ that have highest utilizations, and assigns priorities according to deadline to jobs generated by all other tasks in τ. (Thus, "pure" EDF is EDF$^{(1)}$.)

We introduce the notation $\tau^{(i)}$ to refer to the task system comprised of the $(n-i+1)$ minimum-utilization tasks in τ:

$$\tau^{(i)} \stackrel{\text{def}}{=} \{\tau_i, \tau_{i+1}, \ldots, \tau_n\} .$$

(According to this notation, $\tau \equiv \tau^{(1)}$.)

From [5], proof of Theorem 8 we get:

Theorem 2 *Periodic task system τ will be scheduled to meet all deadlines on m s-speed processors by Algorithm EDF$^{(k)}$ provided*

$$s = \max\{U_{max}(\tau), U_{sum}(\tau^{(k)}) + (m-k)U_{max}(\tau^{(k)})\} \tag{3}$$

Corollary 1 *Periodic task system τ will be scheduled to meet all deadlines on m*

$$s_{\min}(\tau) \stackrel{\text{def}}{=} \min_{k=1}^{n} \left\{\max\{U_{max}, U_{sum}(\tau^{(k)}) + (m-k)U_{max}(\tau^{(k)})\}\right\} \tag{4}$$

$s_{\min}(\tau)$-capacity processors by EDF$^{(\ell)}$ (ℓ corresponds to the semi-partioning which requires m $s_{\min}(\tau)$-processors).

Example 1 *Consider a task system τ comprised of five tasks:*

$$\tau = \{(9,10), (14,19), (1,3), (2,7), (1,5)\};$$

for this system, $u_1 = 0.9$, $u_2 = 14/19 \approx 0.737$, $u_3 = 1/3$, $u_4 = 2/7 \approx 0.286$, and $u_5 = 0.2$; $U_{sum}(\tau)$ consequently equals ≈ 2.457.

Suppose we have to schedule τ on 3 processors, according to Equation 2 the minimal speed is $\frac{2.457+2\cdot 0.9}{3} = 1.41$ but using EDF$^{(2)}$ from Equation 3, we know that the processor speed is $\max\{0.9, \frac{2.457-0.9+1\cdot 0{,}737}{2} = 1.147$.

4 On-line speed reduction : Multiprocessor One Task Extension

Open questions.

- do we allow non synchronous system ($R_i \neq 0$) ? No problem but the static solution must be able to cope with it.
- shutdown if idle.
- does the heuristic actually requires identical processors ?
- two cases : vitesses différentes ou pas
- attention $C_{k,n}$must be defined as a number of processor cycles

In this section, we present a low-complexity on-line algorithm that aims to further reduce the frequencies of the CPUs by performing "local" adjustments, when it is safe to go beyond the Minimum Required Speed (MRS) computed in Section 3. In particular, this technique takes advantage of early task completions (i.e. WCET>ACET) and uses this "slack time" to decrease the CPU speeds for the pending tasks.

This problem has been extensively studied in the uniprocessor case (see, for instance, [10, 2, 8, 1]) and solutions that perform close to the optimal have been devised [1]. In the multiprocessor case, the problem is particularly intricate because it is very difficult to predict at run-time how a change in the frequency of a CPU will affect feasibility. An increase in the frequency might even lead a feasible system to become unfeasible \CITE{Joel-processor-anomaly}. A nice solution has been presented for the multiprocessor case in [13] but it is targeted to a frame-based tasks model in the non-preemptive case and not for the preemptive scheduling of sporadic tasks, which is the context of our paper.

We term our proposal MOTE for Multiprocessor One Task Extension since it is a mutiprocessor version of the technique proposed in [10] and usually refered to as OTE. The idea is simple: the frequency of a CPU can safely be reduced below the minimum required speed (MRS, see Section 3) during the execution of a job if the reduced speed does not change anything w.r.t. the schedule at the MRS for the subsequent jobs scheduled on that CPU. More precisely, subsequent jobs will not be delayed by more higher-priority workload than under MRS.

4.1 Notations

One denotes by $E_{k,n}$ the actual end-of-execution time of $\tau_{k,n}$, the n^{th} instance of task τ_k, $A_{k,n}$ the release time of $\tau_{k,n}$ and $B_{k,n}$ the actual time at which, for the first time, $\tau_{k,n}$ is granted a CPU. The i^{th} out of the m CPUs of the system is denoted by $\mathcal{P}_i$. The ready queue, denoted *ready-Q*, holds all the pending jobs (i.e. ready to be executed but waiting for a CPU) sorted according to the EDF rule where ties are broken according to an arbitrary rule. Another queue, denoted *NextArrival-Q*, stores, sorted by increasing release dates, the set of upcoming jobs that will be released over a time period at least equal to the maximum relative deadline of all tasks.

The function $\Pi(t_1, t_2) \in \mathbf{Z}$ indicates the minimum number of unused processors at time t_2 estimated at time $t_1 \leq t_2$. By convention, $\Pi(t_1, t_2)$ returns minus the number of tasks waiting in the *ready-Q* if no processor is available at time t_2. $\Pi(t_1, t_2)$ is defined to be left continuous in its second variable: $\lim_{\epsilon \to 0, \epsilon > 0} \Pi(t_1, B_{k,n} - \epsilon) = \Pi(t_1, B_{k,n})$. Since the scheduling is non-idling, it should be noted that if $\Pi(t_1, t_2^+) > 0$ there cannot be one or several pending jobs at time t_2. Figure 1 shows an example of the evolution of $\Pi(A_{k,n}, t_2)$, that is the estimation made at the arrival time of instance $\tau_{k,n}$ of the minimum number of unused processors in the future.

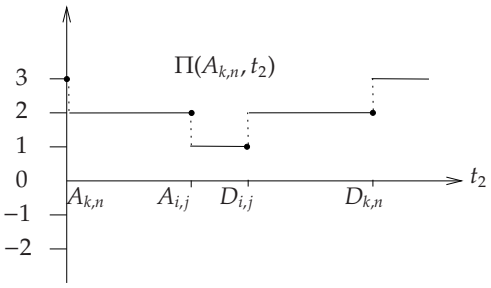

Figure 1: Evolution of $\Pi(A_{k,n}, t_2)$ over time on a 3 processor system. The amount of available processors is lowered by one after each job arrival and increased by one at each deadline since the system feasibility ensures that the corresponding job will be finished.

4.2 MOTE scheme

Under global EDF, since the priorities of the jobs are constant over time, the job executed on a CPU can only change upon the end-of-execution of a job or the release of a job. In our scheme, the speed reduction holds for the whole execution of a job and is decided when the job is allocated to a CPU, for the first time (*i.e.* at time $B_{k,n}$ for $\tau_{k,n}$) or when it resumes after being preempted. Upon its release, a job migrates from the *NextArrival-Q* to the *ready-Q* if it cannot obtain 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".

4.2.1 Principle

When a job $\tau_{k,n}$ is to be allocated to a CPU i, at a time t which is either its arrival or the end-of-execution of a higher priority job, one conservatively estimates the first point in time at which an upcoming job may not find any free CPU. This point in time is $t_{next} = \min\{t_1 \geq t \,|\, \Pi(t, t_1) \leq 0\}$.

Since it will not induced any additional interferences between jobs, speed for $\tau_{k,n}$ can be safely reduced in such a way as $\tau_{k,n}$ finishes at $\min\{D_{k,n}, t_{next}\}$ if the corresponding speed is lower than MRS. Let $s_{k,n}$ denotes the processor speed for $\tau_{k,n}$, one has $s_{k,n} = \min(MRS, \frac{C_{k,n}}{\min\{D_{k,n}, t_{next}\}})$. The evaluation of $\Pi(A_{k,n}, t)$ can be done with the knowledge of the deadlines of the jobs currently under execution and by sweeping the *NextArrival-Q* and *ready-Q* with a running time linear in the size of the queues (see Figure 1 for an example).

4.2.2 Algorithmic description

There are two situations where the decision to reduce or not the CPU speed for a job $\tau_{k,n}$ can be taken:

1. upon its release at time $A_{k,n}$; three cases arise:

   (a) $\Pi(A_{k,n}, A_{k,n}) \geq 2$: the job is allocated to any available CPUs, speed may be reduced according to §4.2.1.

   (b) $\Pi(A_{k,n}, A_{k,n}) \geq 1$: the job is allocated to the single available CPU, speed may be reduced according to §4.2.1.

(c) $\Pi(A_{k,n}, A_{k,n}) \leq 0$: the EDF rule applies, $\tau_{k,n}$ either preempts one of the jobs currently under execution or is inserted in the *ready-Q*. No speed reduction is possible at that point in time.

2. job $\tau_{k,n}$ is at the head of the ready-Q and at time $E_{i,j}$, at least one job has just finished to be executed, 2 cases arise:

   (a) $\Pi(E_{i,j}, E_{i,j}) \geq 2$: the job is allocated to any available CPUs, speed may be reduced according to §4.2.1.

   (b) $\Pi(E_{i,j}, E_{i,j}) \geq 1$: the job is allocated to the single available CPU, speed may be reduced according to §4.2.1.

It worth noting that a job whose speed has been changed upon its arrival (case 1) will not be preempted in the future and thus will not be stored in the *ready-Q* before its end of execution. Furthermore, a job can go through case 2 only once since it will not preempted again. The speed of a job in situation 1.(c) at its arrival can possibly be reduced below MRS in the future when the job is at the head of the *ready-Q*.

5 Perspective

1. Intuitivement, la règle d'allocation Round-Robin des proc doit faire mieux que "smaller CPU index first" - à voir sur des simulations ..

2. On se situe a un instant $A_{k,n}$, on peut faire mieux en assignant les n prochains jobs dans la file NextArrival à des processeurs particuliers. Il y a des morceaux de temps CPU libre et il faut faire rentrer des jobs dedans .. c'est un pb de bin packing, il existe des heuristiques en-ligne à performances garanties (best-fit, first-fit, ..) .. sort certainement du cadre de ce papier.

6 Experiments

References

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

[2] H Aydin, Melhem R., D. Mossè, and Mejia-Alvarez P. Dynamic and agressive scheduling techniques for power aware real-time systems. In *22th Real-Time Systems Symposium*, pages 95–105, 2001.

[3] 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.

[4] J.-J. Chen, H.-R. Hsu, K.-H. Chuang, C.-L. Yang, A.-C. Pang, and T.-W. Kuo. Multiprocessor energy-efficient scheduling with task migration considerations. In *ECRTS*, pages 101–108, 2004.

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

[6] F. Gruian. *Energy-Centric Scheduling for Real-Time Systems*. PhD thesis, Lund Institute of Technology, 2002.

[7] D. Mossè, H Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *Workshop on Compiler and Operating Systems for Low-Power*, 2000.

[8] P. Pillai and K.G.Shin. Real-Time Dynamic Voltage Scaling for Low Powered Embedded Systems . *Operating Systems Review*, 35:89–102, October 2001.

[9] D. Shin, J. Kim, and S. Lee. Intra-task voltage scheduling for low-energy hard real-time applications. *IEEE Design & Test of Computers*, 18(2), 2001.

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

[11] W. Wolf. *Computers as Components: Principles of Embedded Computer Systems*. Morgan Kaufmann, 2000. 155860541X.

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

[13] D. Zhu, R. Melhem, and B. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, page 84, Washington, DC, USA, 2001. IEEE Computer Society.