2010 31st IEEE Real-Time Systems Symposium

Minimizing task preemptions and migrations in multiprocessor optimal real-time schedules

Thomas Megel, Renaud Sirdey, and Vincent David
CEA, LIST, Embedded Real Time Systems Laboratory
Point Courrier 94, F-91191 Gif-sur-Yvette, France.
Email: Firstname.Lastname@cea.fr

***Abstract*—We present a new approach to decrease task preemptions and migrations in optimal global real-time schedules on symmetric multiprocessors. Contrary to classical approaches, our method proceeds in two steps, one off-line to place jobs on intervals and one on-line to schedule them dynamically inside each interval. We propose a new linear programming formulation and a local scheduler which exhibits low complexity and produces few task preemptions and migrations. We compare our approach with other optimal scheduling algorithms, using the implicit-deadline periodic task model. Simulation results illustrate the competitiveness of our approach with respect to task preemptions and migrations.**

I. Introduction

Among the several characteristics of global real-time schedulers for multiprocessor systems, such as embedded multicore architectures, a particularly important one is the optimality of the underlying scheduling algorithm. That is, whether or not the scheduler is guaranteed to obtain a *correct schedule* (i.e. without missing any deadlines) whenever the task set is feasible.

Non-optimal schedulers (e.g. Global-EDF, static-priority class schedulers) generally exhibit fewer task preemptions and migrations than optimal ones. However they do not guarantee real-time constraints beyond specific utilization bounds [8]. It is thus necessary to use more resources to obtain correct schedules given the same task set. Moreover, necessary and sufficient conditions for schedulability do not always exist. Lastly, processors idle time cannot be systematically avoided, so precious resources are wasted.
On-line optimal schedulers, achieving an utilization bound equal to the system capacity, are often criticized with respect to their complexity and the fact that they produce many task preemptions and migrations. Consequently, they are theoretically efficient but not necessarily relevant in practice if we consider preemption and migration costs: context switching and scheduling decisions may lead to high overheads.

Recently, people have tried to improve feasibility tests of non-optimal schedulers [4] because they produce less preemptions than optimal ones. Our goal is to show that it is also possible to use optimal schedulers with few task preemptions and migrations.

There is a trend to privilege scheduling at job-boundary instants such as Bfair [19], LLREF [9]. Our work follows the same idea. We show that our problem is equivalent to scheduling a finite job set on consecutive intervals. Consequently, it is possible to use the job-boundary weighted schedule representation already presented in [14]. This representation allows to express real-time constraints in a linear form. Solving this linear system of inequalities provides an exact feasibility test (necessary and sufficient conditions for schedulability) and the correct amount of jobs to schedule in each interval, dynamically. The linear system, solved for example using the simplex algorithm, usually has several solutions. In order to select a better solution with respect to job migrations and preemptions we propose a mixed integer formulation with suitable objective functions.

Most schedulers are either entirely static or entirely dynamic. Static ones generate a fixed schedule, so they cannot handle unpredictable tasks as such (e.g. non-critical event-triggered tasks with unknown release dates) or execution hazards. We think that it is worth dividing the scheduler work to decrease on-line complexity while still allowing dynamic behavior, which leads to a more robust system. From a methodological standpoint, it is usual practice to invest in compilation time in the embedded context, at least towards the end of the software cycle.

We consider in this study independent periodic tasks to ease the comparison with other optimal/non-optimal schedulers. Note that independent does not mean that we cannot handle communication: for example, the synchronization problem is possibly solved using lock-free mechanisms. As we will see in section 3, our work can be extended to more general task models such as the OASIS task model [10].

In the next section, we present related works. Section 3 defines our general approach and section 4 describes our linear model. Section 5 evaluates the effectiveness of the approach compared to known algorithms. Section 6 concludes about our contribution and presents future works.

II. Related Works

In the remaining sections, we use the following notations: we consider a set Γ of N tasks executed on M processors, each task T releases jobs periodically every period $T.p$. We use the following terminology: we define H_Γ the *hyperperiod* as the least common multiple and G_Γ as the greatest common divisor of all task periods in the Γ task set.

A. Optimal scheduling

Pfair class schedulers (PF [6], PD [5], PD^2 [1]) achieve optimal scheduling for periodic real-time tasks on multiprocessors. These algorithms are based on fluid scheduling and try to follow closely the ideal allocation using a dynamic behavior and a quantum-based approach: each task is divided into quantum-sized pieces, called subtasks, which have pseudo deadlines. As a consequence, all these schedulers generate many preemptions, at most $M{*}H_\Gamma$ per hyperperiod. In the Pfair family, PD^2 is known to be the most effective of this class in terms of complexity ($O(min(N, MlogN))$.
Khemka *et al.* proposed a static optimal scheduling [13], which is incrementally built on intervals based on G_Γ multiples. Tasks allocation priorities are given by increasing task period. However, the complexity and the number of job preemptions depend of the ratio between H_Γ and G_Γ which leads to high overheads when periods are mutually prime.
LLREF [9] is another optimal algorithm which takes dynamic decisions on local intervals defined by job releases. A time and local remaining execution-time plane abstraction is used to show the two possible events on an local interval: either a job ends or a job gets the priority (this job has a zero local laxity). Jobs with largest remaining time are running until one of these two events occurs. An upper bound of this scheduler invocations is $O(N)$.
EKG [2] is an algorithm based on EDF allowing task splitting in groups of K processors. It can be used as an optimal one if there is no group ($K = M$, tasks can migrate on all processors) or as a partitioned one (task can only migrate within their group) but with limited utilization bound. It produces job preemptions in function of the number of *jobs scheduled*, at most $2 * K * N'$ where $N' = \sum_{T\in\Gamma}^{N} \frac{H_\Gamma}{T.p}$.

B. Improvements

Some works have tried to improve existing optimal algorithms in terms of task preemptions or migrations.
Zhu *et al.* proposed Boundary fair [19], an improvement of Pfair that applies fluid scheduling only at job boundaries. At the beginning of each interval, a Fair approach chooses which job to place and McNaughton's algorithm [15] schedules them: the number of job preemptions is bounded by $M - 1$ on an interval. This scheduling is pseudo dynamic (because scheduling is static between two job boundaries) and its complexity is $O(N)$. The authors show from experimental results that job preemptions are reduced from 25 to 50 % compared to Pfair.

Table I
Optimal scheduler characteristics comparative

Algorithm Name	Pfair [6] [5] [1]	Bfair [19]	LLREF [9] [11]	SA [13]	EKG [2]
Allocation type	dynamical	pseudo dynamical	dynamical	static	dynamical
Off-line complexity	-	-	-	O(N$*$ H_Γ/G_Γ)	-
On-line complexity	O(min(N, Mlog(N))[1]	O(N)	O(M)[11] O(N)[9]	O(1)	O(N)
Preemptions bound on interval	-	M-1	N+1	-	-
Preemptions bound on hyperperiod	M$*H_\Gamma$	(M-1)$*$ (1+ $\sum_{T\in\Gamma} \frac{H_\Gamma}{T.p}$)	(N+1)$*$ (1+ $\sum_{T\in\Gamma} \frac{H_\Gamma}{T.p}$)	(M+N-1)$*$ H_Γ/G_Γ	2M$*$ $\sum_{T\in\Gamma} \frac{H_\Gamma}{T.p}$
Processor type	identical	identical	identical	uniform	identical

Pfair migration improvement is also possible by adding some heuristics. Aoun *et al.* [3] proposed some heuristics to guide subtasks allocation such as keeping the scheduling of a task on the same processor as much as possible or reducing the migration of the subtasks of the same job. Their experimental results show that the total number of migrations is reduced from 40 to 60% compared to Pfair.
Funaoka *et al.* [11] showed that it is possible to significantly reduce the number of task preemptions with an LLREF-based algorithm. This is possible because this scheduler is work-conserving (i.e. the algorithm never idles processors if there is at least one task awaiting the execution in the system): if there is idle time on an interval, the algorithm tries to fill it with time apportionment. Intuitively, if a task set uses less than the total processors capacity, then a work-conserving scheduler allows less job divisions, and thus causes less preemptions.

Table I summarizes a number of global optimal schedulers theoretical properties. We use the fact that there are at most $1 + \sum_{T\in\Gamma} \frac{H_\Gamma}{T.p}$ intervals during an hyperperiod to evaluate the maximum number of job preemptions for interval-based schedulers. We compare the allocation type, off-line/on-line complexity, job preemptions upper bounds on job-boundary interval and on hyperperiod for full utilization task set, and the type of processors schedulers are running.

III. General approach

In this section we present the definitions needed to understand the schedule representations and the equivalence between them. More precisely, we seek in our context to execute a finite job set. We consider that scheduling a periodic task set is equivalent to schedule a finite number of jobs on intervals (from different lengths) defined by job releases and repeatedly executed at each *hyperperiod*.
We use the fact that the job-boundary weighted schedule

($\mathcal{BWS}$) representation is adapted to our problem. As a consequence, we express the corresponding linear system Σ for the general approach and show how to solve it and schedule its solution.

A. Definitions

Here we recall some definitions and results based on those published in [14]. We consider a task set Γ composed by N independent implicit-deadline periodic tasks.

Definition 1 Jobs and job set.
Given a task set Γ, each task T releases periodically ($T.p$) jobs, each job j is described by its needed execution time $j.e$ and its deadline $j.d$.
All jobs scheduled during the *hyperperiod* H_Γ belong to the job set J_Γ. As we aim to divide H_Γ into several intervals, we define J_k the subset of J_Γ that contains all active jobs in the k^{th} interval (see **Definition** 3).

Definition 2 Schedule representations.
Given a job set J, a schedule representation associates two objects: the set of its correct schedules S_J and an execution function which represent the time spent executing J. Classically we express four scheduling representations:

- $\mathcal{CS}$: the concrete one (indicates when each job is executed and on which processor, i.e. a Gantt chart).
- $\mathcal{AS}$: the anonymous one (indicates when a job is executed on a processor).
- $\mathcal{WS}$: the weighted one (indicates which fraction of a processor is allocated for a job and at which moment).
- $\mathcal{BWS}$: the job-boundary weighted one (indicates which fraction of a processor is allocated for a job with the restriction that fractions only change at job boundaries).

Result 1
It has been shown that these 4 representations are equivalent, the reader is referred to [14] for proofs and thorough illustration of these representations.

Definition 3 Hyperperiod and job-boundary interval I_k.
It is sufficient to execute a task set on the *hyperperiod* to find a complete schedule. Considering all jobs of all tasks J_Γ, it is possible to divide the hyperperiod duration in a finite number of consecutive intervals defined by job releases (see example Fig. 1). Given I the set of intervals and I_k the k^{th} interval, we define the duration $|I_k|$ between time t_k and time t_{k+1}:

$$|I_k| = t_{k+1} - t_k$$

In our context, we always know the next job boundary i.e. when time t_k is reached then t_{k+1} is known. This is the case for periodic task model, or more generally *time-triggered* task model.

Definition 4 Job and subjob weight.
The weight w_j of a job $j \in J$ is defined as the fraction of processor necessary to execute the job during the time between its release date and its deadline. Given the fact that a job j can be present on different intervals (i.e. a job can be divided in several subjobs), we denote $w_{j,k}$ the subjob weight of job j on interval I_k. As for jobs, the weight of a subjob is defined as the amount of processor necessary to execute job j but only on interval I_k.

Definition 5 E_j set.
E_j corresponds to the set of interval numbers in which subjobs of job j can run: it can be one or several intervals. For example job j_{22} on Fig. 1-a can be scheduled on interval I_2 and I_3, so $E_{j_{22}} = \{2, 3\}$.

Result 2
As shown hereafter, a scheduling problem for finite job sets can be expressed as a simple linear programming problem[1].

B. Linear equalities and inequalities system presentation

We define our linear equation system Σ that allows:

- To verify schedulability of a given job set on a multiprocessor (composed of M identical processors).
- To obtain subjob weights on each interval, by solving Σ (if the job set is schedulable).

First, we express the validity inequalities:

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

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

The first inequality (1) means that the sum of all subjob weights on an interval does not exceed the processors maximum capacity. The second one (2) expresses the fact that each subjob weight does not exceed each processor maximum capacity: it is a necessary condition to avoid that the same subjob is executed at the same time on different processors. Then we have the set of correctness equations,

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

describing that the duration of a job is equal to the sum of its corresponding subjob durations, i.e. jobs must be completely executed.

C. Solving approach

Solving the linear program Σ given by Eqs. (1), (2) and (3) leads to find the set of weights $w_{j,k}$ if the task set is *feasible*. If it is the case, the scheduling problem is reduced to execute subjobs on consecutive intervals. On each interval I_k, the corresponding subjobs have the same start time and deadline, defined by interval boundaries and their execution time are given by $w_{j,k} * |I_k|$. See example on Fig. 1-a: we consider 3 periodic tasks T_1, T_2 and T_3 releasing jobs

[1]which turns out to be a network flow problem, as shown in [7]

periodically (6, 9 and 18 respectively) with execution times 4, 6 and 12 respectively. Processor utilization is equal to $M = 2$. We can write its corresponding system Σ_1 as seen in the previous section: solving Σ_1 gives a set of weights for its corresponding intervals (see Fig. 1-b). So it becomes possible to apply appropriate algorithms to allocate these subjobs on processors for each interval. There are several schedulers for jobs with identical start times and deadlines such as McNaughton [15], Gonzalez [12] and more recently EDZL [18].

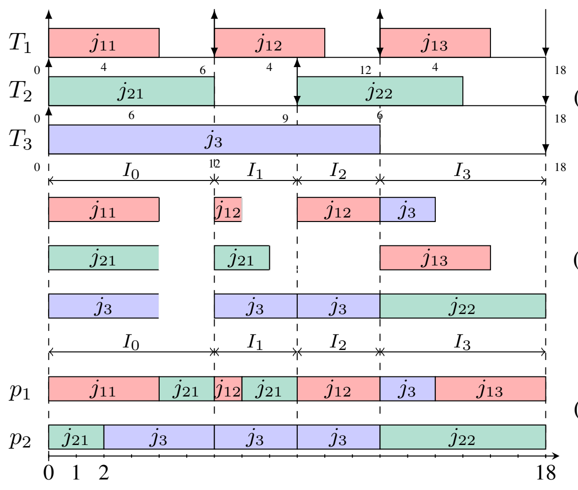

Figure 1. Example with 3 periodic tasks on 2 processors p_1, p_2: (a) tasks T_1, T_2 and T_3 release jobs periodically (6, 9 and 18 respectively) with execution times 4, 6 and 12 respectively. (b) subjob durations sorted by increasing execution time for each interval and for each job solving Σ_1. (c): a corresponding schedule with IZL.

All are suitable (optimal) but we propose another one that is more appropriate because of its dynamic behavior, its low complexity as well as its low task preemptions and migrations bounds.

D. Scheduling description

Our scheduling algorithm, IZL[2] (Incremental scheduling with Zero Laxity), is an improved variant of an algorithm from Lemerre *et al.* (called *Algorithm 1* in their paper [14]). This algorithm uses the following data structures:

- $\mathcal{S}$ is an *array* containing the set of running subjobs. If $\mathcal{S}_i$ is null, then processor i will be idle.
- $\mathcal{P}$ is a *deque* containing the processors with non-urgent jobs (i.e. with positive local laxity). When a processor runs a zero local laxity job, it is removed from $\mathcal{P}$.
- $\mathcal{Q}$ is a *deque* containing the unfinished jobs sorted by increasing execution time. Jobs in $\mathcal{Q}$ are the longest remaining execution time jobs.

[2]IZL has been already succinctly described in short paper [16]

Function `schedule`($\mathcal{S}$, t) puts the jobs given by the array $\mathcal{S}$ for the duration t on their respective processors, and for each scheduled job j decrements its *remaining time* $j.e$, by t. It also decrements the *global remaining time* R by t. Finally, `is_empty`, `pop_`$*$, `push_`$*$ are standard operations on deque.

Algorithm IZL (inputs: J_k, I_k) on M identical processors

```
1  $\mathcal{Q} \leftarrow J_k$ (requirement: jobs sorted by increasing execution time)
2  $R \leftarrow |I_k|$
3  $\mathcal{S} \leftarrow (null, null, ..., null)$
4  for $p$ from 1 to M do
5      if is_empty($\mathcal{Q}$) then do
6          schedule($\mathcal{S}, R$)
7          return
8      end if
9      $\mathcal{S}_p \leftarrow$ pop_first($\mathcal{Q}$)
10     push_last($p$ in $\mathcal{P}$)
11 end for
12 while $\neg$is_empty($\mathcal{Q}$) do
13     $p_{min} \leftarrow$ first($\mathcal{P}$)
14     $L \leftarrow$ last($\mathcal{Q}$)
15     if $\mathcal{S}_{p_{min}}.e \geq R - L.e$ then do
16         schedule($\mathcal{S}, R - L.e$)
17         $p_{max} \leftarrow$ pop_last($\mathcal{P}$)
18         push_first($\mathcal{S}_{p_{max}}$ in $\mathcal{Q}$)
19         $\mathcal{S}_{p_{max}} \leftarrow L$
20         pop_last($\mathcal{Q}$)
21     end if
22     else do
23         schedule($\mathcal{S}, \mathcal{S}_{p_{min}}.e$)
24         $p_{min} \leftarrow$ pop_first($\mathcal{P}$)
25         $\mathcal{S}_{p_{min}} \leftarrow$ pop_first($\mathcal{Q}$)
26         push_last($p_{min}$ in $\mathcal{P}$)
27     end if
28 end while
29 for all $p$ in $\mathcal{P}$ do
30     schedule($\mathcal{S}, \mathcal{S}_p.e$)
31     $\mathcal{S}_p \leftarrow$ null
32 end for
33 schedule($\mathcal{S}, R$)
```

As in Lemerre *et al.*, IZL determines the next preemption or migration instant only at the previous one. It works by monitoring the longest tasks of $\mathcal{Q}$ and exclusively reserving processors when they become urgent ("zero local laxity events", see j_3 at t=2, Fig. 1-c). However, these events are handled differently (*l.15-21*): reserved processors are actually the one executing the longest running jobs (these jobs are then removed and inserted at the head of $\mathcal{Q}$ to free reserved processors). This method guarantees less operations and jobs preemptions.

Invariants First, we have to identify the following necessary invariants to prove algorithm correctness:

I.1 $\mathcal{Q}$ is sorted w.r.t the job durations, $\forall j_1, j_2 \in \mathcal{Q}$:

$$j_1 \text{ placed before } j_2 \text{ in } \mathcal{Q} \Leftrightarrow j_1.e \leq j_2.e$$

I.2 $\mathcal{P}$ sorts the processors w.r.t the length of the job they schedule, $\forall p_1, p_2 \in \mathcal{P}$:

$$p_1 \text{ placed before } p_2 \text{ in } \mathcal{P} \Leftrightarrow \mathcal{S}_{p_1}.e \leq \mathcal{S}_{p_2}.e$$

I.3 A job can appear at most once in $\mathcal{S} \cup \mathcal{Q}$

I.4 Jobs scheduled on $\mathcal{P}$ have the smallest remaining time:

$$\forall j_S \in \mathcal{S}_\mathcal{P},\ j_Q \in \mathcal{Q},\ j_S.e \leq j_Q.e$$

I.5 At any moment, R is the remaining time before the algorithm has been run for $|I_k|$

I.6 The jobs always meet the identical M-multiprocessors schedulability conditions:

$$\forall j \in J_k, j.e \leq R \text{ (a) } \& \sum_{j \in J_k} j.e \leq M * R \text{ (b)}$$

Proofs of correctness

I.1) This invariant is true at the beginning ($\mathcal{Q} = J_k$). Then, there only are two types of operations on $\mathcal{Q}$:

- Extract operation: we extract the first job of $\mathcal{Q}$ during the first **for** loop (*l.9*) and when a job is finished (*l.25*). Moreover, we only extract the last job of $\mathcal{Q}$ when its laxity is zero (*l.20*). Remaining elements of the deque are thus always sorted.
- Add operation: we only add a job that was running at the beginning of $\mathcal{Q}$ (*l.18*). This means that this job was selected because its execution time was less than those remaining jobs of $\mathcal{Q}$ and as it was executed, its execution time decreased (whereas those of $\mathcal{Q}$ did not change). So adding a running job at the head of the deque leads to a sorted deque.

I.2) $\mathcal{P}$ is always sorted regardless the 3 possible operations: a) in the **for** loop allocating processor to sorted jobs, b) in the **if** branch we extract p_{max} the last processor of $\mathcal{P}$, c) in the **else** branch p_{min} runs a new job so p_{min} is removed and replaced at the end of $\mathcal{P}$.

I.3) One can notice that at each time an element is extracted, it corresponds to an allocation (S_p): *l.9*, *l.14* then *l.19-20* and *l.25*.

I.4) In the **for** loop (*l.4*) shortest jobs are taken. In the **while** loop, there are two cases: either a job is finished so we replace it by the first remaining of $\mathcal{Q}$ (*l.23-26*) or the longest running job is replaced by an urgent one and the corresponding processor is extracted from $\mathcal{P}$. So jobs scheduled on $\mathcal{P}$ are always the shortest.

I.5) R is initially set to interval duration and decreases at each time `schedule` is called (*l.6*, *l.16*, *l.23*, *l.30*, *l.33*).

I.6) At the beginning of the algorithm, conditions (a) and (b) are verified by construction, see Eqs. (1) and (2). We define $t = min(R - L.e, \mathcal{S}_p.e)$ and $L = \texttt{last}(\mathcal{Q})$.

I.6-a) Thanks to invariant I.1, I.4 and I.5 we know that,

$$L.e \leq R - (R - L.e) \Rightarrow L.e \leq t$$

$$\forall j \in \mathcal{S} \cup \mathcal{Q},\ j.e \leq L.e \Rightarrow j.e \leq L.e \leq R - t$$

So, schedulability condition (a) is satisfied.

I.6-b) We divide the proof in two cases. The first case is when $|J_k| \leq M$. Then $\mathcal{Q}$ is empty either before the end of the **for** loop (this case is handled *l.5-8*) or at the end of the second **for** loop (*l.29-32*). All jobs are running, so condition (b) is verified. The second case is when $|J_k| > M$. We always subtract the same quantity to both sides of inequality (b) and only when `schedule` is called: at each **while** loop iteration (either in the **if** branch or in the **else** branch) and at the end (*l.30* and *l.33*). This quantity is $M * t$ in the **while** loop, $M * \mathcal{S}_p.e$ in the second **for** loop and R (at the end). So inequality (b) is always verified.

Proof of termination

Proof of termination is done by proving that the couple $(|\mathcal{P}|, |\mathcal{Q}|)$ is strictly decreasing:

- If the **if** branch is taken in the **while** loop, it becomes $(|\mathcal{P}| - 1, |\mathcal{Q}|)$.
- If the **else** branch is taken in the **while** loop, it becomes $(|\mathcal{P}|, |\mathcal{Q}| - 1)$.

The case $(|\mathcal{P}| = 0, |\mathcal{Q}| > 0)$ is not possible, it would imply that job set was infeasible. This is false by assumption. $|\mathcal{P}| = 0 \Rightarrow |\mathcal{Q}| = 0$, so the couple $(|\mathcal{P}|, |\mathcal{Q}|)$ is strictly decreasing.

Properties

- The schedule is built incrementally which means that it is *work-conserving* on the interval and *dynamic*.
- This algorithm has a low complexity: $O(M)$ at the beginning of an interval (*l.4-11*) and $O(1)$ for all other scheduler calls.
- The schedule exhibits at most $M - 1$ preemptions and migrations for each interval I_k (as competitive as McNaughton's one). Migrations and preemptions only happen when jobs of $\mathcal{Q}$ become urgent: an equivalent number of processors are then removed from $\mathcal{P}$. So, at most $M - 1$ processors can be removed (as we have seen above, $|\mathcal{P}| = 0$ implies that all jobs are scheduled), so there are at most $M - 1$ migrations and preemptions.

In this section, we introduced several notions. We have seen how to express real-time constraints through linear equalities and inequalities (system Σ from Eqs. (1), (2) and (3)). We used the job-boundary weighted schedule concept and its equivalence with the concrete one. This linear program can, for example, be solved by the simplex algorithm. Given a solution, the problem is reduced to schedule subjobs with same start times and deadlines. We have shown an efficient algorithm (IZL) to schedule these subjobs. However, a linear program without an objective function produces arbitrary feasible solutions which are not necessary appropriate to limit job preemptions and migrations (see Fig. 2-e,f). This is why we propose to improve this approach by explicitly taking job preemptions and migrations into account.

IV. An integer linear model

In this section, we propose a suitable mixed integer model in order to guide the solution towards the polyhedron vertices which induce less task preemptions and migrations. The main idea is to add constraints to our linear system as well as an objective function which aims at minimizing the number of job preemptions (and by consequence migrations). It is important to note that adding these constraints leaves the set of fractional solutions unchanged.

A. Improvement using additive constraints

In a mixed integer linear programming (MILP) problem, some of the variables are integer and others are continuous. Our model uses boolean and integer variables: the linear problem LP will be also transformed into an MILP. To translate our needs to minimize task preemptions, one way is to minimize the number of subjobs present on each interval.

Definition 6 Job presence in an interval.
To know if a job is present on a interval we use a boolean variable: if its corresponding weight is strictly positive the job will be executed, otherwise not. Given $x_{j,k}$, a boolean which indicates job presence on an interval I_k. We write it as a constraint like this:

$$x_{j,k} \geq w_{j,k} \tag{4}$$

Remark 1 This variable provides necessary though not sufficient condition for presence of a job on an interval. We will address this later in this section.

In order to complete our method, we define now the job preemption when a job j is divided into subjobs (i.e. $|E_j| > 1$). Subjob weight positive on an interval and becoming zero on the next one is a necessary condition for the job to experience a preemption.

Definition 7 Job preemption.
Let $y_{j,k}$ be a boolean which is true when a preemption occurs between interval I_k and I_{k+1}. We define $y_{j,k}$ such as:

$$y_{j,k} = \begin{cases} 1 & \text{if } x_{j,k} = 1 \text{ and } x_{j,k+1} = 0 \\ 0 & \text{else} \end{cases} \tag{5}$$

Remark 2 It would be the same to define $y_{j,k}$ at 1 if $x_{j,k} = 0$ and $x_{j,k+1} = 1$.
For each $y_{j,k}$, this can be modeled using a quadratic constraint such as:

$$y_{j,k} = x_{j,k}(1 - x_{j,k+1}) \tag{6}$$

Still, it is possible to linearize the above equation (6) by replacing it by the following linear constraint set [17].

$$(6) \Leftrightarrow \begin{cases} x_{j,k} - x_{j,k+1} - y_{j,k} \leq 0 \\ -x_{j,k} + y_{j,k} \leq 0 \\ x_{j,k+1} + y_{j,k} \leq 1 \\ -y_{j,k} \leq 0 \end{cases}$$

With these additional constraints we are able to know where job preemptions can appear and on which interval. Note that for each job, there are less preemption variables than presence variables (one less). In order to minimize the number of times the situation happens, we propose to add these constraints to the system Σ.

Objective functions

A first parameter which can be used is minimizing the maximum number of job preemptions to bound all job preemptions. We introduce an integer y_j to count the number of preemptions per job:

$$\forall j, y_j = \sum_{k \in E_j \setminus max_j E_j} y_{j,k} \tag{7}$$

It remains to express how to influence the number of job preemptions. Four possibilities are considered.

1) *Minimization of the maximum number of preemptions.*
This consists in counting the number of preemptions for each job of each task and minimizing the max.

$$\text{minimize } (\max_j y_j) \text{ with } y_j \in \mathbb{N} \tag{8}$$

It can be linearized as follows:

$$\begin{cases} \forall j, b_1 \geq y_j \text{ with } b_1 \in \mathbb{N} \\ \text{minimize } b_1 \end{cases}$$

2) *Minimization of the total number of preemptions.*
This second way consists in adding preemptions of all jobs of all tasks and to minimize the sum.

$$\text{minimize } (\sum_j y_j) \tag{9}$$

3) *Minimization of the total number of job presences.*
This third approach expresses the need to minimize subjobs presence: if we allow less subjob presences then we minimize the number of job preemptions in each interval. We first need to count subjob presences for each job, so we introduce an integer x_j:

$$\forall j, \sum_{k \in E_j} x_{j,k} = x_j \tag{10}$$

It remains to minimize the sum of all x_j:

$$\text{minimize } (\sum_j x_j) \tag{11}$$

4) *Minimization of the total number of job preemptions and presences.*
This last approach consists in summing preemptions and presences of all jobs (or possibly weighted subjobs) of all tasks and to minimize the sum of both. We express an economic function using Eqs. (7) and (10):

$$\text{minimize } (\sum_j x_j + y_j) \tag{12}$$

Necessary limit for $\mathbf{x_{j,k}}$

A simple solution to solve our integer model is to assign all the $x_{j,k}$ variables to 1, whatever minimization criteria chosen. However $w_{j,k}$ will not be necessary strictly positive. To avoid this and to solve issue discussed in *remark 1*, the following approach is proposed.

Maximization of the subjob weights
This technique aims to maximize subjob weights when subjobs are present on an interval ($x_{j,k} = 1$). We use a real variable: $\alpha \in [0;1]$. The goal is to search the maximum value of α which allows to find a solution: the more α is closer from 1, the lesser jobs are divided. The procedure performs a dichotomy to find the critical value. Given $\alpha \in [0,1]$,

$$\begin{cases} \forall j, \forall k, w_{j,k} * |I_k| \geq \min(\alpha * j.e, |I_k|) * x_{j,k} \\ \text{maximize } \alpha \end{cases} \tag{13}$$

The value $\alpha * j.e$ could exceed $|I_k|$ if α is too high which leads to find no solutions (Eq. (2) would be in contradiction with Eq. (13)), so we bound it using a min function.

We now summarize the 4 linear programs corresponding to our 4 objectives functions:

$$\begin{cases} \text{Minimize } b_{1/2/3/4} \text{ with } \begin{cases} b_1 \geq y_j, b_1 \in \mathbb{N}, \forall j \\ b_2 = \sum_j y_j, b_2 \in \mathbb{N} \\ b_3 = \sum_j x_j, b_3 \in \mathbb{N} \\ b_4 = \sum_j x_j + y_j, b_4 \in \mathbb{N} \end{cases} \\ \text{s. t.} \\ \forall j,\ x_j = \sum_{k \in E_j} x_{j,k} \text{ and } y_j = \sum_{k \in E_j \setminus max_j E_j} y_{j,k} \\ \forall y_{j,k}, \begin{cases} x_{j,k} - x_{j,k+1} - y_{j,k} \leq 0 \\ -x_{j,k} + y_{j,k} \leq 0 \\ x_{j,k+1} + y_{j,k} \leq 1 \\ -y_{j,k} \leq 0 \end{cases} \\ \forall j, \forall k, \begin{cases} x_{j,k} \geq w_{j,k} \\ w_{j,k} * |I_k| \geq \alpha * j.e * x_{j,k},\ \alpha \in [0,1] \end{cases} \\ \forall j, \begin{cases} \sum_{j \in J_k} w_{j,k} \leq M, \forall k \\ 0 \leq w_{j,k} \leq 1, \forall k \\ \sum_{k \in E_j} w_{j,k} * |I_k| = j.e \end{cases} \end{cases} \tag{14}$$

All these criteria form our integer model: four linear programs are considered separately (objective b_1, b_2, b_3 and b_4 correspond to Eqs. (8), (9), (11) and (12)) i.e. we have exactly four distinct objective functions.
We propose several ways to minimize job preemptions because these different minimization criteria may generate different results and sometimes one of those may be more appropriate for particular task sets. In order to illustrate our approach we present in the next section the scheduling results for one of these objective functions compared to other global schedulers.

B. Scheduling example

We consider the following example initially proposed by Zhu *et al.* [19] to compare Pfair and Bfair: 6 implicit-deadline periodic tasks ($T.e$,$T.p$): T_1=(2,5), T_2=(3,15), T_3=(3,15), T_4=(2,6), T_5=(20,30), T_6=(6,30). The system utilization is $\sum_{T \in \Gamma}^{6} \frac{T.e}{T.p} = 2$ and $H_\Gamma = 30$. We can notice

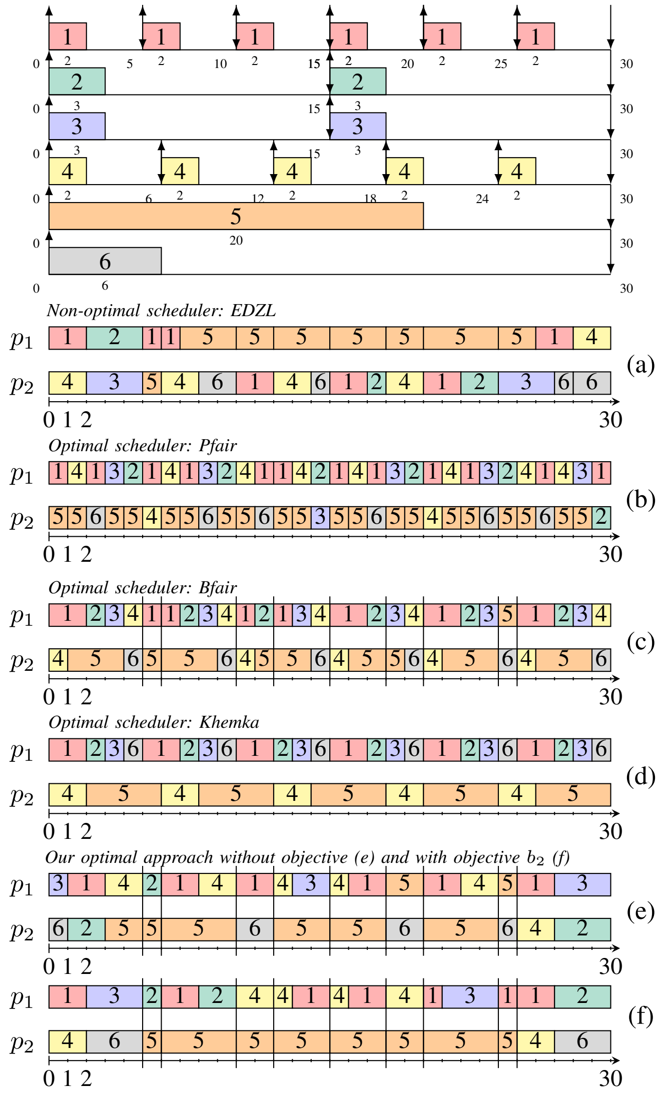

Figure 2. Scheduling example with different algorithms. From top to bottom, context switches and migrations (for 2 processors) are (19,5), (47,7), (40,9), (32,0), (25,6) and (19,2) respectively.

for example that our approach (Fig. 2-f) does not divide T_5 which is the largest task of the set contrary to all other schedulers and contrary to our approach without objective (Fig. 2-e). We also induce less migrations than EDZL on this example. Our solution presented Fig. 2-f is provided by objective function 2. The other objective functions are slightly less powerful but remain interesting: they all produced less context switches than other optimal schedulers. Table II shows scheduling results for each objective function. It confirms that :

1) Several correct solutions can be found from our different objective functions.

2) Solutions are not necessary equivalent in terms of context switches and migrations.

Table II
SCHEDULING COMPARISON BETWEEN THE 4 LINEAR PROGRAMS WITH OBJECTIVE FUNCTIONS ON ZHU EXAMPLE

Objective	Context switches	Task migrations
without (only Σ)	25	6
1 (minimize b_1)	22	4
2 (minimize b_2)	19	2
3 (minimize b_3)	23	9
4 (minimize b_4)	22	6

Switching technique
As we dynamically schedule subjob sets interval after interval, there is no link between intervals: our dynamic scheduling is not aware of the past. For example, if a subjob j is at the end of I_{k-1} on processor p, and is also present on the next interval, the scheduler decision will not take this into account to place this subjob on the same processor p. In order to mitigate this issue and so reduce job migrations, we propose a simple on-line method: we use the fact that, given a valid schedule allocating each task to a processor, the schedule remains valid if we permute a processor with another one.

Procedure
Given a local scheduler and J_k the job set to execute on I_k:

1) At setup time, the scheduler chooses the M first allocated jobs $J \subseteq J_k$.
2) If there were jobs ($J' \subseteq I_{k-1}$) scheduled at the end of interval I_{k-1}, select the C common jobs ($C = J \cup J'$) one by one and permute allocated processors in order to avoid job migration.
3) If $C \neq \varnothing$, select the $J - C$ remaining jobs and allocate them to the remaining processors.

This permutation takes place only at each job boundary and needs to be transparent for the scheduler so there is a temporary virtual mapping of processors. Moreover, the complexity of this procedure is $O(M)$. For the local scheduler, if we use IZL, we slightly increase its complexity at setup time. See for example Fig. 3, with J' = $\{j_{12}, j_3\}$, J = $\{j_3, j_{22}\}$ and $C = \{j_3\}$.

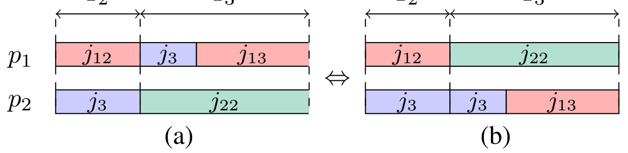

Figure 3. (a) End of scheduling example from fig. 1. (b) Equivalent schedule by permutation of p_1 and p_2 at the beginning of interval I_3.

In this section, we proposed a augmented integer model of our linear program. This model is composed of four objective functions that lead to minimize job preemptions on each interval defined by job boundaries. By reducing job preemptions, we limit also job migrations. In order to improve our model we have seen how to link consecutive intervals with a simple switching technique.

V. EXPERIMENTATION

In order to estimate the practical relevance of our approach in terms of task preemptions and migrations, we conducted a simulation-based experimental study. To that end, we developed a scheduler simulator on the hyperperiod.

A. Context and simulation setup

We consider here four optimal schedulers: Pfair, Bfair, Khemka's one and our approach. In our simulation, we use 100 task sets for each system utilization. Each task set is generated with uniform period and execution time. Periods are randomly generated between 10 and 100, execution times between 1 and the period. We choose integer values for period and execution times because of Pfair and Bfair requirements. Utilization of each task ($u_i = T_i.e/T_i.p$) is in the range [0.01,1] and system utilization ($U = \sum_i u_i/M$) varies between 0.1 and 1. We reject task set with an hyperperiod larger than 2^{32}.

We consider the following rules:

- We count each context switch[3] as the exception of subjobs running on the same processor during consecutive intervals (presents at the end of one and at the beginning of the next one), see for example j_3 on Fig. 3-b.
- We count a job migration if a processor p executes a job j at time t and another processor p' executes the same job at time t' ($t' > t$). We count a job migration also if job does not belong to the same task release.

In order to measure the average number of context switches and job migrations, 100 simulations are conducted (with different task sets).

We then generate corresponding constraints for each task set for all four objective functions. These constraints represent our integer linear program. As we are looking for the critical α value, we use a dichotomic procedure. Actually the solver provides either no result (infeasible because of too high α) or subjob weight results for each interval. After a success, we schedule all subjob weights in each interval with IZL. We choose finally for each U, the objective function which on average produces the best schedule in terms of number of context switches and task migrations. Note that it is possible that a solution is not found for a particular task set on the allotted time (the solving time is limited to 60 sec for each

[3]Note that we propose to count the number of context switches instead of the number of "job preemptions" to ease the measures. There is no difference in our case because minimizing job preemptions leads to less context switches. In the sequel, we will use both terms without distinction.

task set). The solver used is ILOG Cplex. Three possible solving cases occur for any objective functions:

- The solver finds the best integer solution (the best case).
- After the time limit, the solver finds an approximate solution (near the best).
- After the time limit, the solver finds no integer solution.

As we will see in the next section, the two last cases occur marginally in our practical simulations.

B. *Results*

We call our approach : MILP with IZL. Figure 4 and 5 show the results of job preemptions/migrations for each algorithm according to system utilizations: horizontal axis corresponds to the system utilization and vertical axis corresponds to the average number of job preemptions *on the hyperperiod* normalized by the number of task sets. Pfair schedules produce a huge number of preemptions from $U = 0.6$, so we decided to truncate the highest numbers on the curves. Note that this is the same behavior for Bfair and Khemka's one. In Fig. 4, our approach is the lowest curve followed by Bfair, Khemka's one and Pfair. Job preemptions are between two and three times inferior to others optimal schedulers for task utilization comprised from $U = 0.1$ to 0.5. For higher utilization, the gap considerably increases.

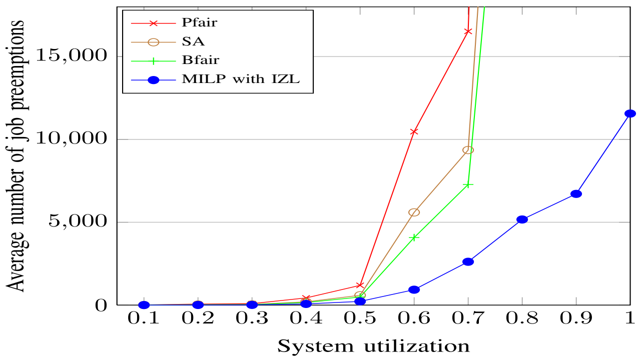

Figure 4. Number of job preemptions for $M = 4$, $U = [0.1, 1]$

Figure 5 shows the results of job migrations for each algorithms. For $U = 0.1$: Bfair is first followed by our approach and the others. From $U = 0.2$ to 0.7, our approach and Khemka's one are very close, followed by other schedulers. For higher utilization, our approach outperforms the others.

Our four approaches generate different schedules which are not equivalent with respect to system utilization: for example best results are obtained either with objective 3 (minimization of job presences) for low utilization or objective 4 (minimization of both presences and preemptions) for high utilization. The α factor varies practically between 0 and 1, and as expected the more utilization system is high the more α decreases: values close to zero are often found for high utilization system.

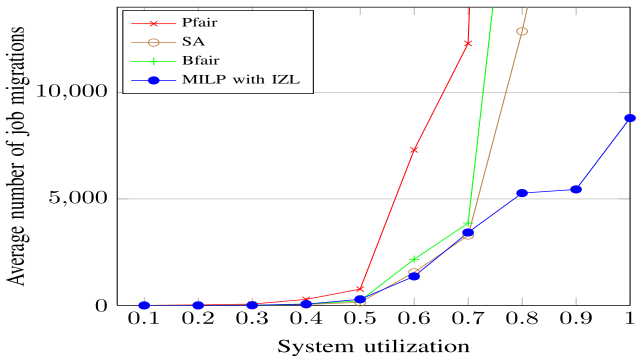

Figure 5. Number of job migrations for $M = 4$, $U = [0.1, 1]$

Towards a lower bound

We are also interested to know how far we are from a lower bound of task preemptions and migrations. Without knowing it, we compared our approach to EDZL, a non-optimal scheduler (*for periodic tasks*) known to be more efficient than Global-EDF and to produce few task preemptions [18]. Simulations show close results for low utilizations. We generate more preemptions than EDZL: between 3% and 31% for system utilization between 0.1 and 0.7. For higher utilization, percentage increases until 300% but EDZL starts to miss its deadlines: the number of correct schedules drops significantly. At $U = 0.8$, EDZL fails on 20% of task sets and until near from 100% for $U = 1$. Concerning migrations, EDZL schedules produce the smallest numbers except for a system utilization of $U = 0.1$. For higher system utilization, our approach produces until five times more migrations than EDZL, but EDZL starts to miss deadlines.

No solutions or approximate solution found

As we expect, some constraint sets (from task sets) provided to the solver do not allow to find a solution for our four objective functions during the solving pass, but there are only few cases on high system utilization, as shown in table III. Another case is met and corresponds to approximate solutions. The time limit invested to let Cplex solving is relatively short (60 sec). It is acceptable to increase this solution time if the user always wants solutions or only exact solutions. There is a trade-off to find between invested time and finding exact solutions. During our experiments, the off-

Table III
NUMBER OF UNSOLVED TASK SETS FUNCTION OF SYSTEM UTILIZATION

System utilization	percentage of missed solutions (with 60 sec for the solver)
0-0.5	0%
0.6	3%
0.7	6%
0.8-0.9	8%
1	10%

line part (solver, dichotomic procedure) takes between 10 sec and 10 min: this time increases with respect to system utilization.

VI. Conclusion

We proposed a new approach to find optimal global real-time schedules on symmetric multiprocessors with task preemption and migration constraints. Contrary to classical approaches, our method is divided in one off-line part and one on-line part. As we have seen, our problem is equivalent to scheduling finite job sets on consecutive intervals and thus it is possible to base our method on the job-boundary weighted schedule representation. This representation allows to express real-time constraints with a linear program. Solving this program provides an exact feasibility test and finds a valid weight set for each interval. We also proposed an integer linear model: we expressed job preemption and presence and we proposed four suitable objective functions which lead to different correct schedules. The on-line part consists in scheduling with an appropriate local scheduler. We chose IZL for its low complexity, its dynamic behavior, its job preemptions and migrations bounds.

Our approach allows to significantly decrease job preemptions and migrations for the periodic task model. Results show that we generate less job preemptions and migrations than other optimal schedules. However, it should be emphasized that our approach may require a significant (several minutes) but acceptable investment in off-line computation time when the number of tasks, the system utilization or the hyperperiod, increase. Still, the improvements induced at run-time appears to be worth that investment (furthermore, as already stated, relatively long compilation durations are common place in the embedded market). Furthermore, it is possible to find a correct trade-off between time used for off-line part and available user time.

In future works, we want to complete our experiments varying the number of processors and adding some other optimal schedulers such as [11] or other non-optimal ones. Moreover, we need to investigate the computational complexity of our MILP and subsequently study faster (though not necessarily exact) resolution algorithms to tackle bigger instances. From a practical viewpoint, it would be interesting to extend schedulability test so as to account for preemption and migration costs. Also, the issue of optimizing the memory footprint of the scheduler working data (weight sets) should be given further attention, for instances with large hyperperiod. Finally we would like to apply our method to time-triggered task models [10].

Acknowledgment

The authors wish to thank Matthieu Lemerre for relevant advices while the present research was carried out.

References

[1] J. H. Anderson and A. Srinivasan. *Early-Release Fair Scheduling*. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 3543, 2000.

[2] J. Andersson, Bjorn and Tovar, Eduardo, *Multiprocessor Scheduling with Few Preemptions*. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 322–334, 2006.

[3] D. Aoun, A.M. Déplanche, Y. Trinquet, *Pfair scheduling improvement to reduce interprocessor migrations*, 16th Int. Conf. on *Real-Time and Network Systems, RTNS'08*, 2008.

[4] T. P. Baker. Comparison of empirical success rates of global vs. partitioned fixed-priority EDF scheduling for hard real time. Technical Report TR-050601, Department of Computer Science, Florida State University, Tallahassee.

[5] S. Baruah, J. Gehrke, and C. Plaxton. *Fast Scheduling of Periodic Tasks on Multiple Resources*. In *Proceedings of the 9th IPPS*, pages 280-288, 1995.

[6] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel. *Proportionate progress: a notion of fairness in resource allocation*, Algorithmica 15 (1996) 600-625.

[7] P. Bratley, M. Florian, and P. Robillard. *Scheduling with earliest start and due date constraints*, *Nav. Res. Log. Quart.*, vol. 18, Dec. 1971.

[8] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson and S. Baruah, *Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms*, in *Handbook on Scheduling Algorithms, Methods, and Models*, 2004, Chapman Hall/CRC, Boca.

[9] H. Cho, B. Ravindran, and E. D. Jensen. *An Optimal Real-Time Scheduling Algorithm for Multiprocessors*. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 101-110, 2006.

[10] D. Chabrol, V. David, C. Aussaguès, S. Louise and F. Daumas, *Deterministic distributed safety-critical real-time systems within the oasis approach*. In *17th IASTED PDCS'05*, 2005.

[11] K. Funaoka, S. Kato, and N. Yamasaki. *Work-Conserving Optimal Real-Time Scheduling on Multiprocessors*. In *Proc. of the 20th Euromicro Conference on Real-Time Systems*, 2008.

[12] T. Gonzalez and S. Sahni. *Preemptive scheduling of uniform processor systems*. *J. ACM*, 25(1):92-101, 1978.

[13] Khemka, Ashok and Shyamasundar, R. K.. *An optimal multiprocessor real-time scheduling algorithm*, In *J. Parallel Distrib. Comput.*, vol.3, n1, pages 37-45, 1997.

[14] M. Lemerre, V. David, C. Aussaguès and G. Vidal-Naquet, *Equivalence between schedule representations: theory and applications*. In *Proceedings of the 14th IEEE RTAS'08*, 2008.

[15] R. McNaughton, *Scheduling with deadlines and loss functions*, In *Management Science*, 1959.

[16] T. Megel, V. David, D. Chabrol and C. Fraboul, *Dynamic Scheduling of Real-Time Tasks on Multicore Architectures*, In *Colloque du GdR Soc/SiP*, 2009, Orsay.

[17] M. Padberg, *Zero-one decision problems*, GBA-Report No. 76-29, April 1976, New York University. Published (in German) in: M. Beckmann et al. (eds.), HandwiSrterbuch der Mathematischen Wirtschafts-wissenschaften (Gabler-Verlag, Wiesbaden, 1978) pp. 187-229.

[18] Hsin-Wen Wei, Yi-Hsiung Chao, Shun-Shii Lin, Kwei-Jay Lin and Wei-Kuan Shih, *Current Results on EDZL Scheduling for Multiprocessor Real-Time Systems*, In *Proceedings of the 13th IEEE RTCSA*, pp.120-130, 2007.

[19] D. Zhu, D. Mossé and R. Melhem. *Multiple-resource periodic scheduling problem: how much fairness is necessary?* In *RTSS'03: Proceedings of the 24th IEEE International Real-Time Systems Symposium*, page 142, 2003.