Scheduling an aperiodic flow within a real-time system using Fairness properties

Annie Choquet-Geniet, Sadouanouan Malo

Scheduling an aperiodic flow within a real-time system using Fairness properties

Annie Choquet-Geniet* — Sadouanouan Malo**

* University of Poitiers. Laboratory of Applied Computer Science. 1 Av. Clément Ader BP 40109 - 86961 Futuroscope Chasseneuil-France.
annie.geniet@univ-poitiers.fr

** Polytechnic University of Bobo Dioulasso. Information Technology High School. 01 BP 1091 Bobo Dioulasso 01, Burkina Faso.
sadouanouan.malo@laposte.net

ABSTRACT. We consider hard real-time systems composed of periodic tasks and of an aperiodic flow. Each task, either periodic or aperiodic, has a firm deadline. An aperiodic task is accepted within the system only if it can be completed before its deadline, without causing temporal failures for the periodic tasks or for the previously accepted aperiodic tasks. We propose an acceptance test, linear in the number of pending accepted aperiodic tasks. This protocol can be used provided the idle slots left by the periodic tasks are fairly distributed. We then propose a model-driven approach, based on Petri nets, to produce schedules with a fair distribution of the idle slots for systems of non independent periodic tasks.

RÉSUMÉ. Nous considérons des systèmes temps-réel composés de tâches périodiques et d'un flux apériodique. Toutes les tâches, périodiques comme apériodiques, sont soumises à des échéances strictes. Une tâche apériodique n'est acceptée que si elle ne remet pas en cause le respect des échéances par les tâches périodiques et par les tâches apériodiques déjà acceptées. Nous proposons un protocole d'acceptation des tâches apériodiques de complexité linéaire en le nombre de tâches apériodiques acceptées présentes dans le système. Ce protocle est utilisable dès lors que les temps creux sont répartis de manière équitable. Nous proposons donc une approche modèle, à base de réseaux de Petri, pour produire des séquences où les temps creux sont équitablement répartis, pour des systèmes de tâches interdépendantes.

KEYWORDS : real-time systems, multiprocessor scheduling, aperiodic flow, Petri nets

MOTS-CLÉS : Systèmes temps-réel, ordonnancement multiprocesseur, flux apériodique, réseaux de Petri

Received, April 20, 2012
Revised, July 5, 2013
Accepted, November 22, 2013

ARIMA Journal, vol. 18 (2014), pp. 93-116.

1. Introduction

1.1. Real-time systems

A real-time system is one whose logical correctness is based both on the correctness of the outputs and on their timeliness [26]. It is often designed to control physical processes, thus it must permanently react to its environment and interact among components within the systems. It is composed of periodic tasks, dedicated to the control activities (temperature acquisition in a nuclear station, robot's trajectory computation, processing of informations provided by a synchronous link, etc.), and of aperiodic tasks which are triggered by aperiodic events (human interaction, alarm activation, error detection etc.). It must satisfy explicit (bounded) response-time constraints or malfunctions might occur, which may have unacceptable consequences e.g. loss of human lives. Thus, a key requirement for real-time systems is the end-to-end delay in the task execution, which is a critical issue in the design and analysis of time critical systems. Here, we focus on hard real-time applications: the tasks (either periodic or aperiodic) have firm deadlines by which they must be completed for safety reasons. E.g. a late computed value can be obsolete, using it may be misleading and even dangerous. Moreover, concurrency, resource sharing, synchronization, and deadlock resolution are also essential issues. Besides, as systems are increasingly complex, the trend is toward the use of several processors to process a large number of tasks. We thus consider a platform composed of m identical processors (for instance a symetric multicore platform), and we assume that the system is preemptive. Moreover, we assume that the preemption costs and the inter-processor migration costs are negligible. We have chosen such architectures firstly because they are rather common, and secondly, because the scheduling results that we use have been set for this kind of architecture. We focus on the temporal validation of the application i.e. on the timeliness which is based on the choice of an appropriate scheduling policy, which can be proved to respect all the temporal constraints.

1.2. Real-time periodic scheduling

Real-time multiprocessor scheduling techniques fall into two general categories: partitioned and global scheduling. Under partitioning, each processor independently schedules tasks from a local ready queue. Each task is definitively assigned to a given processor and is only scheduled on that processor. The main challenge here is the distribution of the tasks among the processors, which is mostly based on bin-packing [18]. In contrast, all ready tasks are stored in a single queue under global scheduling. A task is selected to execute whenever the scheduler is invoked, regardless of which processor is being scheduled. It means that the tasks can run at any time on any processor, they are never definitively assigned to a given one. They may start on one processor and resume on another. Nevertheless, parallelism is prohibited in both schemes: at any time, a task can run on at most one processor. None of these approaches can be considered as better than the other since there exists applications which are feasible under one of the approaches, but not under the other one, and conversely [9]. An application is said to be feasible if there exists a valid schedule which is a schedule which meets all the temporal constraints. We will only consider global scheduling in this paper.
Here again, two approaches are usually considered in order to solve the schedulability

problem, the on-line and the off-line methods. For on-line methods, the scheduling policy is classically priority driven and is implemented within the scheduler. The processors are assigned to the pending tasks with the highest priorities. If independent periodic tasks with implicit deadlines (the due date of any instance is equal to the release date of the next instance) are considered, optimal strategies have been proposed [5, 1] for platforms composed of identical processors, which are based on the fairness property, where an algorithm is said to be optimal if for any application, either it computes a valid schedule or there exists no such schedule at all. But if deadlines may occur before the next release, [17, 29] have proved that no on-line algorithm can be optimal. Furthermore, if critical resources are used, the scheduling problem is NP-hard even for single processor systems [22]. Therefore, only sufficient conditions exist for the schedulability analysis of real-time tasks in multiprocessor environment. On the other hand, off-line methods perform a static pre-runtime schedulability analysis. A valid schedule is computed and then used by the dispatcher at run-time. Most off-line scheduling strategies rely on exhaustive branch-and-bound enumeration techniques. These approaches are generally model-driven and are based on task system modelling using e.g. Petri nets, automata, geometric models...[6, 35, 21, 23, 25, 13].

1.3. Motivation and contribution

We adress the problem of scheduling aperiodic tasks. We assume that they have firm deadlines. Thus, either they can be processed on time, or they must be rejected by the system, e.g. to avoid them to handle obsolete values. Our concern is here to propose an efficient acceptance protocol, where the efficiency is measured by the amount of accepted aperiodic load. The main rule is that an aperiodic task can be accepted only if it can complete execution before its deadline and if it doesn't cause any deadline failure, neither for the periodic tasks, nor for the already accepted aperiodic tasks. We propose an acceptance protocol, which relies on a fair distribution of the idle time units. The accepted aperiodic tasks are scheduled in background, which means that an aperiodic task can be scheduled each time an idle slot occurs in the periodic schedule. This problem has been largely adressed for uniprocessor systems [27, 11, 10]. As to the multiprocessor case, most papers deal with hard periodic tasks but soft aperiodic ones [3, 33, 30], and they consider joint scheduling: a scheduling algorithm schedules the periodic and the accepted aperiodic tasks together. Some authors consider sporadic tasks (there exists a minimum delay between two consecutive releases) [24, 34, 30]. In [31], a method based on the computation of the response time of the aperiodic tasks is proposed. Each time new aperiodic tasks arrive in the system, the response time of the aperiodic traffic is computed, and if required, some of the new tasks are dismissed. Here, the complexity depends on the complexity of the response time computation, which is in O(number of tasks $\times$ the cumulated aperiodic demand). Our objective is to propose a test with a lower complexity. In a first time, we consider independent periodic tasks with implicit deadlines. For such task sets, PFair scheduling strategies are optimal. We thus choose to schedule the periodic tasks with a PFair algorithm, such as PF [5] or PD^2 [1] (PFair scheduling is presented in section 2). In addition, there exists a very simple feasibility test, which is linear in the number of tasks: the task set is feasible if and only if the utilization factor, which is equal to the ratio of the processors activity dedicated to the execution of the tasks, is at most equal to the number of processors. Now, if the application is feasible and has a utilization factor less than the number m of processors, some idle times take place, during which some processors remain idle. We show that it is possible to distribute the idle times in a

regular way and to process aperiodic load each time they occur. We propose an acceptance test which makes use of the regular distribution of the idle times, and which is linear in the number of already accepted pending aperiodic tasks.
Next we consider sets of non independent periodic tasks: they may exchange messages, and use critical resources, i.e. resources which must be used in mutual exclusion. To overcome the lack of optimal on-line strategy, we use an off-line model-driven approach, based on the modelling of the application, including the temporal constraints, by a Petri net. We here adapt the approach proposed in [21]. Then we show how to use the model to get schedules that contain regularly distributed idle times.
The rest of the paper is organized as follows: in section 2 we present the basic definitions and we state our assumptions. In section 3, we present the acceptance protocol and its performance when periodic tasks are independent. Finally, in section 4, we consider non independent tasks, we present our Petri net based model, and show how to use the Petri-net based analysis for the computation of the appropriate schedules. The paper ends with some concluding remarks and perperspectives.

2. Basic definitions and assumptions

2.1. Real-time applications

We consider a multiprocessor platform, with m identical processors and a real-time application composed of a periodic task set $\tau = \{\tau_1,\ldots,\tau_n\}$ and of aperiodic tasks.
Each **periodic** task $\tau_i =< r_i, C_i, D_i, T_i >$ is characterized by four temporal parameters: its first release time r_i, its period T_i, its worst-case execution time (WCET) C_i, and its relative deadline D_i, which is the maximum delay allowed from the release of any instance of the task to its completion (see figure 1). We here assume that $C_i \leq D_i \leq T_i, \forall \tau_i \in \tau$.
If the first release times are all equal, the application is said to have **simultaneous first releases**, otherwise it is said to have **deffered first releases**.
A task τ_i consists of an infinite set of instances (or jobs) $\tau_{i,k}$ ($k \in \mathbb{N}$). The instance $\tau_{i,k}$ is released at time $r_{i,k} = r_i + k \times T_i$, and must be completed by its (absolute) deadline $d_{i,k} = r_i + k \times T_i + D_i$.

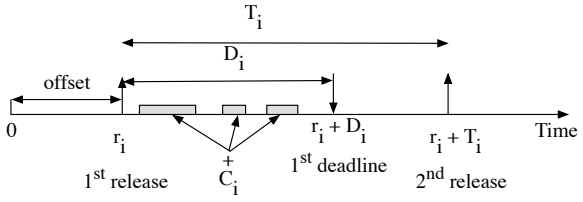

Figure 1. *Temporal modelling of a real time periodic task.*

The **hyperperiod** of a periodic task set is the least common multiple of the periods of the tasks: $H = lcm(T_i)_{i=1..n}$.
The processor **utilization factor** characterizes the processor workload due to the task set.

It is defined by $U = \sum_{i=1}^{i=n} \frac{C_i}{T_i}$. If $U > m$ the system is over-loaded and temporal faults cannot be avoided [8].
An **aperiodic** task τ_s is characterized by an arrival time r_s, a worst case execution time C_s and a relative deadline D_s. Its absolute deadline is $d_s = r_s + D_s$. An important difference between periodic tasks and aperiodic ones is that the release time of each instance of each periodic task is known before runtime, whereas the arrival time r_s of an aperiodic task is not known.
An aperiodic **flow** is a series of aperiodic tasks $\tau_{s,1}, \tau_{s,2}, ...$ (with $r_{s,1} \leq r_{s,2} \leq .....$).

2.2. Schedules

For any $t \in \mathbb{N}$, the **slot** t denotes the time interval [1] $[t, t+1)$. A task is said to be processed at time t if it is processed by one processor during the slot t. We then define the notion of schedule.

Definition 2.1. *Let $O_m(\tau)$ denotes the set of subsets of τ of cardinality less than or equal to m. A* **schedule** *on m processors is defined by $S : \mathbb{N} \to O_m(\tau)$ such that $\tau_i \in S(t) \Leftrightarrow \tau_i$ is scheduled at time t on one processor.*

Let S_i be such that $S_i(t) = \begin{cases} 1 \text{ if } \tau_i \in S(t) \\ 0 \text{ else} \end{cases}$. In order to be temporally **valid**, a schedule must respect the temporal constraints:

$$(1)\ \forall i \in 1..n, \sum_{t=0}^{r_i-1} S_i(t) = 0 \text{ and,}$$

$$(2)\ \forall k \in \mathbb{N}^* \sum_{t=0}^{r_i+(k-1)T_i+D_i-1} S_i(t) = \sum_{t=0}^{r_i+kT_i-1} S_i(t) = k \times C_i.$$

The equation (1) insures that the task doesn't start execution before its first release date, and the equation (2) insures that exactly one instance of the task is processed between any release and the next deadline. If there exists a valid schedule, the system is said **feasible**.
If $\mid S(t) \mid < m$ there are [2] $m - \mid S(t) \mid$ **idle time units** at time t.
For any periodic (resp. aperiodic) task τ_i (resp. τ_{s_i}), $RCT_i(t)$(resp. $RCT_{s_i}(t)$) denotes the **remaining computation time** of the pending instance of τ_i (resp. of τ_{s_i}) at time t. Finally, for any times t and t', and for any task τ_i, we define $W_i(t, t')$ as the **processed execution time** of task τ_i between time t and time t'.
Since the processors are identical, we do not consider the allocation problem. We assume that heuristics are used in order to carry out the allocation process, which generally aims to mimize the number of context switches and of interprocessor migrations.

2.3. PFair scheduling

In a first time, we consider independent periodic tasks with **implicit deadlines** ($D_i = T_i$). Such tasks can be optimally scheduled on a platform composed of identical processors by a PFair strategy [5, 1]. The basic idea is that each task is processed at "regular

1. [a, b) = {u |a ≤ u < b}
2. |A| denotes the cardinality of the set A.

rate". At each time t, the number of processed slots is proportional to t, with a proportionality coefficient $u_i = \frac{C_i}{T_i}$. We state $Ideal_i(t) = u_i \times t$. Since the number $W_i(0,t)$ of processed slots at time t must be integer, $Ideal_i(t)$ is approximated by either $\lfloor u_i \times t \rfloor$ or $\lceil u_i \times t \rceil$ [3] . Thus the difference between the ideal and the actual number of processed time units must be bounded by 1 in absolute value. This is formally expressed by the following definition:

Definition 2.2. *A schedule is* **PFair** *iff we have:*

$$\forall i \in \{1, ..., n\}, \forall t \in \mathbb{N}, -1 < u_i \times t - \sum_{j=0}^{t-1} S_i(j) < 1.$$

The figure 2 illustrates the PFairness. For any task τ_i, the actual execution curve W_i must strictly remain between both limit lines $W_i^- = u_i \times t - 1$ and $W_i^+ = u_i \times t + 1$.

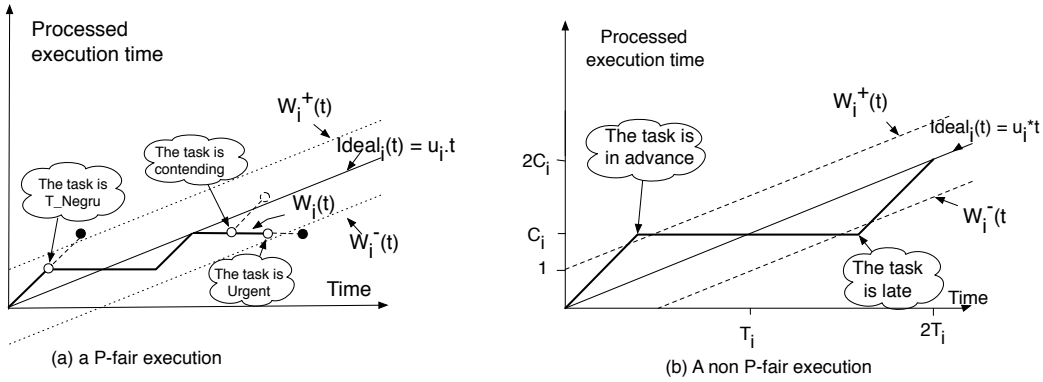

Figure 2. *(a) a PFair execution and (b) a non PFair execution - Different status of a task.*

All the PFair strategies follow the global scheme described below.

1) The task set is partitioned into three sets (see figure 2).

- the **Urgent** set contains the tasks which would be too late (under the lower bound) if they were not processed at time t. These tasks must be processed at time t.

- the **Tnegru** set contains the tasks which would be too in advance (over the upper bound) if they were processed at time t. These tasks must not be processed at time t.

- the **Contending** set contains the other tasks: the PFairness is neither violated if they are processed nor if they are not.

2) The urgent tasks are processed.

3) Then $m - \mid Urgent(t) \mid$ contending tasks are processed, their choice depends on the chosen PFair strategy.

Several PFair versions have been proposed in the literature (PF, PD and PD^2 [5, 4, 1]). These algorihms differ in the way they select the tasks to process among the contending tasks. These scheduling strategies are very efficient, where efficiency means that they can

3. $\lfloor x \rfloor$ (resp. $\lceil x \rceil$) denotes the largest (resp. the lowest) integer lower (resp. higher) than or equal to x

correctly schedule a large set of applications, as stated in the theorem 2.3.

Theorem 2.3. *[5, 4, 1] The scheduling algorithms PF, PD and PD^2 are optimal for systems of periodic independent tasks with implicit deadlines and simultaneous first releases in a multiprocessor context. Moreover, the system is feasible if and only if $U \leq m$ where m is the number of processors.*

For tasks with simultaneous first releases and implicit deadlines, the implementation of the classical PFair strategies (PF, PD and PD^2) relies on the decomposition of the execution of each task $\tau_i =< 0, C_i, D_i, T_i >$ into the execution of unitary subtasks τ_i^j ($j > 0$). Each subtask τ_i^j has a **pseudo-release** date and a **pseudo-deadline** defined by: $r_i^j = \left\lfloor \frac{j-1}{u_i} \right\rfloor$ and $d_i^j = \left\lceil \frac{j}{u_i} \right\rceil$. The time interval $[r_i^j, d_i^j)$ is the j^{th} **feasibility window** of the substask. PFairness can be characterized by the following property:

Property 2.4. *The task τ_i is PFairly processed iff each of its subtask is scheduled within its feasability window.*

The feasibility windows are computed before run-time using the WCET of the tasks. Should the actual execution time be shorter than the WCET, the task will nevertheless be scheduled within its feasibility windows. The task will thus still meet its deadlines. But some processor time units allocated to the last subtasks will be lost: the concerned processors will simply idle.

3. Independent periodic task sets and Aperiodic flow

We first consider independent periodic tasks with implicit deadlines and an aperiodic flow. The aperiodic tasks are assumed to have firm deadlines, which must be respected, else the task must be rejected. Our aim is to propose an acceptance protocol, which must obey the following rules:

1) An accepted aperiodic task must complete at the latest by its deadline.

2) The acceptance of a new aperiodic task must not cause any periodic task to miss its deadline.

3) A new accepted task must not cause a previously accepted aperiodic task to miss its deadline.

Related works

This issue has already been studied for uniprocessor systems. One of the most effective solution consists in scheduling tasks according to EDF [26, 10, 27]: the processed task has the nearest deadline. There is one single queue, sorted in increasing order of deadlines, which contains the pending periodic tasks as well as the accepted aperiodic ones. The acceptance test is then very simple. It consists in considering the aperiodic task as a new periodic one.

In multiprocessor context, we use a PFair strategy to schedule the periodic task set, for optimality reasons and because of the simplicity of the feasibility test ($U \leq m$). Here again, a solution for the acceptance test could be to consider each new aperiodic task as a new periodic one, and to use the global feasability test. This method has nevertheless two

drawbacks.
- Firstly, periodic tasks would no more be steadily scheduled, in the sense that the periodic schedule can vary from one hyperiod to another. It is thus impossible to predict when periodic tasks will occur if the aperiodic flow is not known before run-time, and thus it is impossible to compute the periodic schedule before run-time.
- Secondly, considering each aperiodic task as a periodic one would suppose to require possibly much more slots than actually necessary. Indeed, the feasibility test guarantees that there is enough time to schedule each instance of the task, thus processor time units are reserved for the processing of all these instances, even if only the first one should actually occur. And this could lead to refusing a further aperiodic task whereas it could possibily have used the processor time units reserved for instances that will never occur. Therefore, the test is sometimes pessimistic.
Finally, [30] uses the notion of spare capacities, which represent the number of available slots for the execution of aperiodic tasks within some specific windows. We use a close notion, but the use of a PFair distribution of the idle slots enables to compute these capacities more efficiently: we have neither to analyse the periodic schedule, nor to adapt the periodic temporal parameters.

Our methodology
Our aim is to propose a method which guarantees a planned, steady and fair periodic processing, in association with an acceptance test which considers only the required slots as reserved, so that is less pessimistic than the global PFair acceptance test. Aperiodic tasks are scheduled in background: they use the idle time units left by the periodic tasks. If the considered periodic scheduling is conservative (a processor never intentionally idles), then the aperiodic tasks have lower priorities than the periodic ones. But if the periodic scheduling is not conservative, it is not the case anymore, and some aperiodic tasks may have priority over some periodic tasks.
Another benefit of the use of a PFair strategy is that it enables to estimate the number of idle time units within any temporal interval with a complexity O(1) and with only a very small error. Our acceptance test relies on this predictability property.

3.1. The aperiodic queue

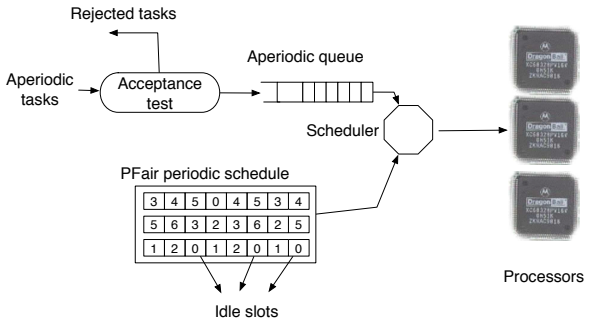

Figure 3. *The aperiodic queue.*

We assume that the operating system maintains an aperiodic queue as shown in the figure 3, which contains the pending aperiodic tasks. The periodic schedule can either be computed on line, or, for more efficiency, have been previously computed. In this case, the scheduler knows the periodic schedule. We furthermore assume that the aperiodic queue is sorted according to EDF (Earliest Deadline First [26]), i.e. in increasing order of deadines. The task with the nearest deadline is processed first. An aperiodic task can be processed at time t only if a processor is idle at time t. Thus an important issue is to control the distribution of the idle time units. We do it through the use of a specific task, which can be considered as an idle time server.

3.2. Idle time units

We assume that $m - 1 < U < m$. If $m = U$, then no aperiodic traffic can take place since the processors are fully busy. If $m < U$, then the periodic task set is not feasible. Finally if $U \leq m - 1$, we have $m - \lfloor U \rfloor \geq 1$, thus there are $m - \lfloor U \rfloor$ fully idle processors. The aperiodic tasks can thus be distributed onto these processors, using a bin-packing strategy (e.g. Worst-Fit [15]). And then scheduling results for non periodic tasks can be used [7]. We here focus on the management of the aperiodic tasks that must share the processor services with the periodic ones, thus our assumption.
Our method is based on the predictability of the idle time unit distribution. We require them to be PFairly distributed so that we can compute their number on-line within any time interval. For that aim, we add a new task to the application, which models the idleness of the processors. Our motivation for the addition of this task is the following. If $U < m$, there are $H \times (m - U)$ idle time units each hyperperiod. The simpler way would be to schedule them each time there is less than m tasks to process. But unfortunately, this doesn't guarantee a PFair distribution of the idle time units as illustrated by the figure 4. We have considered a system of 5 processors, and an application τ_exple composed of 16 tasks with implicit deadlines and first release times equal to 0, depicted in the table 1.

task	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8
C_i	14	26	14	59	48	87	120	9
$D_i = T_i$	60	300	50	50	100	300	120	20
task	τ_9	τ_{10}	τ_{11}	τ_{12}	τ_{13}	τ_{14}	τ_{15}	τ_{16}
C_i	63	82	50	76	4	16	9	65
$D_i = T_i$	300	200	200	300	10	100	20	300

Table 1. *A real time application τ_exple with 16 tasks.*

We have $U = 4.647$, $H = 600$ and there are 212 idle time units. A PFair distribution of the idle slots guarantees that at any time t, either $\lfloor \frac{212}{600} \times t \rfloor$ or $\lceil \frac{212}{600} \times t \rceil$ idle time units have taken place. Figure 4 shows the background distribution of the idle time units for the 80 first slots, which doesn't respect the PFairness criteria. E.g., at time 10, either 3 or 4 idle time units should have occured, but no one has occured yet. Furthermore, we can see that several idle time units may occur simultaneously. But we do not want several idle time units to occur simultaneously, in order to be able to decide wether an aperiodic task can be accepted or not from the only knowledge of the number of idle time units. If nbi idle time units are available between two times t and t', which never occur simulta-

neously, any aperiodic task with deadline at or after t', and such that $C_s \leq nbi$ can be accepted. But if several idle time units occur simultaneously, we cannot conclude. For instance, if $C_s = nbi$ and $d_s = t'$, the task cannot be processed on time. Thus the decision process requires to detailled the precise location of the idle time units, and is thus much more complex. Of course, avoiding simultaneous idle time units forbids to process two aperiodic tasks simultaneously, but it offers a better control. And an idle time unit can be used each time an aperiodic task is pending, which would not always be the case if several idle time units were simultaneously available.

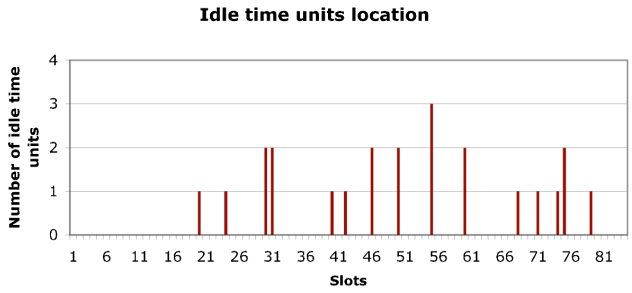

Figure 4. *Background distribution of the idle time units for the application τ_exple.*

We thus introduce a further task, called **idle task** defined by $\tau_0 =< 0, H \times (m - U), H, H >$. We have $C_0 \leq T_0$ since we have assumed that $m - 1 < U < m$. The system is now fully loaded ($U = m$) but is still PFair feasible according to Baruah's theorem (th. 2.3). By construction, an idle time unit takes place each time the idle task is processed. Then, since τ_0 is scheduled by a PFair algorithm, the idle time units are PFairly distributed, and because a task cannot be parallelized, several idle time units never occur simultaneously. Thus the idle time units distribution meets our requirements.
Now, for any time interval $[t, t']$, we have, since τ_0 is PFairly scheduled:

$$\begin{cases} \lfloor u_0 \times t \rfloor \leq W_0(0,t) \leq \lceil u_0 \times t \rceil \\ \lfloor u_0 \times t' \rfloor \leq W_0(0,t') \leq \lceil u_0 \times t' \rceil \end{cases}$$

We deduce that: $\lfloor u_0 \times t' \rfloor - \lceil u_0 \times t \rceil \leq W_0(t,t') \leq \lceil u_0 \times t' \rceil - \lfloor u_0 \times t \rfloor$.
We thus dispose of a minimal value for the number of idle time units within any time interval. Furthermore, the difference between the upper and the lower bounds is at most equal to 2. Thus, the rate of non considered idle time units will be rather small, provided the considered intervals are wide enough. In the sequel, we denote by $W\_Min(t,t')$ this minimal value ($W\_Min(t,t') = \lfloor u_0 \times t' \rfloor - \lceil u_0 \times t \rceil$).

3.3. The acceptance protocol

The periodic tasks, including the idle task τ_0, are scheduled by means of a PFair strategy. Each time the idle task is scheduled, if there is no pending aperiodic task, the slot is lost, i.e. an effective idle time unit occurs, else an aperiodic task is scheduled.
Let $Ap = (\tau_{s_1}, \tau_{s_2}, \ldots, \tau_{s_k})$ be the set of the pending accepted aperiodic tasks. We assume that Ap is ordered in increasing deadline order ($d_{s_1} \leq d_{s2} \leq ... \leq d_{sk}$). Let

$\tau_s = (t, C_s, D_s)$ be a new aperiodic task with arrival time t and deadline $d_s = t + D_s$. We decompose Ap into two subsets: $B(d_s)$ is the (possibly empty) set of the pending aperiodic tasks with deadline at or before d_s, and $A(d_s)$ is the (possibly empty) set of the pending aperiodic tasks with deadline after d_s. Thus there exists $i_0 \in \{0, ..., k\}$ such that $1 \leq i \leq i_0 \Rightarrow \tau_{si} \in B(d_s)$ and $i_0 + 1 \leq j \leq k \Rightarrow \tau_{sj} \in A(d_s)$. The acceptance test uses the approximation of the number of idle slots within a time interval by its minimal value. It runs as follows:
The aperiodic tasks in $B(d_s)$ won't be affected by the execution of τ_s, but should τ_s be processed, then it will delay the completion of the tasks in $A(d_s)$. We must then check this delay will not cause the temporal failure of some of them. We therefore verify:

1) τ_s can meet its deadline, i.e. there are enough idle time units before d_s to process the aperiodic tasks of $B(d_s)$ and the task τ_s: $W\_Min(t, d_s) \geq \sum_{i=1}^{i_0} RCT_{s_i}(t) + C_s$

2) each delayed task will still be processed on time: before its deadline, there are enough idle time units to processs all the pending aperiodic tasks with earlier deadline, and the task τ_s: $\forall p \in i_0 + 1 \ldots k$, $W\_Min(t, d_{s_p}) \geq \sum_{i=1}^{p} RCT_{s_i}(t) + C_s$

We can note that we never have to consider the periodic tasks, which simplifies the decision process. The periodic schedule can thus be computed before run-time. If the tasks have simultaneous first releases, it is computed on the interval $[0, H]$ and then iterated. For tasks with deffered first release times, the schedule is cyclic with a period equal to the hyperperiod H [12, 16, 14]. Now we have no theoretical value for the start date of the steady state, but simulations have always produced a start time that occurs before $Max_{1..n}(r_i) + H$. Then, only the acceptance test and the aperiodic scheduler are processed at run-time. Each time the idle task is planned to be processed, the scheduler is invoked, and if the aperiodic queue is not empty, the first aperiodic task (as to the EDF order) is processed. To run the acceptance test, we must maintain the remaining processing times of every pending aperiodic tasks. Thus, we maintain the list of the already accepted aperiodic tasks sorted in increasing deadline order. For each task τ_i, we store in a table TAP (see table 2):

- its deadline d_{s_i},

- its remaining computing time RCT_{s_i},

- the cumulated remaining aperiodic processing time $C\_RCT_{d_{s_i}}$ that must be completed at the latest by d_{s_i}.

The full acceptance algorithm is given in appendix A.

id	s_1	s_2	...	s_k
dl	d_{s_1}	d_{s_2}	...	d_{s_k}
RCT	RCT_{s_1}	RCT_{s_2}	...	RCT_{s_k}
C_RCT	RCT_{s_1}	$RCT_{s_1} + RCT_{s_2}$	...	$RCT_{s_1} + \ldots + RCT_{s_k}$

Table 2. *The aperiodic table TAP used by the acceptance algorithm.*

3.4. Performance analysis

We first evaluate the complexity of our method. Since the periodic schedule is never revised, the periodic schedule has to be computed only once, before run-time for efficiency reasons. Then, the computation of W_Min within any time interval is performed in O(1). Thus, the acceptance test requires O(k) additions and O(k) comparisons, where k is the number of pending aperiodic tasks. Updating the list of accepted tasks also runs in O(k), so does the function Schedule (see Appendix). Thus our method is globally linear in the number of accepted pending aperiodic tasks.
The second point of interest is to compare our method to previously existing methods and to optimal strategies. The main competitor for our method consists in scheduling together periodic and aperiodic tasks at run-time according to a PFair strategy. The acceptance test is very simple [2]: if $\sum_{i=1}^{n} \frac{C_i}{P_i} + \sum_{j=1}^{k} \frac{C_{s_j}}{D_{s_j}} + \frac{C_s}{D_s} \leq m$ then the task (t, C_s, D_s) is accepted else it is rejected. We call this method the *joined PFair method*. Note that it considers each aperiodic task as a periodic task. Therefore, more slots are reserved for each aperiodic task than required (corresponding to the different instances supposed to occur within the next hyperperiod). Moreover, periodic tasks must be scheduled on line, and the periodic schedule is not steady, i.e. the periodic schedule can be different on two different hyperperiods. Because of a more precise idle time unit reservation, our method will produce better results, in the sense that more aperiodic traffic will be accepted. In order to illustrate it, some simulations have been carried out.
We first created samples of periodic task sets in order to deal with different values of the utilization factor. A sample $Sa(m, i)$ consists of 500 task sets, where m is the number of processors and $i \in 1, \ldots, 9$. $Sa(m, i)$ is characterized by:

- the utilization factor of any task set of the sample, which must belong to the interval $[m - 1 + \frac{i}{10}, m - 1 + \frac{i+1}{10})$,

- the periods which are chosen according to Goossens' method [28], which aims to avoid too large hyperperiods,

- the WCET C_i which are chosen uniformly within the set $\{1, \ldots, \frac{T_i}{2}\}$.

For each task set, we then generated an aperiodic task flow $AP(x, D\_Max)$ which is characterized by:

- the interarrivals, which in a classical way obey an exponential distribution, with a mean equal to x,

- the relative deadlines D_{s_i} which are uniformaly chosen within the set $\{10, \ldots, D\_Max\}$,

- the deadlines, which must occur before H, thus the generation process ends when a task with deadline higher than H is generated,

- the WCET of a task τ_{s_i}, which is uniformaly chosen within the set $\{\lceil \frac{D_{s_i}}{10} \rceil, \ldots, \lfloor \frac{D_{s_i}}{2} \rfloor\}$.

For each sample, we carried out 3 simulations, over the time interval $[0, H]$:

1) we used *our method*: periodic tasks are scheduled by PD^2

2) we used the *joined PFair method*,

3) we used the *exact PFair based method*: each time we need to accept or reject a task, we count in the PF schedule the exact number of idle time units. Then we proceed exactly as in our method, using the exact number of idle time units instead of our approximation.

We then used the competitive analysis, with the *exact PFair based method* as reference method. For any pair (task set, aperiodic flow), we computed the ratio of the cumulated accepted aperiodic demand by *our method* (resp. by the *joined PFair method*) over the cumulated accepted aperiodic demand for the *exact PFair based method*. We repeated the experiment for different tuples (m, x, D_max). Figure 5 presents the results for 4 processors, with a mean interarrival x = 40 and a maximal relative deadline D_Max = 200. Results obtained for other values (2 or 4 processors, x = 20 or 40, D_Max = 40 or 200) lead to similar conclusions.

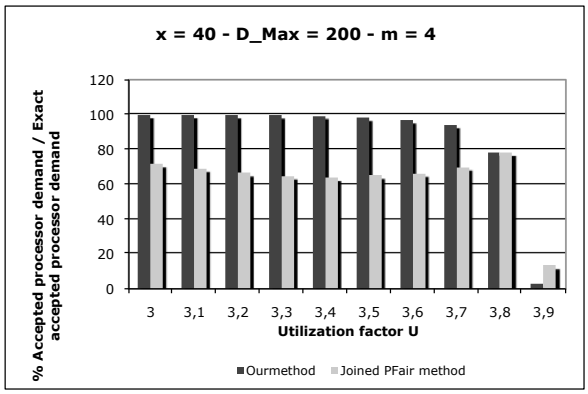

Figure 5. *Comparison of our method with the joined PFair method*

We can see that for most values of U, *our method* behaves almost like the *exact PFair based method*, whereas the *joined PFair method* has lower performance, in the sense that it accepts less aperiodic load (between 60 and 70 percent of the load accepted by the *exact PFair based method*). The performance of our test really decreases only for high values of U ($m - 0.1 < U < m$). With such utilization factors, there are few remaining idle slots, and thus the error due to the approximation of the number of idle slots becomes significant. In such cases, the *joined PFair method* becomes competitive. But in almost all cases, our method has higher performance. Besides, if U is too close to m, it will be quite impossible to include aperiodic traffic, or it would require that the designer adds a processor to the platform. So, to summarize: if $u_0 \geq 0.2$, then our method is the most efficient, but if $u_0 < 0.2$ then a global method gives better results.
Note that equivalent obervations can be made for uniprocessor systems, for which the challenger method to our's is EDF [27]. Experiments show that, except for high values of U, *our method* has significantly better results than EDF. Here again, the EDF acceptance test is very simple: an aperiodic task is accepted only if $\sum_{i=1}^{n} \frac{C_i}{P_i} + \sum_{j=1}^{k} \frac{C_{sj}}{D_{sj}} + \frac{C_s}{D_s} \leq 1$. And

again, once the task is accepted, it is considered as a periodic task thus processor activity is considered as reserved for each instance of the task, not only for one single instance.

4. Non independent periodic tasks and Aperiodic flow

We now consider non independent tasks, which may use critical resources (other than the processors), which must be used in mutual exclusion, and exchange messages through mailboxes. We assume that the reception of messages is blocking, but not the emission. Thus, a task can be represented as a sequence of blocks, whose worst case durations are known: $block(d)$ still denotes a block of duration d; and of real-time primitives (Lock/Unlock resources, Send/Receive messages), whose durations are assumed to be equal to 0, which in fact means that their actual durations are included in the durations of the neighbouring blocks.
The periodic tasks may have constrained deadlines ($D_i \leq T_i$), and finally, aperiodic tasks are still assumed to be independent (they neither use critical resourcse nor exchange messages). We still aim to propose joined scheduling: the periodic tasks are scheduled before run-time, and the aperiodic flow is managed on-line, using the method presented in the previous section. The fundamental requirements to use this method is the PFair distribution of the idle time units. But no specific requirements concern the periodic schedule, provided it is valid. We thus adopt an off-line model-driven approach to compute valid schedules with a PFair idle time unit distribution.

4.1. Model-driven scheduling: Petri net-based approach

When tasks are not independent, not only PFair strategies are no longer optimal but also there exists no on-line optimal strategy [17, 22]. Furthermore, in this case, the scheduling problem is NP-hard. We thus analyse the periodic set before run-time. We consider a model driven approach using a Petri net based modelling, which is presented in the next sections. We recall that a task consists of a sequence of blocks whose durations are known ($Block(d)$ is a block with a WCET equal to d) and of real-time primitives.

4.1.1. Basic definitions

A **marked Petri net** is a pair (N, M_0) with $N = (Q, T, W)$ where Q is a finite set of places; T a finite set of transitions; $W : Q \times T \cup T \times Q \to \mathbb{N}$ is the valuation function; and $M_0 : Q \to \mathbb{N}$ the initial marking, where $M_0(p)$ is the number of tokens initially held by the place p.

N gives a static description of the system and M_0 is its initial state. The dynamic behaviour is described by the firing rules: a transition t is **enabled** or **firable** for a marking M if and only if $\forall p \in Q$, $M(p) \geq W(p, t)$. Its firing leads to the marking M' defined by: $\forall p \in Q$, $M'(p) = M(p) - W(p, t) + W(t, p)$.
A Petri net runs under the **maximal firing rule** [32] if only maximal (as to the inclusion) transition sets are fired.
Coloured places [23] contain marks of different colours[4]. A finite set C of colours is associated to each coloured place. We redefine the valuation function by $W : Q \times T \times$

4. We use a simplified version of the definition of the coloured Petri nets

$C \cup T \times Q \times C \to \mathbb{N}$. Then a transition is enabled from the marking M if and only if $\forall p \in Q$ and $\forall c \in C$, $M(p,c) \geq W(p,t,c)$. Its firing produces the marking M' defined by: $\forall p \in Q, \forall c \in C$, $M'(p,c) = M(p,c) - W(p,t,c) + W(t,p,c)$.
Finally, the evolution of the net may be constrained by a **terminal set** [36] whose aim is to restrict the possible behaviours of the net to the only ones that guarantee that all the reached states verify some properties. We thus add a marking set $\mathcal{I}$, that is defined using properties on markings, to the net, and we only consider firing sequences such that the successive reached markings belong to $\mathcal{I}$.

4.1.2. The model

The model (see [21] for a more detailled presentation) consists of a coloured Petri net with a terminal set, running under the maximal firing rule. It is composed of two parts, both modelled by a place/transition net (see figure 6): the task system, which is obtained through a classical modelling of the functional description of the tasks, and the clock system, which, together with the maximal firing rule, models the time.

4.1.3. The clock system

This part of the net consists of:

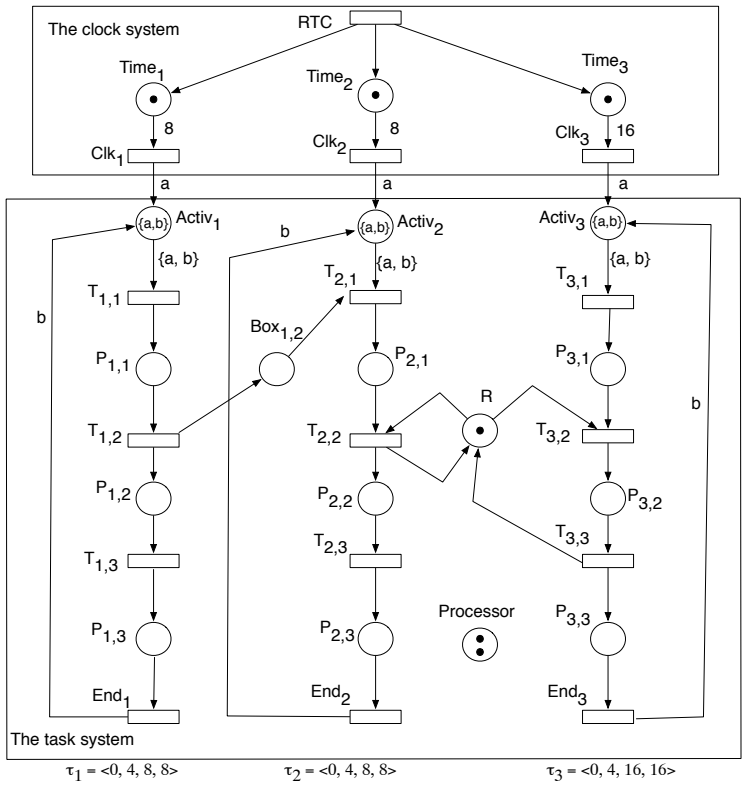

Figure 6. *Modelling of an application. Each transition $T_{i,j}$ and End_i (respectivelly the j^{th} and the last transition of the task τ_i) have the place Processor as input and output. This is not represented on the net for readibility reasons. P_i, j is the $(j+1)^{th}$ place of the task τ_i, and $Activ_i$ is its first place.*

1) A global clock RTC (Real Time clock), which fires each time a set of transitions fires, since this transition is always enabled. We thus get a logical representation of the time: we assimilate the firing of RTC to an execution time unit.

2) The local clocks of the tasks which are used to periodically release the related tasks. The clock of a task τ_i is composed of an accumulative place $Time_i$, which memorizes the elapsed time since the last release of the task and a transition Clk_i which fires each period. The firing of Clk_i models the reactivation of the task which occurs each time the place $Time_i$ contains T_i tokens. It produces a token of colour **a** (for activation) in the place $Activ_i$ which is the first place of the task τ_i in the task system.

4.1.4. The task system

This part of the net models the task set (Figure 7). Each task τ_i starts with a coloured place $Activ_i$, to which two colours **a** and **b** are associated. When the place holds a token **a**, it means that a new instance has been released, but has not yet started execution, and when it holds a token **b**, it means that the previous instance has completed execution. Finally, each last transtion End_i of the task τ_i verifies $W(End_i, Activ_i)$= **b** i.e. a token of colour **b** is produced when an instance completes execution. Then the task can start execution only if this place holds a token **a** and a token **b** i.e. if it has been released and if the previous instance has completed execution. The firing of one transition corresponds to the processing of the associated task during one slot. A block of duration d is modelled by one single transition if $d = 1$, two transitions if $d = 2$ and three transitions if $d > 2$ (see the three blocks in Figure 7). Communications between tasks are modelled by mailbox places, and each shared resource by one place. Processors are modelled by one place with m tokens if we consider a platform of m processors, and each transition of the tasks admit this place as input and output. Accordingly, at most m transitions of the task system can fire simultaneously. The edges from and to the place Processor are not represented on the figure 6 for clarity reasons. Figure 7 presents the modelling of a task with 3 blocks of durations 5, 2 and 1 respectively and whose first release time is $r = 2$. It sends a message after its activation, locks the shared resource R at the beginning of the second block and unlocks it at the end of this block. The platform consists of 2 processors.

4.1.5. Integration of the temporal parameters

The initial marking of the places $Activ_i$ and $Time_i$ takes the first release dates into account. This initial marking is defined as follows (we assume that $0 \leq r_i < T_i$, see [21] for the case $T_i \leq r_i$): $\forall i = 0..n$,

$$M_0(Time_i) = \begin{cases} 1 \ if \ r_i = 0 \\ T_i - r_i + 1 \ if \ r_i > 0 \end{cases}, \quad M_0(Activ_i) = \begin{cases} \{\mathbf{a}, \mathbf{b}\} \ if \ r_i = 0 \\ \mathbf{b} \ if \ r_i > 0 \end{cases}$$

If $r_i = 0$, then the task can start execution, thus M_0 holds a token **a** and a token **b**, else, no instance is running, there is a token **b**, but the task must wait until the next release, thus there is no token **a**.
Let $\Re = \left\{R_1, ..., R_{|\Re|}\right\}$ denotes the places associated to the shared resources and $\wp = \left\{B_1, ..., B_{|\wp|}\right\}$ the mailbox places. $|R_i|$ is the number of instances of the shared resource R_i. The initial markings of the other places are defined by: $M_0(Processor) = m$, $\forall i = 1..\left|\Re\right|, M_0(R_i) = |R_i|$, $\forall i = 1..\left|\wp\right|, M_0(B_i) = 0$, $M_0(P) = 0$ for all other places of the task system (i.e. the places $P_{i,j,k}$).

Since we are only interested in valid behaviours, further constraints on allowed markings are added. A terminal set is introduced to consider deadlines:

$$\forall i = 0..n, \begin{cases} 1.\ M(Time_i) = 1 \Rightarrow M(Activ_i) = \{\mathbf{a},\ \mathbf{b}\}\ or\ \mathbf{b} \\ 2.\ M(Time_i) \geq D_i + 1 \Rightarrow M(Activ_i) = \mathbf{b} \end{cases}.$$

The point 1. means that at each release, the task must have completed the previous instance. The second case ($M(Activ_i)$ = **b**) corresponds to the initialization of the net in the case $r_i = P_i - 1$. And the point 2. means that after deadline, the pending instance must have completed execution. It is used only for the tasks such that $D_i < T_i$.

4.1.6. Valid schedules

Then we label the net, using the alphabet $\sum_\tau = \{\tau_0, \tau_1, ..., \tau_n\}$ and the labelling function $\sigma_\tau : T \to \sum_\tau$: $\forall\ T_{i,j} \in T, \sigma_\tau(T_{i,j}) = \tau_i$, $\forall\ End_i \in T$, $\sigma_\tau(End_i) = \tau_i$ and $\sigma_\tau(t) = \epsilon$ for any other transition $t \in T$. For the net of figure 6, we have for instance $\sigma_\tau(T_{1,1}) = \sigma_\tau(T_{1,2}) = \sigma_\tau(T_{1,3}) = \sigma_\tau(End_1) = \tau_1$, or $\sigma_\tau(Clk_1) = \epsilon$.
Each time a transition set fires, we only keep the informations coming from the task system, more precisely, the labeling function gives the set of the currently processed tasks. Because of the place $Processor$, we are sure that there are at most m such tasks. For instance, in the net of figure 6, only 2 transitions of the task system can fire at once.
We then construct the labelled terminal marking graph. Each edge is labelled by a set of tasks, which belongs to $O_m(\tau)$ and each vertex by a marking. If the set $\{t_1, .., t_p\}$ fires, the edge is labelled by the set $\{\sigma_\tau(t_i)/\sigma_\tau(t_i) \neq \epsilon\}$. For instance, again for the net of figure 6, if $\{RTC, T_{1,2}, T_{3,2}\}$ fires, the associated edge is labelled by $\{\tau_1, \tau_3\}$.
Each path starting at the vertex labelled by M_0 is labelled by a word ω defined as $\omega =$

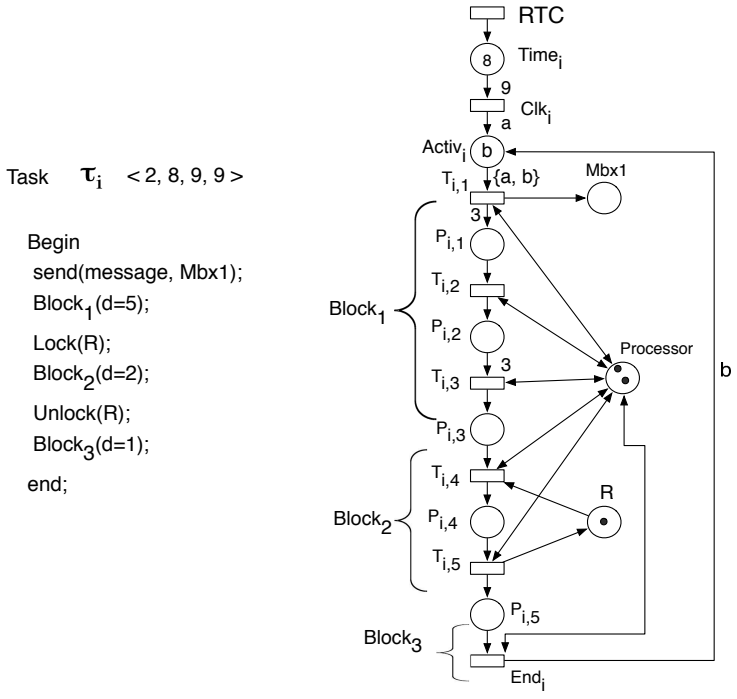

Figure 7. *Model of the task $\tau_i =< 2, 8, 9, 9 >$, for a platform of 2 processors.*

$\omega_0\omega_1\omega_2...$ where for $t \in \mathbb{N}$, $\omega_t \in O_m(\tau)$. The associated schedule S is then defined by $S(t) = \omega_t$ where ω_t is the set of the tasks which are scheduled at time t. Since a marking that meets the terminal constraints corresponds to a state of the task system where all temporal constraints are met, this schedule is valid. We have the following proposition a proof can be found in [19, 20]:

Proposition 4.1. *[19, 20] By construction, the valid schedules exactly correspond to the infinite paths in the labelled terminal marking graph if the net runs under the maximal firing rule.*

4.2. Idleness control using Petri nets

We still assume that $m - 1 < U < m$. Our concern here is the off-line scheduling of the periodic tasks in such a way that we still can use our acceptance protocol to manage the aperiodic flow. For that aim, we again add an idle task to the model, and then we compute valid schedules which ensure a PFair execution of this idle task.

4.2.1. Model of the idle task

The idle task is defined as in section 3.2: $\tau_0 = < 0, C_0 = H(m-U), D_0 = H, T_0 = H >$. For instancce, for the net of the figure 6, we have $\tau_0 = < 0, 12, 16, 16 >$. We use a global approach to model the idle task, and we consider the PFairness requirement only during the schedule computation, by means of the labelled terminal marking graph. We thus model the idle task like any other task, and leave the PFairness requirement to the schedule extraction step. We add a local clock$(Time_0, Clk_0)$, a place $Activ_0$, two places $P_{0,1}$ and $P_{0,2}$ and two transitions $T_{0,1}$ and $T_{0,2}$ (see figure 8 for the case $C_0 \geq 2$). The terminal condition concerning this task is $M(Time_0) = 1 \Rightarrow M(Activ_0) = \{a, b\}$. Even if the PFairness requirement satisfaction is left for the next step, we can note that it nevertheless has an impact on the modelling of the task. Indeed, here, the idle task cannot be parallelized, but in the general case, without specific requirement, it should be possible to have several simultaneous idle time units (to model the fact that several processors are simultaneously idle). Thus it would require another modelling of the idle task (we don't detail it here since we don't need it).

4.3. Schedule production

The first step consists of the construction of the labelled terminal marking graph. Then, we must find the paths in the graph starting at the vertex labelled by M_0, in which the idle time units are PFairly distributed. For that aim, we valuate the graph. Let M be a marking. We can deduce the number of already elapsed idle time units for the current hyperperiod, when the system is in the state modelled by the marking M, which is denoted by $W_0(M)$.

– if $\mid M(Activ_0) \mid_a = 1$, then $W_0(M) = 0$: the presence of the token a means that τ_0 has been released for the current hyperperiod, but doesn't have yet started execution [5] ,

– else if $\mid M(Activ_0) \mid_b = 1$ then $W_0(M) = C_0$: the absence of the token a means that the current instance has already started execution, and the token b means that this instance is completed,

5. $\mid M(Activ_0) \mid_x$ denotes the number of x in the place $Activ_0$

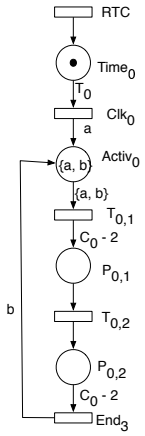

Figure 8. *Modelling of the idle task.*

– else $W_0(M) = 1 + M(P_{0,2})$: the current instance has started execution (no token a) but is not completed (no token b). The transition $T_{0,1}$ has been fired, which corresponds to the occurrence of a first idle time unit. Then the transition $T_{0,2}$ has been fired a certain number of time, say nbf ($0 \leq nbf \leq C_0 - 2$) corresponding to the occurrences of nbf further idle time units. nbf is equal to the marking of the place $P_{0,2}$. Finally the transition End_0 hasn't been fired. Thus the global number of already elapsed idle time units is equal to $1 + nbf = 1 + M(P_{0,2})$.

Then let N_i and N_j be two nodes of the graph, labelled by the markings M_i and M_j, such that the edge (N_i, N_j) exists, and let ω_j be the label of the edge (N_i, N_j). The weight associated to the edge is defined as [6]:

$$Weight(N_i, N_j) = \begin{cases} \lfloor Abs(W_0(M_j) - M_j(Time_0)u_0) \rfloor \; if \; \tau_0 \in \omega_j \\ 0 \; else \end{cases}$$

Let us consider the state of the system modelled by the marking M_j: $W_0(M_j)$ corresponds to the number of elapsed idle time units since the beginning of the hyperperiod, $M_j(Time_0)$ is the current time (modulo H), and $M_j(Time_0)u_0$ is the ideal number of elapsed idle time units since the begining of the current hyperperiod. Thus, if the actual number of idle time units verifies the PFair equation (definition 2.2), the valuation of the edge is equal to 0. If the PFair property is not met; i.e. the task τ_0 is either late or in advance (see figure 2 (b)), the valuation is strictly positive. A path corresponding to a schedule where the idle slots are PFairly distributed has thus a valuation equal to 0. Furthermore, since the terminal marking graph collects all the possible valid schedules (Proposition 4.1), if no nul valuated path can be found, then no schedule exists which ensure a PFair distribution of the idle time units. We thus have the following result.

Proposition 4.2. *If the valuated terminal marking graph contains a path with a nul valuation (thus whose edges are all labelled by 0), then the idle time units are PFairly distributed within the associated schedule. If the valuated terminal marking graph doesn't contain such a path, then it is not possible to schedule the periodic task set without temporal failures and together to PFairly distribute the idle time units.*

6. $Abs(x)$ denotes the absolute value of x

The problem of finding such a schedule is thus reduced to the problem of searching a minimal-valuated path starting at the vertex labelled by M_0 in the graph.
Note that, since we are only interested in the edges that are valuated by 0, we can directly include this restriction into the construction process of the terminal labelled marking graph. This will reduce the graph construction duration, since the final graph is smaller than the full graph, and useless edges and vertices are not constructed.
Then, if one solution has been found, we can again use the acceptance protocol presented in section 3.3 to manage the aperiodic flow on-line. As to the performance of the method (the acceptance rate), provided the idle task is PFairly scheduled, the acceptation rate, only relies on the idle task location, and not on the periodic schedule. Thus the results got for the systems of independent tasks still hold.

5. Conclusion

We have proposed a protocol to schedule aperiodic hard real-time tasks, which is linear in the number of already accepted pending aperiodic tasks. The periodic tasks can be pre-scheduled (i.e. scheduled off-line before run-time). The protocol is based on a PFair distribution of the idle times within the periodic schedule. The idle time units are used for the aperiodic taks progression, and their number can be estimated in O(1) within any time interval. They are managed by a dedicated task, the idle task, which is PFairly scheduled. For that aim, if the periodic tasks are independent, a simple way to PFairly schedule τ_0 is to schedule the whole periodic set (including τ_0) with a PFair strategy. If the periodic tasks are non independent, we use a model-driven off-line strategy, based on Petri nets. We then take the PFairness requirement into account during the analysis of the net.
Now, an open problem is to bound the depth of the marking graph. If the periodic tasks have simultaneous first releases, the depth is equal to the hyperperiod H, else, as mentionned earlier, we have no efficient bound. So either we chose an arbitrarily depth (e.g. $Max_{i\in 1..n}(r_i) + H$) or go on with the construction until each path loops.
Then, we can also use our method to include a new periodic task within an already scheduled task set, without recomputation of the schedule. We consider a flow composed of all the instances of the task for the next hyperperiod. If each instance can be accepted, then the task can be added to the task set, and the schedule be modified according to the acceptance protocol, else, either the system must be completely re-scheduled, or the addition in not possible.

6. References

[1] J. Anderson, P. Holman, and A. Srinivasan. Fair scheduling of real time tasks on multiprocessors. *Handbook of scheduling : Algorithms, Models and Performance analysis*, pages 31.1–31.21, 2004.

[2] J. H. Anderson and A. Srinivasan. Pfair scheduling: beyond periodic task systems. In *RTCSA*, pages 297–306, 2000.

[3] J. Banús, A. Arenas, and J. Labarta. An efficient scheme to allocate soft-aperiodic tasks in multiprocessor hard real-time systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications - Volume 2*, PDPTA '02, pages 809–815. CSREA Press, 2002.

[4] S. Baruah, J. Gehrke, and C.G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the 9^{th} International Parallel Processing Symposium*, pages 280–288, April 1995.

[5] S.K. Baruah, N.K. Cohen, C.G. Plaxton, and D.A. Varvel. Proportionate progress : a notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[6] B. Berthomieu and M. Diaz. Modelling and verification of time dependent systems using time petri nets. *IEEE Transactions on software engineering*, 17(3), 1991.

[7] J. Blazewicz. Scheduling dependent tasks with different arrival times to meet deadlines. In E. Gelenbe and H. Bellner, editors, *Modelling and performance evaluation o colputer systems*, pages 57–65. North-Holland, 1976.

[8] G.C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1997.

[9] J. Carpenter and all. A categorization of real-time multiprocessor scheduling problems and algorithms. *Handbook of scheduling: Algorithms, Models and Performance Analysis*, 2003.

[10] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real time tasks under precedence constraints. *The Journal of the Real time Systems*, 2:115–127, 1990.

[11] Seonho Choi and Ashok K. Agrawala. Scheduling aperiodic and sporadic tasks in hard real-time systems. Technical report, Institute for Advanced Computer Sciences, University of Maryland, 1997.

[12] A. Choquet-Geniet. Un premier pas vers l'étude de la cyclicité en environnement multiprocesseurs. *Actes de la conférences RTS'05*, pages 289–302, 2005.

[13] A. Choquet-Geniet, G. Largeteau-Skapin, and A. Ouattara. Integration of pfairness within a modelled based scheduling tool. In *Fourth International Workshop on Verification and Evaluation of Computer and Communication*, ewics series of the British Computer Society, 2010.

[14] A. Choquet-Geniet and S. Malo. Finding cyclicity behavior in multiprocessor scheduling. Technical report, LISI - ENSMA and University of Poitiers, 2009.

[15] E. Coffman, G. Galambos, S. Martello, and D. Vigo. *Handbook of Combinatorial Optimization*, chapter Bin Packing Approximation Algorithms: Combinatorial Analysis. Kluwer, 1998.

[16] L. Cucu and J. Goossens. Feasibility intervals for fixed-priority real-time scheduling on uniform multiprocessors. In *The 11^{th} IEEE International Conference on Emerging Technologies and Factory Automation*, pages 397–405. IEEE Computer Society Press, 2006.

[17] M.L. Dertouzos and A.K.L. Mok. Multiprocessor scheduling in hard real-time environment. *IEEE transactions on sofware Engineering*, 15(12):1497–1506, 1989.

[18] N. Fisher, S. Baruah, and T. Baker. The partionned scheduling of sporadic tasks according to static-priorities. In 18^{th} *euromicro Conference on real-time systems*, pages 118–127, 2006.

[19] E. Grolleau. *Ordonnancement Temps-Réel Hors-Ligne Optimal à l'Aide de Réseaux de Petri en Environnement Monoprocesseur et Multiprocesseur*. PhD thesis, Univ. Poitiers, 1999.

[20] E. Grolleau and A. Choquet-Geniet. Real-time scheduling in multiprocessor environment by means of Petri nets. *Proceedings of RTS 2001*, pages 189–206, 2001.

[21] E. Grolleau and A. Choquet-Geniet. Off line computation of real time schedules by means of petri nets. *Journal of Discrete Event Dynamic Systems*, 12:311–333, 2002.

[22] K.S. Hong and J.Y Leung. On-line scheduling of real-time tasks. *IEEE transactionson Computers*, 41(10):1326–1331, 1992.

[23] K. Jensen. *Coloured Petri nets, basic concepts, analysis methods and pratical use*. Monographs in theoretical Computer Science. Springer Verlag, 1997.

[24] S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *Proceedings of the 21st Euromicro Conference on Real-time Systems (ECRTS)*, pages 249–258. IEEE Computer Society, 2009.

[25] G. Largeteau, D. Geniet, and E. Andres. Discrete geometry applied in hard real-time systems validation. In *DGCI 2005 12th International Conference, Poitiers*, volume 3429 of *LNCS*, pages 23–33. Springer Verlag, 2005.

[23] G. Largeteau, D. Geniet, and J.P. Dubernard. Validation of distributed periodic real-time systems using can protocol with finite automata. In *Proc. of* 5^{th} *S.C.I.* I.I.I.S., 2001.

[26] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[27] J.W.S Liu. *Real-Time Systems*. Prentice Hall, 2000.

[28] C. Macq and J. Goossens. Limitation of the hyper-period in real-time periodic task set generation. In Teknea, editor, *Proceedings of the 9th international conference on real-time systems*, pages 133–148, Paris France, March 2001. ISBN 2-87717-078-0.

[29] A.K. Mok and M.L. Dertouzos. Multi processor scheduling in a hard real-time environment. In *Proc. of* 7^{th} *Texas Conference on Computer Systems*, 1978.

[30] N. Pernet. Implantation distribuée temps-réel de programmes conditionnés à l'aide d'ordonnancement mixte hors-ligne en)-ligne de tâches périodiques avec contraintes de latence et acceptation de tâches apériodiques. *PhD thesis*, Université Paris 6, 2006.

[31] A. Srinivasan, P. Holman, and J.H. Anderson. Integrating aperiodic and recurrent tasks on fair- scheduled multiprocessors. In *14th Euromicro Conference on Real- Time Systems*, pages 189–198, Vienna, Austria, June 2002. IEEE Computer Society.

[32] P. Starke. Some properties of timed petri nets under te earliest firing rule. In *Advances in Petri nets '90*, LNCS 424. Springer Verlag, 1990.

[33] H. Tang, P. Ramanathan, and K. Compton. Combining hard periodic and soft aperiodic real-time task scheduling on heterogeneous compute resources. *Parallel Processing, International Conference on*, 0:753–762, 2011.

[34] J. Theis and G. Fohler. Transformation of sporadic tasks for off-line scheduling with utilization and response time trade-offs. In *Proceedings of the 19th International Conference on Real-Time and Network Systems (RTNS'11)*, Proceedings of 19th International Conference on Real-Time and Network Systems (RTNS11), pages 119–128, September 2011.

[35] J. Tsai, S. Yang, and Y. Chang. Schedulability analysis of real time systems using timing constraint petri nets. In *Proc. of Comp Euro 93*, pages 375–382, 1983.

[36] R. Valk and G. Vidal-Naquet. Petri nets and regular languages. *Journal of Computer and System Sciences*, 23(3), 1981.

A. The acceptance test

We first give the function which decides whether an aperiodic task can be accepted.

```
Function Accept
 input
 u0: float -- utilization factor of the idle task τ0
 TAP -- accepted aperiodic table
 k : integer -- number of already accepted aperiodic tasks
 τ = (t, C, D) -- new aperiodic task
 output
 accepted: boolean -- true if the task is accepted, false else
d := t + D
W_Min :=⌊ u0 × d ⌋ − ⌈ u0 × t ⌉
If TAP is empty then -- there is no aperiodic pending task
   accepted :=(W_Min ≥ C)
else
-- Computation of i0 for the partition of the pending aperiodic task set
  i0:= 0
  While (i0 < k) and TAP(i0+1).dl ≤ d loop
      i0 := i0+1
  end loop
if i0 > 0 then -- the set B(d) is non empty
  if ⌊ (u0 × TAP(i0).dl ⌋ − ⌈ u0 × t ⌉ ≥ C + TAP(i0).C_RCT
    then accepted := true
    else accepted := false -- there are not enough idle time units to add the task
  end if
else accepted := true
end if
if accepted then
      for j in i0+1..k loop
        if ⌊ u0 × TAP(j).dl ⌋ − ⌈ u0 × t ⌉ < C +TAP(j).C_RCT
          then accepted := false -- a task will miss its deadline if the new aperiodic task is processed
         end if
       end loop
end if
return accepted
```

The list of the pending accepted aperiodic tasks must be updated after acceptation of a new task. We use an insertion function Insert ((dl, RCT, C_RCT), j, TAP) which inserts the tuple (dl, RCT,C_RCT) in position j in the table TAP.

```
Function insert_task
 input
TAP -- pending accepted aperiodic tasks
τ = (t, C, D) -- new aperiodic task
```

Precondition
Task τ is accepted
output
the updated table of accepted tasks
d := t + D
If TAP is empty then -- *insertion in first position*
 insert((d, C, C), 1, TAP)
else i_0:= 0
  While (i_0 < k) and TAP(i_0+1).dl $\leq$ d loop
    i_0 := i_0+1
  end loop
-- *the insertion position is i_0+1*
  for j in i_0+1..k loop
-- *the next C_RCT are increased*
    TAP(j).C_RCT := TAP(j).C_RCT + C
  end loop
-- *insertion in position i_0+1*
  insert((d,C,TAP(i).C_RCT + C),i_0+1,TAP)
end if
return(TAP)

Finally, the global scheduling algorithm is the following. We use a deletion function del(TAP, k) which deletes the k^{th} item from the table TAP.

Function schedule
input
TAP -- *pending accepted aperiodic tasks*
L -- *list of the new arrived aperiodic tasks*
t -- *current time*
output
updated table of pending aperiodic tasks
identity of the processed task
while L is not empty loop
  τ := head(L)
  unqueue(L)
  If accept(u_0,TAP,τ) then
     insert_task(TAP,τ)
  end if
end loop
if TAP is not empty then
   id :=TAP(1).id
   TAP(1).RCT := TAP(1).RCT - 1
   for i in 1..k loop
     TAP(i).C_RCT := TAP(i).C_RCT - 1
   end loop
   If TAP(1).RCT = 0 then del(TAP,1)
   end if
end if;
return(TAP, id)