Scheduling of real-time applications with variable utilization factor using a PFair based aperiodic server

Annie Choquet-Geniet[1]
annie.geniet@univ-poitiers.fr

Christian Fotsing[1]
fotsingc@ensma.fr

Sadouanouan Malo[1,2]
sadouanouan.malo@ensma.fr

[1]University of Poitiers. Laboratory of Applied Computer Science
1 Av. Clément Ader BP 40109-86961 Futuroscope Chasseneuil-France

[2]Polytechnic University of Bobo Dioulasso. Information Technology High School.
01 BP 1091 Bobo Dioulasso 01

Abstract

We consider the scheduling problem for real time applications with variable processor utilization factor. They are composed of a set kernel of periodic tasks and of a flow of aperiodic tasks, all with firm deadlines. Periodic tasks are scheduled according to a PFair policy. We propose an efficient acceptance test for aperiodic traffic, which guarantees steady and predictable periodic scheduling, a very small error rate and which has a low complexity. We then compare its performances to the performances of challenger acceptance tests.

1 Introduction

We consider real-time applications composed of a set periodic task kernel and of an aperiodic task flow. Periodic tasks are mostly dedicated to control, e.g. temperature acquisition in a nuclear station, robot's trajectory computation, processing of informations provided by a synchronous link.... Aperiodic tasks arise as answers to aperiodic events: human interaction, alarm activation, error detection.... We consider here only hard real time tasks. Thus we deal with applications composed of n periodic tasks $\tau_1,\dots,\tau_n$ and of aperiodic tasks. Both periodic and aperiodic tasks are submitted to 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.

One of the main challenge for system designer is to guarantee that all deadlines will be met. This is the concern of scheduling. Now, the architecture of the application, due to aperiodic traffic, is variable, thus scheduling must be adaptative, so that it can take at any time the actual processor workload into account. For periodic traffic, the scheduling problem consists in defining a suited strategy and to prove that all temporal constraints will be respected. We consider here only global scheduling: tasks can run at any time on any processor, they may start and resume on any one. For aperiodic tasks, the problem consists in defining an acceptance test which obeys the following criteria :

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

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

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

This issue has already been studied for uniprocessor systems. One of the most effective solution consists in scheduling tasks according to EDF [7, 8]: the processed task has the nearest deadline. There is one single queue, sorted in increasing order of deadlines, which contains periodic tasks as well as 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, the most efficient scheduling strategy is PFair scheduling [3, 4]. Furthermore, the feasability test consists only in verifying if the utilization factor is at most equal to the number of processors. Here again, a solution for the acceptance test is to consider each new aperiodic tasks as a new periodic one, and to use the global feasability test. This method has nevertheless two drawbacks. Firstly, periodic tasks are not steadily scheduled, there may be rather much jitter, and it is not possible to predict when periodic tasks will occur if the aperiodic flow is not known. Secondly, since each aperiodic task is considered by the test as a periodic one, it is supposed to require possibly much more slots than effectively necessary (several instances may be considered). The test is therefore sometimes pessimistic. 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, so which is less pessimistic than the global PFair acceptance test. The accepted aperiodic queue is sorted according to EDF. Aperiodic tasks are scheduled in background: they use the idle time units left by the periodic tasks. For efficiency reasons, the PF schedule can be computed before run-time, since it will never be affected by the aperiodic traffic. Then only the acceptance test and the aperiodic task scheduling run online. This will reduce the overhead due to scheduling. We have chosen to PF strategy for two reasons :

1. PF is optimal in multiprocessor context, and the feasibility test is very simple.
2. PF enables to predict with a complexite O(1) the number of idle time units within any temporal interval, with only a very small error. Our acceptance test relies on this predictability property.

The paper is organised as follows. In part 2, we introduce the task model and our notations. We also briefly present PFair scheduling. In part 3, we discuss the location of the idle time units. In part 4, we describe our acceptance test. And finally, in part 5, simulation results are presented, which illustrate the performances of our method.

2 Task model and PFair scheluling

We consider multiprocessor systems, with m processors. For any real x, $\lfloor x \rfloor$ denotes the greatest integer less than or equal to x and $\lceil x \rceil$ the smallest integer greater than or equal to x.

2.1 The task model

We consider applications composed of n independent synchronous periodic tasks with implicit deadlines $\tau_i(C_i, P_i)$. Each task is submitted to hard temporal constraints. We adopt the classical modeling of tasks [7]. Each **periodic** task τ_i is characterized by its worst-case execution time C_i and its period P_i, and consists in an infinite set of instances (or jobs), released at times $k \times P_i$, with k $\in \mathbb{N}$. At each release, the precedent instance must have complete execution (deadlines are implicit). We assume parallelism to be forbidden: at any time, a task can run on at most one processor. Finally, we suppose that temporal parameters are known and deterministic.

In the sequel, P denotes the **hyperperiod** of the system defined as P = lcm($P_1, P_2, \ldots, P_n$).

The processor **utilization factor** characterizes the processor workload due to the application. It is defined by U = $\sum_{i=1}^{n} \frac{C_i}{P_i}$. If U > m (m being the number of processors), the system is over-loaded and temporal faults cannot be avoided [6]. In the sequel, we assume that $m-1 < U < m$.The case U = m is avoided because we need to dispose of some processor capacity for aperiodic traffic.

In the further, **slot** t denotes the time interval [t, t+1)[1]. A task is **scheduled at time t** means that one processor processes it during slot t.

A **schedule** is defined by S : $\mathbb{N} \times \{1, \ldots, n\} \rightarrow \{0, 1\}$ such that $\sum_{i=1}^{n} S(t,i) \leq m$. We have S(t,i) = 1 $\Leftrightarrow$ τ_i is scheduled at time t, for i = 1 ... n and if $\sum_{i=1}^{n} S(t,i) = k < m$ then (m - k) idle time units occur at time t. An **idle time unit** corresponds to idleness for one processor.

Let S_i be such that S_i(t) = $\begin{cases} 1 \text{ if } S(t,i) = 1 \\ 0 \text{ else} \end{cases}$.

For any times t and t', and for any task τ_i, we define $W_i(t,t')$ as the **processed execution time** for task τ_i between time t and time t'. We have $W_i(t,t') = \sum_{u=t}^{t'-1} S_i(u)$.

An **aperiodic** task τ_s consists in an arrival time r_s, a worst case execution time C_s and a relative deadline D_s. Its deadline is $d_s = r_s + D_s$. For any aperiodic task τ_{s_i}, $RCT_{s_i}(t)$ denotes the **remaining computation time** of τ_{s_i} at time t:

$$RCT_{s_i}(t) = C_{s_i} - \sum_{u=r_{s_i}}^{t-1} S_{s_i}(u) = C_{s_i} - W_{s_i}(r_{s_i}, t)$$

.

2.2 PFair scheduling

PFair scheduling strategies have been proposed in the multiprocessor context, for which they are very efficient [4]. The basic idea is that each task is processed at "regular rate". This means that at each time t, the number of processed slots $W_i(0,t)$ is proportionnal to t, with coefficient $u_i = \frac{C_i}{P_i}$. But, since the number of processed slots at time t must be integer, $u_i \times t$ is approximated by either $\lfloor u_i \times t \rfloor$ or $\lceil u_i \times t \rceil$.

This is formally expressed by the following definition:

A schedule is **PFair** iff we have:

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

Figure 1 illustrates PFairness. For any task τ_i, the broken line W_i must remain strictly between both limit lines $W^- = u_i \times t - 1$ and $W^+ = u_i \times t + 1$.

At any time t, a task is said to be:

- **ahead** if $W_i(t)$ is above the ideal line $W_i(t) = u_i \times t$. It has been processed a little bit more than in the ideal case. We have: $u_i \times t - \sum_{j=0}^{t-1} S_i(j) < 0$.
- **punctual** if it has been processed for exactly $u_i \times t$ slots. We have: $u_i \times t - \sum_{j=0}^{t-1} S_i(j) = 0$.
- **behind** if $W_i(t)$ is under the ideal line $W_i(t) = u_i \times t$. It has been processed a little bit less than in the ideal case. We have: $u_i \times t - \sum_{j=0}^{t-1} S_i(j) > 0$.

PFair strategies follow the global frame described below.

1 - The task set is partitioned into three sets.

[1] [a, b) = {u | a $\leq$ u < b}

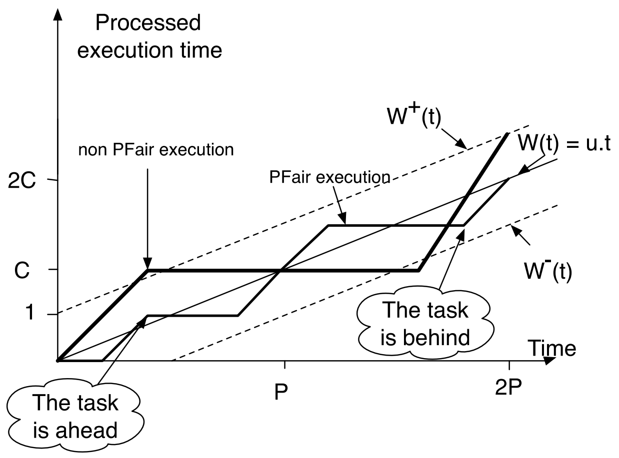

Figure 1. PFair and non PFair executions: PFair execution curve must be located between both dotted lines

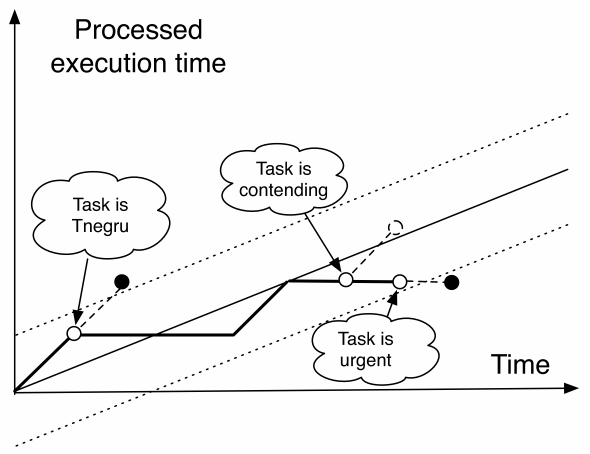

Figure 2. Tnegru, contending and urgent tasks

- the **Urgent** set collects all the behind tasks which would be late (under the lower bound) if they were not processed at time t. These tasks must be processed at time t, else the PFairness condition would be violated.
- the **Tnegru** set collects the ahead tasks which would be in advance (over the upper bound) if they were processed at time t. These tasks must not be processed at time t, else the PFairness condition would be violated.
- the **Contending** set collects the other tasks: the PFairness won't be violated neither if they are processed nor if they are not.

2 - Urgent tasks are executed.
3 - Contending tasks are sorted. The m - $\mid Urgent(t) \mid$ first contending tasks are processed[2].
The different status of tasks is illlustrated by figure 2. Note that usually, PFair strategies are depicted by means of feasibility windows of unitary subtasks. Feasability windows are deduced from the lag inequations. Figure 3 illustrates the windows construction and figure 4 presents an example.

Several PFair versions have been proposed in the litterature (PF, PD and PD^2 [4, 5, 1, 2]). These algorihms differ in the way they select tasks to process among the contending tasks. The remainder of the paper holds whatever the chosen PFair algorithm. These scheduling strategies are very efficient, as stated in theorem 1.

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

[2] $\mid A \mid$ denotes the cardinality of set A

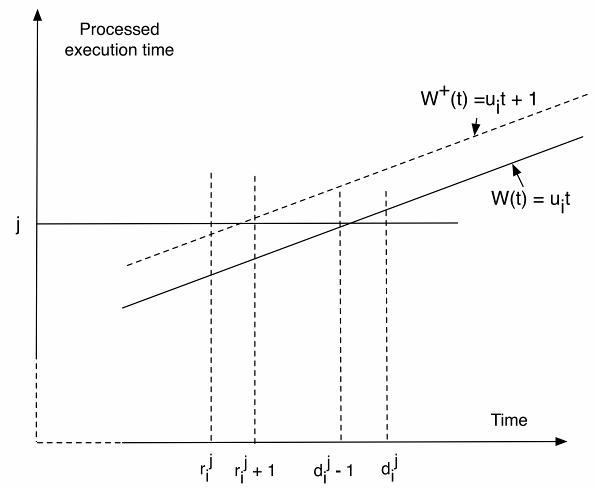

Figure 3. Feasibility windows: construction

–

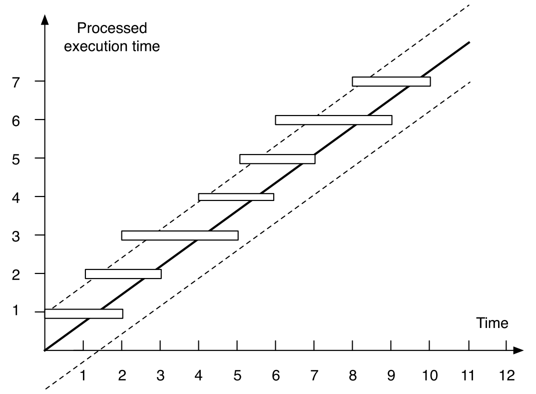

Figure 4. Feasibility windows: example

For multiprocessor systems, Pfair scheduling strategies are the only known optimal strategies. In the sequel, we assume periodic tasks to be scheduled according to a PFair strategy.

2.3 The aperiodic queue

We suppose that the operating system maintains an aperiodic queue as shown on figure 5. The periodic schedule can either be computed on line, or, for more efficiency, have been previously computed. In this case, the scheduler disposes of the periodic schedule. We furthermore

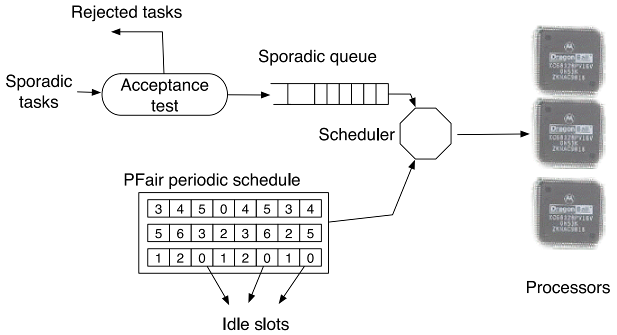

Figure 5. The aperiodic queue

assume the aperiodic queue to be sorted according to EDF (Earliest Deadline First [7]), i.e. in increasing order of deadines. The task with the neartest deadline is processed first. We adopt the background approach: aperiodic tasks are scheduled when some processors have no periodic task to process.

3 Idle time units

Aperiodic tasks can be scheduled each time an idle time unit occurs. The efficiency of our method is based on the predictability of the idle time unit location. For that purpose, we require them to be distributed according to PFair rule. We first have to discuss the way idle time units are supported: they can either take place when no periodic task can be processed, or they can be modeled by a specific task, the idle task. We investigate their location and numbering in both cases.

3.1 Idle time units management

We consider systems with m processors and assume the utilization factor of the application to verify $m - 1 < U < m$. The processor is thus idle $P \times (m - U)$ slots each hyperperiod. The simpler way is to schedule idle time units in background, i.e. to schedule idle time units each time there is less than m ready tasks to process. Note that here, the notion of ready task differs from the classical one: a task is ready if it hasn't completed execution, and if its execution won't lead the task to violate the PFair condition. We can adopt this solution only if we can guarantee a PFair-like distribution of idle time units. This is of matter for us since we want to count on-line the number of idle slots within any time interval, what we can do only if they are PFair distributed. Unfortunately, this is not the case, as illustrated by figure 6. We have considered a system of 5 processors, and an application composed of 16 tasks: ($\tau 1$ = <14, 60>, $\tau 2$ = <26, 300>,$\tau 3$ = <14,50>,$\tau 4$ = <59, 150>,$\tau 5$ = <48, 100>,$\tau 6$ = <87, 300>,$\tau 7$ = <0, 120>,$\tau 8$ = <9, 20>,$\tau 9$ = <63, 300>,$\tau 10$ = <82, 200>,$\tau 11$ = <50,200>,$\tau 12$ = <76, 300>,$\tau 13$ = <4, 10>,$\tau 14$ = <16, 100>,$\tau 15$ = <9, 20>,$\tau 16$ = <65, 300>). We have U = 4.647, P = 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 times units have taken Figure. place 6 shows the distribution of the idle time units for the 80 first slots. We first note that the background idle time distribution doen't respect PFairness. E.g., at time 10, according to PFairness, either 3 or 4 idle time units should have occur, but no one has still occured. Furthermore, we can see that several idle time units may occur simultaneously. In such a case, even if enough idle time units are available for an aperiodic demand, we cannot conclude (i.e. guarantee that the aperiodic demand can be processed on time). Indeed, the demand may come from one single aperiodic task, and an aperiodic task cannot run on several processors at the same time. Thus the number of idle time units it actually may use can be smaller that the required number.

We thus introduce a further task, called **idle task** defined by $\tau_0 =< P \times (m - U), P >$ (remember that we have assumed $m - 1 < U < m$). The system is thus fully loaded (U = m) but is still PFair feasible according to Baruah's theorem. By construction, an idle time unit takes place each time the idle task is processed. Thus, since τ_0 is scheduled by a Pfair algorithm, idle time units are PFairly distributed, and because a task cannot be parallelized, several idle time units can never occur simultaneously.

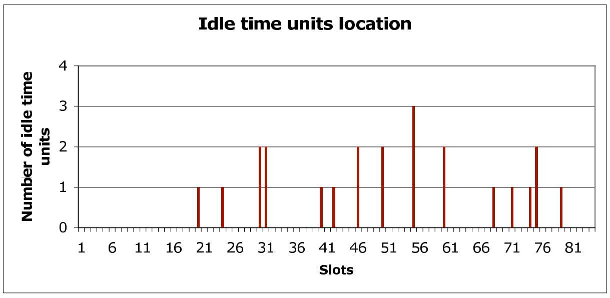

Figure 6. Idle time units repartition for a multiprocessor system: some occur simultaneously

From now on, we adopt the idle task solution.

3.2 Idle time units location

Let us consider a time interval [t, t']. We focus on the idle task τ_0. Beause of PFairness, we have:

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

We can deduce:

$$\lfloor u_0 \times t' \rfloor - \lceil u_0 \times t \rceil \leq W_0(t, t') \leq \lceil u_0 \times t' \rceil - \lfloor u_0 \times t \rfloor$$

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 counted idle time units will be rather small, provided considered intervals are wide enough. In the sequel, we denote by M_W(t, t') this minimal value.

4 The acceptance process

We assume periodic tasks to be scheduled by means of a PFair strategy, and aperiodic tasks to be supported by the idle task. The idle task acts as a PF aperiodic server. Each time the server is scheduled, if there is no pending aperiodic task, the slot is lost, i.e. a real idle time unit occurs. We first present the principle of the acceptance test, and we then give some indications about its implementation.

4.1 The acceptance test

Let $(\tau_{s_1}, \tau_{s_2}, \ldots, \tau_{s_k})$ be the set of the pending accepted aperiodic tasks. We assume the set to be ordered in increasing deadlines order. Let $\tau_s = (t, C_s, D_s)$ be a new aperiodic task with arrival time t. We denote d_s its deadline ($d_s = t + D_s$). We have $d_{s_1} < d_{s2} < \ldots < d_{sk}$.The acceptance test relies on the approximation of the number of idle slots within a time interval by its minimal value. It runs as follows:

- if $d_{s_k} \leq d_s$
  The new aperiodic task has a greater deadline than any pending aperiodic task, so it won't have any impact on their execution. The decision depends on the number of idle time units between t and d_s: either there are enough ones to process τ_s after completion of all pending aperiodic tasks and before d_s, then the task can be accepted or the number of remaining idle time units after completion of the pending aperiodic tasks and before d_s is less than C_s, then the task must be rejected. Thus the task is accepted if and only if

$$W\_M(t, d_s) \geq \sum_{i=1}^{k} RCT_{s_i}(t) + C_s$$

- If $\exists\, j \in 1 \ldots k-1$ such that $d_{s_j} \leq d_s < d_{s_{j+1}}$
  The j^{th} first aperiodic tasks won't be affected by the execution of τ_s, but should τ_s be processed, then it will delay the completion of $\tau_{s_{j+1}}, \ldots, \tau_{s_k}$. We must then make sure that this delay will not cause the temporal failure of some of them. We therefore verify:

  1. τ_s can meet its deadline:

$$W\_M(t, d_s) \geq \sum_{i=1}^{j} RCT_{s_i}(t) + C_s$$

  2. each delayed task will still be processed on time:

$$\forall p \in j+1 \ldots k,$$

$$W\_M(t, d_{s_p}) \geq \sum_{i=1}^{p} RCT_{s_i}(t) + C_s$$

- If $d_s < d_{s_1}$. Each pending aperiodic task will be affected by the execution of τ_s. Thus we must verify that

  1. W_M(t, d_s) $\geq C_s$ i.e. the new task can respect its temporal contraint,
  2. each delayed task will still be processed on time:

$$\forall p \in 1 \ldots k, W\_M(t, d_{s_p}) \geq \sum_{i=1}^{p} RCT_{s_i}(t) + C_s$$

We can note that we never have to consider periodic tasks, which simplifies the decision process.

4.2 Implementation

We first compute the periodic schedule before runtime. It is computed on the interval [0, P] and then iterated. 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 invocated, and if the aperiodic queue is not empty, the first aperiodic task is processed. For the sake of the acceptance test, we must maintain the remaining processing times of every pending aperiodic tasks. Thus, we maintain a list of aperiodic accepted tasks sorted in increasing deadline order. For the i^{th} task, we store: - its deadline d_{s_i}, - its remaining computing time RCT_{s_i}, - the cumulated remaining aperiodic processing time that must be completed at the latest at d_{s_i} (see table 1).

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

Table 1. Aperiodic table used by the acceptance algorithm

The acceptance algorithm is then

`Function Accept`
input
u_0: `float --` *utilization factor of the idle task* τ_0
`T --` *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
M_W :=⌊ u0 × d ⌋ − ⌈ u0 × t ⌉
If T is empty then
-- there is no aperiodic pending task
   accepted :=(M_W ≥ C)
 end if
elsif d < T(1).dl then
-- the new task will postpone completion times of all pending aperiodic tasks
 If M_W ≥ C
   then accepted := true
   for i in 1..k loop
     if ⌊ u0 × T(i).dl ⌋ − ⌈ u0 × t ⌉
                     < C + T(i).C_RCT
       then accepted := false
      end if
    end loop
    else accepted := false
  end if
elsif T(k).dl ≤ d then
-- the new task will have no incidence on other aperiodic tasks
  If ⌊ u0 × d ⌋ − ⌈ u0 × t ⌉
             ≥ C + T(k).C_RCT
     then accepted := true
     else accepted := false
  end if
else i:= 1
  While T(i+1).dl ≤ d loop
      i := i+1
  end loop
-- we have d_i ≤ d ≤ d_{i+1}
  if ⌊ (u0 × T(i).dl ⌋ − ⌈ u0 × t ⌉
                   ≥ C + T(i).C_RCT
    then accepted := true
      for j in i+1..k loop
        if ⌊ u0 × T(j).dl ⌋ − ⌈ u0 × t ⌉
                             < C + T(j).C_RCT
          then accepted := false
         end if
       end loop
   else accepted := false
   end if
end if
return accepted
```

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

```
Function insert_task
 input
T -- pending accepted aperiodic tasks
τ = (t, C, D) -- new aperiodic task
 Precondition
Task τ is accepted
 output
the updated list of accepted tasks

d = t + D
If T is empty then
 insert((d, C, C), 1, T)
elsif d < T(1).dl then
 for i in 1..k loop
   T(i).C_RCT := T(i).C_RCT + C
 end loop
 insert((d, C,C), 1, T)
elsif T(k).dl ≤ d then
 insert((d, C, T(k).C_RCT + C), k+1, T)
else i:= 1
  While T(i+1).dl ≤ d loop
    i := i+1
  end loop
  for j in i+1..k loop
    T(j).C_RCT := T(j).C_RCT + C
  end loop
  insert((d, c, T(i).C_RCT + C), i+1, T)
end if
return(T)
```

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

```
Function schedule
 input
T -- pending accepted aperiodic tasks
L -- list of the new arrived aperiodic tasks
t -- current time
 output
updated list of pending aperiodic tasks
identity of the processed task

while L is not empty loop
  τ := head(L)
  unqueue(L)
  If accept(u0,T,τ) then
     insert_task(T, τ)
  end if
end loop
if T is not empty then
   id :=T(1).id
   T(1).RCT := T(1).RCT - 1
   for i in 1..k loop
     T(i).C_RCT := T(i).C_RCT - 1
   end loop
   If T(1).RCT = 0 then del(T, 1)
   end if
end if;
return(id, T)
```

5 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, on the interval [0, P]. Then, computation of the number of idle time units within any time interval is done 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.
The second point of interest is to compare our method to previously existing methods and to optimal strategies. The greatest challenger for our method consists in scheduling at run-time together periodic and aperiodi tasks according to PFair policy. The acceptance test is very simple: if $\sum_{i=1}^{n} \frac{C_i}{P_i} + \sum_{j=1}^{k} \frac{C_{s_j}}{D_{sj}} + \frac{C}{D} \leq m$ then the task (t, C, D) 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. 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 prove it, some simulations have been carried out. We first create samples of periodic task sets. A sample S(U,m) consists in 500 task sets, where m is the number of processors and is characterized by:

- the utilization factor of any task set of the sample must belong to the interval $[m-1+\frac{i}{10}, m-1+\frac{i+1}{10}]$, for i in 2...9.
- periods are chosen according to Goossens' method [9].
- WCET C_i are chosen uniformly within the interval $[1, \frac{P_i}{2}]$

For each task sets, we then generate a aperiodic task flow. A flow is characterised by:

- the interarrival obeys an exponential law, with mean x
- deadlines must be less than P
- relative deadlines D_{s_i} are uniformaly chosen within the interval [10, D_Max]
- the WCET of a task τ_{s_i} is uniformaly chosen within the interval $[\frac{D_{s_i}}{10}, \frac{D_{s_i}}{2}]$

For each sample, we carry out 3 simulations, over the time interval [0, P):

1. we first use our method, periodic tasks are scheduled by PF
2. we implement the joined PFair method
3. we also implement an exact method based on PF periodic scheduling. 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 use competitive analysis, using the exact PFair method as referent method. For any couple (tasks set, aperiodic flow), we compute 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 method. We have repeated the experience for different tuple (m, x, D_max). Figure 7 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 ou 4 processors, x = 20 or 40, D_Max = 40 or 200) lead to similar figures.

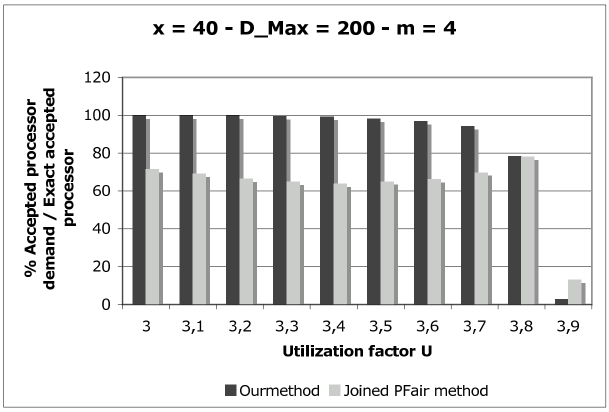

Figure 7. Comparison our method and joined PFair method

We can see that for almost all values of U, our test behaves almost like the exact test, meanwhile the joined PFair method has lower performances, in the sens that it accepts less aperiodic load. Performances 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 idle slots number becomes significant, and in such cases, the joined PFair method becomes competitive. But in almost all cases, our method has higher performances. Note that equivalent obervations can be made for uniprocessor systems, for which the challenger method to our ones is EDF [8]. Experimentations show that, except for high values of U, our method has significantly better results than EDF.

6 Conclusion

We have proposed an efficient acceptance test for aperiodic tasks, with firm deadlines. Periodic tasks are scheduled by a PFair algorithm, e.g. PF [4]. The periodic schedule is steady, new accepted aperiodic tasks do not interfere with periodic tasks. This has two advantages: the periodic schedule can be computed before run-time over the interval [0, P), and is then iterated, and the periodic task location is deterministic. The acceptance test runs in O(k) where k is the number of pending aperiodic tasks. We have here assumed tasks to be synchronous (they are first released at the same time), with implicit deadlines, but we can note that the synchronous assumption can be omitted [10]. The method we propose can be used for uniprocessor systems as well as for multiprocessor systems if m - 1 < U < m, where m is the number of processors. If U ≤ m - 1, the problem is more intricated, since there must be more than one idle task. In this case, there is one (or more) idle processor, and we must first partition aperiodic tasks between idle processor(s) and active processors. Tasks assigned to the active set are scheduled according to our methodology. Further investigation will deal with the way tasks can be partitionned, and look for the best way to schedule tasks on the idle processor(s). Figure 8 illustrates the complete method for $m - 2 \le U \le m - 2$.

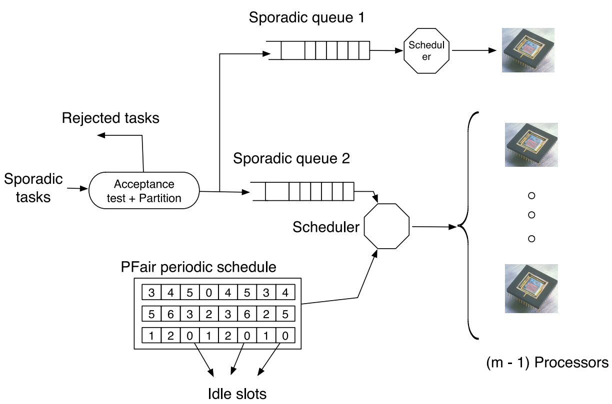

Figure 8. The method when $m - 2 \le U \le m - 1$. Aperiodic tasks are distributed in two separate queues. One of the processor is completely dedicated to aperiodic service. The (m-1) others are used to schedule conjointly periodic and aperiodic tasks according to our methodology.

References

[1] J. Anderson, A. Block, and A. Srinivasan. Pfair scheduling : Beyond periodic task sytems. In *Proceedings of the* 12^{th} *Euromicro Conference on Real-Time Systems*, pages 35–43. Chapman and Hall, 2000.

[2] 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.

[3] S. Baruah. Fairness in periodic real-time scheduling. In *Proceedings of the* $16^t h$ *IEEE Real-Time Systems Symposium*, pages 200–209, 1995.

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

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

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

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

[8] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[9] 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.

[10] S. Malo, A. Choquet-Geniet, and M. Bikienga. Extension of pfair scheduling application context. Technical report, LISI, ENSMA, 2008.