Finding cyclic behavior in multiprocessor real-time scheduling

Annie Choquet-Geniet [1]

Laboratoire d'Informatique Scientifique et Industrielle
Université de Poitiers & E.N.S.M.A.
Téléport 2, Site du Futuroscope, F-86961 Futuroscope Chasseneuil Cédex

&

Sadouanouan Malo

Ecole Supérieure d'Informatique - Université Polytechnique de Bobo Dioulasso,
01 BP 1091 Bobo Dioulasso 01, Burkina Faso

Abstract

This paper concerns sets of periodic independent real-time tasks with hard deadlines, in a multiprocessor context. We address the cyclicity problem for global multiprocessor scheduling. Our aim is to prove the existence of a steady state after a transcient state in valid schedules. This could be helpfull for performing exact schedulability tests as well as for the sake of simulation. First, we underline the main differences between the uniprocessor and the multiprocessor cases. Then we consider the case of fixed-priority scheduling strategies, and finally, we extend our results to a wider class of scheduling algorithms. And finally, we present some amazing results as to the date of the begining of the steady state.

Key words: Real-time systems, Scheduling, Multiprocessor Systems

---

[1] author for all correspondance
Email addresses: agenieт@ensma.fr (Annie Choquet-Geniet), sadouanouan.malo@laposte.net (Sadouanouan Malo).

1 Introduction

1.1 The problem

The general use of multiprocessor architectures gave rise to many investigations in the field of multiprocessor scheduling. For uniprocessor systems, scheduling has been widely studied and many solutions have been proposed. However in the case of the multiprocessor context it becomes very complex, and many questions still remain to be answered. One of the most difficult questions concerns the existence of a steady state, after the system loading phase. Solving the cyclicity problem implies first to prove the occurrence of the steady state, and second to characterize the moment at which it begins. This issue has been successfully addressed for uniprocessor systems [9]. The key point of the proofs is the analysis of the processor activity which is closely related to the processor demand or backlog (the sum of the remaining processing times of pending tasks): the processor is shown to be idle only if there are no pending tasks. Unfortunately, this result does not hold anymore in multiprocessor context. This is due to the assumption that a task cannot run on two different processors at the same time. Thus uniprocessor methods cannot be extended to multiprocessor systems. The problem must be considered in a completely new way. The aim of this paper is to provide a first approach. Since the problem turns out to be very intricate, we have restricted ourselves to sets of independent periodic hard real-time tasks, with offsets. Furthermore we first focuse on global fixed priority scheduling algorithms and then we extend the results to a wider class, wich includes global EDF and global LLF scheduling.

1.2 Multiprocessor scheduling

Scheduling real-time applications on multiprocessor architectures can be solved using either global or partitioned approaches. In partitioned methods, all instances of a task run on the same processor. The problem is here to determine the most suited allocation of tasks onto the different processors. Then each set of tasks is scheduled on its own processor using a given uniprocessor scheduling algorithm. In global methods, tasks can run at any time on any processor. So a task is never definitively assigned to a given processor, it may start and resume on any one. Both methods cannot be compared in the sense that there exist tasks systems that can be scheduled when using one of the method, but not when using the other one. Furthermore, the problem of deciding whether there exists a valid schedule for a given tasks systems is NP-hard for both methods. See [2, 17] for a comparison between both approachs and [23] for an overview on the complexity of the problem of the existence of a valid schedule.
The cyclicity problem can easily be solved for partitioned scheduling. The solution is a straightforward extension of the uniprocessor result. Indeed, the schedule computed on a processor P_i (tasks are assumed to be independent) is cyclic from a given time t_i that can be deduced from the tasks parameters (see [9]) with a period equal to the least common multiple of the periods of the tasks assigned to this

processor. Therefore, the whole schedule is cyclic from the greatest t_i, with period equal to the least common multiple of the periods of all the tasks.
So we focuse on global scheduling, for wich the problem is still open. In [10] it has been proven that no on-line optimal [2] scheduling algorithm can exist, in the general case. Nevertheless, there still exists an optimal polynomial algorithm, as well as a necessary and sufficient schedulability condition, if only synchronous [3] independent tasks with implicit deadlines [4] are considered [1,6]. Beside, many investigations have been made on schedulability tests, leading to sufficient schedulability condition (see e.g. [3–5, 8, 12]) but for asynchronous systems, there exists no exact schedulability test, and only simulation can lead to conclusion when the sufficient conditions are not met. Furthermore on-line scheduling for multiprocessor systems comes up against the problem of scheduling anomalies, even if tasks are independent: the scheduling strategy is validated using worst case execution times, however if the actual computing time of some instance of a task is shorter, some temporal faults may occur [13, 19, 23]. An alternative solution consists in considering off-line scheduling: a schedule previously computed is stored within a table used by the dispatcher. Since these strategies are clairvoyant [5] their decisions are made according to a complete knowledge of the systems, whereas on-line algorithms rely on the instantaneous state of the system. Off-line strategies are thus more powerful than on-line strategies. The counterpart is the rather high cost of off-line methods. Indeed, they are often based on an exhaustive enumeration of the set of possible solutions. Most off-line strategies that can be found in the literature deal with non periodic applications. Only finite sets of jobs, which may be independent or not, are scheduled possibly using some heuristics or branch-and-bound algorithms in order to bound the cost of the method ([11, 21, 22, 24]).
There are significantly less results about the off-line scheduling of periodic tasks sets: methodologies based respectively on a geometrical and a Petri nets based modeling of the application have been developped in [16] and [14], but these methodologies can only deal with synchronous systems.

1.3 Our contribution

A systematic investigation of the cyclicity problem in the multiprocessor context would help to overcome the lack of off-line scheduling strategies for periodic asynchronous applications. As a matter of fact, if the system is proven to behave cyclically after the loading phase, and if the start date t of the cycle is characterized, we can demonstrate that strategies developed for single-instance tasks can be enlarged to periodic tasks sets. Only instances of periodic tasks of which arrival dates stand before $t + P$ (where P is the least common multiple of the periods) must be con-

[2] an algorithm is said to be **optimal** if for any application, either it computes a feasible schedule (a schedule where all the deadlines are met) or there exists no such schedule at all
[3] the release times are all the same
[4] due dates are equal to the periods
[5] the release dates of every instances of every tasks are known

sidered. A feasible schedule [6] involving these instances would lead to an infinite feasible schedule. Without such a cyclicity result, the size of the schedule that has to be computed cannot be bounded. Consequently off-line methods would not work either. Another interesting application of our result concerns simulation: simulating the behaviour of the application according to the chosen scheduling strategy can be used either to analyse the performances of this strategy (see e.g. [2] which compare global and partitioned fixed-priority scheduling for synchronous systems) or to produce a validation test, when available schedulability tests cannot be used (either a sufficient condition is not verified or there exists no such condition). The main question is: how long must the simulation be carried on? The cyclicity result would answer the question.
We consider sets of n independent tasks, and m processors. We first investigate the cyclicity properties of fixed-priority scheduling strategies. Our goal is to characterize the beginning of the schedule cyclic part. We then enlarge the results to a wider class of algorithms. The remainder of the paper is organized as follows. In part 2, our assumptions and notations are stated. In part 3, we introduce partial and complete acyclic idle slots, and show that the core results of the uniprocessor investigation do not hold anymore under the multiprocessor assumption. In part 4, it is shown that the cycle corresponds to the first schedule of size lcm [7] of the periods without any acyclic idle slots, if the system is fully loaded. In part 5, the results are extended to not fully loaded systems. Finally, part 6 extends our results to further algorihtms, including EDF and LLF, and presents the perspectives of our researchs. In particular, we present some amazing examples, which show that that for EDF and LLF, the beginning of the steady state can occur very late compared to the uniprocessor bound $(r + P)$.

2 System model

The PRAM model [15] with m processors is considered: processors are all identical, each has its own memory, but a common memory can be accessed by every one, in constant time. We adopt the global assumption: a tasks is never definitively assigned to a given processor, it can at any time resume on any processor. We consider applications composed of n independent periodic tasks,$\tau_1(r_1,\ C_1,\ D_1,\ P_1)$, $\tau_2(r_2,\ C_2,\ D_2,\ P_2)$, $\ldots$, $\tau_n(r_n,\ C_n,\ D_n,\ P_n)$ (in the sequel, $\mathcal{E}$ denotes the set $\{\tau_1, \tau_2, \ldots, \tau_n\}$). Each task is submitted to hard temporal constraints. We adopt the classical modeling of tasks [18]. Each task τ_i is characterized by four temporal parameters as described in figure 1: first release date or offset r_i; worst-case execution time C_i; relative deadline D_i, which corresponds to the maximal delay allowed between the release and the completion of any instance of the task; and period P_i. Each task τ_i consists of an infinite set of instances (or jobs), released at times $r_i + k \times P_i$, with $k \in \mathbb{N}$. We assume that parallelism is forbidden (at any time, a task can run on at

[6] a schedule is **feasible** if all the deadlines are met
[7] lcm denotes the least common multiple

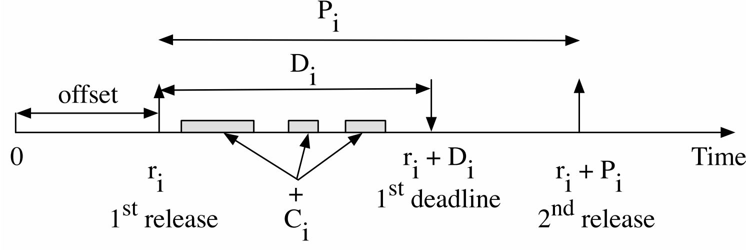

Fig. 1. Temporal modeling of a real time periodic task

most one processor) and that temporal parameters are known and deterministic. P denotes the **hyperperiod** of the system defined as P = lcm(P_1, P_2, $\ldots$, P_n), and r is the latest release time: r = Max$\{r_1, \ldots, r_n\}$.
The processor **utilization factor** characterizes the processors 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 [7].
For any times t and t', and for any task τ_i, the following terminology is used (see figure 2):

- $PI_i(t)$ is the **pending instance** of task τ_i at time t (the last instance released at or before t).
- $RCT_i(t)$ is the **remaining computation time** for the instance $PI_i(t)$ of tasks τ_i.
- $\overline{RCT}_i(t)$ corresponds to the **elapsed computation time** of the instance $PI_i(t)$.
- $W(t, t')$ denotes the **cumulated processed execution time** between time t and time t'.
- $W_i(t, t')$ is the **processed execution time** for task τ_i between time t and time t'.

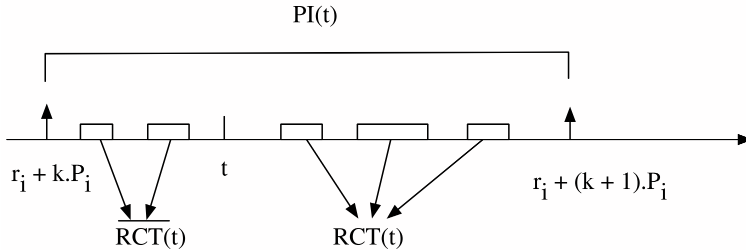

Fig. 2. Pending instance, elapsed computation time and remaining computation time

Following the definitions, RCT_i(t) + $\overline{RCT}_i$(t) = C_i. Furthermore, if processors are never idle between t and t', W(t, t') = m $\times$ (t'- t).
For the sake of the instantaneous description of the system, the notions of state for a task and for the whole system are introduced.

Definition 1 *The* **state of a task** τ_i *at time t is defined by :*

$$st_i(t) = \begin{cases} \bullet\ (0,\ dist_i(t) = r_i - t) \text{ if } \tau_i \text{ is not yet released } (t < r_i) \\ \bullet\ (RCT_i(t),\ dist_i(t)) \text{ else, where } dist_i(t) \text{ denotes the remaining time until the next release.} \end{cases}$$

The **state of the system** *S at time t is defined by* $ST_S(t) = (st_1(t), st_2(t), \ldots, st_n(t))$.

In the further, **slot** t denotes the time interval [t, t+1]. A task is **scheduled at time t** means that one processor processes it during slot t.
We recall that $\mathcal{E}$ is $\{\tau_1, \tau_2, \ldots, \tau_n\}$, let $\mathcal{P}_m(\mathcal{E})$ denotes the set of subsets of $\mathcal{E}$ of size less than or equal to m.
A **schedule** on m processors is defined by O : $\mathbb{N} \rightarrow \mathcal{P}_m(\mathcal{E})$ such that $\tau_i \in$ O(t) $\Leftrightarrow$ τ_i is scheduled at time t.
Let O_i be such that $O_i(t) = \begin{cases} 1 \text{ if } \tau_i \in O(t) \\ 0 \text{ else} \end{cases}$.
A schedule is **feasible** if and only if $\forall i \in 1..n$, $\sum_{t=0}^{r_i-1} O_i(t) = 0$ and, $\forall k \in \mathbb{N}^*$,
$\sum_{t=0}^{r_i+(k-1)P_i+D_i-1} O_i(t) = \sum_{t=0}^{r_i+kP_i-1} O_i(t) = k \times C_i$.
A scheduling strategy is **conservative** if a task never intentionally waits [8] :

$$\mid O(t) \mid = \begin{cases} \mid \{i \mid st_i(t) = (a,b) \text{ with } a>0\} \mid \text{ if } \mid \{i \mid st_i(t) = (a,b) \text{ with } a>0\} \mid < m \\ m \text{ otherwise} \end{cases}$$

A schedule is **deterministic** if and only if scheduling decisions are the same each time the states are the same : ST_S(t) = ST_S(t') $\Rightarrow$ O(t) = O(t').
In the sequel, only conservative and deterministic schedules are considered. Furthermore the allocation problem is not addressed.
A schedule is **cyclic** with period P from time t_c if and only if $\forall$ t $\geq$ t_c, O(t) = O(t+P).
If the schedule is deterministic, the cyclicity can be deduced from state examination, as stated by the following lemma.

Lemma 2 *Let O be a deterministic schedule. If there exists* t_0 *such that* $ST_S(t_0) = ST_S(t_0 + P)$ *and* $t < t_0 \Rightarrow ST_S(t) \neq ST_S(t + P)$ *then O is cyclic with period P from* t_0.

It comes from the fact that, thanks to determinism, O(t) depends only on $ST_S(t)$.

[8] If A is a set, | A | is the cardinality of A

3 Idle slots

First special attention is paid to processor activity. In the sequel, only feasible schedules are concerned.

3.1 Acyclic idle time units

When the processor utilization factor is less than m, processors are globally idle for at least $P \times (m - U)$ processing time units [9] each hyperperiod. These idle time units are called **cyclic** since they occur regularly. If the processor utilization factor is equal to m, there are no such idle time units. But nevertheless, some further idle time units may still occur in the transient state, as underlined by the following example. We consider 2 processors and a system S1 composed of 5 tasks : S1 = $\{\tau_1(0, 1, 3, 3), \tau_2(0, 1, 3, 3), \tau_3(0, 4, 9, 9), \tau_4(0, 2, 3, 3), \tau_5(8, 2, 9, 9)\}$. Tasks are assumed to be ordered in decreasing priority order (τ_1 having the highest priority and τ_5 the lowest). The schedule produced by the associated fixed-priority algorithm is depicted on figure 3. The processors utilization factor equals to 2 (note that none of

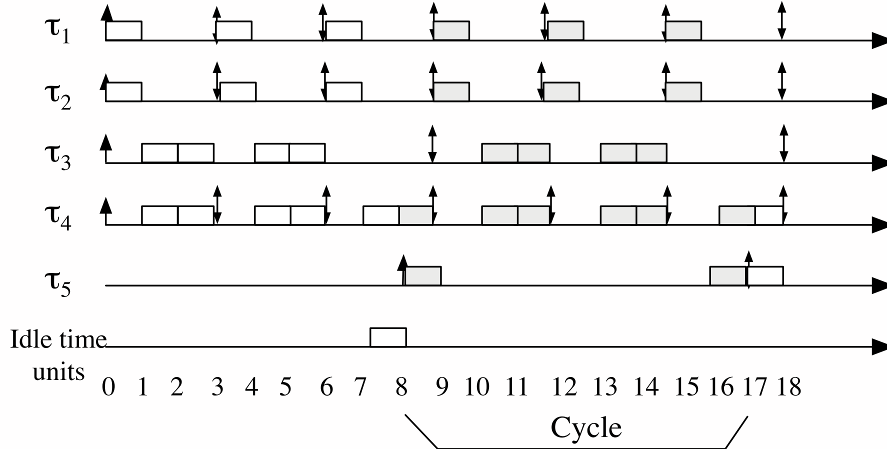

Fig. 3. Schedule for system S1. An idle time unit occurs at time 7: one processor is idle at time 7.

the sufficient fixed-priority schedulability condition can be applied). There are no cyclic idle time units, but nevertheless, an idle time unit occurs at time 7. Besides, the system is in the same state at times 8 and 17 = 8 + P ($ST_{S1}(8) = ST_{S1}(17)$). The schedule is thus cyclic from 8 (lemma 2). The idle time unit observed at time 7 never appears again. It is called an **acyclic** idle time unit.
In uniprocessor case, we have shown that the number and the location of the acyclic idle slots depend only on the application, not on the scheduling strategy, provided it is conservative [9]. Unfortunately, this does not hold for multiprocessors. Figure 4 presents another deterministic and conservative schedule (not provided by a fixed priority strategy) again for tasks system S1.There are two idle time units, at times

[9] a processor processes one processing time unit during each slot. There are thus globally $m \times P$ processing time units each hyperperiod

2 and 5.
Another point had been deduced from uniprocessor investigations: it is always the

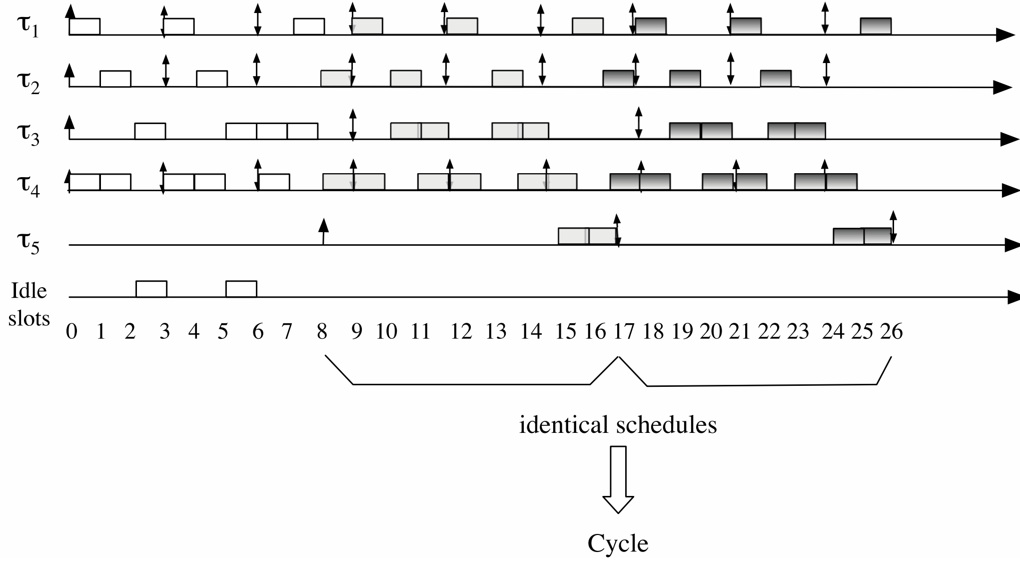

Fig. 4. Another feasible schedule for system S1. There are two idle time units, the last one occurs at time 5 and the cycle begins at time 8: $ST_{S1}(8) = ST_{S1}(17)$ but $ST_{S1}(7) \neq ST_{S1}(16)$.

case that the cyclic behaviour starts directly after the last acyclic idle time unit. Here again, this properties cannot be generalized to multiprocessors, as shown by figure 4: the cycle starts at time 8, but the last idle time unit occurs at time 5. We can note that the cycle cannot start at time 6, since between times 6 and 15, task τ_5 is never scheduled. This put an end to any attempt to generalize uniprocessor results to multiprocessors.
These rather negative observations incited us to restrict our investigations field. In paragraphe 4, we will thus consider only fixed-priority scheduling strategies.

3.2 Partial and complete idle slots

A slot t is called **idle slot** if at least one idle time unit occurs during slot t. In uniprocessor systems, there is no distinction between idle slots and idle time units. For uniprocessors, idle slots implies no pending task, so no remaining processing time, but this does not hold for multiprocessors. It just implies that there are less than m pending tasks. Two kinds of idle slots can be distinguished (see figure 5):

- **Complete idle slots**: all processors are idle, so consequently there are no pending tasks. There are thus m idle time units during this slot.
- **Partial idle slots**: only q processors among m ($0 < q < m$) are not idle, so there are q pending tasks, and the cumulated remaining processing time is not equal to 0. There are $m - q$ idle time units during this slot.

We first prove that the number of acyclic idle time units is bounded.
Proposition 3 *Let S be a fully loaded system of tasks, and O a feasible schedule. Then the number of acyclic idle time units within O is bounded.*

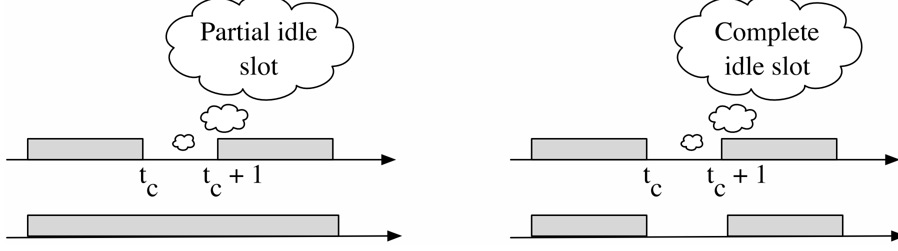

Fig. 5. Partial and complete idle slots, with two processors.

PROOF. Consider a time interval $[0, r + k \times P]$ (k $\in \mathbb{N}$ and r = max$\{r_1, \ldots, r_n\}$). A task τ_i completes at least [10] $\lfloor \frac{r-r_i}{P_i} \rfloor + \frac{k \times P}{P_i}$ times within this window. Thus, the execution of task τ_i requires at least $(\lfloor \frac{r-r_i}{P_i} \rfloor + \frac{k \times P}{P_i}) \times C_i$ processing time units. It follows that the processors have globally to execute at least $(\sum_{i=1}^{n} (\lfloor \frac{r-r_i}{P_i} \rfloor + \frac{k \times P}{P_i}) \times C_i)$ $= (\sum_{i=1}^{n} \lfloor \frac{r-r_i}{P_i} \rfloor) \times C_i + m \times k \times P$ time units (remember that U = m). There are thus at most $m \times (r + k \times P) - [(\sum_{i=1}^{n} \lfloor \frac{r-r_i}{P_i} \rfloor \times C_i + m \times k \times P] = m \times r - (\sum_{i=1}^{n} \lfloor \frac{r-r_i}{P_i} \rfloor) \times C_i$ processing time units left for idle slots. This number does not depend on k, so this bound hold for infinite schedules. Therefore, the number of acyclic idle slots is bounded. □

4 Fully loaded processors and fixed-priority algorithms

We focuse on fixed-priority scheduling strategies. For the sake of determinism, we assume that two different tasks have different priorities. And we assume the utilization factor to be equal to m. There are thus no cyclic idle slots. Our aim is to characterize the steady state start point, by means of the last acyclic idle slot. We claim that the steady state corresponds to the first part of the schedule of size P which contains no idle slots.

Theorem 4 *In a fixed-priority schedule O, if there is an idle slot at time t_c followed by P time units without idle slots, then the schedule obtained within the window $[t_c + 1, t_c + P + 1)$ defines the steady state of the application. This can be formally expressed by:*

$$\begin{cases} \mid O(t_c) \mid < m \\ \forall t \in [t_c + 1, t_c + P], \mid O(t) \mid = m \end{cases} \Rightarrow \textit{O is cyclic from } t_c + 1.$$

The remainder of this section consists of the proof of this result. We discuss on the nature of the idle slot occurring at time t_c (either complete or partial). If no idle slot occurs at all, we state t_c = -1.

[10] $\lfloor$ x $\rfloor$ denotes the greatest integer less than or equal to x

4.1 *Complete idle slot*

We assume first the idle slot at t_c to be complete. The task set can be decomposed in three parts:

- **Re** is the set of tasks for which t_c + 1 is a (possibly first) release date:
  $\tau_i \in RE \Leftrightarrow (t_c + 1 - r_i)\ mod\ P_i = 0$ and $r_i \leq t_c + 1$
- **LR** is the set of the late released tasks, whose offsets are greater than t_c + 1:
  $\tau_i \in LR \Leftrightarrow r_i > t_c + 1$
- **AR** is the set of the already released tasks. Their offsets are less than t_c and t_c + 1 is not a release date:
  $\tau_i \in AR \Leftrightarrow (t_c + 1 - r_i)\ mod\ P_i \neq 0$ and $r_i \leq t_c$

4.1.1 *Processed execution time between $t_c + 1$ and $t_c + P + 1$*

The basis of the proof consists in the estimation of $W(t_c + 1, t_c + 1 + P)$, which is the cumulated processed execution time between t_c + 1 and t_c + P + 1. Because of the assumption that no idle slot takes place between these two dates, this execution time equals to $m\times$ P. The contribution of each set of tasks to the whole execution time is computed as follows:

- **Load due to tasks in Re**: each task τ_i within Re is processed exactly $\frac{P}{P_i}$ times, thus Re generates a processor load equal to $\sum\limits_{\tau_i \in Re} \frac{P}{P_i} \times C_i$.
- **Load due to tasks in LR**: each task τ_i within LR completes $\lfloor \frac{t_c+P+1-r_i}{P_i} \rfloor$ instances, and starts a last one. Furthermore, since τ_i is not yet released at $t_c + 1$, we have RCT($t_c + 1$) = 0. Thus the cumulated processed time equals to $\sum\limits_{\tau_i \in LR} \lfloor \frac{t_c+P+1-r_i}{P_i} \rfloor \times C_i + \overline{RCT}_i(t_c + P + 1)$.
- **Load due to tasks in AR**: each task τ_i within AR completes $\frac{P}{P_i} - 1$ instances, and starts a last one. Furthermore, at time t_c + 1, there is no pending task left, since the idle slot at time t_c is complete, and t_c +1 is not a release date for tasks in AR. Thus, $RCT_i(t_c + 1) = 0$ and the cumulated load coming from AR equals to $\sum\limits_{\tau_i \in AR} (\frac{P}{P_i} - 1) \times C_i + \overline{RCT}_i(t_c + P + 1)$.

We thus have the following load equation:

$$\begin{aligned} W(t_c + 1, t_c + P + 1) &= \sum_{\tau_i \in Re} \frac{P}{P_i} \times C_i \\ &+ \sum_{\tau_i \in LR} (\lfloor \frac{t_c+P+1-r_i}{P_i} \rfloor \times C_i + \overline{RCT}_i(t_c + P + 1)) \\ &+ \sum_{\tau_i \in AR} ((\frac{P}{P_i} - 1) \times C_i + \overline{RCT}_i(t_c + P + 1)) \\ &= m \times P \end{aligned}$$

4.1.2 *Late first releases*

In order to refine the estimation of the processed execution time due to tasks with late first releases, we must investigate their offsets. Lemma 5 states that these tasks must start less than a period after $t_c + 1$. Otherwise, some further idle slots would occur after t_c.

Lemma 5 $\forall \tau_i \in LR$*, we have* $t_c + 1 < r_i < t_c + P_i + 1$.

PROOF.
a - We first assume there exists i_0 such that $r_{i_0} > t_c + P_{i_0} + 1$. It follows that $\lfloor \frac{t_c+P+1-r_{i_0}}{P_{i_0}} \rfloor < \frac{P}{P_{i_0}} - 1$. Besides, we know that $\overline{RCT}_i(t_c + P + 1) \leq C_i$ for each task, and $\lfloor \frac{t_c+P+1-r_i}{P_i} \rfloor \leq \frac{P}{P_i} - 1$ for each task τ_i in LR but τ_{i_0} (because $t_c + 1 < r_i \Rightarrow \frac{t_c+1-r_i+P}{P_i} < \frac{P}{P_i}$). Thus we have:

$$\begin{aligned} m \times P &= W(t_c+1, t_c+P+1) \\ &\leq \sum_{\tau_i \in Re \cup AR \cup (LR \setminus \{\tau_{i_0}\})} \frac{P}{P_i} \times C_i + \lfloor \frac{t_c+P+1-r_{i_0}}{P_{i_0}} \rfloor \times C_{i_0} + \overline{RCT}_{i_0}(t_c+P+1) \\ &< \sum_{i=1}^{n} \frac{P}{P_i} * C_i - C_{i_0} + C_{i_0} \\ &< m \times P \end{aligned}$$

So the contradiction.
b - Let us now assume there is some i_0 such that $r_{i_0} = t_c + P_{i_0} + 1$. We have $\lfloor \frac{t_c+P+1-r_{i_0}}{P_{i_0}} \rfloor = \frac{P}{P_{i_0}} - 1$ and $\overline{RCT}_{i_0}(t_c + P + 1) = 0$. This produces the same inconsistency as in case a. □

4.1.3 Proof of the theorem

We can now estimate the contribution of each tasks subset, and complete the proof. From lemma 5, we have for each task τ_i in LR $\lfloor \frac{t_c+P+1-r_i}{P_i} \rfloor = \frac{P}{P_i} - 1$. We thus have:

$$\begin{aligned} m \times P &= W(t_c+1, t_c+P+1) \\ &= \sum_{i=1}^{n} \frac{P}{P_i} \times C_i + \sum_{\tau_i \in LR \cup AR} (\overline{RCT}_i(t_c+P+1) - C_i) \\ &= m \times P + \sum_{\tau_i \in LR \cup AR} (\overline{RCT}_i(t_c+P+1) - C_i) \end{aligned}$$

Now, for each task τ_i in LR ∪ AR, $\overline{RCT}_i(t_c + P + 1) - C_i \leq 0$. It follows that $\overline{RCT}_i(t_c + P + 1) = C_i$. Thus

$$\forall \tau_i \in LR \cup AR, RCT_i(t_c+1) = RCT_i(t_c+P+1) = 0$$

Besides

$$\tau_i \in Re \Rightarrow RCT_i(t_c+1) = RCT_i(t_c+P+1) = C_i$$

Finally, for each task τ_i, we have $dist_i(t_c + P + 1) = dist_i(t_c + 1)$ since $dist_i$ is periodic, with period P_i. Thus for each task τ_i, we have $st_i(t_c+P+1) = st_i(t_c+1)$. From lemma 2, we conclude that O is cyclic from $t_c + 1$. □

4.2 Partial idle slot

At time t_c, q tasks are pending, with 0 < q < m. Without loss of generalities, we can assume that tasks $\tau_1, \tau_2, \ldots, \tau_q$ are processed at time t_c. Set $\mathcal{E}$ is decomposed into four subsets:

- **PT** is the set of the pending tasks at time t_c (PT = $\{\tau_1, \tau_2, \ldots, \tau_q\}$).
- **Re**, **LR** and **AR** are defined as in previous section, but considering only tasks which do not belong to PT.

The contribution to the cumulated processed execution time between t_c + 1 and t_c + P + 1 of any task τ_i in PT is $(\frac{P}{P_i} - 1) \times C_i + RCT_i(t_c+1) + \overline{RCT}_i(t_c+P+1)$. We consider the complete instances, plus what is still to be processed at time t_c + 1 plus what has already been processed at time t_c + P + 1. The load equation becomes:

$$\begin{aligned}
W(t_c+1, t_c+P+1) = & \sum_{\tau_i \in Re} \frac{P}{P_i} \times C_i \\
& + \sum_{\tau_i \in PT} (\frac{P}{P_i} \times C_i) \\
& + \sum_{\tau_i \in PT} [RCT_i(t_c+1) + \overline{RCT}_i(t_c+P+1) - C_i]) \\
& + \sum_{\tau_i \in LR} (\lfloor \frac{t_c+P+1-r_i}{P_i} \rfloor \times C_i + \overline{RCT}_i(t_c+P+1)) \\
& + \sum_{\tau_i \in AR} ((\frac{P}{P_i} - 1) \times C_i + \overline{RCT}_i(t_c+P+1)) \\
& = m \times P
\end{aligned}$$

The remainder of the proof relies on the behaviour of tasks in PT. To adapt to the partial idle time case the proof of the complete idle slot case, we first prove that a task in PT cannot have performed more computation at time t_c + P + 1 than at time t_c + 1.

Lemma 6 $\forall \tau_i \in PT, RCT_i(t_c+1) + \overline{RCT}_i(t_c+P+1) \leq C_i$.

This lemma can also be expressed as follows: $\forall \tau_i \in PT, \overline{RCT}_i(t_c+P+1) \leq \overline{RCT}_i(t_c+1)$.

4.2.1 Proof of theorem

Assume lemma 6 to be established. The end of the proof is very close to the proof of the complete idle slot case. In a first step we prove that lemma 5 holds in that context to. The proof is quite the same as before: in the computation of the cumulated processed time, the contribution of task τ_i from PT, consists of $\frac{P}{P_i} \times C_i$, which is included in U, and $RCT_i(t_c+1) + \overline{RCT}_i(t_c+P+1) - C_i$, which is less than or equal to 0. The end of the proof is the same as in the previous case.

We then have:

$$\begin{aligned}
m \times P = & W(t_c+1, t_c+P+1) \\
= & \sum_{\tau_i \in Re} \frac{P}{P_i} \times C_i \\
+ & \sum_{\tau_i \in PT} (\frac{P}{P_i} \times C_i + [RCT_i(t_c+1) + \overline{RCT}_i(t_c+P+1) - C_i]) \\
+ & \sum_{\tau_i \in LR} [(\frac{P}{P_i} - 1) \times C_i + \overline{RCT}_i(t_c+P+1)] \\
+ & \sum_{\tau_i \in AR} [(\frac{P}{P_i} - 1) \times C_i + \overline{RCT}_i(t_c+P+1)]
\end{aligned}$$

$= m \times P$

$+ \sum_{\tau_i \in PT} (RCT_i(t_c+1) + \overline{RCT}_i(t_c+P+1) - C_i)$

$+ \sum_{\tau_i \in LR} (\overline{RCT}_i(t_c+P+1) - C_i)$

$+ \sum_{\tau_i \in AR} (\overline{RCT}_i(t_c+P+1) - C_i)$

It follows that

$\sum_{\tau_i \in PT} (RCT_i(t_c+1) + \overline{RCT}_i(t_c+P+1) - C_i)$

$+ \sum_{\tau_i \in LR} (\overline{RCT}_i(t_c+P+1) - C_i)$

$+ \sum_{\tau_i \in AR} (\overline{RCT}_i(t_c+P+1) - C_i)$

$= 0$

Now we have:

- $\forall \tau_i \in PT$, $RCT_i(t_c+1) + \overline{RCT}_i(t_c+P+1) - C_i \leq 0$ (from lemma 6).
- $\forall \tau_i \in LR \cup AR$, $\overline{RCT}_i(t_c+P+1) - C_i \leq 0$: an instance cannot process more than the global execution time.

Thus, each term of the previous sum is negative and the sum equals to 0. It follows that

- $\forall \tau_i \in PT$, $RCT_i(t_c+1) + \overline{RCT}_i(t_c+P+1) - C_i = 0$
- $\forall \tau_i \in LR \cup AR$, $\overline{RCT}_i(t_c+P+1) - C_i = 0$.

And we conclude as in the case of a complete idle slot.□

4.2.2 Proof of the lemma

The key lemma (lemma 6) must now be proven. Let τ_i be any task in set PT. We must consider three cases:

1 - An instance of τ_i is released at time t_c + 1.

Then $RCT_i(t_c+1) = C_i$, t_c + P +1 is also a release time for τ_i, so $\overline{RCT}_i(t_c+P+1)$ = 0. Thus the lemma holds.

2 - $t_c + 1$ is not a release time for τ_i, and $PI_i(t_c+1)$ has continuously been processed from its release.

Let $r_i + k_c \times P_i$ be its release date. Then $\overline{RCT}_i(t_c+1)$ = t_c + 1 - (r_i + $k_c \times P_i$). And we obviously have $\overline{RCT}_i(t_c+P+1) \leq$ (t_c + P + 1) - (r_i + P + $k_c \times P_i$). Thus, $\overline{RCT}_i(t_c+P+1) \leq \overline{RCT}_i(t_c+1)$. So the lemma.

3 - $t_c + 1$ is not a release time for τ_i, and τ_i has not been continuously processed from its released, but has been preempted at some points of time, by some tasks with highest priorities.

In order to deal with that case, we introduce some further definitions.

Definition 7 *(1) A time t is called a* **preemption point** *if there exists at least one pending task which is not processed during slot t.*

(2) Let t be a premption point.

Its **preemption context** *is the tuple Ctx(t) = ($\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_m}$, list) such that:*

- $\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_m}$ *are processed at time t.*
- *list is the set of the pending but not processed tasks at time t.*

According ot the definition, if t is a preemption point and Ctx(t) its context, Ctx(t).list is non empty. Furthermore, if τ_k is a task in Ctx(t).list, tasks τ_{i_j}, for j =1, .., m have higher priority than task τ_k.
Let $(tp_1, tp_2, \ldots, tp_f)$ denotes the increasing sequence of preemption points which occur before t_c. Notice that, since t_c is an idle slot, it cannot be a preemption point. We now prove that a pending task which is not processed at time t can neither be processed at time t + P.

Lemma 8 *Let tp be a preemption point and τ_k a task in Ctx(tp).list. Then τ_k is not processed at time tp + P, i.e. O_k(tp + P) = 0.*

PROOF. We prove the lemma by induction on the sequence of preemption points.
I - Let tp1 be the very first preemption point tp_1, and $(\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_m}, list_1)$ its context. Let τ_k be any task in $list_1$. We consider the next three cases:

(1) $\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_m}$ have been released before tp_1. They have been continuously processed (because they havec never been preempted), they have highest priorities than τ_k, thus, since tp_1 is the first preemption point, $PI_k(tp_1)$ is released precisely at tp_1. Besides, we have $\overline{RCT_{i_j}}(tp_1) \geq \overline{RCT_{i_j}}(tp_1 + P)$ (because of the continuous processing of τ_{i_j}). Furthermore, since it is processed at time tp_1, τ_{i_j} does not have completed execution at time tp_1, so it has neither complete execution at time $tp_1 + P$. Thus tasks τ_{i_j} (j = 1 . . . m) are still pending at time $tp_1 + P$, they have priority over τ_k, thus τ_k cannot be processed at time $tp_1 + P$.
(2) Assume that some of the tasks τ_{i_j} are released before tp_1 and some other at tp_1. Tasks released at tp_1 are also released at $tp_1 + P$. Let τ_{i_u} be a task released before tp_1. It has been processed continuously from release time up to tp_1, so we have $\overline{RCT_{i_u}}(tp_1)$(which is maximal) $\geq \overline{RCT_{i_u}}(tp_1 + P)$. Thus here again, every tasks τ_{i_j} (j = 1..m) are still pending at time $tp_1 + P$, and we conclude as for the previous case.
(3) Every tasks of the context of tp_1 are released at time tp_1. They are thus also released at time $tp_1 + P$, and we conclude again in the same way.

II - Assume the lemma to hold for i = 1. . . s. We first prove the following corollary:

Corollary 9 *Assume that, for any preemption point tp $\in \{tp_1, tp_2, \ldots, tp_s\}$, and for any task τ_k in Ctx(tp).list, $O_k(tp{+}P) = 0$ (i.e. lemma 8 holds for $\{tp_1, tp_2, \ldots, tp_s\}$). Then we have:*
A - $\forall tp_r$ *($r \leq s$),* $\forall \tau_i$ *such that* $r_i \leq tp_r + 1$, $\overline{RCT_i}(tp_r + 1) \geq \overline{RCT_i}(tp_r + P + 1)$
B - $\forall t < tp_{s+1}$, $\forall \tau_i$ *such that* $r_i \leq t + 1$, $\overline{RCT_i}(t + 1) \geq \overline{RCT_i}(t + P + 1)$

Proof of the corollary
A - By assuption, each time τ_i is pending but idle at a time tp_u, it is also idle one hyperperiod later. So τ_i is idle at least as often between the release of $PI_i(tp_r+P+1)$ and tp_r+P+1 as between the release of $PI_i(tp_r+1)$ and tp_r+1. It follows that $\overline{RCT_i}(tp_r+1) \geq \overline{RCT_i}(tp_r+P+1)$.

B - We note tr the release date of $PI_i(t)$.

1. If $t+1$ is a release date for τ_i, we have $\overline{RCT_i}(t+1) = 0 = \overline{RCT_i}(t+P+1)$
2. If $t+1$ is not a release date for τ_i and τ_i has been continuously processed from tr up to $t+1$, then $\overline{RCT_i}(t+1)$ is maximal, thus $\overline{RCT_i}(t+1) \geq \overline{RCT_i}(t+P+1)$.
3. Else τ_i has been preempted at some points of time. Let tp_u be the last preemption point before t ($tp_u \leq$ t).
   (a) If $t = tp_u$ then we apply point I.
   (b) If $t \neq tp_u$ (t is thus not a preemption point) and if $PI_i(t+1)$ completes at or before $t+1$, $\overline{RCT_i}(t+1) = C_i \geq \overline{RCT_i}(t+P+1)$.
   (c) In the other cases, we have $tr \leq tp_u$ (else, τ_i would have been continuously processed). From point I, we have $\overline{RCT_i}(tp_u+1) \geq \overline{RCT_i}(tp_u+P+1)$. Furthermore, $PI_i(t+1)$ is continuously processed from tp_u+1 up to $t+1$. Thus $W_i(tp_u+1, t+1) = t-tp_u \geq W_i(tp_u+P+1, t+P+1)$. We then deduce the result. □

III - Consider the $(s+1)^{th}$ preemption point tp_{s+1} and its context ($\tau_{j_1}, \tau_{j_2}, \ldots, \tau_{j_m}$, $list_{s+1}$). We use point B of corollary 9 with t = tp_{s+1} - 1, and a task τ_{j_k} of the context of tp_{s+1}. We thus have $\overline{RCT_{j_k}}(tp_{s+1}) \geq \overline{RCT_{j_k}}(tp_{s+1}+P)$. As before, we conclude that none of the tasks of the context of tp_{s+1} can have completed execution at time $tp_{s+1}+P$, so τ_k which has a lower priority cannot be processed at that time. □

We can now complete the proof of lemma 6. It corresponds to the point II of the corollary, with s = f and k = i (τ_i being the task of PT under consideration). □

5 Not fully loaded processors

We now get rid of the fully loaded processors assumption, and consider the case $U < m$. There are $P \times (m-U)$ cyclic processing time units each hyperperiod P. Let k be equal to $P \times (m-U)$. We introduce further tasks $\{\tau_{n+1}, \tau_{n+2}, \ldots, \tau_{n+k}\}$, called idle tasks, with the following temporal parameters: $r_{n+i} = 0, C_{n+i} = 1, D_{n+i} = P_{n+i} = P$. These tasks support processors cyclic idleness. Then we assume task τ_{n+i} (with $1 \leq i \leq k$) to have priority level i and the tasks τ_i with $i \leq n$ to have priority levels between $k+1$ and $k+n$. Thus, the idle tasks have the lowest priorities, according to the work conserving property of fixed-priority scheduling. Then we can apply the result of the fully loaded processors section, and the result

still holds.

6 Extensions and perspectives

Future works must address two issues. First, the results we got for fixed-priority scheduling strategies must be enlarged to other strategies. And then, it would be helpful to determine an upper bound for the date of the beginning of the steady state.

6.1 Other scheduling strategies

The proof in the complete idle slot case does not depend on the chosen scheduling strategies. Thus it can be extended to any deterministic and conservative strategies. Only the partial idle slot case uses the fixed priority assumption. And more precisely, it is used for proving lemma 8, which in turn was used to prove that a task already released at time t can never have processed more processing time units at time t + P than at time t. We now examine the case of Earliest Deadline First and Least Laxity First.

6.1.1 Earliest Deadline First

We first consider the earliest deadline first strategy: the processed tasks are those with the nearest deadlines [18]. If we focus on our proof, we can note that it is not necessary to link priorities to the tasks, but only to instances, provided the following property holds (which we call monotony property): for any two tasks, say τ_1 and τ_2, if PI_1(t) have an higher priority than PI_2(t), then PI_1(t + P) also have an higher priority than PI_2(t + P). This property holds for ED, so does our result. Furthermore, our result holds for any job-level dynamic priorities strategy [8] which obey the monotony property.

6.1.2 Least Laxity

Here, the running tasks are those with the lowest laxities, were the laxity is defined by Lax(τ_i, t) = deadline of PI_i(t) - RCT_i(t) [20]. We prove directly that a task cannot have processed more execution time units at time t + P than at time t (t being greater than its release time). That is to say, we prove that:

$$\forall \text{ t}, \forall \tau_i \text{ , t } \geq r_i \Rightarrow \overline{RCT}_i(t) \geq \overline{RCT}_i(t+P)$$

Assume this result not to hold. We then define t_0 as the smallest t such that there exists some task τ_i such that $\overline{RCT}_i$(t) < $\overline{RCT}_i$(t + P) (with t $\geq r_i$).
Let τ_{i_0} be such a task. We have

$$\overline{RCT}_{i_0}(t_0) < \overline{RCT}_{i_0}(t_0+P) \quad (1)$$

$$\overline{RCT}_{i_0}(t_0-1) = \overline{RCT}_{i_0}(t_0+P-1) \quad (2)$$

We deduce from (1) that t_0 is not a release time for τ_{i_0} (else, we would have $\overline{RCT}_{i_0}(t_0)$ = $\overline{RCT}_{i_0}(t_0$ + P) = 0) and τ_0 has been processed at time t_0 + P - 1, but not at time t_0 - 1. We also deduce that τ_0 is not completed at time t_0 - 1. Thus, because of conservatism, there are m tasks, says $\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_m}$ that are processed at time t_0 - 1. Following the definition of t_0, we have

$$\overline{RCT}_{i_j}(t_0 - 1) \geq \overline{RCT}_{i_j}(t_0 + P - 1) \tag{3}$$

And according to the LLF strategy, we have

$$Lax(\tau_{i_j}, t_0) \leq Lax(\tau_{i_0}, t_0) \tag{4}$$

Besides, from (2) we get $Lax(\tau_{i_0}, t_0 + P - 1) = Lax(\tau_{i_0}, t_0) + P - 1$. From (3) we get $Lax(\tau_{i_j}, t_0 + P - 1) \leq Lax(\tau_{i_j}, t_0) + P - 1$. Combined with (4) it gives $Lax(\tau_{i_j}, t_0 + P - 1) \leq Lax(\tau_{i_0}, t_0 + P - 1)$. Then, thanks to determinism, tasks $\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_m}$ have again priority over τ_0 at time t_0 + P. Thus τ_0 cannot have been processed at time t_0 + P. So the contradiction. Thus, theorem 4 also holds for LLF schedules.

6.1.3 Further investigations

Considering the cases where the theorem holds, and the example of a scheduling strategy for which it does not, we can underline the differences. In case of fixed-priority assignments, of EDF and of LLF, one just have to know their states to decide which one of two tasks has priority over the other. We call this property "local determinism". In the case of the example of figure 4, deciding which of two tasks has priority required the knowledge of the state of the whole system. We conjecture that our result holds for all local deterministic scheduling strategies.

6.2 An upper bound for t_c ?

In the uniprocessor case, we have proven that t_c is less than r + P [9]. For multi-processor case, we have surprising results. Indeed, this bound holds neither for ED nor for LL. The two next examples give evidence of the late occurrence of the last acyclic idle slot.

We first consider system S2 = <τ_1 (5, 6, 11, 11), τ_2 (0, 6, 11, 11), τ_3 (0, 6, 11, 11), τ_4 (3, 4, 11, 11)>. We compute the EDF schedule on the time interval [0, 66], for 2 processors P1 and P2. It is given figure 6.

We note that the system is in the same state at times 55 and 66, there is an idle slot at time 54 followed by 11 time units without idle slot, so the schedule is cyclic only from time 55. And we have 54 = r + 4P + 5.

If we consider the system S3 = <τ_1 (225, 90, 161, 161), τ_2 (115, 40, 161, 161), τ_3 (0, 72, 161, 161), τ_4 (129, 120, 161, 161)>, again schedulted by EDF, we find the last acyclic idle slot at time 7037 = r + 42P + 50.

So the last acyclic idle slot can be very delayed. And it seems to be difficult to find

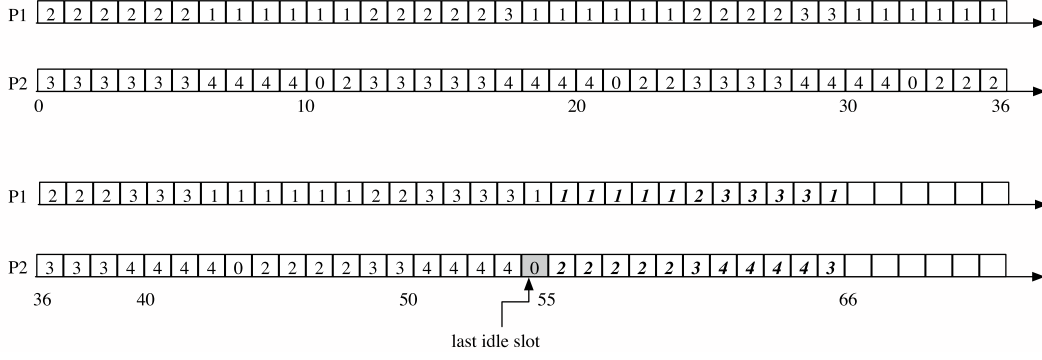

Fig. 6. Schedule O1. The last idle slot occurs at t = 54 = r +4P + 5.

an interesting upper bound in the general case.

As to LLF, we have the same problem. Consider system S4 = <τ_1 (5, 4, 11, 11), τ_2 (0, 6, 11, 11), τ_3 (4, 6, 11, 11), τ_4 (3, 6, 11, 11)>. For the sake of determinism, we adopt the following static rule when we have to chose among several tasks having the same laxity: we chose the tasks with the least numbers. We get the schedule O3 (figure 7). We have $ST_{S3}(25) = ST_{S3}(36)$ and $ST_{S3}(24) \neq ST_{S3}(35)$. So the

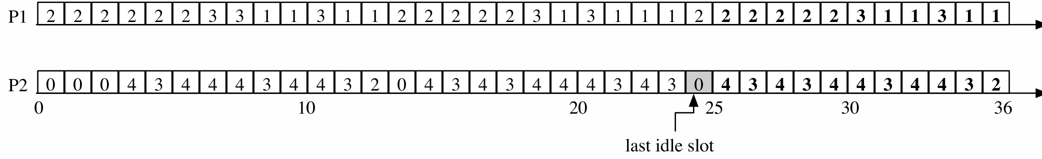

Fig. 7. Schedule O3. The last idle slot occurs at t = 24 = r + P + 9.

schedule is cyclic from 25.

Another point must be underlined: it is not true that there exists a feasible schedule if and only if there exists a feasible schedule on the time interval [0, r + P]. As an example, consider the system S5 = <τ_1 (8, 3, 16, 16), τ_2 (0, 3, 4, 4), τ_3 (2, 9, 16, 16), τ_4 (4, 1, 2, 2)>. The LLF-schedule in the interval [0, 24] is feasible. Nevertheless a temporal fault occurs at time 66 = R + 3P + 10.

As to the fixed-priority case, things seems to be different: we made some experimentations, using a configurations generator, and we always find the last acyclic idle slot before r + P. So we conjecture this result still holds for fixed priorities.
Then simulating the system for P slots after the last acyclic idle slot still provides an exact schedulability test, which can be helpful when sufficient conditions do not hold. This is e.g. the case when $U = m$ (see figure 3) for fixed-priority scheduling. And if our conjecture is true, the simulation won't exceed $r + 2 \times P$ slots.

7 Conclusion

Because it is mandatory for off line scheduling or for validation by simulation, we have investigated the cyclicity problem for multiprocessor systems. We have first shown that neither the approach developed in the uniprocessor context nor the results got in this context can be extended to the multiprocessor context. We have then considered independent tasks systems, and fixed-priority scheduling strategies. We have derived from the observation of the processors idleness a characterization of the beginning of the steady state: it corresponds to the beginning of the first interval of size P without acyclic idle slot. This result has also been proven for EDF and LLF. But we also have shown that some deterministic and conservative strategies do not meet this characterisation. We have conjectured that our result holds for any locally deterministic and conservative scheduling strategy. We also have wondered whether there exists an upper bound for the steady state beginning. In uniprocessor context, this upper bound had been shown to be equal to $r+P$. For the sake of multiprocessor context, we made some experimentations: in the fixed-priority case, the steady state always started before $r + P$, but for EDF and LLF, it could start later. For the time, we don't have any interesting conjecture, since this date can be very late compared to $r+P$. This will be one of our future investigation fields.

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] B. Andersson and J. Jonsson. Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition. In *Proceedings of the conference on Real-Time Computing Systems and Applications*, pages 337–346, December 2000.

[3] T.P. Baker. Multiprocessor edf and deadline monotonic schedulability analysis. In *Proceedings of the* $24^t h$ *IEEE Real-Time Systems Symposium*, 2003.

[4] T.P. Baker. An analysis of edf schedulability on a multiprocessor. *Transactions on Parallel and Distributed Systems*, 16(8):760–768, 2005.

[5] T.P. Baker. An analysis of fixed-priority schedulability on a multiprocessor. *The Journal of Real-Time Systems*, 32:49–71, 2006.

[6] 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.

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

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

[9] A. Choquet-Geniet and E. Grolleau. Minimal schedulability interval for real time systems of periodic tasks with offsets. *Theoretical of Computer Science*, pages 117–134, 2004.

[10] 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.

[11] M.R. Garey and D.S. Johnson. Schedulling tasks with nonuniform deadlines on two processors. *Journal of the Association for Computing Machinery*, 23(3):461–467, 1976.

[12] J. Goossens, S. Funk, and A. Baruah. Priority-driven scheduling of periodic tasks systems on multiprocessors. *The journal of Real Time Systems*, 25:187–205, 2003.

[13] R. Graham. Bounds on the performance of scheduling algorithms. In M. Frappier and H. Habrias, editors, *Computer and job shop scheduling theory*. John Wiley and Sons, 1976.

[14] 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.

[15] R.M. Karp and V. Ramchandani. Parallel algorithms for shared-memory machines. In J.V. Leuwen, editor, *Algorithms and complexity*, pages 869–935. MIT press, 1990.

[16] G. Largeteau, D. Geniet, and E. Andres. Discrete geometry applied in hard real-time systems validation. In *Proc. of* 12th *Discrete Geometry for Computer Imagery*, LNCS 3429, pages 23–33. Springer Verlag, 2005.

[17] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, pages 237–250, 1982.

[18] 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.

[19] J. Liu and R. Ha. Efficient methods of validating timing constraints. *Advances in Real-Time Systems*, pages 199–223, 1995.

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

[21] R.R Muntz and E.G. Coffman jr. Preemptive scheduling of real-time tasks on multiprocessor systems. *Journal of the association for computing Machinery*, 17(2):324–338, 1970.

[22] K. Ramamritham, J.A. Stankovic, and P. Shiah. O(n) scheduling algorithms for real-time multiprocessor systems. *International Conference on Parallel Processing*, 3:143–153, 1989.

[23] J.A. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo. Implications of classical scheduling results for real time systems. *IEEE Computer*, 28(9):16–25, 1995.

[24] J. Xu. Multiprocessor scheduling of processes with release times, deadlines, precedence and exclusion relations. *IEEE Transactions on Software Engineering*, 19(2):139–153, 1993.