Negative results for scheduling independent hard real-time tasks with self-suspensions

Frédéric Ridouard, Pascal Richard, Francis Cottet
LISI-ENSMA
Av C. Ader, Téléport 2 BP 40109
86961 Futuroscope Cedex, France
{frederic.ridouard,pascal.richard,francis.cottet}@ensma.fr

Abstract

In most real-time systems, tasks use remote operations that are executed upon dedicated processors. External operations introduce self-suspension delays in the behavior of tasks. This paper presents several negative results concerning scheduling independent hard real-time tasks with self-suspensions. Our main objective is to show that well-known scheduling policies such as fixed-priority or Earliest Deadline First are not efficient to schedule such task systems. We prove the scheduling problem to be $\mathcal{NP}$-hard in the strong sense, even for synchronous task systems with implicit deadlines. We also show that scheduling anomalies can occur at run-time: reducing the execution requirement or the suspension delay of a task can lead the task system to be infeasible under EDF. Lastly, we present negative results on the worst-case performances of well-known scheduling algorithms (EDF, RM, DM, LLF, SRPTF) to maximize tasks completed by their deadlines.

1 Introduction

Efficient real-time systems exploit the power of dedicated processors. Tasks prepare specific computations such as signal processing (e.g., *FFT*) and then wait until these external operations complete. When a task invokes an external operation, that task is suspended by the real-time kernel and the scheduler chooses the next task ready to run according to an on-line scheduling policy. The execution requirement of a remote operation invoked by a task can be modeled as a self-suspension delay. Next, we consider real-time scheduling of independent tasks with self-suspension allowed upon a uniprocessor system. Let $\tau_{i,1}$ and $\tau_{i,2}$ be two parts of a task τ_i separated by a self-suspension delay. Self-suspensions are modeled differently according to the scheduling environment (*time-driven* or *priority-driven* scheduling policies). In a time-driven system, a self-suspension can be modeled as a time-lag between the end of a subtask $\tau_{i,1}$ and the start time of $\tau_{i,2}$. In this former approach, the maximum self-suspension delay is enforced as a hard timing constraint between the end of $\tau_{i,1}$ and the starting time of $\tau_{i,2}$. Nevertheless, self-suspension delays change from one execution to another since they model execution requirements of external operations. As a consequence, time-lags modeling external operations cannot be assumed to be constant in a priority-driven system. At run-time, the pending task is resumed when an external operation completes. Thus, self-suspension delays can-

not be modeled as time-lags associated to precedence constraints in the on-line setting.

Several feasibility tests are known for analysing tasks allowed to self-suspend. In [3] is presented a test based on the utilization factor of the processor. For fixed-priority task systems, tests are based on the computation of worst-case response time of tasks [5, 9, 8]. Such an approach can also be used for *EDF* scheduling [9]. But, to the best of our knowledge few have been published on the efficiency of classical scheduling priority-driven policies for dealing with tasks allowed to self-suspend.

We next show that well-known on-line scheduling algorithms are not efficient to schedule tasks with self-suspensions. Section 2 defines formally task systems with self-suspensions considered in the remainder. We first show in Section 3 that there exists neither optimal polynomial time, nor pseudo-polynomial time, scheduling algorithm. Furthermore, we show that if there exists an universal scheduling algorithm for tasks with at most one self-suspension per task then $\mathcal{P} = \mathcal{NP}$. We also present scheduling anomalies occurring while scheduling tasks with self-suspensions under EDF. To the best of our knowledge, this is the first time that such anomalies are exhibited for scheduling independent tasks upon an uniprocessor system. In Section 4, we show that classical scheduling algorithms fail to schedule task systems having arbitrary small utilization factors whereas there exist trivial off-line feasible schedules. Lastly, using resource augmentation technique (for instance see [10]), we show that there is no competitive on-line scheduling algorithm using a k-speed processor against an off-line scheduler run under a unit-speed processor.

2 Tasks with Self-Suspensions

Real-time softwares are usually based on a collection of *recurring tasks*. Every task τ_i, $1 \leq i \leq n$ has an upper limit to its execution requirement C_i (worst-case execution time), a relative deadline D_i to its release date and a period T_i. If $D_i = T_i$ for a task τ_i, then the task has an *implicit deadline*, if $D_i \leq T_i$ then it has a *constrained deadline*. Every occurrence of a task is called a *job*. We assume next that tasks can be preempted at any time and resumed later without any incurring costs (no overhead). The utilization factor of a periodic task τ_i, is the ratio of its execution requirement to its period: $U(\tau_i) = C_i/T_i$. The utilization factor of a task system τ is the sum of the utilization factors of all tasks: $U(\tau) = \sum_{i=1}^{n} U(\tau_i)$. A task set is said *feasible* if there exists a schedule such that all tasks are completed by their deadlines at run-time. Classical on-line schedulers use priority rules such as Rate Monotonic - (*RM*) and Deadline Monotonic - (*DM*), Earliest Deadline First (*EDF*) and Least laxity First (*LLF*) policies.

Tasks are scheduled on a single processor whereas external operations that they perform are executed on remote dedicated processors. We study the scheduling of periodic tasks having at most one self-suspension each. We limit ourselves to this simple case to simplify the presentation of our results.

Definition 1 *A task τ_i with a self-suspension is a task defined by two subtasks ($\tau_{i,1}$ and $\tau_{i,2}$) separated by a maximum self-suspension delay between the completion of the first subtask and the start of the second subtask. A task τ_i, $1 \leq i \leq n$ has the following sequence at run-time:*

- *an input subtask $\tau_{i,1}$ having an execution requirement of at most $C_{i,1}$,*
- *a suspension delays modeling an external operation with a length of at most $X_i \geq 0$,*
- *an output subtask $\tau_{i,2}$ having an execution requirement of at most $C_{i,2}$.*

If a task τ_i has no self-suspension (i.e, $X_i = 0$), then its subtasks are merged as a single one with an execution requirement $C_i = C_{i,1} + C_{i,2}$. A task system is a collection of independent tasks with self-suspensions.

3 Complexity of the Run-Time Scheduling Problem

We next show that the feasibility problem of scheduling synchronously released tasks, with implicit deadlines having at most one self-suspension each, is $\mathcal{NP}$-hard in the strong sense. We also prove that scheduling anomalies can occur under *EDF*.

3.1 Computational Complexity

In [11], we proved the feasibility problem of scheduling synchronous periodic task systems to be $\mathcal{NP}$-hard in the strong sense when tasks are allowed to self-suspend and have constrained deadlines. In this previous paper, we left open the case of tasks having at most one self-suspension and *implicit-deadlines* (the deadline is equal to the period for every task). Please notice that in this particular case, the feasibility problem of scheduling tasks when self-suspensions are not allowed is solved in $O(n)$ by checking that the utilization factor of the processor satisfies $U \leq 1$. Theorem 1 establishes that the feasibility problem of scheduling tasks with self-suspensions is $\mathcal{NP}$-hard in the strong sense, even in this restrictive case.

Theorem 1 *The feasibility problem of scheduling periodic tasks with at most one self-suspension per task and implicit deadlines is $\mathcal{NP}$-hard in the strong sense.*

Proof: We shall transform from 3-Partition, known to be strongly $\mathcal{NP}$-Complete.
Instance: Set A of $3m$ elements, a bound $B \in N$, and a size $s_j \in N$ for each $j = 1..3m$ such that $B/4 < s_j < B/2$ and such that $\sum_{j=1..3m} s_j = mB$.
Question: Can A be partitioned into m disjoint sets $A_1, A_2, ..., A_m$ such that, for $1 \leq i \leq m$, $\sum_{j \in A_i} s_j = B$ (each A_i must therefore contain exactly three elements from A)?
For every 3-Partition instance we define an instance of the scheduling problem with $3m + 1$ tasks:

- the tasks $\tau_1, ..., \tau_{3m}$:

$$1 \leq i \leq 3m \qquad \begin{cases} C_{i,1} &= C_{i,2} = s_i \\ X_i &= (2m-1)B \\ D_i &= T_i = 4mB \end{cases}$$

- a task τ_{3m+1} with:

$$\begin{aligned} C_{3m+1,1} &= \left\lceil \frac{B}{2} \right\rceil \qquad C_{3m+1,2} = \left\lfloor \frac{B}{2} \right\rfloor \\ X_{3m+1} &= B \\ D_{3m+1} &= T_{3m+1} = 2B \end{aligned}$$

We can now prove that we have a solution to the 3-Partition instance if, and only if, there is a feasible schedule for the previously defined task system with self-suspensions. The hyperperiod of the task set is $H = lcm(T_1, \ldots, T_{3m+1})$. It is easy to show that its utilization factor is exactly 1:

- the workload generated by the first set of tasks within the hyperperiod is:

$$\sum_{i=1}^{3m} (C_{i,1} + C_{i,2}) = 2 \sum_{i=1}^{3m} s_i = 2mB$$

- the workload generated by the task τ_{3m+1} within the hyperperiod is:

$$\frac{4mB}{2B} \left(\left\lceil \frac{B}{2} \right\rceil + \left\lfloor \frac{B}{2} \right\rfloor \right) = 2mB$$

Hence, the workload of the task set is $4mB$ within the hyperperiod having exactly the same length. Thus, the utilization factor of the previously defined task set is exactly 1. As a consequence, there is no idle-time in any feasible schedule. The task τ_{3m+1} has no laxity in every feasible schedule. Thus, its execution leaves idle-blocks of length B in the schedule that are separated by the execution of the last subtask of τ_{3m+1} and the first subtask of the next job of τ_{3m+1}. Except for the first job of τ_{3m+1}, the execution of τ_{3m+1} starts its execution for B units of times and then leaves an idle-block of length B in every feasible schedule (Figure 1

presents the pattern of every feasible schedule; $\uparrow$ is a release date and $\downarrow$ a deadline). A *block* is such an interval left idle by the execution of τ_{3m+1}. In every feasible schedule, the kth block is defined as follows:

$$\left[2(k-1)B + \left\lceil \frac{B}{2} \right\rceil ; 2kB - \left\lfloor \frac{B}{2} \right\rfloor \right) \qquad \forall k \geq 1$$

(If Part) Consider a 3-Partition of A, then we can define a feasible schedule as follows. We first consider the subset A_1, that contains exactly 3 elements and has a size B. We schedule the first subtask of the corresponding tasks in the first block and the second one into the block $m+1$. The end of the first subtask and the start of the second one are separated by an interval of time of length $(2m-1)B$. Thus, suspension delays are respected. The same principle is used to sequence tasks corresponding to elements in A_2, in subsequent blocks $(2, m+2)$; and so on. This method leads to a feasible schedule.

(Only if Part) Assume that we have a feasible non-preemptive schedule. We shall consider the case of preemptive schedules later. As a consequence, tasks having their first subtasks in the first block of the schedule cannot have their second subtasks in the subsequent $m-1$ blocks. Since the utilization factor of the task system is 1, then these second subtasks can only be scheduled in the block $m+1$, otherwise we necessarily introduce an idle time in this block. Furthermore, every block has 3 subtasks since their execution requirements verify $B/4 < C_{i,j} < B/2, i = 1..3m, j = 1,2$. According to these facts, we can set elements corresponding to tasks in the ith block into the subset $A_i, 1 \leq i \leq m$, leading to a feasible 3-Partition. We now have to consider preemptive schedules by showing that no subtask can be scheduled in more than one block. We use a contradiction argument. Assume there exists such a subtask that is started in the first block and completed in block 2, for instance. All the other subtasks are started and completed within this block. Then, due to the size of jobs, there is no more than two subtasks completed in block k. As a consequence, in block $k+m$, only two tasks having subtasks completed in block k can be scheduled while respecting the self-suspension delays. As a consequence, there is an idle-time in block $k+m$, that contradicts the fact that the utilization factor is equal to 1. $\Diamond$

We next show that there is no *universal* scheduling algorithm to schedule tasks with self-suspensions, unless $\mathcal{P} = \mathcal{NP}$. Please notice that, a scheduling algorithm is said to be universal if the algorithm takes a polynomial amount of time (in the length of the input) to make each scheduling decision [4].

Theorem 2 *If there exists an universal scheduling algorithm for tasks with at most one self-suspension per task then $\mathcal{P} = \mathcal{NP}$.*

Proof:
To prove this theorem, we use a classical proof approach, such as presented in [4]. Precisely, we show that if a such an algorithm exists, and if it takes a polynomial amount of time (in the length of the input) to choose the next processed job, then $\mathcal{P} = \mathcal{NP}$. Because, one can find a pseudo-polynomial time algorithm to solve the 3-PARTITION problem.
We assume that there exists a scheduling algorithm for scheduling independent periodic tasks with at most one self-suspension upon a uniprocessor system, we denote this algorithm A. From an instance of the 3-PARTITION problem, to define a set I of tasks, we use the same reduction technique as that in the proof of Theorem 1. Since the hyperperiod of the schedule is $4Bm$ and A is assumed to be a polynomial time scheduling algorithm, then the whole algorithm for checking deadline is at most pseudo-polynomial (i.e, it is clearly performed in time proportional to Bm). Thus, the solution delivered by the algorithm A gives a solution to solve the 3-PARTITION problem.

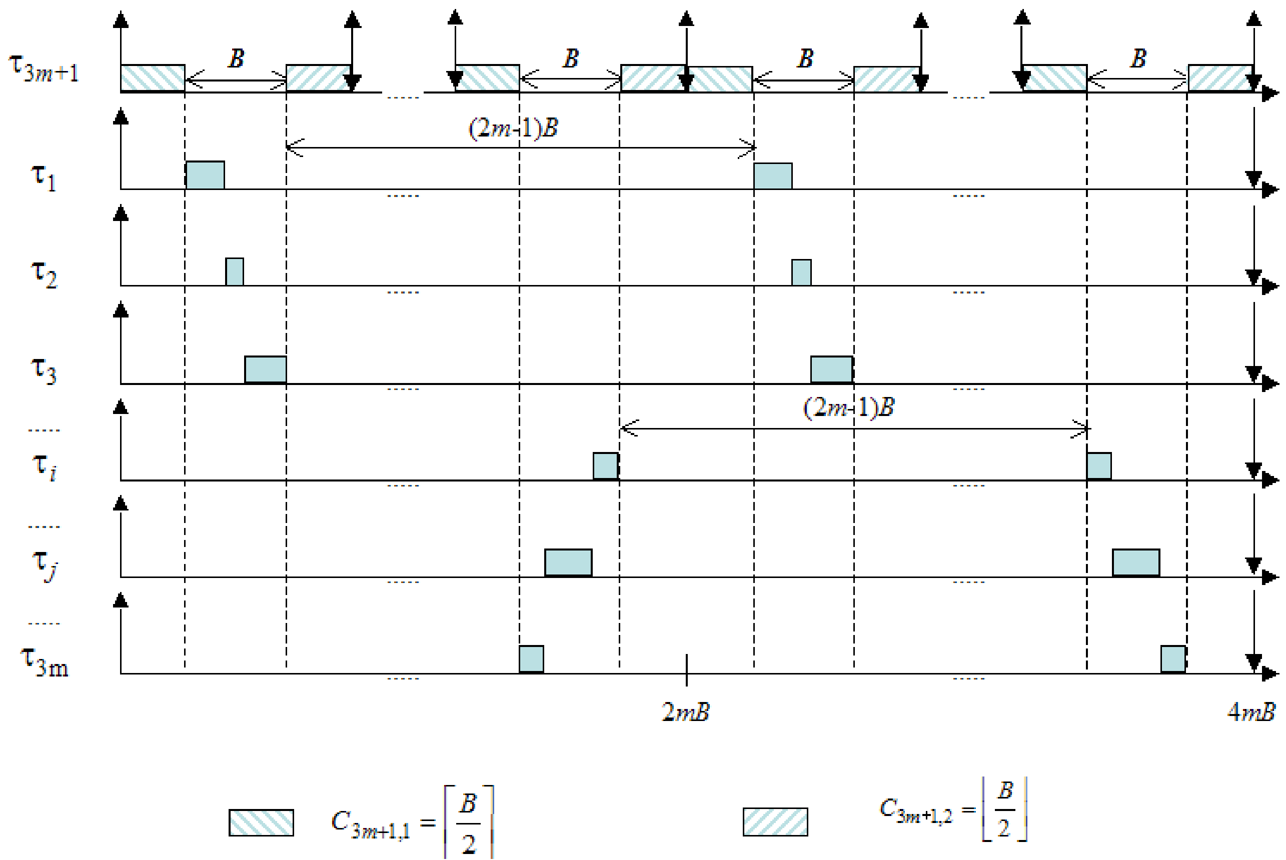

Figure 1. A feasible schedule of the instance (Theorem 1).

Therefore we have found a pseudo-polynomial time algorithm to solve the 3-PARTITION problem. But 3-PARTITION problem is $\mathcal{NP}$-complete in the strong sense. As a consequence, if the algorithm A exists then $\mathcal{P} = \mathcal{NP}$. This is a contradiction. We can then conclude that such an algorithm does not exist. $\diamondsuit$

3.2 Scheduling anomalies under *EDF*

The validation problem is difficult when the scheduling algorithm is priority-driven. Execution requirement of jobs can vary at run-time. An *anomalous behavior* occurs when reducing the execution requirement of a task can lead to miss a deadline whereas the same task system is feasible if all jobs are run with their worst-case execution requirements.

In uniprocessor system, scheduling independent tasks without self-suspension can never lead to scheduling anomalies under *EDF* [6]. Thus, if all deadlines are met while considering the worst-case execution times for all tasks, then reducing the execution requirement of a task cannot lead *EDF* to miss a deadline at run-time. According to this result, considering the worst-case execution requirements of tasks in the feasibility analysis leads to a necessary and sufficient schedulability condition. We prove hereafter that the sufficient part of this result does not hold when tasks are allowed to self-suspend.

Theorem 3 *EDF has anomalies to schedule independent tasks with self-suspensions upon one*

processor.

Proof: To prove this theorem, we define an instance of tasks I and we show that if an execution requirement of a task or a suspension delay are decreased, then a deadline will be missed. The instance I contains three tasks with the following characteristics :

$$\tau_1 : r_1 = 0, D_1 = 6, T_1 = 10,$$
$$C_{1,1} = 2, X_1 = 2, C_{1,2} = 2$$
$$\tau_2 : r_2 = 5, D_2 = 4, T_2 = 10,$$
$$C_{2,1} = 1, X_2 = 1, C_{2,2} = 1$$
$$\tau_3 : r_3 = 7, D_3 = 3, T_3 = 10,$$
$$C_{3,1} = 1, X_3 = 1, C_{3,2} = 1$$

EDF defines the following schedule when all tasks use their worst-case execution requirements and worst-case suspension delays: at time 0, τ_1 is scheduled and self-suspended at time 2. At time 4, τ_1 is released, immediately scheduled and completed at time 6. Then, at this instant, τ_2 is released, scheduled and self-suspended at time 7. τ_3 is released at time 7 and immediately scheduled. At time 8, τ_3 self-suspends and τ_2 is resumed after its self-suspension and completed by time 9. Lastly, τ_3 is resumed and completed by time 10. Figure 2.a presents the schedule obtained under *EDF*.

Now, we show that $C_{1,1}$, X_1 or $C_{1,2}$ are decreased of one unit of time, then τ_3 is not completed by its deadline. For instance, let us consider that $C_{1,1} = 1$ and all other job requirements are still unchanged then τ_1 is completed by time 5. Then, τ_2 is released and immediately run. At time 7, τ_2 is resumed from its self-suspension and τ_3 is delayed since it has a larger deadline than τ_2. τ_3 starts its execution at time 8 ans is completed by time 11, thus one unit of time after its deadline. The corresponding schedule is presented in Figure 2.b. The same anomaly occurs if X_1 or $C_{1,2}$ are decreased (i.e., $X_1 = 1$ or $C_{1,2} = 1$).

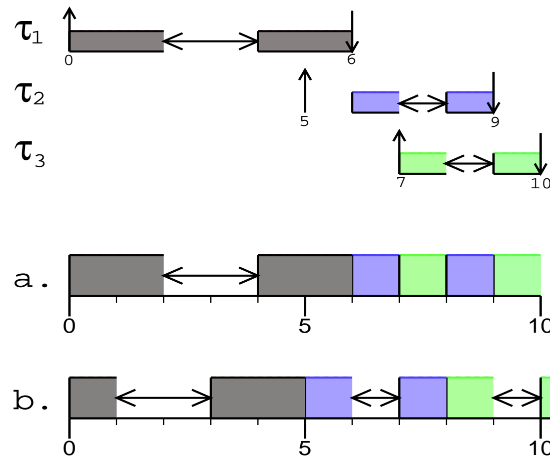

Figure 2. Example of execution-time anomaly for *EDF* by decreasing $C_{1,1}$ of one unit of time

It is easy to show that in all these cases there exist feasible schedules while *EDF* always fails. $\diamondsuit$
According to these results, if a processing time or a self-suspension delay decreases then scheduling anomalies can arise. The previous result can be easily extended to fixed-priority task systems.

4 Competitive Analysis

In this section, in order to simplify the result presentations, we assume that periods are larger enough so that exactly are job of each task belongs to the hyperperiod. We first recall known results on on-line scheduling to maximize tasks completions (maximizing the number of early tasks or equivalently minimizing the number of tardy tasks). We shall use the *competitive analysis* to compare these classical scheduling algorithms against an optimal clairvoyant algorithm (the adversary).

The competitive analysis allows to determine the performance guarantee of an on-line algorithm. This approach compares an on-line algorithm to an optimal *clairvoyant* algorithm: *the adversary*. A good adversary defines instances of problems so that the on-line algorithm achieves its worst-case performance. An algorithm that minimizes

a measure of performance is c-competitive if the value obtained by the on-line algorithm is less than or equal to c times the optimal value obtained by the adversary. We also say that c is the performance guarantee of the on-line algorithm. An algorithm is said to be *competitive* if there exists a constant c so that it is c-competitive. More formally, given an on-line algorithm A, let I be an instance, then, $\sigma_A(I)$ is the value obtained by A and $\sigma^*(I)$ is the value obtained by the optimal clairvoyant algorithm, then A is c-competitive if there exists a constant c so that $\sigma_A(I) \leq c\sigma^*(I)$. The competitive ratio c_A of the algorithm A is the worst-case ratio while considering any instance I: $c_A = \sup_{anyI} \frac{\sigma_A(I)}{\sigma^*(I)}$. The competitive ratio of an algorithm A is greater than or equal to 1. If $c_A = 1$, then A is an optimal algorithm.

There is no competitive algorithms for general preemptive task systems, but competitive algorithms are known for special cases [1, 2]. In the same context, we then prove that if tasks are allowed to self-suspend at most once, then classical on-line scheduling algorithms are not competitive. Note that our results are also valid from the feasibility point of view since we always consider task sets having an arbitrarily small utilization factor such that there exists a feasible schedule whereas classical on-line algorithms miss most of the deadlines. Lastly, we show that using a k-speed processor cannot help to achieve a feasible schedule against a clairvoyant scheduling algorithm using a unit-speed processor. So extra resources is not useful for scheduling tasks with self-suspensions.

4.1 Known results

Baruah et al. [1, 2] proved that there is no competitive on-line preemptive scheduling algorithm to maximize task completions for uniprocessor systems. But, to obtain such a result, the adversary defines a task set under overloaded conditions. But, these authors show that there are also positive results for special cases [1, 2]. We next present one of these special cases that will be used after:

Definition 2 *Monotonic Absolute Deadlines (MAD):*
A task system is said to be MAD if each newly-arrived task will not have a absolute deadline before that of any task that has previously arrived.

We also recall the definition of the SRPTF scheduling rule:

Definition 3 *Shortest Remaining Processing Time First (SRPTF):*
SRPTF is an on-line scheduling algorithm that allocates the processor at any time to the task having the shortest remaining processing time.

In [1, 2] is proved that if the task system has the *MAD* property, then the on-line scheduling algorithm *SRPTF* is 2-competitive to minimize the number of tardy tasks. Furthermore, this rule yields a best possible on-line algorithm.

4.2 Negative results for scheduling tasks with self suspensions

We first prove that SRPTF is no longer competitive to maximize task completions for MAD task sets and self-suspensions allowed.

Theorem 4 *For task systems with arbitrarily small utilization factor, the on-line scheduling algorithm SRPTF is not competitive to maximize the number of early tasks allowed to self-suspend at most once.*

Proof: To demonstrate this theorem, we study the instance I generated by the clairvoyant algorithm. I is an instance of $n + 1$ tasks: τ_0 arrives in the system at the time 0 with only one subtask ($C_{0,1} = 1$, $X_0 = C_{0,2} = 0$) and its deadline is at time $K - 1$ (where K is an arbitrary large number). The other tasks have the following characteristics: for $i \in \{1, \ldots, n\}$, $r_i = i - 1$,

$C_{i,1} = C_{i,2} = 1$, $X_i = K-2$ and $D_i = K$. In Figure 3, we show the outcomes of the scheduling of I by *SRPTF* and by an optimal clairvoyant algorithm. At time 0, τ_0 and τ_1 are available, *SRPTF* schedules τ_0 because τ_0 has the shortest remaining processing time. After, the first subtask of task τ_i is scheduled at time i ($1 \leq i \leq n$) and the second at time $K + (i-1)$. The clairvoyant algorithm schedules τ_1 at time 0 and it schedules every task τ_i ($i \in \{2, \ldots, n\}$) at time $i-1$. To finish, the clairvoyant algorithm schedules τ_0.

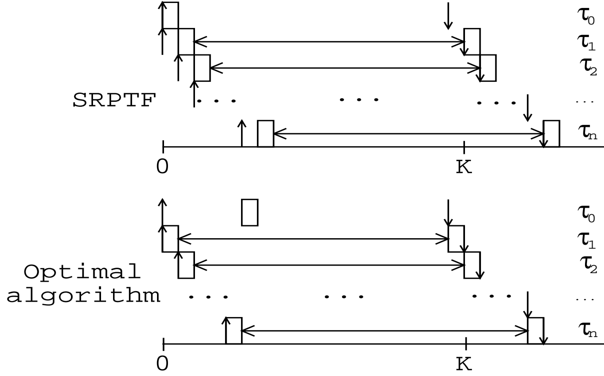

Figure 3. *SRPTF* **is not competitive**

Consequently, the competitive ratio of *SRPTF* is:

$$c_{SRPTF} = \frac{\sigma_{SRPTF}}{\sigma_{Opt}} = \lim_{n \to \infty} \frac{1}{n+1} = 0$$

The factor of utilization is:

$$\begin{aligned} U_I &= \textstyle\sum_{i=0}^{n} \frac{C_{i,1}+C_{i,2}}{T_i} = \frac{1}{K-1} + \sum_{i=1}^{n} \frac{2}{K} \\ &= \lim_{K \to \infty} \frac{2n+1}{K} = 0 \end{aligned}$$

To conclude, we have an instance with an arbitrarily small utilization factor such as *SRPTF* is not competitive to maximize the number of early tasks. $\diamondsuit$

With the instance of the task system used in the proof of the Theorem 4, we can extend the previous result for *EDF*, *DM* and *RM*.

Corollary 1 *For task systems with arbitrarily small utilization factor, the scheduling algorithms EDF, DM and RM are not competitive to maximize the number of early tasks when self-suspensions are allowed.*

Proof: We use the same instance I that in the proof of the Theorem 4. For this instance, *EDF*, *DM* and *RM* assign priorities to the tasks exactly as SRPTF do. Consequently, we obtain the same conclusions for all these scheduling algorithms.$\diamondsuit$

We now consider the *L*east *L*axity *F*irst scheduling algorithm (*LLF*) [7].

Theorem 5 *For task systems with arbitrarily small utilization factor, the scheduling algorithm LLF is not competitive to maximize the number of early tasks when self-suspensions are allowed.*

Proof: To prove this theorem, we study an instance I with n identical tasks. Every task τ_i ($1 \leq i \leq n$) is released at time $r_i = 0$ and if K is a large integer then $C_{i,1} = 3$, $X_i = K - 3(n+1)$, $C_{i,2} = 3$; and its deadline is $D_i = K$. Figure 4 presents the outcomes of the scheduling of I by *LLF* and by an optimal algorithm. At time 0, the first subtask of τ_1 is scheduled by *LLF*. But at time 1, the priorities of the tasks τ_i ($2 \leq i \leq n$) are greater than the priority of τ_1. Consequently, the task τ_2 is scheduled. But at time 2, the others tasks have a priority greater than the priority of τ_2. Therefore, every task τ_i with $1 \leq i \leq n$ always have the same laxity leading *LLF* to preempt the active job after one unit of its execution. The clairvoyant algorithm schedules in the order, the first subtask of $\tau_1, \tau_2, \ldots, \tau_n$ and in the same order the second subtask of these tasks.
Figure 4 presents the outcomes of the scheduling of I by *LLF* and by an optimal algorithm. Consequently, the competitive ratio of *LLF* is:

$$c_{LLF} = \frac{\sigma_{LLF}}{\sigma_{Opt}} = \frac{0}{n} = 0$$

The factor of utilization is:

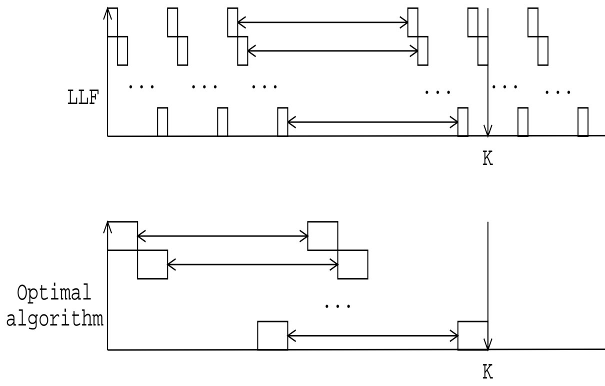

Figure 4. *LLF* is not competitive

$$U_I = \sum_{i=1}^{n} \frac{C_{i,1}+C_{i,2}}{T_i} = \sum_{i=1}^{n} \frac{6}{K} = \lim_{K\to\infty} \frac{6n}{K} = 0$$

To conclude, we have an instance with a processor utilization factor close to zero leading *LLF* to non-competitiveness. Consequently, for any processor utilization factor, *LLF* is not competitive to minimize the number of tardy tasks. $\diamondsuit$

4.3 Resource augmentation

In the competitive analysis, the on-line algorithm and the optimal one use the same processor having a unit speed. A simple way to improve the competitive ratio is to give a faster processor to the on-line algorithm whereas the off-line algorithm is still running on a unit speed processor. This technique is called *resource augmentation*. It has been proved in [10], that *EDF* is still optimal under overloaded conditions if it is run under a two-speed processor while the optimal algorithm is run under a unit speed processor. Thus, if a feasible schedule is determined by a clairvoyant algorithm with a 1-speed processor, then *EDF* will define a feasible schedule under a 2-speed processor.

We next show that when tasks are allowed to self-suspend, then *EDF* cannot define a feasible schedule under a k-speed processor while there exists an off-line feasible schedule under a 1-speed processor (determined by an optimal clairvoyant algorithm). As a consequence, allocating extra resources does not help to define a simple on-line scheduling policy.

Theorem 6 *EDF is not optimal even with a k-speed processor, for any poositive integer k.*

Proof: We use a contradiction argument. Let k be an integer such that $k > 1$ and such that if there exists a feasible schedule under a 1-speed processor then there exists a feasible *EDF* schedule under k-speed processor. Let I be an instance with two tasks. These two tasks of I arrive in the system at time 0 with the following characteristics:

$\tau_1 : C_{1,1} = 2k, X_1 = 0, C_{1,2} = 0, D_1 = 4k - 1$
$\tau_2 : C_{2,1} = 1, X_2 = 4k - 2, C_{2,2} = 1, D_2 = 4k$

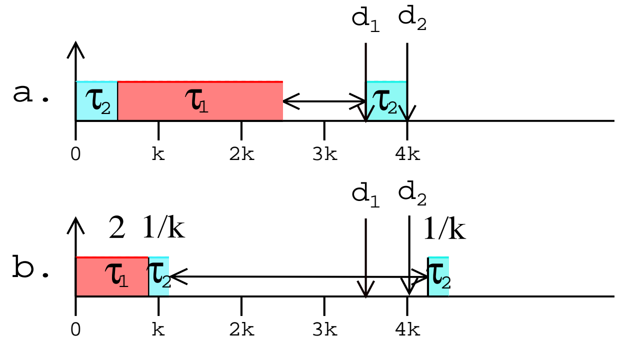

Figure 5. The schedule of I under *EDF*

- At time 0, τ_2 is scheduled and self-suspended at time 1. Then τ_1 is immediately scheduled and completed at time $2k+1$. At time $4k-1$, τ_2 is resumed after its self-suspension. τ_2 is completed by time $4k$. Figure 5.a shows a feasible schedule of I under a 1-speed processor.

- At time 0, τ_1 is scheduled since its absolute deadline is before the deadline of τ_2. Then, at time 2, τ_1 is completed and τ_2 is scheduled and is completed at time $4k + 2/k$. Figure 5.b shows that *EDF* cannot schedule I under a k-speed system.

So, the assumption that there is a feasible schedule under a k-speed processor is false and the theorem is demonstrated for any integer $k > 1$. $\diamondsuit$

This result is not so surprising since when a faster processor is used by the on-line algorithm then no extra resources are given to the processors running remote operations. Thus, the length of external operations are still unchanged (self-suspension delays are not decreased since the modeled external operations are still running on unit speed remote processors).

5 Conclusion

We have presented some negative results to schedule tasks allowed to self-suspend when external operations are executed upon dedicated processors. We have firstly proved that scheduling synchronous tasks having at most one self-suspension and implicit deadlines is a strongly $\mathcal{NP}$-hard problem and there is no universal scheduling algorithm, unless $\mathcal{P} = \mathcal{NP}$. Then, we have shown that under the *EDF* scheduling policy, scheduling anomalies can occur at run-time. Using adversary arguments, we have shown that classical scheduling rules can miss deadlines, even if the utilization factor of the processor is arbitrarily small, whereas an off-line feasible schedule can be easily defined. Lastly, we also have proved that allocating extra resources does not help to schedule tasks with self-suspensions.
In further works, we will try to define practical solutions for scheduling such task systems. An other interesting issue will be to consider non independent tasks.

6 Acknowledgements

The authors wish to express their gratitude to the anonymous reviewers for their helpful comments. We also would like to thank Joël Goossens for his comments on an earlier version of this paper.

References

[1] S. Baruah, J. Haritsa, and N. Sharma. On-line scheduling to maximize task completions. *The Journal of Combinatorial Mathematics and Combinatorial Computing*, 39:65–78, 2001.

[2] S. Baruah, J. Haritsa, and N. Sharma. On-line scheduling to maximize task completions. *In Proceedings of the 15th IEEE Real-TIme Systems Symposium, San Juan, Puerto Rico*, pages 228–237, Dec 1994.

[3] U. C. Devi. An improved schedulability tast for uniprocessor periodic task systems. *proc. Euromicro Conference on Real-Time Systems (ECRTS'03)*, pages 23–30, 2003.

[4] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. *proc. Real-Time Systems Symposium*, pages 129–139, 1991.

[5] I. G. Kim, K. Choi, S. K. Park, D. Y. Kim, and M. P. Hong. Real-time scheduling of tasks that contain the external blocking intervals. *proc. Conference on Real-Time Computing Systems and Applications*, pages 54–59, 1995.

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

[7] A. K. L. Mok. *Fundamental design problems of distributed systems for hard real-time environment*. PhD thesis, MIT, 1983.

[8] J. C. Palencia and M. Gonzalez-Harbour. Schedulability analysis for tasks with static and dynamic offsets. *Proceedings of the 19th Real-Time Systems Symposium, IEEE Computer Society Press*, pages 26–37, December 1998.

[9] J. C. Palencia and M. Gonzalez-Harbour. Response time analysis of edf distributed real-time systems. *www.ctr.unican.es/publications*, December 2003.

[10] C. A. Philips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. *proc.* 29th *Ann. ACM Symp. on Theory of Computing*, pages 110–149, 1997.

[11] P. Richard. On the complexity of scheduling tasks with self-suspensions on one processor. *proc. Euromicro Conference on Real-Time Systems (ECRTS'03)*, pages 187–194, 2003.