Some results on scheduling tasks with self-suspensions

F. Ridouard P. Richard F. Cottet K. Traoré
{frederic.ridouard@ensma.fr} {pascal.richard@ensma.fr} {francis.cottet@ensma.fr} {karim.traore@ensma.fr}
Tel. (+33/0)5 49 49 83 36 Tel. (+33/0)5 49 49 80 69 Tel. (+33/0)5 49 49 80 52 Tel. (+33/0)5 49 49 83 36

Laboratoire d’Informatique Scientifique et Industrielle
École Nationale de Mécanique et d’Aérotechnique
Téléport 2 – BP 40109 F-86961 Chasseneuil Futuroscope Cedex, France
fax : (+33/0)5 49 49 80 64

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 and to minimize the maximum response time of tasks.

**Keywords**: Real-time, On-line Scheduling, Self-suspension, Competitive Analysis.

### 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 \( \tau_i \) 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
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 cannot 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 [5] 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 [7, 9, 12, 11]. Such an approach can also be used for EDF scheduling [12]. 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. This paper summarizes and extends results presented in [15, 16]. 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 $P = NP$. 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 [13]), 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 $\tau_i$, $1 \leq i \leq n$ 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 $\tau_i$, 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 $\tau_i$, is the ratio of its execution requirement to its period: $U(\tau_i) = C_i/T_i$. The utilization factor of a task system $\tau$ 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 preemptive 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 \(\tau_i\) with a self-suspension is a task defined by two subtasks \((\tau_{i,1} \text{ and } \tau_{i,2})\) separated by a maximum self-suspension delay between the completion of the first subtask and the start of the second subtask. A task \(\tau_i, 1 \leq i \leq n\) 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 \(\tau_i\) 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 [14], 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 $\sum_{j=1..3m} s_j = mB$. 

6
Question: Can $A$ be partitioned into $m$ disjoint sets $A_1, A_2, \ldots, A_m$ 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 $\tau_1, \ldots, \tau_{3m}$:

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

- a task $\tau_{3m+1}$ with:

\[
C_{3m+1,1} = \left\lfloor \frac{B}{2} \right\rfloor \quad C_{3m+1,2} = \left\lceil \frac{B}{2} \right\rceil \\
X_{3m+1} &= B \\
D_{3m+1} &= T_{3m+1} = 2B
\]

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 $\tau_{3m+1}$ 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 $\tau_{3m+1}$ 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 $\tau_{3m+1}$ and the first subtask of the next job of $\tau_{3m+1}$. Except for the first job of $\tau_{3m+1}$, the execution of $\tau_{3m+1}$ 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; ↑ is a release date and ↓ a deadline). A block is such an interval left idle by the execution of $\tau_{3m+1}$. In every feasible schedule, the $k^{th}$ block is defined as follows:

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

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.

Conversely, 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 \(i^{th}\) block into the subset \(C_{3m+1,1} = \lfloor B/2 \rfloor\) and \(C_{3m+1,1} = \lceil B/2 \rceil\).

Figure 1: A feasible schedule of the instance (Theorem 1).
$A_i, 1 \leq i \leq m$, 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.

\[ \square \]

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 [6].

**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 [6]. Precisely, we show that if 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.

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.

3.2 Scheduling anomalies under $EDF$

The validation problem is difficult when the scheduling algorithm is priority-driven. The execution requirement of jobs can vary at run-time. An anomalous behavior occurs when reducing the execution requirement of a task leads 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 [8] (anomalies can occur when using EDF in multiprocessors [17] and the anomalies that can occur on a uniprocessor when varying the processor speed with non-preemption or blocking [4]. 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:

\[
\begin{align*}
\tau_1 &: r_1 = 0, D_1 = 6, T_1 = 10, C_{1,1} = 2, X_1 = 2, C_{1,2} = 2 \\
\tau_2 &: r_2 = 5, D_2 = 4, T_2 = 10, C_{2,1} = 1, X_2 = 1, C_{2,2} = 1 \\
\tau_3 &: r_3 = 7, D_3 = 3, T_3 = 10, C_{3,1} = 1, X_3 = 1, C_{3,2} = 1
\end{align*}
\]

**EDF** defines the following schedule when all tasks use their worst-case execution requirements and worst-case suspension delays: at time 0, \( \tau_1 \) is scheduled and self-suspended at time 2. At time 4, \( \tau_1 \) is released, immediately scheduled and completed at time 6. Then, at this instant, \( \tau_2 \) is released, scheduled and self-suspended at time 7. \( \tau_3 \) is released at time 7 and immediately scheduled. At time 8, \( \tau_3 \) self-suspends and \( \tau_2 \) is resumed after its self-suspension and completed by time 9. Lastly, \( \tau_3 \) 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 \( \tau_3 \) 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 \( \tau_1 \) is completed by time 5. Then, \( \tau_2 \) is released and immediately run. At time 7, \( \tau_2 \) is resumed from its self-suspension and \( \tau_3 \) is delayed since it has a larger deadline than \( \tau_2 \). \( \tau_3 \) 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 \( C_{1,2} = 1 \)).

It is easy to show that in all these cases there exist feasible schedules while **EDF** always
Figure 2: Example of execution-time anomaly for EDF by decreasing $C_{1,1}$ of one unit of time

fails.

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 On-line scheduling algorithms

4.1 Introduction

In this section, in order to simplify the result presentations, we assume that periods are larger enough so that exactly one job of each task belongs to the hyperperiod. In this
part, we analyse the competitiveness of the classical on-line scheduling algorithms for
the optimization of two performance criteria:

- To maximize the number of early tasks (or equivalently minimizing the number
  of tardy tasks).

- To minimize maximum response time (or flow time).

For each performance measure, we first recall known results, then we present and
demonstrate our results. We shall use the *competitive analysis* to compare these classical
scheduling algorithms against an optimal clairvoyant algorithm (the adversary).

### 4.2 Competitive analysis

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* a-
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 \( I \): 

\[ c_A = \sup_{\text{any} I} \frac{\sigma_A(I)}{\sigma^*(I)} \]

The competitive ratio of an algorithm \( A \) is greater than or equal to 1. If \( c_A = 1 \), then \( A \) is an optimal algorithm.

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.

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.
4.3 Maximizing the number of early tasks

4.3.1 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 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.

Using the resource augmentation technique, it has been proved in [13], 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 an optimal clairvoyant algorithm with a 1-speed processor, then EDF will define a feasible schedule under a 2-speed processor.

4.3.2 Non-competitiveness results

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: $\tau_0$ 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 $0$, $\tau_0$ and $\tau_1$ are available, SRPTF schedules $\tau_0$ because $\tau_0$ has the shortest remaining processing time. After, the first subtask of task $\tau_i$ is scheduled at time $i$ ($1 \leq i \leq n$) and the second at time $K + (i - 1)$. The clairvoyant algorithm schedules $\tau_1$ at time 0 and it schedules every task $\tau_i$ ($i \in \{2, \ldots, n\}$) at time $i - 1$. To
finish, the clairvoyant algorithm schedules $\tau_0$.

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:

$$U_I = \sum_{i=0}^{n} \frac{C_{i,1} + C_{i,2}}{T_i} = \frac{1}{K-1} + \sum_{i=1}^{n} \frac{2}{K}$$

$$= \lim_{K \to \infty} \frac{2n+1}{K} = 0$$

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

□

19
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.

□

We now consider the **Least Laxity First** scheduling algorithm (LLF) [10].

**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 $\tau_i (1 \leq i \leq n)$ 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 $\tau_1$ is scheduled by LLF. But at time 1, the priorities of the tasks $\tau_i (2 \leq i \leq n)$ are greater than the priority of $\tau_1$. Consequently, the task $\tau_2$ is scheduled. But at time 2, the
others tasks have a priority greater than the priority of $\tau_2$. Therefore, every task $\tau_i$ 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: $LLF$ is not competitive

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:

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

$$= \lim_{K \to \infty} \frac{6n}{K} = 0$$

21
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.

\[\square\]

### 4.3.3 Resource augmentation

To schedule tasks with self-suspensions, our system has several processors, one main processor to schedule tasks and the dedicated processors run external operations. Consequently, we can increase the speed of two types of processors. This section is subdivided in two parts. In the first part, we increase the speed of the main processor and during the second part, the speed of the dedicated processors.

**Increasing the speed of the main processor.** We next show that when tasks are allowed to self-suspend, then $EDF$ cannot define a feasible schedule under a $s$-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 competitive to minimize the number of tardy tasks even with a $s$-speed processor, for any positive integer $s$. 

22
Proof: We use a contradiction argument. Let \( s \) be an integer such that \( s > 1 \) and such that if there exists a feasible schedule under a 1-speed processor then there exists a feasible EDF schedule under \( s \)-speed processor. Let \( I \) be an instance with \( n + 1 \) tasks. Every task of \( I \) arrives in the system at time 0 with the following characteristics with a unit speed:

\[
\tau_0 : C_{0,1} = 2s, X_0 = 0, C_{0,2} = 0, D_0 = 4s + 1 \\
\tau_i : C_{i,1} = \frac{1}{n}, X_i = 4s, C_{i,2} = \frac{1}{n}, D_i = 4s + 2 \quad 1 \leq i \leq n
\]

Figure 5: The schedule of \( I \) under EDF with a 2-speed main processor (\( s = 2 \))

- At time 0, all the tasks are released, the optimal clairvoyant algorithm under an 1-speed processor schedules \( \tau_1 \) in first position and after it schedules \( \tau_i \) (\( i \) between 2 and \( n \)). To finish, it schedules \( \tau_0 \). All deadlines are respected.

- At time 0, EDF under a \( s \)-speed processor schedules \( \tau_0 \) in first position, since it has the shortest deadline. After, it schedules every task \( \tau_i \) (\( 1 \leq i \leq n \)). Consequently, every task \( \tau_i \) (\( 1 \leq i \leq n \)) is delayed and misses its deadlines.
Consequently, the on-line algorithm \( EDF \) even with \( s \)-speed main processor cannot obtain a feasible schedule of the instance \( I \) whereas there exists a feasible schedule under an 1-speed processor.

Figure 5.a (Resp. Figure 5.b) presents the schedule of task system \( I \) under an optimal clairvoyant algorithm (Resp. \( EDF \)) with \( s = 2 \).

Now, we determine the competitive ratio (to minimize the number of tardy jobs) of the scheduling algorithm \( EDF \) with \( s \)-speed main processor. The optimal algorithm meets all deadlines (cf. Figure 5.a). The on-line algorithm \( EDF \) meets an unique deadline: the deadline of \( \tau_0 \) (cf. Figure 5.b).

Consequently, increasing the number of jobs \((n+1)\) to infinity, the competitive ratio of \( EDF \) can be arbitrary small:

\[
c_{EDF} = \frac{\sigma_{EDF}}{\sigma_{Opt}} = \lim_{n \to \infty} \frac{1}{n+1} = 0
\]

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

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).

**Increasing the speed of dedicated processors.** We demonstrate in this part that increasing the speed of every dedicated processor does not improve the performance of the scheduling algorithm *EDF*. The main processor is unit speed for both systems.

**Theorem 7** Increasing the speed of the dedicated processor for the on-line scheduling algorithm *EDF* does not improve these performances when tasks are allowed to self-suspend at most once.

**Proof:** To prove this theorem, we use the same method as in Theorem 6: a contradiction argument. We assume that there exists an integer \(s, s > 1\) such as the on-line algorithm *EDF* uses \(s\)-speed dedicated processors. Consequently, if there exists a feasible schedule of an instance \(I\) then there exists a feasible schedule of the instance \(I\) under *EDF* with \(s\)-speed dedicated processors.

Let \(I\) be the following instance:

\[\tau_0 : r_1 = 0, C_{1,1} = 2s, X_1 = 0, C_{1,2} = 0, D_1 = 2s + 1\]

\[\tau_i : r_i = 0, C_{i,1} = 1/n, X_i = 2s, C_{i,2} = 1/n, D_i = 2s + 2 \quad 1 \leq i \leq n\]

- at time 0, all tasks are available, the optimal clairvoyant algorithm with 1-speed dedicated processors schedules \(\tau_1\) at time 0 and after it schedules \(\tau_i\) (\(i\) between 2 and \(n\)). To finish, it schedules \(\tau_0\). All deadlines are respected.

25
At time 0, $EDF$ with $s$-speed dedicated processors schedules $\tau_0$ in first position, since it has the shortest deadline. After, it schedules the tasks $\tau_i$ ($1 \leq i \leq n$).

Consequently, every task $\tau_i$ ($1 \leq i \leq n$) are delayed and miss their deadlines.

Consequently, the on-line algorithm $EDF$ even with $s$-speed dedicated processors cannot obtain a feasible schedule of $I$ whereas there exists a feasible schedule with 1-speed processor.

The Figure 6 presents the schedule of $I$ under an optimal algorithm and under $EDF$ with $s = 2$.

To finish this proof, we determine the competitive ratio (to minimize the number of tardy jobs) of $EDF$ with $s$-speed dedicated processors. In the optimal schedule, all deadlines are respected as it is shown by the Figure 6.a. But, the on-line algorithm $EDF$ meets only one deadline: the deadline of $\tau_0$ (cf. Figure 6.b).

So the competitive ratio of $EDF$ is obtained in tending the number of jobs $(n + 1)$ to infinity:

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

To conclude, $EDF$ is not competitive to minimize the number of tardy jobs even in increasing the speed of dedicated processors.

So, the assumption that there exists a feasible schedule under a $s$-speed processor is false and the theorem is demonstrated for any integer $s > 1$. 
4.3.4 Conclusion

As we showed, increasing the speed of the main processor or the speed of dedicated processors does not help to schedule tasks systems when self-suspension are allowed. An interesting open issue is to increase the speed of all processors simultaneously.

4.4 Minimizing the Maximum response time

4.4.1 Known results

Several response time analysis of tasks with self-suspension have been proposed in the past [7, 9, 12, 11]. But, as far as we know, no result is known about any guarantee about such worst-case response time upper bounds. Next, we show that response time analysis of on-line scheduler cannot be better than 2-competitive. To show this result, we focus on a subproblem: minimizing the maximum response time that can be achieved by a
task during the system life.

For the on-line problem, the scheduling algorithm \textit{FIFO} (First In First Out) is \((3 - \frac{2}{P})\)-competitive for parallel machine with \(P\) processors [3]. As a direct consequence, under an uniprocessor system \((P = 1)\), \textit{FIFO} is optimal.

If tasks have no self-suspension, then for any non-idling scheduling (i.e. the processor is never idle if there are some available jobs) then the worst-case response time of a task cannot be greater than the length of the \textit{Synchronous Busy Period}.

We recall that a busy period is an interval of time in which the processor is always busy. In a synchronous busy period tasks are synchronously released at the beginning of the period. Its length is computed as the smallest solution of the following equation:

\[
w(t) = t \quad \text{where} \quad w(t) = \sum_{i=1}^{n} \left\lceil \frac{t}{T_i} \right\rceil C_i
\]

### 4.4.2 Competitiveness of EDF, RM and DM

Our first result shows that the scheduling algorithm \textit{EDF} is not better than 2-competitive to minimize the maximum response time.

**Theorem 8** \textit{The scheduling algorithm EDF is not better than 2-competitive to minimize the maximum response time if tasks are allowed to self-suspend.}
Proof: Let $I$ be the following task system with self-suspensions:

$$\tau_1 : r_1 = 0, C_{1,1} = \epsilon, X_1 = K, C_{1,2} = \epsilon, D_1 = 4K$$

$$\tau_2 : r_2 = 0, C_{2,1} = K, X_2 = 0, C_{2,2} = 0, D_2 = 4K - 1$$

Where $K$ is an integer greater than 1 and where $\epsilon$ is an arbitrary number between 0 and 1.

The Figure 7 presents the schedule of the instance $I$ under $EDF$ and under an optimal clairvoyant algorithm. At time zero, the algorithm $EDF$ schedules $\tau_2$ since it has the shortest deadline. But the clairvoyant algorithm schedules $\tau_1$ in the first position and after that it schedules $\tau_2$ during the suspension of the task $\tau_1$.

![Figure 7: Competitiveness of $EDF$ to minimize the maximum response time](image)

The maximum response time obtained by $EDF$ for the task system $I$ is equal to $2K + 2\epsilon$.

Whereas the maximum response time of the optimal algorithm is just equal to $K + 2\epsilon$.

So assuming that $\epsilon$ tend to 0, we obtain the competitive ratio of $EDF$:

$$c_{EDF} = \frac{\sigma_{EDF}}{\sigma_{OPT}} = \lim_{\epsilon \to 0} \frac{2K + 2\epsilon}{K + 2\epsilon} = \frac{2K}{K} = 2$$ (1)
As a conclusion, EDF is not better than 2-competitive to minimize the maximum response time.

Now with the next corollary, we extend the previous result to RM and DM scheduling policies.

**Corollary 2** The scheduling algorithms RM and DM are not better than 2-competitive to minimize the maximum response time when tasks are allowed to self-suspend.

**Proof:** We use the same task system $I$ used to prove that EDF is at least 2-competitive (Theorem 8). If we use the same instance $I$, DM and RM assign priorities to the tasks exactly as EDF does. Consequently, we obtain the same conclusions for both scheduling algorithms. Thus, the competitive ratio of RM and DM is not better then 2 to minimize the maximum response time.

4.4.3 Competitiveness of LLF

The scheduling algorithm LLF (Least Laxity First) assigns the greatest priority to the task having the shortest dynamic laxity. But if we consider or not the self-suspension as part of the task laxity, we obtain two different definitions to define the laxity of a job:
• Considering the self-suspension as part of the task laxity, the dynamic laxity $L_1$ of a task $\tau_i$ at time $t$ equals: $L_{1,i}(t) = d_i - t - c_i(t)$ where $c_i(t)$ is the remaining processing requirement at time $t$ to schedule.

• Not accounting for the self-suspension as part of the task laxity, the dynamic laxity $L_2$ of a task $\tau_i$ at time $t$ is equal to: $L_{2,i}(t) = d_i - t - c_i(t) - x_i(t)$ where $x_i(t)$ is the remaining suspension delay at time $t$.

In the following theorem, we use the first definition of the dynamic laxity (suspension delays are not considered to compute dynamic laxity of tasks).

**Theorem 9** The scheduling algorithm LLF is not better than 2-competitive to minimize the maximum response time in scheduling tasks with self-suspension.

**Proof:** To prove this theorem, let $I$ be the following task system with self-suspension and we demonstrate that the competitive ratio under $I$ for the algorithm LLF equals 2:

$\tau_1 : r_1 = 0, C_{1,1} = \epsilon, X_1 = K, C_{1,2} = \epsilon, D_1 = 4K$

$\tau_2 : r_2 = 0, C_{2,1} = K, X_2 = 0, C_{2,2} = 0, D_2 = 2K + 2\epsilon$

Where $K$ is an integer greater than 1 and where $\epsilon$ is an arbitrary number between 0 and 1.

The scheduling of the instance $I$ under the on-line algorithm LLF and under an optimal clairvoyant algorithm are presented by Figure 8. The algorithm LLF (cf. Figure 8) schedules task $\tau_2$ in first position, since it has shortest dynamic laxity, whereas the
adversary schedules $\tau_1$ at time zero, and then schedules $\tau_2$ during the self-suspension of $\tau_1$.

\[ \text{LLF} \]
\[\text{Optimal } \]
\[\text{algorithm} \]

Figure 8: Competitiveness of LLF to minimize the maximum response time

The worst-case response time obtained with the on-line algorithm LLF is equal to $2K - 2\epsilon$ whereas the maximum response time obtained with the optimal algorithm equals $K + 2\epsilon$. Considering these results, we obtain for the on-line algorithm LLF, the following competitive ratio:

\[
c_{\text{LLF}} = \frac{\sigma_{\text{LLF}}}{\sigma_{\text{OPT}}} = \lim_{\epsilon \to 0} \frac{2K + 2\epsilon}{K + 2\epsilon} = \frac{2K}{K} = 2
\]

(2)

To conclude this proof, we have proved that the on-line scheduling algorithm LLF is not better than 2-competitive to minimize the maximum response time.

\[ \square \]

**Corollary 3** If we consider suspension delays to compute the dynamic laxity of tasks, LLF is still not better than 2-competitive to minimize the maximum response time.

**Proof:** To prove this Corollary, we use the same task system as in Theorem 9. The same scheduled is obtained since $D_1$ is large and the same competitive ratio is derived.
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. Such a result is still valid even if the on-line scheduler uses a faster processor than the optimal clairvoyant algorithm (thus, speed does not help to schedule tasks allowed to self-suspend). We then show that classical scheduling policies cannot be better than 2-competitive for minimizing the maximum response time of tasks. We showed that response times achieved by on-line scheduler can be at least twice these ones achieved by an off-line scheduler. Note that, response time analysis introduces another gap since such tests are usually based on pseudo-polynomial algorithms to solve a strongly $\mathcal{NP}$-hard problem. An interesting issue is to analyse schedulability tests presented in [7, 9, 12, 11] in order to analyse their worst-case performance against an exact response time analysis.
(necessarily based on an exponential time computational complexity).

In further works, we will try to define practical solutions for scheduling such task systems. Another interesting issue will be to consider non-independent tasks.
References


