A New Approach for Limited Preemptive Scheduling in Systems with Preemption Overhead

Mitra Nasri[1], Geoffrey Nelissen[2], Gerhard Fohler[1]
[1] *Chair of Real-time Systems, Technische Universität Kaiserslautern, Germany*
[2] *CISTER/INESC-TEC, ISEP, Porto, Portugal,*
Emails: {nasri,fohler}@eit.uni-kl.de, grrpn@isep.ipp.pt

***Abstract*—This paper considers the problem of reducing the number of preemptions in a system with periodic tasks and preemption overhead. The proposed solution is based on the key observation that for periodic task sets, the task with the smallest period plays an important role in determining the maximum interval of time during which a lower priority task can be executed without being preempted. We use this property to build a new limited preemptive scheduling algorithm, named RS-LP, based on fixed-priority scheduling. In RS-LP, the length of each task's non-preemptive region is varying during the system execution so as to keep the preemptions aligned with the releases of the highest priority task. This simple mechanism allows us to reduce the overall number of preemptions. The proposed algorithm, decides whether or not to preempt the currently executing task based on the maximum blocking tolerance of the higher priority tasks. In any case, the preemptions are authorized only at release instants of the task with the smallest period, thereby limiting the maximum number of preemptions to the number of releases of the highest priority task. Moreover, in this paper, we provide two different preemption overhead aware schedulability tests for periodic and loose-harmonic task sets (i.e., where each period is an integer multiple of the smallest period), together with a lower bound on the maximum number of preemptions. To conclude, extensive experiments comparing RS-LP with the state of the art limited preemptive scheduling algorithms are finally presented.**

***Keywords*-limited preemption scheduling; real-time systems; schedulability analysis; cache related preemption delay;**

I. Introduction

Preemptive scheduling is used in most real-time systems because it allows the operating system to allocate the processor to the incoming tasks with urgent timing requirements, and hence, increases the overall schedulability [1]. For example, the preemptive earliest deadline first (EDF) scheduling algorithm is able to schedule a system up to 100% utilization if: i) tasks are periodic with implicit deadlines, ii) there is no shared resource or dependency among the tasks, and iii) the preemption overhead is negligible [2]. In reality, however, the preemption overhead is not negligible [3]. It has been shown that it can be as large as 40% of the worst-case execution time (WCET) of the task measured in isolation [4].

The preemption overhead has different sources. In cache-enabled real-time systems, *cache related preemption delays* (CRPD) happen when a task preempts another and evicts some cache blocks which have been used and will later be reused by the preempted task [3]. Preemption overhead can also happen in the context of networked systems where messages must be segmented in order to not cause a long blockage for messages with higher frequency and shorter deadline. In such conditions, a header will be added to each segment of the message which in turn, increases the total transmitted data, and hence, decreases the throughput. In reservation based scheduling, which are commonly used in hypervisors [5] or integrated modular avionics (IMA) systems [6], preemption overheads can be significant too. For instance, according to the ARINC 653 avionics standard, real-time applications run within a *partition server* which is the basic execution environment of software applications [6]. Since IMA supports the temporal and spatial isolation of the partition servers from one another, the various real-time avionics functions (having various safety-assurance levels) can be developed independently. However, in order to preserve the isolation between the partition servers, overheads for partition server context-switch (save of the data transmitted by the preempted partition to other partitions, load of the data received by the preempting partition, switching of execution stack and processor registers, cache flushes implemented to enforce the space partitioning, etc) must be considered [7].

There have been different approaches to cope with preemption overhead. One of them is to provide schedulability analysis for the scheduling algorithms such as rate monotonic (RM) and EDF which are implemented in many real-time operating systems [8]. However, RM and EDF have a relatively large number of preemptions. In order to reduce the number of preemptions, several scheduling algorithms have been developed including: fixed-priority scheduling with floating non-preemptive regions (FP-NPR) [9], fixed-priority scheduling fixed-preemption points (FP-FPP) [9], preemption threshold (PT) approaches [10], [11], and limited preemption EDF (LP-EDF) [12], [13].

FP-NPR and LP-EDF defer the preemption of a low priority job if a high priority job is released. In both algorithms, the maximum length of the *non-preemptive region* (NPR) is calculated at design time in the way that the schedulability is guaranteed. In LP-EDF, the length of NPR is a function of the remaining time to the deadline of the executing job while in FP-NPR it is a constant value for all jobs of a task. In our paper, we show that the maximum length of the NPR obtained in [1], [9] and [12], [13] is valid only for sporadic tasks, not for periodic tasks. In FP-FPP, a preemption can only happen at the preemption points of the executing task. The maximum length of NPR between every two preemption points has been calculated in [1], [9] using the same approach as for

FP-NPR, namely, it is based on the maximum tolerable blocking time of the tasks. Therefore, FP-FPP has the same problem as FP-NPR with periodic tasks. In the preemption threshold approach [10], a low priority task can disable the preemption up to a certain priority level, called preemption threshold. In most cases, however, the low priority tasks are not allowed to have preemption threshold 1, because otherwise, they will not let the highest priority task (denoted by τ_1) be scheduled. As a result, every release of τ_1 causes a preemption, which in many cases, could be postponed as we will show later.

In [14], an EDF-based approach has been introduced that decides whether to preempt an executing job or not based on the newly released high priority job. At design time, a binary variable is assigned to each task which determines whether jobs of that task has the right to preempt any other job in the system or not. This approach has two disadvantages; a) in order to assign the binary variables it is required to iterate over all possible combinations of the variables and for each combination, a pseudo-polynomial time schedulability test must be evaluated, and b) since the binary variables are not related to the jobs, but to the tasks, in many task sets, τ_1 will always be a preempting task, meaning that as soon as a job of τ_1 is released, it preempts the executing job. As a result, similar to the preemption threshold approach, this approach cannot take advantage of postponing τ_1 to provide large non-preemptive execution segment for the task.

A *non-preemptive execution segments* (NPS) is defined as an interval of time during which a task is executed non-preemptively after it became the task with the highest priority ready to execute. As observed in [15], τ_1 plays an important role in the length of the NPS of other tasks in the system because, in a periodic task set, τ_1 has the highest release frequency and the shortest deadline. Nasri et al., [15], [16] have used this observation to build a non-work-conserving non-preemptive scheduling algorithm based on RM that schedules a task only if it does not cause a deadline miss for the next instance of τ_1, otherwise, it inserts an idle time until the next release of τ_1. However, this approach may reduce the schedulability of highly utilized systems due to the inserted idle times.

In order to reduce the number of preemptions, we propose to dynamically compute the length of the NPS' as a function of the higher priority jobs that are being released rather than, as currently done in most approaches of the state of the art, statically maximizing the length of the NPR as a function of the sole task about to be preempted.

In this paper, we present an online fixed-priority scheduling algorithm which synchronizes the execution windows of the low priority tasks with the releases of τ_1. Therefore, it allows the low priority tasks to be scheduled non-preemptively for the largest feasible non-preemptive interval of time equal to two times the slack of τ_1. Yet, to cope with the situations where the maximum tolerable blocking of a high priority task is not as large as this window, we let a high priority task with a low blocking tolerance force a preemption. That additional preemption is however postponed until the next release of τ_1. In other words, our approach can be seen as a limited preemption strategy with a varying length of the non-preemptive region, which is determined with respect to the parameters of the released tasks. Besides, due to the fact that even if a preemption is required, we postpone it to the next release of τ_1, we synchronize the execution segments of the tasks with the releases of τ_1 which in turn, gives them a chance to maximize their next NPS.

Contributions. The contributions of our paper are listed as follows: 1) showing that the existing approaches, i.e., RM, EDF, FP-NPR, FP-FPP, and LP-EDF are not able to minimize the number of preemptions, 2) introducing a scheduling solution based on fixed-priority scheduling which is able to increase the length of NPS, 4) providing schedulability tests for periodic and loose-harmonic task sets that are aware of the preemption overhead, 5) deriving the lower bound on the number of preemptions and proposing a new feasibility test, and 6) showing that verifying the feasibility of a task set which must be scheduled with the minimum number of preemptions obtained from our lower bound, is an NP-Hard problem.

Paper organization. The remainder of the paper is organized as follows; Sect. II describes the system model and backgrounds. In Sect. III we present some motivational examples to show differences between the existing limited preemption scheduling solutions. In Sect. IV we introduce our fixed priority scheduling solution. Our two schedulability tests for periodic and loose-harmonic task sets are presented in Sect. V. In Sect. VI, we derive and discuss the lower bound on the number of preemptions. Sect. VII presents the experimental results, and the paper is concluded in Sect. VIII.

II. System Model and Backgrounds

A. System Model

We consider a uniprocessor system executing a set of independent hard real-time periodic tasks denoted by $\tau : \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task τ_i in the task set is identified by $\tau_i : (C_i, T_i, \Delta_i)$, where $C_i \in \mathbb{R}$ is the worst-case execution time (WCET) of τ_i, $T_i \in \mathbb{R}$ is its period, and $\Delta_i \in \mathbb{R}$ is its preemption overhead. Δ_i can be calculated with one of the several methods introduced in the state of the art (e.g., [17] or [7]). In this paper, we assume that the deadlines are implicit (i.e., the deadline of each task is equal to its period) and that there is no release offsets, (i.e., all tasks are released at time 0). The task set is said to be *schedulable* by a given scheduling algorithm if for all sequences of jobs generated by the tasks from time 0 to ∞, no job misses its deadline when scheduled with that algorithm. In the paper, we use term *periodic* to refer to the periodic tasks with no release offset.

The system utilization is denoted by $U = \sum_{i=1}^{n} u_i$ where $u_i = C_i/T_i$ is the utilization of task τ_i. The hyperperiod is denoted by H and is the least common multiple of the periods. Tasks are indexed according to their periods so that $T_1 < T_2 \leq \ldots \leq T_n$. The period ratio of a task τ_i $(1 < i \leq n)$ is defined as $k_i = T_i/T_{i-1}$, $k_i \in \mathbb{R}$. A task set is *harmonic* iff $\forall i, k_i \in \mathbb{N}$, and is *loose-harmonic* [16] iff $\forall i, T_i/T_1 \in \mathbb{N}$.

We use hp_i to denote the set of tasks with higher priority than τ_i, i.e., $hp_i = \{\tau_1, \tau_2, \ldots, \tau_{i-1}\}$ where tasks are indexed according to their periods. Without loss of generality, we assume that if there are several tasks with a period equal to T_1, then they are represented as a single task τ_1 with a WCET equal to $\sum_{\{\tau_i; T_i = T_1\}} C_i$ and a period equal to T_1. All these tasks are then executed sequentially.

B. Background

This subsection presents the state of the art approaches to calculate the maximum NPR lengths of a set of tasks scheduled with a fixed-priority scheduling algorithm. We discuss two specific policies: FP-NPR and FP-FPP.

With FP-NPR, a task enters in a non-preemptive mode at the release of a higher priority task. The length of the non-preemptive region is calculated at design time according to the maximum tolerable blocking time (denoted by β_i) of each task τ_i. In [9], β_i is calculated using the following equation:

$$\beta_i = \max_{C_i < t \leq T_i} \{t - rbf_i(t)\} \tag{1}$$

where $rbf_i(t)$ is an upper bound on the execution request in a time window of length t in which tasks in $hp_i \cup \tau_i$ execute. $rbf_i(t)$ is defined as

$$rbf_i(t) = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil C_j \tag{2}$$

In [9], Equation (1) is solved by considering value of t obtained from a set of time instants which was defined in [18]. In [18] it has been shown that instead of considering all possible t values from C_i to D_i, it is enough to consider a finite set of values for each task τ_i. This set is obtained with the following recursive expression:

$$\mathcal{TS}(\tau_i) \doteq \mathcal{P}_{i-1}(D_i) \tag{3}$$

$$\mathcal{P}_j(t) = \begin{cases} \{t\} & j = 0 \\ \mathcal{P}_{j-1}\left(\left\lfloor \frac{t}{T_j} \right\rfloor T_j\right) \cup \mathcal{P}_{j-1}(t) & \text{otherwise} \end{cases} \tag{4}$$

After obtaining β_i from Equation (1), the maximum NPR length (denoted by Q_i) of task τ_i is calculated the recursive equation below (starting from $Q_1 = \infty$) [9]:

$$Q_i = \min\{Q_{i-1}, \beta_i\} \tag{5}$$

With FP-FPP, Q_i is used to assign preemption points in the task, i.e., the maximum distance between every two consecutive preemption points is Q_i.

Note that preemption thresholds algorithms [11] use the exact same concept of maximum tolerable blocking to compute the length of the execution segments and their associated preemption threshold.

III. Motivational Examples

In this section, we discuss the natural differences between the state of the art methods for limited preemption scheduling, in order to provide a better understanding about their advantages and weaknesses. We show that in periodic task sets, the maximum length of the non-preemptive execution segment, as well as the maximum length of the NPR of a task can be larger than what is provided by some of the state of the art algorithms.

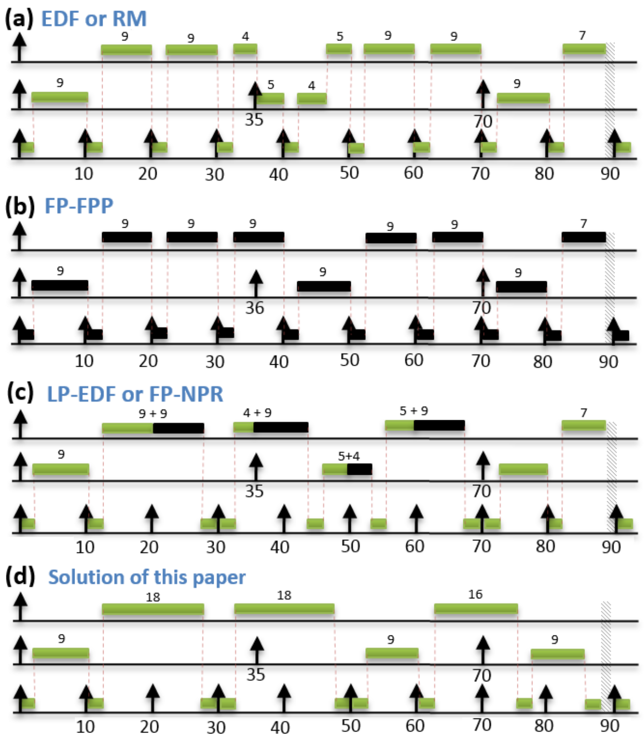

Figure 1. A task set with 3 tasks $\tau_1 : (1, 10)$, $\tau_2 : (9, 35)$, and $\tau_3 : (52, 105)$ scheduled by EDF, RM, FP-FPP, FP-NPR, LP-EDF, and our scheduling solution. The black color shows non-preemptive mode of execution. Using (5), $Q_1 = \infty, Q_2 = 9$, and $Q_3 = 9$ for FP-NPR and FP-FPP. In LP-EDF schedule, τ_3 stops its execution at time 44 while it could have continued to execute until 49 without jeopardizing the schedulability of τ_1 or τ_2.

Fig. 1 shows a task set with 3 tasks $\tau_1 : (1, 10)$, $\tau_2 : (9, 35)$, and $\tau_3 : (52, 105)$ which is scheduled by RM, EDF, FP-FPP, FP-NPR, LP-EDF and the algorithm which will be presented in this paper. As it can be seen, as soon as a job of τ_1 is released both EDF and RM preempt the low priority job even if the preemption can be postponed. For FP-FPP, shown in Fig. 1-(b), we have used [9] to calculate Q_i (i.e., the length of the NPR between every two preemption points). This algorithm executes the tasks non-preemptively until it reaches a preemption point. As can be seen, FP-FPP does not minimize the number of preemption since in this algorithm, Q_i is highly affected by the slack of the first task.

The schedule provided by the floating point limited preemption algorithms, e.g., LP-EDF and FP-NPR, is shown in Fig. 1-(c). However, as one can see, since the non-preemptive mode of a task is activated when *any* higher priority task is released, these algorithms preempt τ_3 at 44 while it could have been executed for 5 more time units without jeopardizing the schedulability of τ_2 or τ_1. The reason is that the maximum length of the NPR (which will be activated at the releases of high priority tasks) should in fact depend on the characteristics of the released tasks. For example, at time 20 when τ_1 is released, the length of the NPR is 9, which will create a non-preemptive execution segment $[11, 29]$ for τ_3. However, when τ_2 is released at

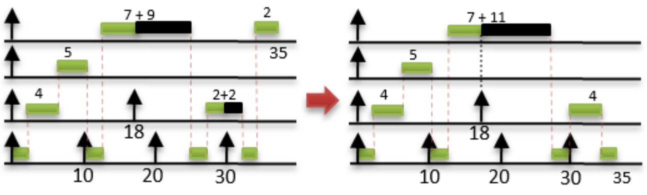

Figure 2. An example with 4 tasks $\tau_1 : (1, 10)$, $\tau_2 : (4, 18)$, $\tau_3 : (5, 45)$, and $\tau_4 : (18, 90)$, to show that the maximum length of the non-preemptive region can be larger than what is calculated in Equation (5).

35, it is possible to have a larger non-preemptive execution segment from 31 until 49 for τ_3. At 49, the slack of the 5th job of τ_1 becomes 0, and hence a preemption is unavoidable. This solution has been shown in Fig. 1-(d). We summarize these properties in the following theorem.

Theorem 1. *In periodic task sets, EDF, RM, LP-EDF, FP-NPR, and FP-FPP neither minimize the number of preemptions nor maximize the length of the non-preemptive execution segment of a task.*

Proof: The proof is based on the counter example shown in Fig. 1 with 3 periodic tasks $\tau_1 : (1, 10)$, $\tau_2 : (9, 35)$, and $\tau_3 : (52, 105)$ with no preemption overhead. As shown in Fig. 1-(d), the largest execution segment is 18 units for τ_3, and it can be feasibly scheduled only with 2 preemptions. ■

Based on the intuition behind the example in Fig. 1, we show that for periodic tasks, the method suggested by Yao et al., [9] does not calculate the largest length of the NPR for FP-NPR (and FP-FPP).

Theorem 2. *The method suggested by Yao et al. in [9] (based on Equation (5)) does not calculate the largest length of the NPR in periodic task sets.*

Proof: The proof is based on a counter example with 4 tasks $\tau_1 : (1, 10)$, $\tau_2 : (4, 18)$, $\tau_3 : (5, 45)$, and $\tau_4 : (18, 90)$ shown in Fig. 2. In this task set, τ_4 releases only one job in the hyper-period. Moreover, it is possible to schedule that only job of τ_4 non-preemptively in the execution window $[11, 29]$, which means that the maximum NPR length for this task is 11 units, while according to Equation (5), this length should not be larger than $T_1 - C_1 = 9$. ■

An interesting conclusion from Theorems 1 and 2 is that the length of the last non-preemptive region of the task is not the only factor that affects the number of preemptions. As shown in Fig. 1-(d) and further analyzed in Section VII, by synchronizing the preemptions generated by any task on the releases of τ_1, the overall number of preemptions can be significatly reduced. Later, in Sect. VI, we derive a lower bound on the number of preemptions for any feasible task set. We show that in order to minimize the number of preemptions, the length of the NPSs must be equal to two times the slack of the highest priority task τ_1 (i.e., $2 \times (T_1 - C_1)$).

IV. A New Scheduling Algorithm

In this section, we introduce our new limited preemptive scheduling algorithm referred to as *release-sensitive limited preemptive* (RS-LP).

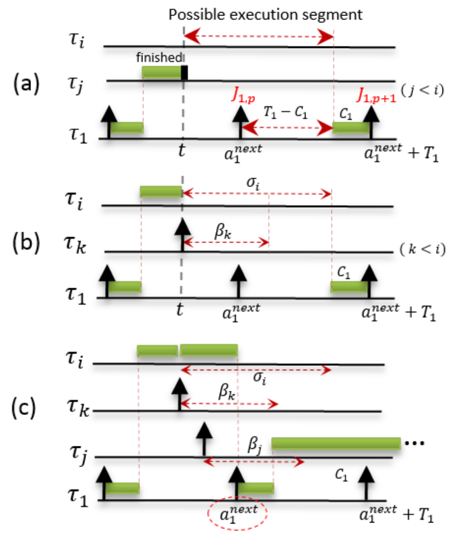

Figure 3. Three examples to explain how the RS-LP method works; (a) shows the situation where Algorithm 1 is activated. In this situation, τ_i starts its execution until $a_1^{next} + T_1 - C_1$. (b) shows the situation where Algorithm 2 is activated due to the release of τ_k. In this case, if $\beta_k < \sigma_i$, task τ_i must stop its execution at a_1^{next}. (c) shows a case where high priority tasks τ_k and then τ_j are released. Since they cannot tolerate a blocking as long as σ_i, task τ_i must be preempted at a_1^{next}.

As stated in Section III, FP-NPR and LP-EDF activate the non-preemptive region of a running task τ_i at the release of any higher priority job regardless of the characteristics of that job. The NPR completes after Q_i time units and τ_i gets preempted. This approach may generate a succession of short execution segments for τ_i when their length could otherwise have been larger. Indeed, if the released higher priority job has more slack than Q_i, there is no need to preempt τ_i after Q_i.

Therefore, instead of having a fixed length for the non-preemptive region as it is the case for PT, FP-NPR and FP-FPP, we propose that whenever a high priority task is released, the decision of whether or not to preempt the running task is dynamically taken based on the maximum blocking tolerance of the released job (and that job only).

As presented in Algorithm 1, when a new task must be chosen for execution, RS-LP picks the task with the highest priority and starts an execution segment as long as the remaining time until the next release of τ_1 plus the whole slack of the next job of τ_1. That is, if τ_i starts executing at time t, then its execution segment will span from t to $a_1^{next} + (T_1 - C_1)$ where a_1^{next} is the next release of τ_1 after time t. As illustrated in Fig. 3-(a), this scheduling decision allows us to 1) maximize the length of the execution segment of τ_i, and 2) push the completion of the job $J_{1,p}$ of τ_1 released at a_1^{next} to coincide with the release of its next job $J_{1,p+1}$ released at time $a_1^{next} + T_1$. Therefore, $J_{1,p+1}$ will directly start executing after the completion of $J_{1,p}$ and one preemption will be avoided.

Furthermore, when a high priority task τ_k releases a job during the execution of a low priority task τ_i, RS-LP may decide to preempt the execution segment of τ_i before its completion. Yet, in order to limit the total number of preemptions, the preemption can only happen at the next

Algorithm 1: Algorithm executed at the completion of a job or an execution segment

```
Input: $t$: the current time
1 $a_1^{next} \leftarrow (\lfloor t/T_1 \rfloor + 1) \times T_1$ ;  // next release of $\tau_1$ after $t$
2 $i \leftarrow$ index of the highest priority task with a pending job;
3 Schedule $\tau_i$ for $a_1^{next} + (T_1 - C_1) - t$ time units;
```

release of τ_1. This technique has two advantages: 1) it allows us to synchronize the execution of the execution segments of the tasks with the release of τ_1, and hence, increase the chance that these tasks are scheduled using the largest possible time interval between two consecutive execution of τ_1; 2) by deferring these preemptions and synchronizing the finishing time of the execution segment of the low priority task τ_i with the releases of τ_1, we actually postpone several potential preemptions to a single time instant. Consequently, the only task which preempts a low priority task will be τ_1 (since τ_1 will be the task with the highest priority at the time τ_i is actually preempted). Fig. 3-(c) shows a case where several high priority tasks (τ_k and then τ_j) have been released during the execution of τ_i. As shown in the figure, the preemption of both τ_k and τ_j are postponed until a_1^{next}. Using this simple technique, we avoid a cascade of preemptions that may happen in RM or EDF due to the consecutive releases of high priority tasks in the reverse order of their priorities. As a result, with RS-LP, the number of preemptions suffered by any task other than τ_1 will be a function of the number of releases of τ_1.

Algorithm 2 describes the procedure followed by RS-LP whenever a new job is released. If the task τ_k which released a job has a lower priority than the currently executing task τ_i or if the maximum blocking tolerance of τ_k (i.e., β_k), is larger than or equal to the remaining length of the execution segment of τ_i (denoted by σ_i), we continue executing τ_i until completion of its execution segment (Line 11). Otherwise, if the maximum blocking tolerance of τ_k, is not larger than the remaining length of the execution segment of τ_i, then we reduce the size of the execution segment of τ_i such that it finishes at the next release of τ_1 happening at a_1^{next}) (Lines 7 to 9). This situation is illustrated in Fig. 3-(b).

Since finding the highest priority task with a pending job can be done in a constant time [19], the computational complexity of Algorithm 1 as well as the amortized complexity of Algorithm 2 per released job is $O(1)$. Same as FP-NPR or FP-FPP, β_k values in Algorithm 2 must be calculated at design time and store in an array (of size n) before the system starts. This process has pseudo-polynomial time computational complexity.

As a property of Algorithms 1 and 2, one can easily derive an upper bound for the maximum length of an execution segment with the RS-LP algorithm.

Lemma 1. *With RS-LP, the length of an execution segment of any task τ_i such that $i \neq 1$, is at most $2(T_1 - C_1)$.*

Algorithm 2: Algorithm executed at a job release

```
Input: $t$: the current time, $\tau_k$: the released task
1  if the processor is idle then
2      Call Algorithm 1;
3  else
4      $i \leftarrow$ index of the task running on the processor;
5      $\sigma_i \leftarrow$ remaining execution time of the current execution segment of $\tau_i$;
6      $\beta_k \leftarrow$ maximum blocking time of $\tau_k$;
7      if $\tau_i$ has a lower priority than $\tau_k$ and $\beta_k < \sigma_i$ then
8          $a_1^{next} \leftarrow \lceil t/T_1 \rceil \times T_1$;  // next release of $\tau_1$ at or after $t$
9          Schedule $\tau_i$ for $(a_1^{next} - t)$ time units;
10     else
11         Schedule $\tau_i$ for $\sigma_i$ time units;
               // continue executing $\tau_i$
12     end
13 end
```

Proof: According to Line 3 in Algorithm 1, the execution segment of any task τ_i is the longest if it starts executing at a time t as far as possible from the next release a_1^{next} of a job of τ_1. Since τ_1 has the highest priority, the last release of τ_1 before t must have completed its execution before τ_i becomes eligible for execution. Hence, $(a_1^{next} - t)$ is upper bounded by $(T_1 - C_1)$. At Line 3 in Algorithm 1, the maximum length of a non-preemptive execution segment of any task τ_i is given by $a_1^{next} + (T_1 - C_1) - t \leq 2(T_1 - C_1)$. ■

V. Schedulability Analysis

A. Schedulability Test for Periodic Tasks

In this section, we first derive an upper bound for the total preemption cost during the execution of a task τ_i scheduled with RS-LP (see Lemma 2 and Corollary 1). We then provide an equation to compute the maximum blocking time β_i tolerable by τ_i (see Equation (9)) and an upper bound on the actual blocking time suffered by τ_i during its execution (Lemma 3). All that information is finally used in Theorem 3 to derive an upper bound on the WCRT of a task τ_i scheduled with RS-LP.

Let $\mathcal{S}_i$ be the set of tasks $\tau_k \in hp_i \setminus \tau_1$ with a maximum tolerable blocking time β_k (computed with Equation (9) below) smaller than $2(T_1 - C_1)$.

Lemma 2. *With RS-LP, in a time window of length t where only tasks in $hp_i \cup \tau_i$ execute (i.e., tasks in $\{\tau_1, \tau_2, \ldots, \tau_i\}$), the total number of preemptions is upper bounded by*

$$\hat{P}_i(t) = \min\left\{ \left\lceil \frac{t}{T_1} \right\rceil,\ \left\lceil \frac{t}{2T_1} \right\rceil + \sum_{\tau_k \in \mathcal{S}_i} \left\lceil \frac{t}{T_k} \right\rceil \right\} \quad (6)$$

Proof: We first prove that the number of preemptions is upper bounded by $\lceil t/T_1 \rceil$ and then by $\lceil t/(2T_1) \rceil + \sum_{\tau_k \in \mathcal{S}_i} \lceil t/T_k \rceil$. Since both expressions are upper bounds, the total number of preemptions is upper bounded by the minimum of those two, which then proves the claim.

1) According to Line 3 in Algorithm 1 and Line 9 in Algorithm 2, the length of an execution segment is based on the arrival time of the next job released by τ_1. This means that only jobs released by τ_1 can preempt a running task. Since there are at most $\lceil t/T_1 \rceil$ jobs released by τ_1 in a window of length t, the number of preemptions is upper bounded by $\lceil t/T_1 \rceil$.

2) According to Line 3 in Algorithm 1, the length of an execution segment of a task τ_i that starts to execute at time t is set to $a_1^{next} + (T_1 - C_1) - t$ (where a_1^{next} is the release of the next job $J_{1,j}$ by τ_1 after t). Three situations may occur:

a) τ_i completes its execution before the end of the segment, in which case no preemption happens and Algorithm 1 is called again;

b) τ_i executes for $a_1^{next}+(T_1-C_1)-t$ and is preempted by τ_1 at time $a_1^{next} + (T_1 - C_1)$. Since τ_1 executes for C_1 time units, it completes at time $a_1^{next} + T_1$, which happens to be the release of a new job $J_{1,j+1}$ by τ_1. Since τ_1 has the highest priority, this new job $J_{1,j+1}$ is elected by Algorithm 1 to be executed right after the completion of the previous job $J_{1,j}$ of τ_1. Therefore, $J_{1,j}$ generates a preemption but $J_{1,j+1}$ does not;

c) Another task τ_k releases a job before the end of the execution segment of τ_i. Algorithm 2 is then called. If τ_k has a higher priority than τ_i and its maximum tolerable blocking time β_k is smaller than the remaining time σ_i until the end of the current execution segment of τ_i, then the execution segment of τ_i is shortened and one more preemption is generated. Note that by Lemma 1, σ_i is upper bounded by $2(T_1-C_1)$. Hence, only tasks in $\mathcal{S}_i$ (i.e., higher priority tasks such that $\beta_k < 2(T_1-C_1)$) can cause such reschedule, and at most $\sum_{\tau_k \in \mathcal{S}_i} \lceil t/T_k \rceil$ jobs can be released by the tasks in $\mathcal{S}_i$ within a time interval of length t. Consequently, the number of preemptions initiated by other tasks than τ_1 is upper bounded by $\lceil t/(2T_1) \rceil + \sum_{\tau_k \in \mathcal{S}_i} \lceil t/T_k \rceil$.

Therefore, by a) and b), only half of the jobs released by τ_1 can generate a preemption, that is, $\lceil 0.5 \lceil t/T_1 \rceil \rceil$. As shown in [20], this ceiling operation can be simplified as $\lceil t/(2T_1) \rceil$. Furthermore, by c), at most $\sum_{\tau_k \in \mathcal{S}_i} \lceil t/T_k \rceil$ jobs of other tasks than τ_1 can be the cause of one more preemption. Hence, the total number of preemptions in a window of length t is upper bounded by $\lceil t/(2T_1) \rceil + \sum_{\tau_k \in \mathcal{S}_i} \lceil t/T_k \rceil$. ■

Corollary 1. *In a time window of length t where only tasks in $hp_i \cup \tau_i$ execute, an upper bound on the total preemption cost is given by*

$$\hat{\Delta}_i(t) = \hat{P}_i(t) \times \max_{k=2}^{i} (\Delta_k) \tag{7}$$

Proof: Because τ_1 has the highest priority, it cannot be preempted. The maximum preemption cost caused by one preemption is therefore upper bounded by $\max_{k=2}^{i} (\Delta_k)$. Since by Lemma 2, there are at most $\hat{P}_i(t)$ preemptions in such a time window, the total preemption cost is upper bounded by $\hat{P}_i(t) \times \max_{k=2}^{i} (\Delta_k)$. ■

As explained in Sect. II-B, Yao et al. [9] have proposed a lower bound β_i on the maximum blocking time tolerable by τ_i defined in Equation (1). In this equation, $rbf_i(t)$ is an upper bound on the execution request in a time interval of length t where tasks in $hp_i \cup \tau_i$ execute. Therefore, the total demand $rbf_i(t)$ is given by the sum of: (i) the execution request of each task in $hp_i \cup \tau_i$, and (ii) the total preemption overhead suffered in a window of length t. As shown in [21](Equation (5)), the former is upper bounded by

$$rbf_k^*(t) = \left(\left\lceil \frac{t}{T_k} \right\rceil - 1 \right) C_k + \min \left\{ C_k,\ t - \left\lfloor \frac{t}{T_k} \right\rfloor T_k \right\} \tag{8}$$

and by Corollary 1, the latter is upper bounded by Equation (7). A lower bound on the maximum tolerable blocking time is thus given by

$$\beta_i = \max_{C_i < t \leq T_i} \left\{ t - \left(\hat{\Delta}_i(t) + \sum_{k=1}^{i} rbf_k^*(t) \right) \right\} \tag{9}$$

Using Equation (9) and Algorithm 2, one can now define the maximum blocking time B_i suffered by any job $J_{i,j}$ of task τ_i due to a lower priority task already running on the processor at $J_{i,j}$'s release.

Lemma 3. *The maximum blocking time suffered by any job released by τ_i due to lower priority tasks is upper bounded by*

$$B_i = \begin{cases} 0 & \textit{if } i = n \\ \min \left\{ T_1 - C_1,\ \max_{j=i+1}^{n} (C_j) \right\} & \textit{if } \beta_i < 2(T_1 - C_1) \\ \min \left\{ 2(T_1 - C_1),\ \max_{j=i+1}^{n} (C_j) \right\} & \textit{otherwise} \end{cases} \tag{10}$$

Proof: The first case is for τ_n which is the lowest priority task, and hence, cannot be blocked by a lower priority task. For the other tasks, we prove the three following properties:

1) Because Algorithm 1 always selects the highest priority task to be scheduled when a job completes an execution segment, at most one job of a lower priority task can block τ_i. For any other task than τ_n, the blocking time is thus upper bounded by $\max_{j=i+1}^{n} (C_j)$.

2) By Lemma 1, the maximum length of a non-preemptive execution segment of any task τ_j with a smaller priority than τ_i is $2(T_1 - C_1)$. Combining this fact with 1), we get that $B_i \leq \min \left\{ 2(T_1 - C_1),\ \max_{j=i+1}^{n} (C_j) \right\}$ for any task τ_i. This proves the third case of Equation (10).

3) According to Line 9 of Algorithm 2, if τ_k has a maximum tolerable blocking time β_k smaller than the remaining execution length σ_i of the current execution segment of the lower priority running task τ_i, then the length of τ_i's execution segment is shortened to $(a_1^{next}-t)$. By Lemma 1, σ_i is upper bounded by $2(T_1 - C_1)$, and $(a_1^{next}-t)$ is smaller than or equal to (T_1-C_1). Combining this with 1), we get that for any task τ_i such that $\beta_i < 2(T_1 - C_1)$, there is $B_i \leq \min \left\{ T_1 - C_1,\ \max_{j=i+1}^{n} (C_j) \right\}$. This proves the second case of Equation (10). ■

We can now derive an upper bound on the WCRT of a task τ_i under RS-LP.

Theorem 3. *An upper bound on the WCRT of a task τ_i scheduled with RS-LP is given by the smallest positive value R_i such that*

$$R_i = B_i + C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \hat{\Delta}_i(R_i) \tag{11}$$

Proof: The WCRT of τ_i happens when (i) it executes for its WCET, (ii) it suffers its maximum blocking time, (iii) it suffers the maximum interference by higher priority tasks and (iv) it suffers the maximum preemption cost.

(i) is given by C_i. (ii) is given by Equation (10) in Lemma 3. (iv) is given by Equation (7) in Corollary 1. Only (iii) must still be quantified. The maximum number of jobs any higher priority task τ_j can release within a time window of length R_i is $\lceil R_i/T_j \rceil$. The maximum interference they can therefore generate is $\lceil R_i/T_j \rceil C_j$. (iii) is thus upper bounded by $\sum_{j=1}^{i-1} \lceil R_i/T_j \rceil C_j$.

Summing all those terms, we get Equation (11), which proves the theorem. ■

Note that Equation (11) is recursive. It can be solved with a fixed point iteration on R_i by initializing R_i to $B_i + C_i$.

B. Schedulability Test for Harmonic and Loose Harmonic Tasks

In harmonic and loose harmonic task sets every release of a job is synchronized to the releases of the jobs of the highest priority task τ_1. This simple fact introduces some interesting properties in RS-LP that results in a reduced number of preemptions in a time window of length t (see Lemma 4) and limits the maximum blocking time that is actually suffered by a task τ_i due to lower priority tasks delaying their preemptions (see Lemma 5). This in turn allows us to reduce the pessimism in the computation of the maximum tolerable blocking time β_i (see Equation 14) and the WCRT of task τ_i (see Theorem 4).

To avoid any confusion with the generic formulas of the previous section, in this section, we append the superscript ^{h} to each quantity (i.e., $\mathcal{S}_i$, $\hat{P}_i$, $\hat{\Delta}_i(t)$, β_i, B_i and R_i) which is related to harmonic (and loose harmonic) tasks.

Hence, we define $\mathcal{S}_i^h$ as the set of tasks $\tau_k \in hp_i \setminus \tau_1$ with a maximum blocking time β_k^h (defined by Equation (14) below) smaller than $(T_1 - C_1)$, i.e., $\beta_k^h < (T_1 - C_1)$. Note that $\mathcal{S}_i^h$ considers only the tasks with a tolerable blocking time smaller than $(T_1 - C_1)$. The set $\mathcal{S}_i$ defined in the previous section for generic periodic tasks, was however containing all the tasks with a maximum tolerable blocking time smaller than $2(T_1 - C_1)$. This difference in their respective definitions is important as it impacts the computation of the total number of preemptions and thus the total preemption cost as demonstrated below.

Lemma 4. *With RS-LP, in a time window of length t where only loose harmonic tasks in $hp_i \cup \tau_i$ execute, the total number of preemptions is upper bounded by*

$$\hat{P}_i^h(t) = \min \left\{ \left\lceil \frac{t}{T_1} \right\rceil, \ \left\lceil \frac{t}{2T_1} \right\rceil + \sum_{\tau_k \in \mathcal{S}_i^h} \left\lceil \frac{t}{T_k} \right\rceil \right\} \tag{12}$$

Proof: The only difference between Equation (6) and (12) resides in the definition of $\mathcal{S}_i^h$. The sum on the elements in $\mathcal{S}_i^h$ is therefore the only term we will discuss in this proof. All the other terms were already proven in point 1. (i.e., $\hat{P}_i^h(t)$ is upper bounded by $\lceil t/T_1 \rceil$) and points 2.a. and 2.b. (i.e., at most $\lceil t/(2T_1) \rceil$ preemptions are initiated by τ_1) of the proof of Lemma 2. However, point 2.c. of Lemma 2's proof must be revisited.

When a task τ_k releases a job before the end of the execution segment of a task τ_i already running on the processor, Algorithm 2 is called. If τ_k has a higher priority than τ_i and its maximum tolerable blocking time β_k^h is smaller than the remaining time σ_i until the end of the current execution segment of τ_i, the execution segment of τ_i is shortened and one more preemption is generated.

From Line 3 of Algorithm 1, the maximum length of an execution segment of τ_i is $a_1^{next} + (T_1 - C_1) - t$. Since the releases of τ_k are synchronized on the releases of τ_1, the earliest time at which a job of τ_k can be released is a_1^{next}. It results that the maximum remaining time until the end of the current execution segment of τ_i at a_1^{next} (and thus at the release of τ_k) is $(T_1 - C_1)$. Hence, only tasks in $\mathcal{S}_i^h$ (i.e., higher priority tasks such that $\beta_k^h < (T_1 - C_1)$) can cause a reschedule shortening the execution segment of τ_i. At most $\sum_{\tau_k \in \mathcal{S}_i^h} \lceil t/T_k \rceil$ jobs can be released by the tasks in $\mathcal{S}_i^h$ within a window of length t. The number of preemptions initiated by other tasks than τ_1 is thus upper bounded by $\lceil t/(2T_1) \rceil + \sum_{\tau_k \in \mathcal{S}_i^h} \lceil t/T_k \rceil$.

Combining this last result with the fact that at most $\lceil t/(2T_1) \rceil$ preemptions are initiated by τ_1 (see points 2.a. and 2.b. in the proof of Lemma 2), we get $\lceil t/(2T_1) \rceil + \sum_{\tau_k \in \mathcal{S}_i^h} \lceil t/T_k \rceil$ as an upper bound on $\hat{P}_i^h(t)$. And because it was already proven (point 1. of the proof of Lemma 2) that $\lceil t/(T_1) \rceil$ is also an upper bound on $\hat{P}_i^h(t)$, the claim follows. ■

Corollary 2. *In a time window of length t where only loose-harmonic tasks in $hp_i \cup \tau_i$ execute , an upper bound on the total preemption cost is given by*

$$\hat{\Delta}_i(t) = \hat{P}_i^h(t) \times \max_{k=2}^{i} (\Delta_k) \tag{13}$$

Proof: Identical to the proof of Corollary 1. ■

Replacing the total preemption $\hat{\Delta}_i(t)$ by $\hat{\Delta}_i^h(t)$ in Equation (9), we get the following lower bound on the maximum tolerable blocking time for loose harmonic tasks

$$\beta_i^h = \max_{C_i < t \le T_i} \left\{ t - \left(\hat{\Delta}_i^h(t) + \sum_{k=1}^{i} rbf_k^*(t) \right) \right\} \tag{14}$$

Using Equation (9) together with Algorithm 2, we now quantify the maximum blocking time that is actually suffered by a task τ_i scheduled with RS-LP when all tasks are loose harmonic.

Lemma 5. *In a loose-harmonic task set, the maximum blocking time suffered by any job released by τ_i due to* *lower priority tasks, is upper bounded by*

$$B_i^h = \begin{cases} 0 & \textit{if } i = n \textbf{\textit{ or}} \\ & \beta_i^h < (T_1 - C_1) \\ \min\left\{(T_1 - C_1),\ \max\limits_{j=i+1}^{n}(C_j)\right\} & \textit{otherwise} \end{cases} \tag{15}$$

Proof: Since τ_n is the lowest priority task, it cannot suffer any blocking due to lower priority tasks. Therefore, we have $B_n^h = 0$.

Furthermore, from Line 3 of Algorithm 1, the maximum length of an execution segment is $a_1^{next} + (T_1 - C_1) - t$. Since the releases of τ_i are synchronized on the releases of τ_1, the earliest time at which a job of τ_i can be released after a lower priority task τ_j started running, is a_1^{next}. It follows that the maximum remaining time until the end of the current execution segment of τ_j at a_1^{next} (and thus at the release of τ_i) is $(T_1 - C_1)$. Two cases must be considered: **Case 1.** $\beta_i < (T_1 - C_1)$. In this case, Algorithm 2 preempts τ_j at time a_1^{next}, which happens to be the release time of τ_i. Therefore, τ_i never suffers any blocking due to τ_j when $\beta_i < (T_1 - C_1)$. This proves the first case of Equation (15). **Case 2.** $\beta_i \geq (T_1 - C_1)$, in which τ_j keeps running and the maximum blocking time that τ_i can suffer due to τ_j, is upper bounded by $(T_1 - C_1)$.

Moreover, because Algorithm 1 always selects the highest priority task to be scheduled when a job completes an execution segment, at most one job of a lower priority task can block τ_i. For any other tasks than τ_n, the blocking time is thus upper bounded by $\max\limits_{j=i+1}^{n}(C_j)$. Combining this with the conclusion of case 2 above, we get that $B_i^h \leq \min\left\{(T_1 - C_1),\ \max\limits_{j=i+1}^{n}(C_j)\right\}$ which proves the second case of Equation (15) ■

Theorem 4. *In a loose harmonic task set, an upper bound on the WCRT of a task τ_i scheduled with RS-LP is given by the smallest value R_i^h larger than C_i such that*

$$R_i^h = B_i^h + C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^h}{T_j} \right\rceil C_j + \hat{\Delta}_i^h(R_i^h) \tag{16}$$

Proof: The proof is identical to the proof of Theorem 3, replacing B_i and $\hat{\Delta}_i(t)$ with B_i^h and $\hat{\Delta}_i^h(t)$, respectively. ■

VI. Discussion on the Number of Preemptions

As mentioned in [15], [22] a necessary condition for the schedulability of non-preemptive periodic tasks is

$$\forall i, 1 < i \leq n,\ C_i \leq 2(T_1 - C_1) \tag{17}$$

meaning that the WCET of any low priority task τ_i, $1 < i \leq n$ must not be larger than two times the slack of τ_1. Based on this observation, we derive the lower bound on the number of preemptions of *any* scheduling algorithm which does not miss a deadline as follows

Theorem 5. *Let $C_{i,j}$ be the actual execution time of a job $J_{i,j}$ of task τ_i ($1 < i \leq n$). If all the jobs released by the tasks in τ respect their deadlines, then the number of preemptions suffered by $J_{i,j}$ is lower bounded by*

$$P(C_{i,j}) = \left\lfloor \frac{C_{i,j}}{2(T_1 - C_1)} \right\rfloor - \begin{cases} 1 & C_{i,j}/(2(T_1 - C_1)) \in \mathbb{N} \\ 0 & \textit{otherwise} \end{cases} \tag{18}$$

Proof: The proof is by contradiction; we show that if $J_{i,j}$ has a smaller number of preemptions than $P(C_{i,j})$, then at least one task in the system will miss its deadline. Assume that $J_{i,j}$ suffers $P_i' < P_i$ preemptions (where $P_i = P(C_{i,j})$ and $P_i' = P'(C_{i,j})$). It means that it will have one non-preemptive execution segment which has a length larger than $2(T_1 - C_1)$, i.e., that execution segment has length $l = 2(T_1 - C_1) + \epsilon$ where $\epsilon \in \mathbb{R}^{>0}$. Let us call that execution segment S_i. Assume that the last release of τ_1 before the execution of S_i was at t. Two cases can happen: τ_1 starts its execution before that execution segment of τ_i, thus, τ_i can be scheduled from $t + C_1$ and ends at $t + C_1 + 2(T_1 - C_1) + \epsilon$. It means that the next job of τ_1 released at $t + T_1$ cannot meet its deadline. In the second case, τ_1 is not yet scheduled when S_i starts. Even in the best-case where S_i start its execution at time t, the first instance of τ_1 released at time t will miss its deadline because $l > t + T_1$. Thus, in both cases, at least one job will miss its deadline if $J_{i,j}$ has a non-preemptive execution segment with length l. As a result, the assumption is not valid, and $J_{i,j}$ cannot have a smaller number of preemptions than $P(C_{i,j})$. ■

If the preemption overhead is considered, the worst-case execution time of the task when it suffers the minimum number of preemptions is given by C_i' such that

$$C_i' = C_i + P(C_i') \times \Delta_i \tag{19}$$

where C_i' is the smallest value larger than C_i which satisfies the equation. Equation (20) can be solved with a fixed point iteration initiating C_i' to C_i. The following two theorems directly follow from that fact.

Theorem 6. *If all the jobs released by the tasks in τ respect their deadlines, then the number of preemptions suffered by τ_i is lower bounded by $P_i^{lb} = P(C_i')$ where*

$$C_i' = C_i + P(C_i') \times \Delta_i \tag{20}$$

Using Equation (20) we derive a necessary condition on the feasibility of a task set τ composed of periodic (or sporadic) tasks executing on a uniprocessor platform in the presence of preemption overhead.

Theorem 7. *Let Δ_i be the preemption cost suffered by $\tau_i \in \tau$ whenever it is preempted by another task, then the task set τ is feasible on a uniprocessor platform only if*

$$\sum_{i=1}^{n} \frac{C_i'}{T_i} \leq 1 \tag{21}$$

where C_i' is obtained from Equation (20).

Proof: This claim follows directly from the feasibility test proposed by Liu and Layland in [2] where instead of the WCET in isolation, we use C_i' which is the smallest

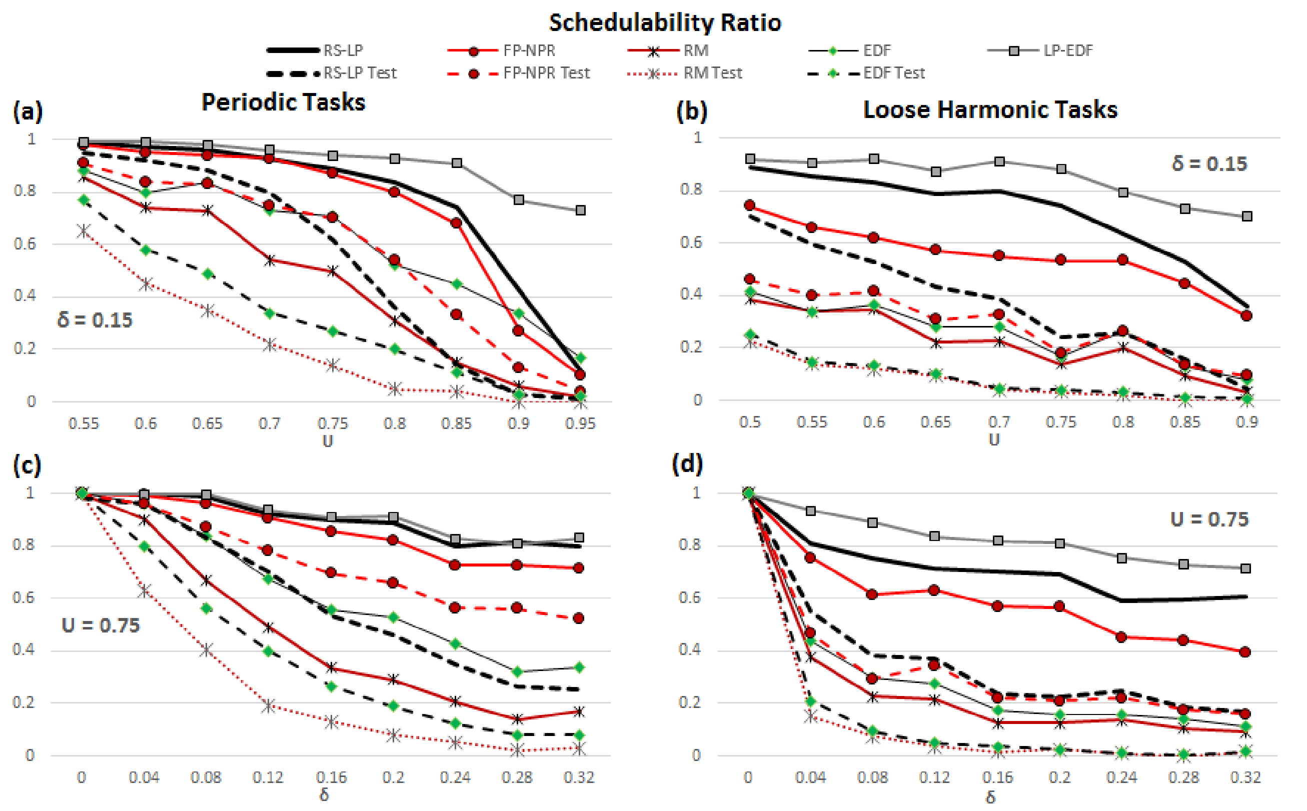

Figure 4. Schedulability ratio as a function of U and δ. In (a) and (b), the horizontal axis is U and δ is set to 0.15 while in (c) and (d) the horizontal axis is δ and U is set to 0.75. Diagrams (a) and (c) are for periodic tasks while (b) and (d) are for loose-harmonic tasks.

worst-case execution time in the presence of preemption overhead. ■

Although Theorem 7 provides a necessary feasibility condition with linear complexity, proving that there exists a feasible schedule such that each job suffers its minimum number of preemptions, is an intractable problem as proven below.

Theorem 8. *Verifying the feasibility of a periodic task set τ with implicit deadlines such that every job of every task τ_i does not suffer more than $P(C'_i)$ preemptions, is a NP-hard problem.*

Proof: The proof is based on the fact that if $P_i(C'_i) = 0$ for all tasks, each task must be scheduled non-preemptively. And, as stated in [23], verifying the feasibility of a non-preemptive periodic task set with implicit deadline is a NP-hard problem. ■

VII. Experimental Results

In this section, we present the experimental results obtained for different scheduling algorithms including RM, EDF, FP-NPR, LP-EDF and RS-LP with randomly generated task sets. The schedule of each task set is simulated over its hyper-period to assess its schedulability. The schedulability ratio of each algorithm, i.e., the number of schedulable task sets over the total number of generated task sets, is then reported in Fig. 4.

In the presented graphs, we also include the schedulability ratio of the existing schedulability tests for the studied scheduling algorithms. For our algorithm, we use Theorem 4 as the schedulability test for loose harmonic task sets and Theorem 3 for periodic task sets. For RM we use the test presented in [24], for FP-NPR we use the test in [9], and for EDF we use the utilization based test presented in [25] (Equations (18) and (19) in [25]).

It is worth noting that EDF and LP-EDF are in the class of dynamic priority scheduling algorithms, which are known to have better schedulability performance than fixed priority scheduling algorithms; particularly if the preemption overhead is negligible. However, we decided to include EDF and LP-EDF in our comparisons so as to get an idea on the gap that still exists between dynamic and fixed priority algorithms when the preemption overhead is taken into account. Yet, one should consider that the real opponents to RS-LP are FP-NPR and RM.

In the conducted experiments, we consider the effect of the task set utilization (denoted by U) and the maximum preemption cost, which is computed as a percentage (denoted by δ) of the WCET of the tasks. Due to space limitation, we could not show the influence of the number of tasks n on the schedulability ratio. However, according to our experiments, the schedulability ratio did not vary much for values of n larger than 4. Therefore, we decided to fix the number of tasks to 8.

We used uUniFast [18] to generate random task sets. We produced $n = 8$ random values $u_i \in [0, 1]$ with a total utilization U. For periodic task sets, periods have been selected randomly in the interval $[10, 500]$ following the same approach suggested in [18]. Note that we only accepted task sets with $T_2 \geq 2T_1$ because then with a very high chance, β_2 becomes smaller than $T_1 - C_1$, which causes deadline misses for τ_2 in RS-LP. For loose-harmonic task sets, we first assigned a random period $T_i \in [1, 10]$ to τ_1, and then generated $n-1$ random values $k_i \in \{2, 3, \ldots, 500\}$ with a uniform distribution. Hence, we defined T_i as $T_i = k_i \times T_1$. Using u_i and T_i, we computed C_i. Finally, we generated the preemption cost Δ_i by selecting a random value $x_i \in [0, \delta]$, and assigning $\Delta_i = \min\{x_i \times C_i, \Delta\}$ where Δ is an upper-bound on the time required to reload the whole cache content. In our experiments, we have assumed $\Delta = 50$. For each data point in the resulting diagrams, we generated 200 task sets.

Fig. 4 shows the schedulability ratios of the scheduling algorithms and their respective schedulability tests for periodic (left column), and loose-harmonic (right column) tasks as a function of U (top row) and the preemption overhead (bottom row). As expected, RM and EDF have considerably lower schedulability ratios than limited preemptive approaches, as they suffer large number of preemptions. Those results are even worse with loose-harmonic tasks. This can be explained by the fact that almost each release of τ_1 during the execution of a low priority job causes a preemption in RM and EDF. However, most of those preemptions could have been postponed until the end of the slack of τ_1. As one can see, the schedulability tests for RM and EDF perform poorly as they over-estimate the number of preemptions. In some cases, e.g., for $U = 0.75$, $\delta > 0.08$ in Fig. 4-(c), RM's test accepts at most 15% out of 90% of the schedulable task sets. If we compare the results of RM's schedulability test with the schedulability tests presented in Section V, we accept almost 2 times more task sets than RM. A more readable comparison of the performances of the schedulability tests can be seen in Fig. 5 for loose harmonic tasks (the parameter is δ and the utilization is 0.75).

As it has been shown in Fig. 4, RS-LP shows a better schedulability ratio than FP-NPR, particularly in loose harmonic task sets (Fig. 4-(b) and (d)). As explained earlier in Sect. IV and V-B, in loose-harmonic task sets tasks are released together with the release of τ_1, hence, the maximum non-preemptive region obtained by Yao et al. [9] in Equation (5) is exact as long as $\beta_i \geq T_1 - C_1$, since no counter example such as Fig. 2-(a) can happen. Thus, RS-LP is able to create larger non-preemptive segments by fitting the execution segments within the slack of two consecutive jobs of τ_1. It directly affects the performance of the test in Theorem 4. As a result, we expect that the schedulability of FP-NPR and RS-LP becomes similar in loose harmonic task sets, however, we still see a clear difference between their performance. This difference comes from the situations where some tasks have $\beta_i \leq T_1 - C_1$. If that happens, the length of Q_i in FP-NPR is reduced according to Equation (5). However, since β_i is calculated according to the worst-case scenario, it is the minimum safe value of blocking tolerance of the jobs of τ_i. In other words, the real blocking tolerance for each job can be larger than β_i as

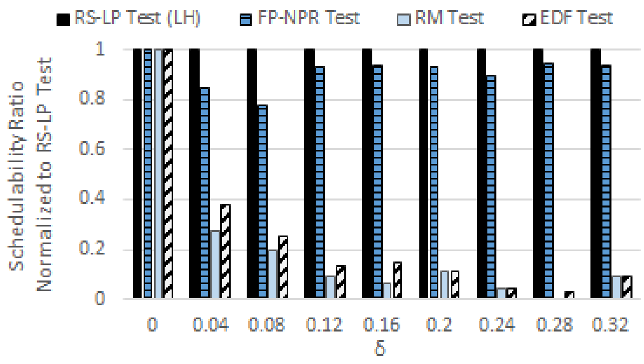

Figure 5. Normalized schedulability ratio of the schedulability tests with respect to RS-LP test (Theorem 4) as a function of δ for loose harmonic tasks.

it depends on the releases of other high priority jobs. FP-NPR ignores this fact and preempts a job after Q_i units of NPR, even if the job can continue its execution. As a result, it creates smaller execution segments which are not synchronized with the releases of τ_1 anymore. Although RS-LP has better schedulability ratio than FP-NPR, it does not dominate it in terms of schedulability because RS-LP may miss a job if β_i is smaller than the time until the next release of τ_1.

VIII. Conclusion

In this paper we have introduced a new limited preemptive scheduling solution based on fixed priorities in order to increase schedulability of the systems with preemption overheads. In the presented method, we try to increase the maximum non-preemptive execution segment length by fitting the execution segments in the continuous slack of two consecutive releases of the task with the smallest period. Consequently, the tasks can be executed with smaller number of preemptions. Furthermore, whenever a higher priority task is released, we decide wether to preempt the low priority task, based on the maximum blocking tolerance of the high priority task. Yet, even if we decide to preempt the running task, the preemption is postponed until the next release of the task with the smallest period. Thus, we keep the execution segments synchronized with the releases of the highest priority task, so as to optimize the overall number of preemptions.

We have provided two schedulability tests one for periodic and one for loose-harmonic task sets. We have derived a feasibility condition based on a proven lower bound on the number of preemptions, and shown that the exact feasibility analysis to check the existence of a schedule which can guarantee the minimum number of preemptions is a NP-Hard problem.

Our experimental results show that our approach has a higher schedulability ratio than the existing limited preemptive fixed-priority scheduling algorithms including FP-NPR, which is one of the most efficient methods in the class of fixed-priority scheduling algorithms. Similarly, the schedulability test for harmonic task sets shows very good performances against its opponents. Yet, we plan to improve the performance of our schedulability test for periodic tasks as future work.

Acknowledgements

This work was partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within project UID/CEC/04234/2013 (CISTER); also by FCT/MEC and the EU ARTEMIS JU within projects ARTEMIS/0003/2012 - JU grant nr. 333053 (CONCERTO) and ARTEMIS/0001/2013 - JU grant nr. 621429 (EMC2).

References

[1] G. C. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems: A survey," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, 2013.

[2] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[3] C.-G. Lee, J. Hahn, Y.-M. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *IEEE Transactions on Computers*, vol. 47, no. 6, pp. 700–713, 1998.

[4] R. Pellizzoni and M. Caccamo, "Toward the Predictable Integration of Real-Time COTS Based Systems," in *Real-Time Systems Symposium (RTSS)*, 2007, pp. 73–82.

[5] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach," in *Dependable Computing Conference (EDCC)*, 2010, pp. 67–72.

[6] "ARINC Specification 653-P1: Avionics Application Software Standard Interface, Required Services," 2015. [Online]. Available: https://www.arinc.com/cf/store/index.cfm

[7] O. Sokolsky, M. Xu, J. Lee, L. T. X. Phan, and I. Lee, "Overhead-aware Compositional Analysis of Real-time Systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE Computer Society, 2013, pp. 237–246.

[8] S. Altmeyer, R. I. Davis, and C. Maiza, "Improved Cache Related Pre-emption Delay Aware Response Time Analysis for Fixed Priority Pre-emptive Systems," *Real-Time Systems, Springer*, vol. 48, no. 5, pp. 499–526, 2012.

[9] G. Yao, G. Buttazzo, and M. Bertogna, "Feasibility analysis under fixed priority scheduling with limited preemptions," *Real-Time Systems*, vol. 47, no. 3, pp. 198–223, 2011.

[10] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *International Conference on Real-Time Computing Systems and Applications (RTCSa)*, 1999, pp. 328–335.

[11] R. J. Bril, M. M. van den Heuvel, and J. J. Lukkien, "Improved feasibility of fixed-priority scheduling with deferred preemption using preemption thresholds for preemption points," in *International conference on Real-Time Networks and Systems (RTNS)*. ACM, 2013, pp. 255–264.

[12] S. Baruah, "the limited-preemption uniprocessor scheduling of sporadic task systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2005, pp. 137–144.

[13] M. Bertogna and S. Baruah, "Limited Preemption EDF Scheduling of Sporadic Task Systems," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 579–591, 2010.

[14] J. Lee and K. Shin, "Preempt a Job or Not in EDF Scheduling of Uniprocessor Systems," *IEEE Transactions onComputers*, vol. 63, no. 5, pp. 1197–1206, 2014.

[15] M. Nasri and M. Kargahi, "Precautious-RM: a predictable non-preemptive scheduling algorithm for harmonic tasks," *Real-Time Systems*, vol. 50, no. 4, pp. 548–584, 2014.

[16] M. Nasri and G. Fohler, "Non-Work-Conserving Scheduling of Non-Preemptive Hard Real-Time Tasks Based on Fixed Priorities," in *International Conference on Real-Time Networks and Systems (RTNS)*. ACM, 2015.

[17] S. Altmeyer and C. Maiza, "Cache-related preemption delay via useful cache blocks: Survey and redefinition," *Journal of Systems Architecture*, vol. 57, no. 7, pp. 707–719, 2011.

[18] E. Bini and G. Buttazzo, "Schedulability analysis of periodic fixed priority systems," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1462–1473, 2004.

[19] G. C. Buttazzo, "Rate monotonic vs. edf: Judgment day," *Lecture Notes in Computer Science*, vol. 2855, pp. 67–83, 2003.

[20] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., 1994.

[21] N. Guan, M. Stigge, W. Yi, and G. Yu, "New response time bounds for fixed priority multiprocessor scheduling," in *30th IEEE Real-Time Systems Symposium, (RTSS)*. IEEE Computer Society, 2009, pp. 387–397.

[22] Y. Cai and M. C. Kong, "Nonpreemptive scheduling of periodic tasks in uni- and multiprocessor systems," *Algorithmica*, vol. 15, no. 6, pp. 572–599, 1996.

[23] K. Jeffay, D. F. Stanat, and C. U. Martel, "On non-preemptive scheduling of period and sporadic tasks," in *IEEE Real-Time Systems Symposium (RTSS)*, 1991, pp. 129–139.

[24] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, and A. Wellings, "Adding instruction cache effect to schedulability analysis of preemptive real-time systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 1996, p. 204212.

[25] W. Lunniss, R. I. Davis, C. Maiza, and S. Altmeyer, "Integrating Cache Related Pre-emption Delay Analysis into EDF Scheduling," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE Computer Society, 2013, pp. 75–84.