Cache-aware Scheduling with Limited Preemptions

Marko Bertogna, Giorgio Buttazzo, Mauro Marinoni, Gang Yao, Francesco Esposito
Scuola Superiore Sant’Anna
Pisa, Italy
Email: {name.surname}@sssup.it

Marco Caccamo
University of Illinois at Urbana Champaign
Urbana, IL
Email: mcaccamo@illinois.edu

Abstract—In safety-critical applications, the use of advanced real-time scheduling techniques is significantly limited by the difficulty of finding tight estimations of worst-case execution parameters. This problem is further complicated by the use of cache memories, which reduce the predictability of the executing threads due to cache misses.

In this paper, we analyze the effects of preemptions on worst-case execution times, taking into account the main factors that affect task response times, such as context switches, cache misses, and bus contention. In particular, a limited preemption model with fixed preemption points is proposed as a viable scheduling strategy to improve the schedulability and the predictability of a given task system. The proposed method provides information on where to place preemption points in the task code to obtain a feasible schedule. A trade-off between preemptive and non-preemptive scheduling is investigated, to balance the increased blocking caused by non-preemptive sections and the beneficial reduction of the cache miss ratio.

I. INTRODUCTION

Most theoretical results on schedulability analysis have been derived assuming a preemption cost equal to zero. Under such an ideal case, preemptive scheduling is often more efficient than non-preemptive scheduling, because of the additional blocking time that can be introduced by the non-preemptive execution of lower priority tasks. In practice, however, preemptions can introduce a significant runtime overhead and may cause high fluctuations in task execution times, so degrading system predictability. In particular, the following types of costs must be taken into account at each preemption:

1) Scheduler cost. It is due to the time taken by the scheduling algorithm to suspend the running task, insert it into the ready queue, switch the context, and dispatch the new incoming task.
2) Pipeline cost. It is due to the time taken to flush the processor pipeline when the task is interrupted and the time taken to refill the pipeline when the task is resumed.
3) Cache-related cost. It is due to the time taken to reload the cache lines evicted by the preempting task. This time depends on the specific point in which preemption occurs and on the number of preemptions experienced by the task [16], [13], [1].

4) Bus contention cost. It is due to the Front Side Bus (FSB) conflicts caused by the extra memory accesses due to cache misses. In fact, whenever data are not found in the cache, they have to be fetched from RAM, using the FSB. Hence, contentions can occur when the FSB is used by I/O peripheral devices through a DMA transfer [19], [18].

These effects are not negligible at all, and may contribute to a great share of the overall worst-case execution time (WCET). To overcome such problems, some authors investigated limited preemption models that can be used to reduce the negative effects of context switches, while limiting the amount of blocking due to non-preemptive regions [25], [7], [2], [26]. From another side, other authors extended the schedulability analysis of preemptive scheduling to take context switch overhead into account [10], [27]. The problem of selecting preemption points in order to achieve the schedulability of the system did not instead receive much attention.

Indeed, such problem is not easy to solve, since it is characterized by a circular dependency. In fact, when taking the overhead into account, the WCET becomes a function of the number preemptions, but the number of preemptions (or equivalently the maximum size of the non-preemptive sections) can only be computed when WCETs are given.

In this paper we analyze the effects of preemptions on worst-case execution times, understanding how they affect the schedulability of the system. Then, a limited preemption model with fixed preemption points is proposed. The advantage of this model is that it is in line with the current practice adopted in critical software development, so that the derived results can be applied to real applications.

A trade-off between preemptive and non-preemptive scheduling is investigated, to balance the increased blocking caused by non-preemptive sections and the beneficial reduction of the cache miss ratio. Finally, we propose a method for automatically selecting the most suitable preemption points in the code in order to guarantee the schedulability of the system.
II. RELATED WORK

A. Non-Preemptive and Limited Preemption scheduling

Non-preemptive EDF scheduling has been studied by Jeffay et al. [14], who showed that EDF is optimal even among non-preemptive work-conserving schedulers\(^1\) for periodic and sporadic task sets. For these systems, a necessary and sufficient schedulability test with pseudo-polynomial complexity was provided. Moreover, it was shown that, for concrete periodic task systems scheduled by non-preemptive algorithms\(^2\), feasibility analysis is NP-hard in the strong sense.

Baruah and Chakraborty [3] analyzed the schedulability of non-preemptive task sets under the recurring task model and showed that there exist polynomial time approximation algorithms for both preemptive and non-preemptive scheduling.

Wang and Saksena [25] proposed a different approach for limiting preemptions, in systems scheduled with FP. Each task is assigned a regular priority and a preemption threshold, and it is allowed to preempt only when its priority is higher than the threshold of the preempted task.

Burns [7] extended the response time analysis to verify the schedulability of fixed priority tasks with fixed preemption points, but did not address the problem of selecting the best location of the preemption points to improve the schedulability of the task set. His work has been later improved by Bril et al. [6].

Baruah introduced limited preemption scheduling for EDF [2], computing the maximum amount of time for which a task may execute non-preemptively without missing any deadline. Yao et al. [26] extended Baruah’s work to fixed priority systems.

B. Cache-aware scheduling

The problem of finding a correct WCET estimation for real-time task sets has been considered by many authors (see [12] for a good survey). The analysis of the Cache-Related Preemption Delay (CRPD) has also been addressed in various papers considering systems scheduled with a fully-preemptive algorithm.

In [8] and [20], two methods have been presented to integrate the classic Response Time Analysis with the penalties associated with CRPD, adding a fixed context-switch cost. A complex but more precise analysis considering common sets of data between preempting and preempted tasks has been described in [11]. With a similar target, Staschulat et al. [24] provided safe estimations of the CRPD, analyzing the intersection between the set of useful data — locations that might be accessed again by a preempted task — and used data — locations that might be accessed by the preemtping task.

In [21], a bound was provided on the Data Cache Related Preemption Delay (D-CRPD), identifying additional data-cache misses due to context switches. Response Time Analysis was then used to check the system schedulability, using the derived bound on the worst-case execution times. In [22], such a bound was tightened by refining the estimation of the maximum number of preemptions a task may experience, considering both best-case and worst-case execution times of higher priority tasks. In a recent work [23], the same authors extended the analysis to tasks having at most one non-preemptive region with a given position inside the task code.

While most of the above works were based on systems scheduled with Fixed Priority, Ju et al. [15] considered the CRPD computation problem for systems scheduled with preemptive EDF.

C. Improvements over previous works

In this paper, we consider the problem of scheduling a set of real-time tasks consisting of a sequence of Non-Preemptive Regions (NPR) separated by Preemption Points (PP). The proposed method helps a designer in selecting the best preemption points inside each task code, exploiting the available slack in the system to reduce the number of preemptions of some selected tasks, without imposing too much blocking on higher priority tasks. The final objective is to achieve a feasible schedule when the task set is not feasible in non-preemptive mode (due to high blocking times), nor in fully preemptive mode (due to the high overhead).

As shown in [26], [4], limited preemption schedulers can significantly reduce the total number of preemptions with respect to fully preemptive algorithms. This happens because the allowed length of non-preemptive execution of a task is often larger than or comparable to that task execution time.

However, directly applying previous theoretical results, like the ones in [2], [4], [26], is not so straightforward, since computing the maximum lengths of non-preemptive regions requires the knowledge of worst-case execution times, which in turn are significantly influenced by the number of preemptions.

In this paper, we show how to deal with such a circular dependency, proposing an iterative algorithm that considers both problems at the same time. Our approach is based on limited preemption model, like the one proposed in [23]. However, there are significant differences between the two approaches. First of all, each task in [23] can have at most one NPR, and, since the complexity of the proposed approach is exponential, it is rather difficult to adapt such method to the limited preemption model considered in this paper. Moreover, the presence of NPRs is considered just to comply with a more general model (i.e., the presence

---

\(^1\)A scheduling algorithm is work-conserving if the processor is never idled when a task is ready to execute. Note that EDF is not optimal among general non-preemptive schedulers (including non work-conserving ones).

\(^2\)A concrete periodic task is a periodic task that comes with an assigned initial activation.
of critical sections of code in which the task must not be preempted) and it is not leveraged to increase the task set schedulability.

The rest of the paper is organized as follows. In Section III, we will present the adopted system model and terminology. Section IV describes a schedulability analysis for task sets scheduled with limited preemption EDF or FP. In Section V, we will show an algorithm to achieve the schedulability of a task set with a proper placement of preemption points inside each task code. In Section VI, we will present some considerations on the proposed method. The effectiveness of such method will be evaluated by a set of simulations, shown in Section VII. Finally, we will draw our conclusions in Section VIII.

### III. System Model

In this paper, we consider a set $\tau$ of $n$ periodic and sporadic real-time tasks that are scheduled on a single processor using either a fixed priority algorithm (FP) or Earliest Deadline First (EDF) [17]. Each task $\tau_i$ is defined by a worst-case execution requirement $C_i$, a relative deadline $D_i$ and a period, or minimum interarrival time, $T_i$. Each task generates an infinite sequence of jobs, with the first job arriving at any time and successive job-arrivals separated by at least $T_i$ time units. We assume that tasks are ordered by decreasing priorities in the FP case, and by increasing relative deadlines in the EDF case, i.e., $\forall i \mid 0 \leq i < n : D_i \leq D_{i+1}$.

Each job of $\tau_i$ consists of a sequence of $p_i$ non-preemptive chunks of code. Preemption is allowed only between chunks by inserting proper preemption points. The $j$-th chunk of task $\tau_i$ is denoted by $\delta_{i,j}$, $1 \leq i \leq n$, $1 \leq j \leq p_i$, and its worst-case execution time by $q_{i,j}$. The maximum chunk length for $\tau_i$ is $\delta_{i}^{\max} = \max\{q_{i,j}\}_{j=1}^{p_i}$.

The memory footprint $F_i$ of a task $\tau_i$ is the cumulative size of the individual memory locations accessed by $\tau_i$ during its execution. A task repeatedly accessing the same set of data will have a smaller footprint than a task accessing multiple different memory locations.

We assume the processor can take advantage of a dedicated cache, of size $L$, from which recently used data and instructions can be loaded. We say that a cache is “hot” if a requested data is present in the cache; otherwise, the cache is “cold”. The cache miss penalty due to the time taken to load data from the main memory to the cache is denoted by $\gamma$. To simplify the analysis, we assume this value is the same for every memory location accessed by each task in the set. Moreover, we ignore any timing anomaly in the cache behavior, assuming each miss increases the observed execution time by $\gamma$.

Finally, we do not take advantage of the positive cache effects due to the subsequent execution of concurrent tasks accessing similar sets of data, nor to the limited number of cache evictions performed by a preempting job with reduced footprint (smaller than the cache size). In other words, we assume the cache being completely cold after any context switch.

The introduced notation is summarized in Figure 1.

#### A. Worst-case execution times

The worst-case execution time $C_i$ of a task $\tau_i$ is the largest amount of processor time a job of $\tau_i$ might need to successfully complete its execution. To perform a precise schedulability analysis, this parameter must include all overhead costs identified in the introduction, and can be expressed as the sum of the net computation time $E_i$, (achieved when all accessed data are always in the cache) plus the overhead.

In particular, the maximum number of cache misses a task $\tau_i$ may experience in the worst-case scenario is denoted by $\mu_i^{\max}$, and it is equal to the maximum number of memory accesses a job of $\tau_i$ may perform. In fact, this is the only bound that can be given when no information is available on the adopted scheduler, nor on the tasks concurrently scheduled with $\tau_i$.

When a particular scheduler is assumed, the estimation of the real number of cache misses may be refined. We call $\mu_i$ the maximum number of cache misses a task $\tau_i$ may experience using a given scheduling algorithm. For instance, using preemptive EDF or FP it has been shown [9] that the number of preemptions on a job of task $\tau_i$ is bounded by the number of higher priority jobs that can be released in $[0, D_i]$, decreasing the number of potential cache misses in the worst-case. When $\tau_i$ is executed non-preemptively, $\mu_i$

\begin{table}[h]
\centering
\begin{tabular}{|c|l|}
\hline
Symbol & Description \\
\hline
$\tau$ & Task set \\
$\tau_i$ & $i$-th task $\in \tau$ \\
$n$ & Number of tasks in the task set \\
$\delta_{i,j}$ & $j$-th chunk of task $\tau_i$ \\
$p_i$ & Number of chunks of task $\tau_i$ \\
$C_i$ & WCET of $\tau_i$ in presence of cache misses \\
$E_i$ & WCET value when the cache is always hot \\
$D_i$ & Relative deadline of $\tau_i$ \\
$T_i$ & Period or minimum interarrival time of $\tau_i$ \\
$q_{i,j}$ & Worst-case execution time of chunk $\delta_{i,j}$ \\
$q_i^{\max}$ & Largest non-preemptive execution of $\tau_i$ \\
$U_i$ & $C_i/T_i$, utilization of $\tau_i$ \\
$U$ & $\sum_{i=1}^{n}(U_i)$, total utilization of task set $\tau$ \\
$\mu_i$ & Worst-case number of cache misses of $\tau_i$ \\
$\mu_i^{\max}$ & Maximum $\mu_i$ among all possible schedulers \\
$\mu_i^{\NP}$ & $\mu_i$ value when $\tau_i$ executes non-preemptively \\
$L$ & Cache size \\
$F_i$ & Memory footprint of task $\tau_i$ \\
$\gamma$ & Cache miss penalty \\
$\sigma$ & Penalty due to load/store the task state \\
$\pi$ & Penalty due to pipeline invalidation \\
$\eta(x)$ & I/O induced delay for $x$ cache misses \\
\hline
\end{tabular}
\caption{Notation used throughout the paper.}
\end{table}
Figure 2. Example of cache accesses: (H) cache Hit, (I) Intrinsic miss, (E) Extrinsic miss.

has the smallest possible value $\mu_i^{\text{NP}}$. Hence, the following relation holds:

$$\mu_i^{\text{NP}} \leq \mu_i \leq \mu_i^{\text{max}}.$$ 

Note that $\mu_i$ depends on the number $p_i$ of non-preemptive regions in which $\tau_i$ is divided. The smaller $p_i$, the fewer the cache misses experienced by $\tau_i$. In fact, each context switch might evict the cache locations commonly accessed by two subsequent chunks. To understand that, consider the example shown in Figure 2, where the memory accesses of the first two chunks of a task $\tau_i$ are shown. The first chunk loads into the cache the memory locations corresponding to $a$, $b$ and $c$. When the second chunk starts executing after a potential preemption, another task might have overwritten the cache content, evicting data commonly accessed by $\delta_{i,1}$ and $\delta_{i,2}$. Therefore, $\delta_{i,2}$’s first accesses to $a$ and $c$ should be accounted as misses. To clarify which misses are due to a possible preemption and which are not, we distinguish between Intrinsic and Extrinsic cache misses. A miss is intrinsic if it occurs independently of the preemption, i.e., when a task accesses a memory location for the first time, or when the miss is caused by a self-eviction\(^4\). An extrinsic miss is instead due to evictions caused by preempting tasks.

As already mentioned in the introduction, the overhead due to cache misses is not the only delay experienced after a preemption, but there is also the scheduler cost $\sigma$, the pipeline cost $\pi$, and the FSB contention cost $\eta(\mu_i)$.

Since there are $p_i$ non-preemptive chunks, the total number of times a job of $\tau_i$ may be preempted is $(p_i - 1)$. Hence, the overall worst-case execution time of $\tau_i$ results to be

$$C_i = E_i + \gamma \mu_i + (\pi + \sigma)(p_i - 1) + \eta(\mu_i).$$

In the next sections, we show how to reduce such a value, by minimizing the number $p_i$ of preemption points inside the code of each task $\tau_i$, as well as the resulting number of cache misses $\mu_i$.

\(^4\)A self-eviction is an eviction performed by the task itself. This can happen whenever the task footprint is larger than the cache size.

IV. Schedulability Analysis

In this section, we present a unified analysis of EDF and FP scheduling under the limited preemption model, reformulating the results derived in [26] (for FP) and in [2], [4] (for EDF), using a homogeneous notation.

For the feasibility analysis under FP, we use the request bound function $\text{RBF}_i(a)$ in an interval $a$, defined as

$$\text{RBF}_i(a) = \left\lceil \frac{a}{T_i} \right\rceil C_i.$$

Under EDF, the analysis is carried out by the demand bound function $\text{DBF}_i(a)$ in an interval $a$, defined as

$$\text{DBF}_i(a) = \left(1 + \frac{a - D_i}{T_i}\right) C_i.$$ 

Moreover, we conventionally set $D_{n+1}$ equal to the minimum between: (i) the least common multiple (lcm) of $T_1, T_2, \ldots, T_n$, and (ii) the following expression\(^5\):

$$\max \left(D_n, \frac{1}{1 - U} \cdot \sum_{i=1}^{n} U_i \cdot \max \left(0, T_i - D_i\right)\right).$$

The largest blocking $B_i$ that a task $\tau_i$ might experience is given, under both FP and EDF, by the length of the largest non-preemptive chunk belonging to tasks with index higher than $i$:

$$B_i = \max_{i < k \leq n+1} \{q_k^{\text{max}}\},$$

where $q_k^{\text{max}} = 0$ by definition. Summarizing the results presented in [26], [2], [4], the next theorem derives a schedulability condition under limited preemptions, for FP and EDF.

Theorem 1. A task set $\tau$ is schedulable with limited preemption EDF or FP if, for all $i \mid 1 \leq i \leq n$,

$$B_i \leq \beta_i,$$  

where, under FP, $\beta_i$ is given by

$$\beta_i^{\text{FP}} = \max_{a \in A|a \leq D_i} \left\{a - \sum_{\tau_j \leq i} \text{RBF}_j(a)\right\},$$

with

$$A = \{kT_j, k \in \mathbb{N}, 1 \leq j < n\},$$

whereas, under EDF, $\beta_i$ is given by

$$\beta_i^{\text{EDF}} = \min_{a \in A|D_i \leq a < D_i+1} \left\{a - \sum_{\tau_j \in \tau} \text{DBF}_j(a)\right\},$$

with

$$A = \{kT_j + D_j, k \in \mathbb{N}, 1 \leq j \leq n\}.$$ 

\(^5\)The expression may in general be exponential in the parameters of $\tau$; however, it is pseudo-polynomial if the system utilization is a priori bounded from above by a constant less than one.
The following theorem presents a different schedulability condition, expressed in terms of a bound \( Q_k \) on the longest non-preemptive region \( q_k^{\text{max}} \) of each task \( \tau_k \).

**Theorem 2.** A task set \( \tau \) is schedulable with limited preemption EDF or FP if, for all \( k \leq n+1 \),

\[
q_k^{\text{max}} \leq Q_k \equiv \min_{1 \leq i < k} \{ \beta_i \},
\]

(6)

where \( \beta_i \) is given by Equation (4) in the FP case, and by Equation (5) in the EDF case.

*Proof:* A sufficient schedulability condition can be obtained combining Theorem 1 with Equation (2):

\[
\bigwedge_{1 \leq i \leq n} \left( \max_{1 \leq k \leq n+1} \{ q_k^{\text{max}} \} \leq \beta_i \right).
\]

The inner inequality can be rewritten as a system of inequalities, as follows:

\[
\bigwedge_{1 \leq i \leq n} \left( \bigwedge_{1 \leq k \leq n+1} \{ q_k^{\text{max}} \} \leq \beta_i \right).
\]

Developing the system, it is possible to obtain

\[
\bigwedge_{1 \leq k \leq n+1} \left( \bigwedge_{1 \leq i \leq n} \{ q_k^{\text{max}} \} \leq \beta_i \right),
\]

which is equivalent to

\[
\forall k \mid 1 < k \leq n + 1 : \quad q_k^{\text{max}} \leq \min_{1 \leq i < k} \{ \beta_i \},
\]

proving the theorem.

Note that the definition of \( Q_k \) can be rewritten in the following iterative form (starting with \( Q_1 = \infty \), for all \( 1 < k \leq n + 1 \):

\[
Q_k = \min\{Q_{k-1}, \beta_{k-1}\}.
\]

(7)

We hereafter prove that the sufficient schedulability condition of Theorem 2 is also necessary under EDF. Suppose the test fails. Consider a \( q_k \) for which condition (6) evaluates to false, i.e.,

\[
q_k^{\text{max}} > \min_{i < k} \{ \beta_i \} = \min_{a \in A} \{ a \mid D_1 \leq a < D_k \} \left\{ a - \sum_{\tau_j \in \tau} \text{DBF}_j(a) \right\}.
\]

Consider the point \( a^* \in A \) that minimizes the RHS of the above inequality. Then, \( q_k^{\text{max}} > a^* - \sum_{\tau_j \in \tau} \text{DBF}_j(a^*) \), and

\[
q_k^{\text{max}} + \sum_{\tau_j \in \tau} \text{DBF}_j(a^*) > a^*.
\]

(8)

Consider a situation in which:

- all tasks with relative deadline \( \leq a^*(< D_k) \) start synchronously at \( t = 0 \);
- task \( \tau_k \) enters its largest NPR of length \( q_k^{\text{max}} \) an arbitrarily small amount of time before \( t = 0 \). Since \( \tau_k \) is the only task executing before \( t = 0 \), it will always be possible to build such a situation.

In the above conditions, the total demand in \([0, a^*)\) is equal to the LHS of Equation (8). Therefore, the total demand exceeds the length of the interval, leading to a deadline miss. Therefore, if the test of Theorem 2 fails, it means that the task set is not schedulable with limited preemption EDF.

Under FP, instead, the test is necessary and sufficient only when no information is available on the location of each non-preemptive region, as in the “floating” NPR model adopted in [26]. When instead the position of the (last) NPR of each task is known — i.e., under the “fixed” NPR model — the theorem is only sufficient. An exact test could be derived significantly complicating the analysis, adopting techniques described in [6].

V. PROPOSED APPROACH

As explained in Section III, limiting preemptions may significantly reduce the number of cache misses \( \mu_i < < \mu_i^{\text{max}} \), as well as the negative effects of context switches, with a beneficial effect on the worst-case timing behavior. On the other side, however, limiting preemptions increases the blocking delay on higher-priority jobs, possibly jeopardizing the task set schedulability.

In this section, we present a method for placing preemption points inside the code of each task. In particular, the number and the position of preemption points will be derived as a function of the task parameters and the major sources of overhead, with the objective of improving the task set schedulability.

The algorithm starts by analyzing the feasibility of the task set when preemption is disabled. If the task set is not schedulable in non-preemptive mode, the algorithm searches for preemption points that generate a feasible schedule, if there exists one.

A. Worst-case parameters computation

From Equations (4) and (5) it is clear that the value of \( \beta_i \) depends on the worst-case execution times \( C_j \), which are significantly influenced by the number of cache misses of each task \( \tau_j \). From Equation (1), it is possible to see that \( C_j \) has a fixed component (equal to \( E_j \)) and a variable component that depends on the total number of preemptions and cache misses. While intrinsic cache misses cannot be avoided by any scheduling policy, extrinsic cache misses can be reduced by decreasing the number of preemptions, adopting a proper scheduling policy. However, large non-preemptive regions increase blocking delays; hence finding the best preemption points (PPs) analytically is rather difficult, due to the interdependencies between PPs and worst-case execution times, as well as between extrinsic cache misses and data reusing patterns among different sections of code. To simplify the problem, we assume that each PP in a task \( \tau_k \) causes a fixed overhead \( \xi_k \).
Under this assumption, we present a method that optimally exploits the schedulability test of Theorem 2 to place PPs inside the code of each task, in order to achieve a schedulable condition. We initially assume that PPs can be inserted anywhere in the code of each task, relaxing this assumption in Section VI.

The proposed algorithm can be summarized with the following steps.

1. The algorithm starts with no preemption points for each task, i.e., setting \( p_i = 1 \) and \( q_i^{\text{max}} = C_i^{\text{NP}} \), \( \forall i \), where \( C_i^{\text{NP}} \) is the worst-case execution time of \( \tau_i \) when it executes non-preemptively. This value can be found using timing analysis tools, without needing to take into account preemptions.

2. Then, \( \beta_i \) is computed by Equations (4) or (5) for increasing indexes, that is, starting from \( \beta_1 \). Note that \( \beta_i \) depends on the \( C_j \) of tasks with indexes \( j \leq i \), given by \( C_j^{\text{NP}} + (p_j - 1) \xi_j \).

3. Then, \( Q_{i+1} \) is computed from \( \beta_{k \leq i} \) using Theorem 2.

4. If \( Q_{i+1} \) is smaller than the maximum non-preemptive region of \( \tau_{i+1} \), procedure \( \text{PPLACE}(Q_{i+1}, i + 1) \) is invoked to place the least number of PPs in \( \tau_{i+1} \) to guarantee \( q_{i+1}^{\text{max}} \leq Q_{i+1} \). This is achieved by placing a first PP after \( Q_{i+1} \) time-units of (non-preemptive) execution from the beginning of \( \tau_{i+1} \). To account for the preemption overhead, further PPs are placed after every \( (Q_{i+1} - \xi_{i+1}) \) time-units, until the end of the code is reached.

5. If \( \text{PPLACE}(Q_{i+1}, i + 1) \) returns false, the algorithm stops, declaring the task set infeasible. The failing condition of \( \text{PPLACE}(Q_k, k) \) is \( (Q_k \leq \xi_k) \). This is because, if \( Q_k \leq \xi_k \), then the execution time available to \( \tau_k \) is entirely dedicated to the preemption overhead.

6. When all \( Q_i \) values have been successfully checked, the algorithm returns, having guaranteed the schedulability of the task set.

The pseudo-code of the algorithm is summarized in Figure 3. We hereafter prove the correctness of procedure \( \text{INSERTPP}(\tau) \) in deriving a schedulable condition. Then, we will show some optimality properties of the adopted method.

**Theorem 3.** Procedure \( \text{INSERTPP}(\tau) \) is correct.

**Proof:** If the procedure succeeds, each \( Q_k \) will be larger than or equal to the maximum non-preemptive region \( q_k^{\text{max}} \) of each task \( \tau_k \), \( i \leq k \leq n \), and \( \beta_n \geq Q_{n+1} \geq 0 \). Note that, both in the EDF and in the FP cases, the \( \beta_i \) value computed at line 3 of the algorithm depends only on \( C_j \) values with \( j \leq i \) (as well as on deadlines and periods, which cannot change). Since none of these values may change in the next iterations (because PPs are inserted only into the code of tasks \( \tau_{k > i} \)), all \( \beta_i \), and therefore \( Q_i \), are correctly computed. By Theorems 1 and 2, the correctness of the procedure is assured.

---

**INSERTPP(\( \tau \))**

Tasks ordered by non-increasing relative deadline.

Initialize: \( \{q_i^{\text{max}} \leftarrow C_i^{\text{NP}}\}_{i=1}^n \), \( q_n^{\text{max}} = 0 \), \( \{p_i \leftarrow 1\}_{i=1}^n \), and \( Q_1 \leftarrow \infty \).

1. for \( (i : 1 \leq i \leq n) \)
2. \( C_i \leftarrow C_i^{\text{NP}} + (p_i - 1) \xi_i \)
3. Compute \( \beta_i \) using Equation (4) or (5);
4. \( Q_{i+1} \leftarrow \min\{Q_i, \beta_i\} \)
5. if \( (q_{i+1}^{\text{max}} > Q_{i+1}) \)
6. if \( (\text{PPLACE}(Q_{i+1}, i+1) = \text{false}) \)
7. return (Infeasible)
8. return (Feasible)

**PPLACE(\( Q_k, k \))**

Let \( \xi_k \) be the preemption overhead of \( \tau_k \), \( 1 \leq k \leq n \), and \( \xi_n+1 \leftarrow 0 \).

1. if \( (Q_k \leq \xi_k) \) return false
2. Place a PP in \( \tau_k \) at \( Q_k \) and after every \( (Q_k - \xi_k) \) time-units.
3. \( p_i \leftarrow \left[ \frac{C_i^{\text{NP}} - Q_k}{Q_k - \xi_k} \right] + 1 \)
4. \( q_k^{\text{max}} \leftarrow Q_k \)
5. return (true)

---

Having proved the correctness of \( \text{INSERTPP}(\tau) \), we now show that the PP placement is optimal under EDF scheduling, meaning that if the algorithm fails, then any other possible PP placement leads to an infeasible schedule.

**Theorem 4.** Procedure \( \text{INSERTPP}(\tau) \) is optimal under EDF.

**Proof:** Suppose, by contradiction, there is a feasible task set \( \tau \) for which procedure \( \text{INSERTPP}(\tau) \) fails. Then, there is at least one task \( \tau_k \) for which procedure \( \text{PPLACE}(Q_k, k) \) fails. Let \( \tau_k \) be the task with the smallest index for which the procedure fails, and let \( \beta_k \) be the value that minimizes \( Q_k \), i.e., \( i = \arg\min_{j=1}^{k-1}(\beta_j) \). As previously mentioned, \( \beta_i \) is a function of the worst-case execution times, deadlines and periods of all tasks \( \tau_{j \leq i} \). While the latter values \( (D_j \text{ and } T_j) \) are fixed, the execution times \( C_j \) may vary for different placements of PPs in the code of each task \( \tau_j \). Note that \( \beta_i \) is a decreasing function of all \( C_{j \leq i} \).

We now prove by induction that procedure \( \text{INSERTPP}(\tau) \) allows finding the smallest values \( C_{j \leq i} \), among PP allocation strategies that are feasible. Therefore, if \( \tau \) is feasible, the largest possible \( \beta_i \) is found with \( \text{INSERTPP}(\tau) \). If such a value is too small (or even negative), so that no PP placement can be found for a task \( \tau_{k > i} \) to satisfy \( q_k^{\text{max}} \leq \beta_i \), then this latter condition will be violated by any other possible strategy, since it cannot lead to a larger \( \beta_i \). Therefore, \( \tau \) is not feasible, reaching a contradiction.

---
Base case: Independently of the number of PPs, task \( \tau_1 \) is always executed non-preemptively, both under FP and under EDF. Note that procedure \( \text{INSERTPP}(\tau) \) does not insert any PP in \( \tau_1 \), leading to the smallest possible value of \( C_1 \).

Inductive step: Let \( j \leq i \). Assume \( \text{INSERTPP}(\tau^{(j-1)}) \) obtained the schedulability of the reduced task set \( \tau^{(j-1)} = \{\tau_1, \ldots, \tau_{j-1}\} \), minimizing the worst-case execution times \( C_1, \ldots, C_{j-1} \). We will prove that \( \text{INSERTPP}(\tau^{(j)}) \) obtains as well the schedulability of the set \( \tau^{(j)} = \tau^{(j-1)} \cup \{\tau_j\} \), minimizing \( C_j \).

It is easy to see that the PP allocation strategy for tasks \( \tau_1, \ldots, \tau_{j-1} \) is the same for \( \text{INSERTPP}(\tau^{(j-1)}) \) and \( \text{INSERTPP}(\tau^{(j)}) \), so that there is no change in any \( C_k \) or \( q_k^{\text{max}} \), for \( 1 \leq k \leq j-1 \). Since \( \tau^{(j-1)} \) was schedulable, the schedulability of \( \tau^{(j)} \) can be obtained, by Theorem 2, if \( q_j^{\text{max}} \leq Q_j = \min\{\beta_{k<j}\} \). By the inductive hypothesis, the procedure obtains the largest possible values for each \( \beta_{k<j} \) (since \( C_1, \ldots, C_{j-1} \) are minimized). Hence, no other possible PP allocation for tasks \( \tau^{(j-1)} \) can result in a larger \( Q_j = \min\{\beta_{k<j}\} \). Since \( Q_j \) is a tight bound on the maximum NPR length in the EDF case, procedure \( \text{PPLACE}(Q_j,j) \) places the least possible number of PPs to \( \tau_j \). Since we assumed that each PP of \( \tau_j \) causes the same overhead \( \xi_j \), procedure \( \text{PPLACE}(Q_j,j) \) obtains the smallest possible \( C_j \), proving the statement.

Note that the optimality of procedure \( \text{INSERTPP}(\tau) \) depends on (i) the tightness of the \( Q_k \) bounds computed with Equation (6), and (ii) the assumption on the identical preemption overheads of the PPs of each task. Regarding the first point, as we explained in Section IV, condition (6) is necessary and sufficient only in the EDF case. In the FP case, instead, condition (6) is tight only when the floating NPR model is adopted. Otherwise, a larger bound \( Q_k \) could be derived considering the exact location of the last NPR of \( \tau_k \). However, this would imply a much more complex analysis, which is beyond the scope of this paper.

Regarding point (ii), the assumption on the preemption cost of each PP might be relaxed, using a more complex timing analysis that considers data reusing patterns inside the code of each task. Tighter estimations of the preemption costs might be derived in this way, leading to an improved placement of PPs. Moreover, the worst-case execution time of each task could be further reduced analyzing how many PPs can effectively cause a preemption. If a task \( \tau_j \) has a small \( \beta_j \) value, all lower priority tasks will have frequent PPs. However, they cannot be preempted by \( \tau_j \) more than once every \( T_j \) time units. Therefore, it may happen that most PPs won’t lead to a preemption. To account for this fact, the worst-case execution time of a task \( \tau_i \) at line 1 of procedure \( \text{INSERTPP}(\tau) \) can be replaced by tighter expressions, derived adapting techniques from [24], [22], [23] to the limited preemption scheduling model adopted in this paper. Again, we believe this to be a very interesting problem, that we intend to address in a future work. In the current paper, we instead assumed a fixed overhead for all preemption points of each task. We will show in our simulations that this assumptions is not overly pessimistic, since the number of inserted PPs is typically very small, even for heavily loaded systems.

VI. CONSIDERATIONS

The placement of PPs inside each task code is subject to constraints such as atomic instructions, critical sections and non-preemptible sections of code in general. Moreover, requiring a task to be decomposable into a sequence of non-preemptive chunks of execution can be a too strong assumption, since task systems are generally better modeled by a tree structure with loops and branches. An optimal placement of PPs would then require to go through each branch in the execution tree of a task, considering the data and instructions accessed along each path.

To simplify the problem, we will assume a set of Potential Preemption Points (PP) be given, each one separating the execution of two consecutive non-preemptive chunks, called Basic Blocks (BB), forming a serial chain of BBs for each task \( \tau_k \). Each loop, conditional branch or non-preemptable section of code is assumed to be entirely contained inside a BB. The most suitable positions where to place a PPP can be chosen examining the task code. Smaller preemption overheads \( \xi_k \) can be found inserting each PPP between sections of code that access few common memory locations. The "minimum execution granularity" \( \Delta_k \) is the maximum "execution distance" between any two consecutive PPs, including the preemption delay \( \xi_k \).

Procedure \( \text{DISCPPPLACE}(Q_k,k) \), whose pseudocode is shown in Figure 4, can be used instead of \( \text{PPLACE}(Q_k,k) \) when a set of PPs is given. The procedure will try to minimize the number of PPs that will be used for the insertion of an actual Preemption Point (PP), without violating the condition on the blocking (\( q_k^{\text{max}} \leq Q_k \)). To do that, subsequent basic blocks will be progressively combined, starting from the first one, as long as the worst-case execution time of the resulting NPR, including the preemption delay \( \xi_k \), is smaller than \( Q_k \). When the addition of the next basic block would cause the resulting WCET to exceed \( Q_k \), a new NPR is initiated with this block, and a PP is inserted immediately before it. Note that no preemption delay is accounted for the first NPR, since \( BB_k \) is decreased by \( \xi_k \). The procedure continues until all BBs have been assigned to a NPR. The failing condition is when the allowed \( Q_k \) is smaller than the minimum execution granularity.
The only modification needed to procedure INSERTPP is at line 6, where procedure PPLACE should be replaced by DISCPPLACE.

As explained in Section III-A, the preemption delay \( \xi_k \) of each task \( \tau_k \) can be bounded by \( (\pi + \sigma) \), plus the cache related preemption delay, which is a function of the number of extrinsic cache misses experienced by \( \tau_k \) when resuming its execution. Such number cannot be larger than (i) the total number of memory locations accessed by \( \tau_k \), equal to its footprint \( F_k \), and (ii) the total number of cache lines, equal to \( L \). Every other memory access is either a hit, or an intrinsic cache miss, which is not due to preemptions. The total overhead introduced by each preemption on a task \( \tau_k \) is therefore smaller than

\[
\xi_k = \pi + \sigma + \gamma \min\{F_k, L\} + \eta(\min\{F_k, L\}).
\]

Although the above bound is rather pessimistic, its effect on the final worst-case execution time is often limited, since the number of preemption points is typically very small, as we will show in our simulations.

The complexity of the proposed approach is pseudopolynomial, both in the EDF and in the FP case. We believe the main complexity lies in finding good estimations of the \( C_i^{NP} \) values that are needed as inputs for procedure INSERTPP. Anyway, timing analysis tools are much more efficient in finding worst-case estimations of these non-preemptive execution times, rather than when needing to consider a preemptive situation.

VII. EXPERIMENTAL RESULTS

In this section, experimental results are presented based on simulations. Randomly generated task sets were used to evaluate the effectiveness of the proposed limited-preemptive policy (procedure INSERTPP) in comparison with non-preemptive and fully preemptive algorithms. We randomly generated one thousand task sets and measured the effectiveness of each scheduling policy, analyzing the percentage of schedulable task sets as a function of total system utilization.

More in detail, each task set was generated as follows. The UN1F2 algorithm, described in [5], was used to generate each set of \( n \) tasks with a given total utilization \( U_{tot} \). Each execution time \( E_k \) was generated as a random integer value uniformly distributed in \([50, 150]\), computing \( T_k \) as \( T_k = E_k/U_k \). The relative deadline \( D_k \) was generated as a random integer value within the range \([E_k + 0.8 \cdot (T_k - E_k), T_k]\).

Due to space reasons, we include here only the results for fixed priority scheduling. The simulations for EDF are rather similar. We considered four fixed priority scheduling policies: non-preemptive (NP), limited-preemptive (LP), fully preemptive without preemption cost (FP w/o cost) and fully preemptive with preemption cost (FP with cost). Task sets schedulability under FP without cost was calculated by using the classical response time analysis, setting \( C_k = E_k \), for all tasks \( \tau_k \in \tau \). It represents an (ideally) optimal scenario for fixed-priority, since it has the minimum possible blocking, with no overhead. Schedulability under NP was verified using the test of Theorem 2, with \( q_{max}^k = C_k = E_k \) for each task \( \tau_k \). When considering FP with preemption cost, we used the equation of classical response time analysis, adding a fixed cost for each preemption. As shown in [8], the response time of \( \tau_k \) is given by the smallest fixed point of:

\[
R_k = \sum_{j \leq k} \left[ \frac{R_j}{T_j} \right] (E_j + \text{cost}).
\]

As in the classical analysis, we iterated the above equation until either convergence was reached or the response time exceeded the deadline.

The preemption cost is a crucial parameter when evaluating the effectiveness of the proposed LP policy. We selected the range of values for this parameter by analyzing the impact of the last level of cache on a typical partition size for an avionic system compliant to ARINC-653\(^7\) (scheduling partitions can be as small as \(2ms\)). Considering the widely used PowerPC processor MPC7410 (with \(2MB\) two-way associative L2 cache), it would take about \(655\mu s\) to reload the whole L2 cache; hence, in such scenario, execution time increment due to cache interference could be as big as \(655\mu s/2ms \approx 33\% \). We chose to show here the cases for a cost value between 5% and 20% of the average worst-case execution time \( \bar{E} = \sum_{\tau_k \in \tau} E_k/n \).

The results are shown in Figure 5. The first three histograms show the cases with \( n = 10 \) tasks and a cost of, respectively, 5% (a), 10% (b) and 20% (c). As it is clear from the plotted graphs, NP and FP-without-cost policies are not affected by a variation of preemption cost. The

\(^7\)http://www.arinc.com/
The superiority of LP policy over FP-with-cost is evident under all three scenarios. Notice that the LP model achieves a better schedulability ratio even when the preemption cost is as low as 5%. The performance of FP-with-cost deteriorates very quickly as the preemption cost increases, while LP is always close to the ideal case of FP w/o cost.

In histogram (d), the preemption cost is set to 10%, as in (c), while the number of tasks is increased to $n = 20$. The performance of FP-with-cost slightly deteriorates, while LP improves in terms of percentage of schedulable task sets. This is expected, because when the number of tasks increases, each task has a larger slack, hence it can tolerate a larger blocking. Therefore, less preemption points are needed in the task code. A similar argument can be applied to non-preemptive scheduling: larger slacks and smaller worst-case execution times imply that a task set is more likely to be feasible.

VIII. Conclusions

We presented an efficient algorithm to obtain the schedulability of a task set in a real-time system scheduled with FP or EDF, using the limited preemption model. A proper number of preemption points is placed inside the code of each task, in order to guarantee the feasibility of the task set, reaching an optimal compromise between small blocking times and reduced preemption overhead. The positive outcomes of the proposed method are not limited to the improved schedulability of the system, but include as well a more predictable behavior of all tasks, reducing the number of locations in which they can be preempted, and simplifying the timing analysis.

As a future work, we plan to find tighter estimations on the preemption delays experienced under the limited preemption model, bounding the number of PPs that can effectively lead to a preemption, and refining the estimation of the overhead.
REFERENCES


