Preemption points placement for sporadic task sets

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*—Limited preemption scheduling has been introduced as a viable alternative to non-preemptive and fully-preemptive scheduling when reduced blocking times need to coexist with an acceptable context switch overhead. To achieve this goal, preemptions are allowed only at selected points of the code of each task, decreasing the preemption overhead and simplifying the estimation of worst-case execution parameters. Unfortunately, the problem of how to place these preemption points is rather complex and has not been solved.**

In this paper, a method is presented for the optimal placement of preemption points under simplifying conditions, namely, a fixed preemption overhead at each point. We will prove that if our method is not able to produce a feasible schedule, then no other possible preemption point placement (including non-preemptive and fully preemptive scheduling) can find a schedulable solution. The presented method is general enough to be applicable to both EDF and Fixed Priority scheduling, with limited modifications.

I. Introduction

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. To simplify the problem, 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, therefore 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 [18], [12], [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 [21], [20].

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 [29], [26], [8], [2], [32]. From another side, other authors extended the schedulability analysis of preemptive scheduling to take context switch overhead into account [11], [33]. The problem of selecting preemption points in order to improve the schedulability of the system has been preliminarily considered in [13] and [17].

Indeed, such a problem is not easy to solve in an optimal way, since it is characterized by a circular dependency. In fact, when considering the context switch overhead in the schedulability analysis, the WCET of a task becomes a function of the number of preemptions it might be subject to; but the number of preemptions depends on its turn by the WCET of the task—the longer a task executes, the more it will be preempted—complicating the analysis.

In this paper we will show how to deal with such circular dependency, when a limited preemption model with fixed preemption points is adopted. The advantage of this model is that it is in line with the current practice adopted in critical software development [8], so that the derived results can be applied to real applications. We will present a method for automatically selecting the most suitable preemption points in the code of each task in order to guarantee the schedulability of the system. The analysis will consider on one hand the increased blocking caused by non-preemptive sections, and on the other hand the beneficial reduction of the preemption overhead. We will prove that the proposed algorithm is optimal when each preemption point is assumed to produce an identical overhead. Even if this assumption could appear rather restrictive, we introduced it to establish the mathematical background for more complex models

without violating the strict page limit of this submission. As preliminarily shown in [31], the presented analysis can be integrated with data collected by timing analysis tools, for the handling of more realistic models with a variable preemption overhead.

The proposed approach is general enough to be applicable to the most used scheduling algorithms, such as EDF and FP. We will show how existing results on limited preemption scheduling can be extended and integrated under a common notational model, in order to derive the necessary information for the optimal placement of preemption points. To comply with more general requirements, we will also analyze the case in which preemption points can be inserted only at a discrete number of points. This will allow our algorithm to deal with user-defined non-interruptible sections of code, as well as avoid complex protocols for access to shared resources. In fact, when it is possible to encapsulate each critical section within a non-preemptive region, a task will never be preempted while holding a lock, solving any mutual exclusion problem in the access to shared resources.

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, an exact 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, deriving polynomial time approximation algorithms for both preemptive and non-preemptive scheduling.

Wang and Saksena [29] 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. This work has been later improved by Regehr in [26].

Burns [8] extended the response time analysis to verify the schedulability of fixed priority tasks with fixed preemption points. His work has been later improved by Bril et al. [7].

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.* [32] extended Baruah's work to fixed priority systems.

B. Preemption overhead

The problem of finding a correct WCET estimation for real-time task sets has been considered in many different papers in the timing analysis domain (see [30] for a good survey). When a preemptive scheduler is adopted, a critical factor in the estimation of a task's WCET is represented by the Cache-Related Preemption Delay (CRPD).

In [9] and [22], 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 [16]. With a similar target, Staschulat *et al.* [28] 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 preempting task. The appropriate selection of preemption points for an easier computation of the CRPD has been addressed in [27].

In [23], 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. This bound was then refined in [24]. In a recent work [25], 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, 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 [32], [4], limited preemption schedulers can significantly reduce the total number of preemptions with respect to fully preemptive algorithms. This happens because the allowed non-preemptive execution length of a task is often larger than or comparable to that task's execution time. However, existing theoretical results on limited preemption scheduling [2], [4], [32] have been derived neglecting the cost of preemptions. Integrating these results with the preemption overhead is not so straightforward, since computing the maximum lengths of the non-preemptive regions requires the knowledge of worst-case execution times, which in turn are significantly influenced by the number of context switches. 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. Earlier attempts

[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.

to reduce context switching overhead delaying preemptions have been presented in [13] and [17].

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's code. In Section VI, we will present some considerations on the proposed method. The effectiveness of this method will be evaluated through a set of simulations, shown in Section VII. Finally, we will draw our conclusions in Section VIII.

III. System Model

We consider a set τ 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) [19]. Each task τ_i is defined by a worst-case execution requirement C_i, a period, or minimum interarrival time, T_i, and a relative deadline $D_i \leq T_i$. 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. The utilization U_i of task τ_i is defined as C_i/T_i. The total utilization U of task set τ is the sum of the utilizations of all tasks in τ. We assume that tasks are ordered by decreasing priorities in the FP case, and by increasing relative deadlines in the EDF case, i.e., $\forall i \mid 0 < i < n : D_{i-1} \leq D_i$. Tasks are either supposed to be independent, or their critical sections are assumed to be entirely contained within a non-preemptive region[3].

Each job of τ_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 τ_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 τ_i is $q_i^{\max} = \max\{q_{i,j}\}_{j=1}^{p_i}$.

The *memory footprint* F_i of a task τ_i is the cumulative size of the individual memory locations accessed by a job of τ_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 "cold" if it does not contain any useful data; otherwise, the cache is "hot". A cache that is always hot is referred to as an "ideal cache". The cache miss penalty due to the time taken to load data from the main memory to the cache is denoted by γ. To simplify the analysis, we assume this value to be 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 γ. Finally, we assume the cache being completely *cold* after any context switch. In other words, 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).

These restricting assumptions will be removed in a future work, where less pessimistic estimations of the CRPD will be considered[4]. Before complicating the model, this paper intends to presents the preliminary results that are needed for a more thorough analysis.

A. Worst-case execution times

The worst-case execution time C_i of a task τ_i is the largest amount of processor time a job of τ_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 such penalties.

In particular, the maximum number of cache misses a task τ_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 τ_i may perform. Indeed, 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 τ_i.

When a particular scheduler is assumed, the estimation of the real number of cache misses may be refined. We call μ_i the maximum number of cache misses a task τ_i may experience using a given scheduling algorithm. For instance, with preemptive EDF or FP it has been shown [10] that the number of preemptions on a job of task τ_i is bounded by the number of higher priority jobs that can be released in $[0, D_i)$[5], decreasing the number of potential cache misses in the worst-case. When τ_i is executed non-preemptively, μ_i has the smallest possible value μ_i^{NP}. Hence, the following relation holds:

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

Note that μ_i depends on the number p_i of non-preemptive regions in which τ_i is divided. The smaller p_i, the fewer the cache misses experienced by τ_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 1, where the memory accesses of the first two chunks of a task τ_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

[3] As critical sections are typically very short [6], they are likely to be accommodated inside a non-preemptive region. When this is not true, some shared resource protocol needs to be adopted.

[4] Some insights of this future work can be found in [31]

[5] See [11], [28], [24], [33] for tighter bounds in the number of preemptions.

Chunk	Code	Hit/Miss
$\delta_{i,1}$	`access(a)`	I
	`access(b)`	I
	`access(c)`	I
	`access(a)`	H
	`access(c)`	H
$\delta_{i,2}$	`access(a)`	E
	`access(d)`	I
	`access(a)`	H
	`access(c)`	E
	`access(d)`	H
	...	

Figure 1. Example of cache accesses: (H) cache Hit, (I) Intrinsic miss, (E) Extrinsic miss.

Symbol	Description
$\delta_{i,j}$	j-th chunk of task τ_i
p_i	Number of chunks of task τ_i
C_i	WCET of τ_i in presence of cache misses
C_i^{NP}	WCET of τ_i when it executes non-preemptively
E_i	WCET value with an ideal cache
$q_{i,j}$	WCET of chunk $\delta_{i,j}$
$q_i^{\max}$	Largest non-preemptive execution of τ_i
μ_i	Worst-case number of cache misses of τ_i
$\mu_i^{\max}$	Maximum μ_i among all possible schedulers
μ_i^{NP}	μ_i value when τ_i executes non-preemptively
L	Cache size
F_i	Memory footprint of task τ_i
γ	Cache miss penalty
σ	Penalty due to load/store the task state
π	Penalty due to pipeline invalidation
$\eta(x)$	I/O induced delay for x cache misses

Figure 2. Notation used throughout the paper.

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[6]. An extrinsic miss is instead due to evictions caused by preempting tasks.

As already mentioned in the introduction, there are also other kinds of penalties associated to each preemption, like the scheduler cost σ, the pipeline cost π, and the FSB contention cost $\eta(\mu_i)$. Since there are p_i non-preemptive chunks, the total number of times a job of τ_i may be preempted is $(p_i - 1)$. Hence, the overall worst-case execution time of τ_i results to be

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

In the next sections, we will present a method to decrease this value by minimizing the number p_i of preemption points inside the code of each task τ_i, resulting in a smaller number of cache misses μ_i.

For convenience, all notations are summarized in Figure 2.

IV. Schedulability analysis

In this section, we present a unified analysis of EDF and FP scheduling under the limited preemption model, extending and reformulating the results derived in [32] (for FP) and in [2], [4] (for EDF), under a common notational model.

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

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

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[7]:

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

The largest blocking B_i that a task τ_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:

$$B_i = \max_{i<k\leq n+1}\{q_k^{\max}\}, \tag{2}$$

where $q_{n+1}^{\max} = 0$ by definition. Summarizing the results presented in [32], [2], [4], the next theorem derives a schedulability condition under limited preemptions, for FP and EDF.

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

$$B_i \leq \beta_i, \tag{3}$$

where, under FP, *β_i is given by*

$$\beta_i^{\text{FP}} \doteq \max_{a\in A | a\leq D_i}\left\{a - \sum_{j\leq i} \text{RBF}_j(a)\right\}, \tag{4}$$

with

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

whereas, under EDF, *β_i is given by*

$$\beta_i^{\text{EDF}} \doteq \min_{a\in A | D_i\leq a<D_{i+1}}\left\{a - \sum_{\tau_j\in\tau} \text{DBF}_j(a)\right\}, \tag{5}$$

with

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

The following theorem presents a different schedulability condition, expressed in terms of a bound Q_k on the longest non-preemptive region $q_k^{\max}$ of each task τ_k.

[6] A *self-eviction* is an eviction performed by the task itself. This can happen whenever the task footprint is larger than the cache size.

[7] The expression may in general be exponential in the parameters of τ; however, it is pseudo-polynomial if the system utilization is a priori bounded from above by a constant less than one.

Theorem 2. *A task set τ is schedulable with limited preemption* EDF *or* FP *if, for all $k \mid 1 < k \leq n+1$,*

$$q_k^{\max} \leq Q_k \doteq \min_{1 \leq i < k} \{\beta_i\}, \tag{6}$$

where β_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):

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

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

$$\bigwedge_{1 \leq i \leq n} \left(\bigwedge_{i < k \leq n+1} (q_k^{\max} \leq \beta_i) \right).$$

Rewriting the system of inequalities,

$$\bigwedge_{1 < k \leq n+1} \left(\bigwedge_{1 \leq i < k} (q_k^{\max} \leq \beta_i) \right),$$

which is equivalent to

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

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 $1 < k \leq n+1$:

$$Q_k = \min\{Q_{k-1}, \beta_{k-1}\}. \tag{7}$$

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.,

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

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

$$q_k^{\max} + \sum_{\tau_j \in \tau} \text{DBF}_j(a^*) > a^*. \tag{8}$$

Consider a situation in which:

- all tasks with relative deadline $\leq a^* (< D_k)$ start synchronously at $t = 0$;
- task τ_k enters its largest NPR of length $q_k^{\max}$ an arbitrarily small amount of time before $t = 0$. Since τ_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 [32]. 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 [7].

V. Proposed approach

As explained in Section III, limiting preemptions may significantly reduce the number of cache misses — so that $\mu_i << \mu_i^{\max}$ — as well as the negative effects of context switches, with a beneficial effect on the worst-case timing behavior. On the other side, 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 β_i depends on the worst-case execution times C_j, which are significantly influenced by the number of cache misses of each task τ_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 adopting a scheduling policy that decreases the number of preemptions. 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 assumed each PP in a task τ_k to cause a fixed overhead ξ_k.

Under this assumption, we present a method that optimally exploits the schedulability test of Theorem 2 to select the PPs inside the code of each task in order to achieve a schedulable condition. The proposed algorithm can be summarized as follows:

- The algorithm starts with no preemption points for each task, i.e., setting $p_i = 1$ and $q_i^{\max} = C_i^{\text{NP}}$, $\forall i$, where C_i^{NP} is the worst-case execution time of τ_i when it executes non-preemptively. This value can be found

INSERTPP(τ)

Initialize: $\{q_i^{\max} \leftarrow C_i^{\mathrm{NP}}\}_{i=1}^{n}$, $q_{n+1}^{\max} \leftarrow 0$, $\{p_i \leftarrow 1\}_{i=1}^{n}$, and $Q_1 \leftarrow \infty$.
1 **for** (i : $1 \leq i \leq n$)
2 $C_i \leftarrow C_i^{\mathrm{NP}} + (p_i - 1)\xi_i$
3 Compute β_i using Equation (4) or (5);
4 $Q_{i+1} \leftarrow \min\{Q_i, \beta_i\}$
5 **if** ($q_{i+1}^{\max} > Q_{i+1}$)
6 **if** (PPLACE($Q_{i+1}, i+1$) = `false`)
7 **return** (`Infeasible`)
endfor
8 **return** (`Feasible`)

PPLACE(Q_k, k)

Let ξ_k be the preemption overhead of τ_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 τ_k at Q_k and after every $(Q_k - \xi_k)$.
3 $p_k \leftarrow \left\lceil \frac{C_k^{\mathrm{NP}} - Q_k}{Q_k - \xi_k} \right\rceil + 1$
4 $q_k^{\max} \leftarrow Q_k$
5 **return** (`true`)

Figure 3. Algorithm for the optimal placement of PPs.

using timing analysis tools [30], without needing to take into account preemptions.

- Then, β_i is computed by Equations (4) or (5) for increasing indexes, that is, starting from β_1. Note that β_i depends only on the C_j of tasks with indexes $j \leq i$ (when computing β_i in the EDF case, $\mathrm{DBF}_j(a) = 0$ for all $j > i$), given by $C_j^{\mathrm{NP}} + (p_j - 1)\xi_j$.
- Then, Q_{i+1} is computed from $\beta_{k \leq i}$ using Theorem 2.
- If Q_{i+1} is smaller than the maximum non-preemptive region of τ_{i+1}, procedure PPLACE($Q_{i+1}, i+1$) is invoked to place the least number of PPs in τ_{i+1} to guarantee $q_{i+1}^{\max} \leq Q_{i+1}$. This is achieved by placing a first PP after Q_{i+1} time-units of (non-preemptive) execution from the beginning of τ_{i+1}. To account for the preemption overhead, further PPs are placed after $(Q_{i+1} - \xi_{i+1})$ time-units, until the end of the code.
- If PPLACE($Q_{i+1}, i+1$) returns false, the algorithm stops, declaring the task set infeasible. The failing condition of 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 τ_k is entirely dedicated to the preemption overhead.
- 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 INSERTPP(τ) in deriving a schedulable condition. Then, we will show the optimality of the adopted method.

Theorem 3. *Procedure* INSERTPP(τ) *is correct.*

Proof: If the procedure succeeds, each Q_k will be larger than or equal to the maximum non-preemptive region $q_k^{\max}$ 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 β_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 β_i, and therefore Q_{i+1}, are correctly computed. By Theorems 1 and 2, the correctness of the procedure is assured. ■

Having proved the correctness of INSERTPP(τ), 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 unfeasible schedule.

Theorem 4. *Procedure* INSERTPP(τ) *is optimal under* EDF.

Proof: Suppose, by contradiction, there is a feasible task set τ for which procedure INSERTPP(τ) fails. Then, there is at least one task τ_k for which procedure PPLACE(Q_k, k) fails. Let τ_k be the task with the smallest index for which the procedure fails, and let β_i be the value that minimizes Q_k, i.e., $i = \mathrm{argmin}_{j=1}^{k-1}\{\beta_j\}$. As previously mentioned, β_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 and T_j) are fixed, the execution times C_j may vary for different placements of PPs in the code of each task τ_j. Note that β_i is a decreasing function of all $C_{j \leq i}$. We now prove by induction that procedure INSERTPP(τ) allows finding the smallest values $C_{j \leq i}$, among PP allocation strategies that are feasible. Therefore, if τ is feasible, the largest possible β_i is found with INSERTPP(τ). 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 $q_k^{\max} \leq \beta_i$, then this latter condition will be violated by any other possible strategy, since it cannot lead to a larger β_i. Therefore, τ is not feasible, reaching a contradiction.

Base case: Independently of the number of PPs, task τ_1 is always executed non-preemptively, both under FP and under EDF. Note that procedure INSERTPP(τ) does not insert any PP in τ_1, leading to the smallest possible value of C_1.

Inductive step: Let $j \leq i$. Assume INSERTPP($\tau^{(j-1)}$) obtained the schedulability of the reduced task set $\tau^{(j-1)} \doteq \{\tau_1, \ldots, \tau_{j-1}\}$, minimizing the worst-case execution times $C_1, \ldots, C_{j-1}$. We will prove that 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 INSERTPP($\tau^{(j-1)}$) and INSERTPP($\tau^{(j)}$), so that there is no change in any C_k or $q_k^{\max}$, 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 $q_j^{\max} \leq Q_j = \min\{\beta_{k<j}\}$. 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 $\in \tau^{(j-1)}$ 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 PPLACE(Q_j, j) places the least possible number of PPs to τ_j. Since we assumed that each PP of τ_j causes the same overhead ξ_j, procedure PPLACE(Q_j, j) obtains the smallest possible C_j, proving the statement. ■

Note that the optimality of procedure INSERTPP(τ) 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 τ_k. However, this would imply a much more complex analysis, which is beyond the scope of this paper[8].

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 [31]. 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 τ_j has a small β_j value, all lower priority tasks will have frequent PPs. However, they cannot be preempted by τ_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 τ_i at line 1 of procedure INSERTPP(τ) can be replaced by tighter expressions, derived adapting techniques from [28], [24], [25] to the limited preemption scheduling model adopted in this paper.

Again, we believe these are 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 assumption is not overly pessimistic, since the number of inserted PPs is typically very small, even for heavily loaded systems, resulting in rather long non-preemptive regions.

VI. CONSIDERATIONS

The placement of PPs inside each task's code is subject to constraints such as atomic instructions, critical sections and non-preemptable 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 (PPP) 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 τ_k. Each loop, conditional branch, critical section or non-preemptable section of code can be accommodated inside a BB. Smaller preemption overheads ξ_k can be found inserting each PPP between sections of code that access few common memory locations. We define the *minimum execution granularity* Δ_k as the maximum "execution distance" between any two consecutive PPPs, including the preemption delay ξ_k.

Procedure DISCPPLACE(Q_k, k), whose pseudocode is shown in Figure 4, can be used instead of PPLACE(Q_k, k) when a set of PPPs is given. The procedure will try to minimize the number of PPPs that will be used for the insertion of an actual Preemption Point (PP), without violating the condition on the blocking ($q_k^{\max} \leq Q_k$). 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 ξ_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_1 is decreased by ξ_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 ξ_k of

DISCPPLACE(Q_k, k)

Let ξ_k be the preemption overhead of τ_k, $1 \leq k \leq n$ and $\xi_{n+1} \leftarrow 0$. Let BB_i be the length of the i-th BB when τ_i executes non preemptively.
Initialize: $j \leftarrow 1$; $C \leftarrow BB_1 - \xi_k$.

```
1  if (Q_k < Δ_k) return false
2  for BB_i in {BB_2, BB_3, ...}
3          C ← C + BB_i
4          if (C + ξ_k > Q_k)
5                  Place a PP before the i-th BB
6                  j = j + 1
7                  C ← BB_i
   endfor
8  p_k ← j
9  q_k^max ← max_{i=1}^{p_k} {q_{k,i}}
10 return (true)
```

Figure 4. Algorithm for the insertion of PPs in a task τ_k, so that $q_k^{\max} \leq Q_k$ is satisfied.

[8]As explained in [7], when the exact length of the last NPR of a task τ_k is fixed and known a priori, the worst-case response time of τ_k is not necessarily given by the first instance of τ_k after a critical instant. A necessary and sufficient schedulability condition would then need to check a large number of possible arrival times for τ_k, resulting in a much more complex schedulability condition. See [7] for further details.

each task τ_k can be bounded by $(\pi + \sigma)$, plus the CRPD, which is a function of the number of extrinsic cache misses experienced by τ_k when resuming its execution. For a fully associative cache, such number cannot be larger than (i) the total number of memory locations accessed by τ_k, equal to its footprint F_k, and (ii) the total number of cache lines[9], 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 τ_k can be therefore bounded by

$$\xi_k \doteq \pi + \sigma + \gamma \min\{F_k, L\} + \eta(\min\{F_k, L\}), \quad (9)$$

where π, σ and γ can be found with a timing analysis of the architecture, and $\eta()$ can be derived using techniques described in [21], [20]. 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 shown in our simulations.

The complexity of the proposed approach is pseudopolynomial, both in the EDF and in the FP case. 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 the system utilization.

More in detail, each task set was generated as follows. The `UUniFast` algorithm, described in [5], was used to generate each set of n tasks with individual utilizations uniformly distributed with a given total utilization U_{tot}. The non-preemptive WCET C_k^{NP} was generated as a random integer value uniformly distributed in $[50, 150]$, computing T_k as $T_k = C_k^{\mathrm{NP}}/U_k$. The relative deadline D_k was generated as a random integer value within the range $[C_k^{\mathrm{NP}} + 0.8 \cdot (T_k - C_k^{\mathrm{NP}}),\ T_k]$.

Due to space reasons, we include here only the results for fixed priority scheduling. The simulations for EDF are very similar. We considered four fixed priority scheduling policies: non-preemptive (NP), the proposed limited-preemptive policy (LP) using procedure PPLACE, 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 $C_k = C_k^{\mathrm{NP}}$, 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 $q_k^{\max} = C_k = C_k^{\mathrm{NP}}$ for each task τ_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 [9], the response time of τ_k is given by the smallest fixed point of:

$$R_k = \sum_{j \le k} \left\lceil \frac{R_k}{T_j} \right\rceil \left(C_j^{\mathrm{NP}} + cost\right).$$

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[10] (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 [20]; 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 $\hat{C_k^{\mathrm{NP}}} = \sum_{k=1}^{n} C_k^{\mathrm{NP}}/n$.

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 the preemption cost. 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.

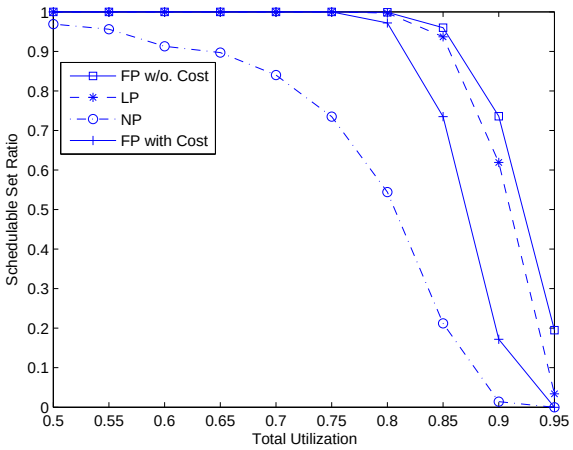
In histogram (d), the preemption cost is set to 10%, as in (c), while the number of tasks is increased to $n = 20$. The performance of FP-with-cost slightly deteriorates, while LP improves in terms of percentage of schedulable task sets. This is expected, because when the number of tasks increases, each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

VIII. Conclusions

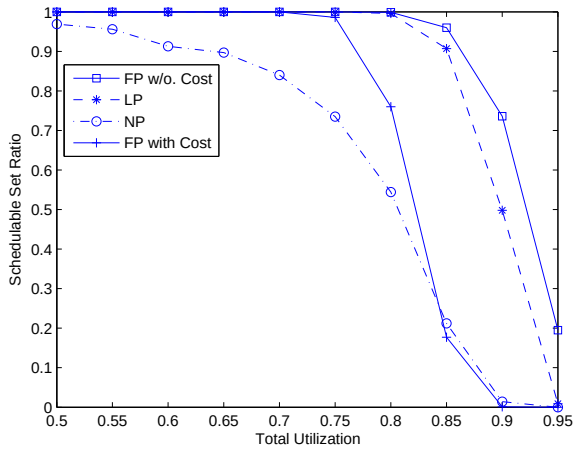
We presented an efficient algorithm to obtain the schedulability of a task set in a real-time system scheduled with FP or EDF, using the limited preemption model. A proper number of preemption points is placed inside the code of each task, in order to guarantee the feasibility of the task set, reaching an optimal compromise between small

[9] Tighter bounds can be found for set-associative or direct mapped caches.

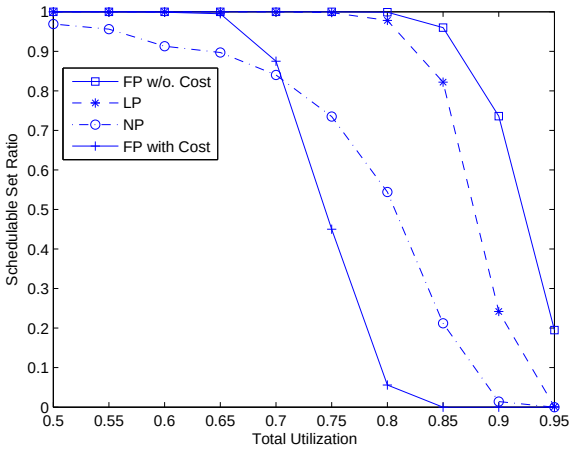
[10] http://www.arinc.com/



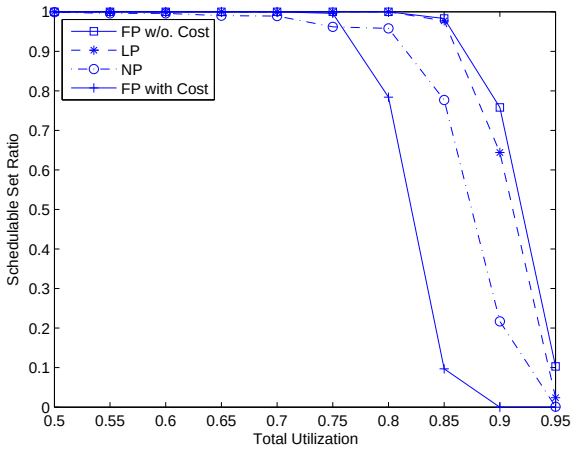
(a) $n = 10$, cost = 5%



(b) $n = 10$, cost=10%



(c) $n = 10$, cost = 20%



(d) $n = 20$, cost=10%

Figure 5. Percentage of schedulable task sets varying n and the preemption cost.

blocking times and reduced preemption overhead. The positive outcomes of the proposed method are not limited to the improved schedulability, but include as well a more predictable behavior of the system. Indeed, reducing the number of locations in which each task can be preempted simplifies the timing analysis, allowing more precise bounds on the worst-case execution times.

As a future work, we plan to adopt tighter estimations on the preemption delays experienced under the limited preemption model, relaxing the assumption on the fixed preemption overhead. We are working on integrating our techniques with existing timing analysis tools that are able to derive tight bounds on the CRPD generated at each potential preemption point. Moreover, we have strategies to extend our limited preemption model considering critical tasks, like interrupt handler, that need to immediately preempt other tasks for a prompt execution.

REFERENCES

[1] S. Altmeyer and G. Gebhard. Wcet analysis for preemptive scheduling. In *Intl. Workshop on WCET Analysis*, Dagstuhl, Germany, 2008.

[2] S. Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *ECRTS*, Palma de Mallorca, Spain, July 2005. IEEE Computer Society Press.

[3] S. Baruah and S. Chakraborty. Schedulability analysis of non-preemptive recurring real-time tasks. In *International Workshop on Parallel and Distributed Real-Time Systems (IPDPS)*, Rhodes, Greece, April 2006.

[4] M. Bertogna and S. Baruah. Uniprocessor scheduling of sporadic task systems under preemption constraints. Manuscript under review. Downloadable of the first author's web page, 2009.

[5] E. Bini and G. Buttazzo. Biasing effects in schedulability measures. In *ECRTS*, Catania, Italy, July 2004.

[6] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *RTAS*, pages 342–353, Washington, DC, USA, 2008. IEEE Computer Society.

[7] R. Bril, J. Lukkien, and W. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited. In *ECRTS '07: Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 269–279, 2007.

[8] A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. *In S. Son, editor, Advances in Real-Time Systems*, pages 225–248, 1994.

[9] J. Busquets-Mataix, D. Gil, P. Gil, and A. Wellings. Techniques to increase the schedulable utilization of cache-based preemptive real-time systems. *J. Syst. Archit.*, 46(4):357–378, 2000.

[10] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park Norwell, MA 02061, USA, 1997.

[11] J. Echague, I. Ripoll, and A. Crespo. Hard real-time preemptively scheduling with high context switch cost. In *7th Euromicro Workshop on Real-Time Systems (EUROMICRO-RTS'95)*, 1995.

[12] G. Gebhard and S. Altmeyer. Optimal task placement to improve cache performance. In *EMSOFT*, pages 166–171, Salzburg, Austria, 2007.

[13] R. Gopalakrishnan and G.M. Parulkar. Bringing real-time scheduling theory and practice closer for multimedia computing. In *Proceedings of the ACM Sigmetrics Conference on Measurement & Modeling of Computer Systems*, pages 1–12, May 1996.

[14] K. Jeffay, D. Stanat, and C. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the 12th Real-Time Systems Symposium*, San Antonio, Texas, December 1991. IEEE Computer Society Press.

[15] L. Ju, S. Chakraborty, and A. Roychoudhury. Accounting for cache-related preemption delay in dynamic priority schedulability analysis. In *DATE*, 2007.

[16] Chang-Gun Lee et al. Bounding cache-related preemption delay for real-time systems. *IEEE Transactions on Software Engineering*, 27(9):805–826, 2001.

[17] S. Lee, C.-G. Lee, M. Lee, S. L. Min, and C.-S. Kim. Limited preemptible scheduling to embrace cache memory in real-time systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 51–64, May 1998.

[18] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, San Diego, California, 2007.

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

[20] R. Pellizzoni, B. Bui, M. Caccamo, and L. Sha. Coscheduling of cpu and i/o transactions in cots-based embedded systems. In *RTSS*, pages 221–231, Barcelona, Spain, 2008.

[21] R. Pellizzoni and M. Caccamo. Toward the predictable integration of real-time cots based systems. In *RTSS*, pages 73–82, Tucson, Arizona (USA), 2007.

[22] S. M. Petters and G. Färber. Scheduling analysis with respect to hardware related preemption delay. In *Workshop on Real-Time Embedded Systems*, London, UK, 2001.

[23] H. Ramaprasad and F. Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. *Real-Time and Embedded Technology and Applications Symposium, 2006. RTAS'06*, April 2006.

[24] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemption points. *Real-Time Systems Symposium, 2006. RTSS '06*, December 2006.

[25] H. Ramaprasad and F. Mueller. Bounding worst-case response time for tasks with non-preemptive regions. *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, pages 58–67, April 2008.

[26] J. Regehr. Scheduling tasks with mixed preemption relations for robustness to timing faults. In *RTSS*, pages 315–326, Cancun (Mexico), December 2002. IEEE Computer Society.

[27] J. Simonson and J.H. Patel. Use of preferred preemption points in cache-based real-time systems. In *IPDS*, pages 316–325, April 1995.

[28] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *ECRTS*, 2005.

[29] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proceedings of the International Conference on Real-time Computing Systems and Applications*. IEEE Computer Society, 1999.

[30] R. Wilhelm et al. The worst-case execution time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, 2008.

[31] Orges Xhani. Effects of real-time scheduling on cache performance and worst case execution times. Master's thesis, Scuola Superiore Sant'Anna, Pisa, Italy, December 2009. Downloadable of the first author's web page.

[32] G. Yao, G. Buttazzo, and M. Bertogna. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. In *RTCSA*, Beijing, China, May–June 2009.

[33] P. Meumeu Yomsi and Y. Sorel. Extending rate monotonic analysis with exact cost of preemptions for hard real-time systems. In *ECRTS*, 2007.