# Optimal Selection of Preemption Points to Minimize Preemption Overhead

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

❖

**Abstract**—A central issue for verifying the schedulability of hard real-time systems is the correct evaluation of task execution times. These values are significantly influenced by the preemption overhead, which mainly includes the cache related delays and the context switch times introduced by each preemption. Since such an overhead significantly depends on the particular point in the code where preemption takes place, this paper proposes a method for placing suitable preemption points in each task in order to maximize the chances of finding a schedulable solution.

In a previous work, we presented a method for the optimal selection of preemption points under the restrictive assumption of a fixed preemption cost, identical for each preemption point. In this paper, we remove such an assumption, exploring a more realistic and complex scenario where the preemption cost varies throughout the task code. Instead of modeling the problem with an integer programming formulation, with exponential worst-case complexity, we derive an optimal algorithm that has a linear time and space complexity. This somewhat surprising result allows selecting the best preemption points even in complex scenarios with a large number of potential preemption locations. Experimental results are also presented to show the effectiveness of the proposed approach in increasing the system schedulability.

## 1 INTRODUCTION

A key element to guarantee the timing constraints of hard real-time systems is a precise knowledge of tasks characteristics. In particular, while periods and deadlines are assigned by the system designer to meet specific performance requirements, task computation times depend on the task code and input data, and hence may be subject to high variations. To perform feasibility analysis in worst-case conditions, task computation times are thus upper bounded by worst-case execution times (WCETs), that are typically estimated by appropriate timing analysis tools. However, a precise WCET estimation is quite difficult to achieve, and current tools can only provide large upper bounds computed under very pessimistic assumptions, dictated by the low-level mechanisms of modern computer architectures. For example, since preemptions destroy program locality, WCET estimates of preemptive tasks are computed by assuming worst-case cache related delays, given by the extra operations needed for refilling the cache lines evicted

by the preempting task. Additional overhead is due to the context switch time and to the time needed to invalidate the instruction pipeline after each preemption. Finding a precise estimation of preemption costs is therefore crucial for deriving tight schedulability conditions.

Less pessimistic WCET bounds can be obtained either by refining the timing analysis tools, or by adopting suitable scheduling algorithms to limit the number of preemptions as much as possible. The first approach has been targeted by many papers in the timing analysis domain [3], [4], [16], [20], [23], [27], [28], [32], and will not be discussed here. The second approach has been introduced in [24], [31] and consists in deferring a preemption request until a point in which the resulting Cache Related Preemption Delay (CRPD) is small, without imposing an excessive blocking on the preempting job. The knowledge of the deterministic location of the preemption points can be exploited to simplify the analysis of the cache state at each point, so improving the estimation of the preemption overhead. Specific experiments on CRPD showed that WCET can increase up to 40% in the presence of preemptions, with respect to a fully non-preemptive execution [26].

In [10], we tackled the problem of finding the best possible placement of preemption points under the restrictive assumption of a fixed context-switch overhead, identical for each preemption point. The analysis was presented for both Fixed Priority (FP) and Earliest Deadline First (EDF) scheduling, by computing the maximum time-interval for which each task can execute in non-preemptive mode without causing any deadline miss. However, the assumption of a fixed preemption cost turns out to be very pessimistic, since to be conservative we considered that each task experiences the maximum preemption overhead at all preemption locations, without taking advantage of points at which a preemption would cause a reduced overhead (e.g., between two frames of an MPEG decoding application).

**Contributions:** In this paper, we improve the task model by removing the restriction of considering a fixed overhead for each preemption point, so obtaining a more realistic and interesting scenario in which the preemption cost depends on the particular location at which a task is preempted. The information on the preemption cost can be provided by exist-

ing timing analysis tools, along with the worst-case execution time of each non-preemptive chunk of code. We show how to optimally select the preemption points under such a model on a single processor system, in such a way that, if a feasible schedule is not found by the proposed method, then no other strategy can lead to a feasible solution. While the problem could be solved using an integer programming formulation, its exponential worst-case complexity would make it intractable in typical scenarios, where the number of potential preemption points can be very high. Instead, we managed to find a much more efficient solution that has a linear complexity both in space and time. The final outcome is a fully integrated approach that allows the automatic placement of preemption points, without user intervention, for applications that can be modeled as a sequential flow of basic blocks of code. A set of experiments is presented to prove the effectiveness of the proposed approach in increasing the system schedulability. First, some tasks coming from real applications are characterized in terms of WCETs and preemption costs, using timing analysis tools, and the results are used to derive a model for generating a synthetic workload with realistic preemption costs. Then, the proposed placement algorithm is applied to a realistic set of synthetic tasks, showing that a smart preemption point selection can significantly reduce the WCET of each task, so improving system schedulability. As already mentioned, the run-time complexity of the selection algorithm is linear in the number of basic blocks, so that the additional computational cost is negligible when compared to the cost of the classic timing analysis.

**Organization of the paper:** The remainder of the paper is organized as follows. Section 2 reviews previously proposed scheduling algorithms that adopt hybrid preemption policies. Section 3 presents the adopted system model and terminology, along with the assumptions made to simplify the analysis. Section 4 reminds some useful results on the schedulability analysis of systems scheduled using a deferred preemption scheduler. Section 5 illustrates the proposed preemption point placement algorithm. The complexity of the proposed approach is evaluated in Section 6, along with implementation related issues. Section 7 reports a set of experiments aimed at evaluating the effectiveness of the method. The validity of the adopted system model is discussed in Section 8, suggesting future research directions that relax the assumptions made in this work. Finally, Section 9 states our conclusions.

## 2 RELATED WORK

The research on hybrid preemption strategies is receiving an increasing attention in the real-time research community [14], [11], [29], [10], [31], [24], [15], [33], [30] as well as in industrial environments [17], [18]. This is mainly due to the problems that fully preemptive approaches create in terms of cache performance and WCET analysis. On the other side, non-preemptive schedulers [19], [7] are not able to achieve a high utilization, because of the large blocking imposed to high priority jobs. For these reasons, alternative preemption strategies have been investigated to achieve higher utilizations at a reduced preemption overhead.

The *deferred preemption model* has been proposed by Burns in [15]. According to this model, each task is composed of a sequence of non-preemptive regions separated by a fixed preemption point. The advantage of this approach is that the timing analysis is significantly simplified since a task can be preempted only at a limited set of pre-defined locations. An exact schedulability analysis for fixed priority scheduling with deferred preemptions has been presented in [13], deriving a pseudo-polynomial test that needs to take into account only the maximum and the last non-preemptive regions of each task.

A different model has been proposed by Baruah [6], who proposed a method for computing the maximum amount of time $Q_i$ for which a task $\tau_i$ may execute non preemptively still preserving feasibility. Under this model, non-preemptive regions are not defined a priori, but are triggered by a preemption request from a higher priority job. In other words, an executing task will switch to *non-preemptive* mode as soon as a higher priority job arrives, postponing the preemption after $Q_i$ time-units (or earlier in case the executing task completes before). Since the location of the preemption points is not deterministic and non-preemptive regions can be everywhere in the task's code (excluding the first $Q_i$ time-units), this model is also called *floating non-preemptive region* model. In [11] and [35], a method is presented to compute the largest non-preemptive region lengths that allow each task to meet its deadline, under both FP and EDF.

An alternative hybrid preemption strategy, based on the concept of *preemption threshold*, has been presented in [33]. According to this policy, each task is assigned a nominal priority and a preemption threshold. A preemption will take place only if the preempting task has a nominal priority greater than the preemption threshold of the executing task. An exact schedulability analysis for FP with preemption thresholds has been presented in [22][1]. Although the number of preemptions is reduced with this method, a preemption can still occur at any time and position, so that timing analysis techniques cannot take advantage of additional information to simplify the estimation of the WCETs.

Finding tight estimations of the WCET of critical tasks is a problem that has been widely considered in the real-time community (a good survey on timing analysis techniques can be found in [34]). As we said, this problem becomes even more complex when considering preemptive systems with caches, for which existing techniques are still far from deriving tight bounds on the preemption overhead. In [16] and [27], 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 presented in [23]. With a similar target, Staschulat *et al.* [32] 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

---

1. The original analysis in [33] was flawed and has been corrected in [30], which in its turn has been improved by [22].

accessed by the preempting task. The impact of the data cache on the overall preemption overhead has been analyzed in [29] considering as well the case in which tasks can contain one non-preemptive region. While most of the above works were based on systems scheduled with FP, Ju *et al.* [20] proposed a CRPD analysis for systems scheduled with EDF.

Closer in scope and motivations to our analysis are the works described in [31], [24]. Both approaches adopt a FP scheduler with deferred preemptions, with the common target of reducing the preemption overhead by properly placing the preemption points. In [31], each task is divided into a set of intervals, each one shorter than the maximum blocking time tolerated by higher priority tasks. Then, a preemption point is placed in each interval, at the location having the smallest number of *useful cache blocks*, i.e., cached data that will be accessed again. In [24], instead, preemption points are inserted at the locations characterized by a number of useful cache blocks below a given threshold. Since both approaches adopt a heuristic strategy for the placement of preemption points, neither method is able to minimize the CRPD to a full extent, obtaining only suboptimal solutions.

Our work improves the above results, deriving an optimal preemption point placement algorithm able to minimize the CRPD of each task and find a feasible solution, if there exists one. In [10], we proposed a simpler algorithm that is optimal when considering a fixed context switch cost at each preemption point. Since this cost was set to the largest preemption overhead experienced by a task, the analysis was rather pessimistic. This work improves the analysis, by taking advantage of more detailed information on the preemption overhead at different locations. This is possible by exploiting the features of modern timing analysis tools – as the Absint's aiT[2] tool [5] – that are able to evaluate the maximum blocking time and the context-switch cost for a set of potential preemption points of a given task.

## 3 TASK MODEL

We consider a set $\tau$ composed of $n$ periodic and sporadic real-time tasks that are scheduled on a single processor using Fixed Priority (FP) or Earliest Deadline First (EDF) [25]. Each task $\tau_i$ generates an infinite sequence of jobs, with the first job arriving at any time and successive job-arrivals separated by at least a minimum inter-arrival time $T_i$, also denoted as period. Each job of $\tau_i$ is assumed to have a relative deadline $D_i \leq T_i$ and to consist of a sequence of $N_i$ non-preemptive Basic Blocks (BBs). Preemption is allowed only at basic block boundaries, so each task has $N_i - 1$ "Potential Preemption Points" (PPPs), one between any two consecutive BBs. Critical sections and conditional branches are assumed to be executed entirely within a basic block. In this way, there is no need for using shared resource protocols to access critical sections.

To limit the preemption overhead, the proposed algorithm identifies a subset of PPPs that minimizes the overall CRPD still preserving the schedulability of the task set. A PPP selected by the algorithm is referred to as an Effective Preemption Point (EPP), whereas the other PPPs are disabled.

2. http://www.absint.com/ait/

Therefore, the sequence of basic blocks between any two consecutive EPPs forms a Non-Preemptive Region (NPR). The following notations are used throughout the paper:

$C_i^{\mathrm{NP}}$  denotes the non-preemptive WCET of $\tau_i$, that is, the WCET (without preemption cost) estimated when $\tau_i$ is executed fully non preemptively.

$C_i$  denotes the preemptive WCET of $\tau_i$, that is, the WCET (with preemption cost) estimated when $\tau_i$ is executed preemptively.

$T_i$  denotes the period of $\tau_i$ or its minimum inter-arrival time.

$D_i$  denotes the relative deadline of $\tau_i$.

$\delta_{i,k}$  denotes the $k$-th basic block of task $\tau_i$.

$b_{i,k}$  denotes the WCET of $\delta_{i,k}$ without preemption cost, that is, when $\tau_i$ is executed non-preemptively.

$N_i$  denotes the number of BBs of task $\tau_i$, determined by the $N_i - 1$ PPPs defined by the programmer.

$p_i$  denotes the number of NPRs of task $\tau_i$, determined by the $p_i - 1$ EPPs selected by the algorithm.

$q_{i,j}$  denotes the WCET of the $j$-th NPR of $\tau_i$, including the preemption cost.

$q_i^{\mathrm{max}}$  denotes the maximum NPR length for $\tau_i$:

$$q_i^{\mathrm{max}} = \max\{q_{i,j}\}_{j=1}^{p_i}.$$

$\xi_{i,k}$  denotes the worst-case preemption overhead introduced when $\tau_i$ is preempted at the $k$-th PPP (i.e., between $\delta_k$ and $\delta_{k+1}$).

To evaluate the preemption cost $\xi_{i,k}$, the following overhead contributions need to be considered:

- CRPD;
- WCET of the scheduler and related RTOS activities;
- cost for flushing the processor pipeline (if there is any);
- bus contention delay.

Assuming a fully timing compositional architecture [21], like the ARM7, that does not exhibit any timing anomalies and with no DMA-capable devices, the above costs can be upper bounded by timing analysis tools.

We assume tasks to be ordered by decreasing priorities in the FP case, and by increasing relative deadlines in the EDF case, i.e., $\forall i \mid 0 < i < n : D_{i-1} \leq D_i$. To simplify the notation, the task index is omitted from task parameters whenever the association with the related task is evident from the context.
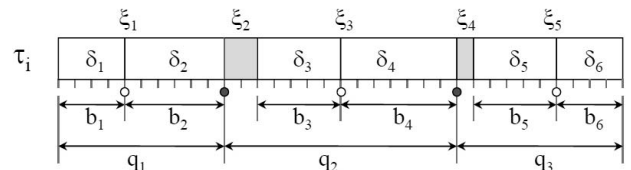


Fig. 1. Example of parameters defined for a task with $6$ BBs and $3$ NPRs. Preemption cost is reported for each PPPs, but accounted only for the EPPs.

Figure 1 illustrates some of the defined parameters for a task with 6 basic blocks and 3 NPRs. PPPs are represented by dots between consecutive basic blocks: filled dots are EPPs selected by the algorithm, while empty dots are PPPs that are disabled. Above the task code, the figure also reports the preemption costs $\xi_k$ for each PPP, although only the cost for the EPPs is accounted in the analysis, in the WCET $q_j$ of the corresponding NPR.

Using the notation introduced above, the non-preemptive WCET of $\tau_i$ may be expressed as follows:

$$C_i^{\mathrm{NP}} = \sum_{k=1}^{N_i} b_{i,k}.$$

The goal of this work is to minimize the overall worst-case execution time $C_i$ of each task $\tau_i$, including the preemption overhead, by properly selecting the EPPs among all the PPPs specified in the code by the programmer, without compromising the schedulability of the task set. To compute the preemption overhead, we make the following simplifying assumptions:

A1. The cache is cold after each context switch.
A2. Each EPP leads to a preemption.

Under these assumptions, the overall worst-case execution time $C_i$ can be computed as follows

$$C_i = C_i^{\mathrm{NP}} + \sum_{k=1}^{N_i-1} \mathsf{selected}(i,k) \cdot \xi_{i,k} \qquad (1)$$

where $\mathsf{selected}(i,k) = 1$ if the $k$-th PPP of $\tau_i$ is selected by the algorithm to be an EPP, whereas $\mathsf{selected}(i,k) = 0$, otherwise.

Note that A1. and A2. are pessimistic assumptions. In particular, if a preempting task has a small cache footprint, the preempted task might experience a smaller preemption overhead than that assumed with A1. However, expressing the preemption overhead as a function of the preempting task would significantly complicate the analysis, needing to evaluate how many cached locations are effectively invalidated at each PPP by each preempting task.

Regarding Assumption A2., there might be cases in which not all EPPs lead to a preemption. As mentioned in [10], the fact that an EPP is inserted in the code of a task $\tau_i$ does not imply that $\tau_i$ will be preempted at that point. For instance, when there is only one higher priority task $\tau_j$ with a large period $T_j$, $\tau_i$ cannot be preempted more than once every $T_j$ time units. Considering which EPP can effectively lead to a preemption would significantly complicate the analysis, needing again to express the preemption overhead as a function of the preempting task. Moreover, when a task $\tau_i$ can be preempted by more than one task, it is difficult to identify when a higher priority instance will cause an additional preemption to $\tau_i$. In fact, multiple higher priority instances could be executed within one single preemption of $\tau_i$, when all such instances arrive before $\tau_i$ resumes the execution. Techniques to detect the worst-case preemption pattern that leads to the largest overhead for $\tau_i$ are presented in [32], [28], [29]. Adapting these techniques to the limited preemption scheduling model is beyond the scope of this paper. A more detailed discussion

on the validity of assumptions A1. and A2. and on the problem of relaxing these assumptions is presented in Section 8.

## 4 SCHEDULABILITY ANALYSIS

This section briefly summarizes some schedulability results useful for computing the upper bound of an NPR for each task. For this purpose, we define the *request bound function* $\mathrm{RBF}_i(a)$ and the *demand bound function* of a task $\tau_i$ in an interval $a$ as

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

and

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

The maximum allowed non-preemptive execution depends on the adopted scheduler. The following theorem, derived in [10], provides an upper bound $Q_i$ for the maximum allowed NPR of a task $\tau_i$ for FP and EDF.

**Theorem 1** (from [10]). *A task set $\tau$ is schedulable with limited preemption EDF or FP if, for all $k \mid 1 < k \leq n+1$,*

$$q_k^{\max} \leq Q_k \doteq \min_{1 \leq i < k} \{\beta_i\}, \qquad (2)$$

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

$$\beta_i^{\mathrm{FP}} \doteq \max_{a \in \{D_i\} \cup A \mid a \leq D_i} \left\{ a - \sum_{j \leq i} \mathrm{RBF}_j(a) \right\}, \qquad (3)$$

*with $A = \{kT_j, \ k \in \mathbb{N}, \ 1 \leq j < n\}$, whereas, under EDF, $\beta_i$ is given by*

$$\beta_i^{\mathrm{EDF}} \doteq \min_{a \in A \mid D_i \leq a < D_{i+1}} \left\{ a - \sum_{\tau_j \in \tau} \mathrm{DBF}_j(a) \right\}, \qquad (4)$$

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

Note that we conventionally set $q_{n+1}^{\max} = 0$, and $D_{n+1}$ equal to the minimum between: (i) the least common multiple (lcm) of $T_1, T_2, \ldots, T_n$, and (ii) the following expression[3]:

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

Notice that, under EDF, Condition (2) is necessary and sufficient, whereas, under FP, it is necessary and sufficient only when there is no information on the length of the final non-preemptive chunk of code of each task (i.e., in the "*floating model*" described in [35]).

---

3. 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, as proved in [8].

# 5 PROPOSED APPROACH

In [10], it was proved that an optimal way for selecting the EPPs to minimize the preemption overhead without violating Condition (2) is to proceed from task $\tau_1$ to $\tau_n$, according to the ordering assumed in Section 3. In this way, the chances for finding a feasible solution are maximized. Since the preemption overhead $\xi_i$ of a task $\tau_i$ in [10] is assumed to be constant for all preemption points, the resulting placement algorithm is rather simple, selecting an EPP every (at most) $Q_i - \xi_i$ units of execution. However, since the cost of each single preemption needs to be safely set to the largest preemption cost experienced by each task, the overhead is significantly overestimated.

In this section, we show how to improve the analysis, considering a different preemption overhead for every PPP and presenting a placement algorithm that selects the EPPs to minimize the overall preemption cost, without violating Condition (2). In the following, we implicitly refer to a generic task $\tau_i$, with maximum allowed NPR length $Q_i = Q$.

It is easy to prove that a naive algorithm, like the one adopted in [10], that activates a PPP after at most $Q$ units of execution from the previous one is *not* optimal, since it does not minimize the overall preemption overhead. When a variable preemption overhead is considered, this algorithm does not even minimize the number of EPPs[4]. Moreover, even an algorithm that minimizes the number of EPPs might be unable to minimize the overall preemption overhead. In general, it may be convenient to insert more preemption points than the least possible number, to take advantage of points with a small CRPD.

Consider, for instance, the task reported in Figure 2, assuming $Q = 8$. With the naive algorithm, only one preemption point is inserted at the end of $\delta_4$. In fact, $\sum_{k=1}^{4} b_k = 2+2+2+1 = 7 \leq Q$, and $\xi_4 + \sum_{k=5}^{6} b_k = 3+2+3 = 8 \leq Q$, meeting Condition (2). The total preemption overhead is $3\cdot$ However, by selecting two EPPs — one after $\delta_1$ and another after $\delta_5$ — we achieve a feasible solution with a smaller total overhead $\xi_1 + \xi_5 = 1 + 1 = 2$. In general, for tasks with a large number of basic blocks with different overhead values, finding the optimal solution is not trivial.

For a generic task, the worst-case execution time $q$ of a NPR composed of the consecutive basic blocks $\delta_j, \delta_{j+1}, \ldots, \delta_k$ can be expressed as

$$q = \xi_{j-1} + \sum_{\ell=j}^{k} b_\ell, \qquad (5)$$

conventionally setting $\xi_0 = 0$. Note that the preemption overhead is included in $q$. Since any NPR of a feasible EPP selection has to meet the condition $q \leq Q$, we must have

$$\xi_{j-1} + \sum_{\ell=j}^{k} b_\ell \leq Q. \qquad (6)$$

Now, let $B_k$ be the WCET, including the preemption overhead, of the first $k$ basic blocks, i.e., from the beginning

---



$$Q = 8$$

Solution with one PP:

$$CRPD = 3$$
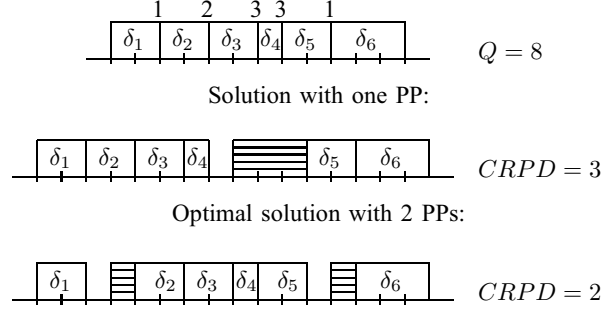
Optimal solution with 2 PPs:

$$CRPD = 2$$

Fig. 2. Two solutions for selecting EPPs in a task with $Q = 8$: the first minimizes the number of EPPs, while the second minimizes the overall preemption cost.

of $\delta_1$ until the end of $\delta_k$. Then, we can express the following recursive expression

$$B_k = B_{j-1} + q = B_{j-1} + \xi_{j-1} + \sum_{\ell=j}^{k} b_\ell. \qquad (7)$$

Note that since $\delta_N$ is the last BB, the worst-case execution time $C_i$ of the whole task $\tau_i$ is equal to $B_N$.

The proposed algorithm for the optimal selection of preemption points is based on the equations presented above and its pseudo-code is reported in Figure 3. The algorithm evaluates all the BBs in increasing order, starting from the first one. For each BB $\delta_k$, the minimum $B_k$ that does not violate Condition 2 is computed as follows.

For the first BBs, the minimum $B_k$ is given by the sum of the BB lengths $\sum_{\ell=1}^{k} b_\ell$ as long as this sum does not exceed $Q$. Note that if $b_1 > Q$, there is no feasible PPP activation, and the algorithm fails. For the following BBs, $B_k$ needs to consider the cost of one or more preemptions as well. Let $Prev_k$ be the set of the preceding BBs $\delta_{j \leq k}$ that satisfy Condition (6), i.e., that might belong to the same NPR of $\delta_k$. If this set is empty, there is no feasible PPP activation, and the algorithm fails. Otherwise, the minimum $B_k$ is given by

$$B_k = \min_{\delta_j \in Prev_k} \left\{ B_{j-1} + \xi_{j-1} + \sum_{\ell=j}^{k} b_\ell \right\}. \qquad (8)$$

Let $\delta^*(\delta_k)$ be the basic block for which the rightmost term of Expression (8) is minimum

$$\delta^*(\delta_k) = \arg\min_{\delta_j \in Prev_k} \left\{ B_{j-1} + \xi_{j-1} + \sum_{\ell=j}^{k} b_\ell \right\}. \qquad (9)$$

If there are many possible BBs minimizing (8), the one with the smallest index is selected. Let $\delta_{Prev}(\delta_k)$ be the basic block preceding $\delta^*(\delta_k)$, if there exists any. The PPP at the end of $\delta_{Prev}(\delta_k)$ – or, equivalently, at the beginning of $\delta^*(\delta_k)$ – is meaningful for the analysis, since it represents the last PPP to activate for minimizing the preemption overhead of the first $k$ basic blocks.

A feasible placement of EPPs for the whole task can then be derived with a recursive activation of PPPs, starting with

---

4. When instead a fixed preemption overhead is considered for each point, the naive algorithm described in [10] is able to minimize the number of EPPs, and, consequently, the total preemption overhead.

the PPP at the end of $\delta_{Prev}(\delta_N)$, which will be the last EPP of the considered task. The penultimate EPP will be the one at the end of $\delta_{Prev}(\delta_{Prev}(\delta_N))$, and so on. If this recursive lookup of function $\delta_{Prev}(k)$ reaches the start of the program, a feasible placement of EPPs has been detected, with a worst-case execution time, including preemption overhead, equal to $B_N$. This is guaranteed to be the placement that minimizes the preemption overhead of the considered task, as proved in the next theorem.

**Theorem 2.** *Under assumptions A1. and A2., the PPP activation pattern detected by procedure $\text{PPP\_SELECT}(Q_i, \tau_i)$ minimizes the preemption overhead experienced by a task $\tau_i$, without compromising the schedulability.*

*Proof:* First, we prove that if procedure $\text{PPP\_SELECT}(Q_i, \tau_i)$ fails, there is no other feasible EPP placement. For the procedure to fail, it is necessary that the condition at line 3 is satisfied. This means that there exists a BB $\delta_k$ for which Condition (6) is violated for any $j \leq k$; That is,

$$\xi_{j-1} + \sum_{\ell=j}^{k} b_\ell > Q, \qquad \forall j \leq k.$$

This means that with any possible PPP activation pattern, the length of the NPR containing $\delta_k$ will be larger than $Q$, violating Condition (2) and leading to a deadline miss[5].

We now consider the minimization of the preemption overhead. Let $C_i$ be the WCET, including the preemption overhead, resulting from the EPP allocation given by $\text{PPP\_SELECT}(Q_i, \tau_i)$. Suppose there exists another feasible EPP allocation that results in a smaller $C_i' < C_i$. We prove by induction that this is not possible. The proof inducts over the index $j$ of the basic blocks $\delta_j$, proving that $B_j$ is minimized for all $j, 1 \leq j \leq N$.

**Base case.** For $j = 1$, $B_1 = b_1$ by definition. This is the minimum possible value of the WCET of the first BB, since it does not experience any preemption.

**Inductive step.** Assume all $B_\ell$, $\forall \ell < j$ are minimized by procedure $\text{PPP\_SELECT}(Q_i, \tau_i)$. We prove that $B_j$ is also minimized. By Equation (8), procedure $\text{PPP\_SELECT}(Q_i, \tau_i)$ computes $B_j$ as

$$B_j = \min_{\delta_\ell \in Prev_j} \left\{ B_{\ell-1} + \xi_{\ell-1} + \sum_{m=\ell}^{j} b_m \right\}.$$

Since, by induction hypothesis, all $B_{\ell-1}$ terms are minimal, also $B_j$ is minimized, proving the statement. Since $C_i = B_N$, a contradiction is reached, proving the theorem. □

As we mentioned in Section 4, the feasibility of a given task set is maximized by applying procedure $\text{PPP\_SELECT}(Q_i, \tau_i)$ to each task $\tau_i$, starting from $\tau_1$ and proceeding in task order. Once the optimal allocation of EPPs has been computed for a task $\tau_i$, the value of the overall WCET $C_i = B_N$ can be used for the computation of the maximum allowed NPR $Q_{i+1}$ of the next task $\tau_{i+1}$, using Equation (2). The procedure is repeated

5. Note that Theorem 1 is necessary and sufficient only in the EDF case.

until a feasible PPP activation pattern has been produced for all tasks in the considered set. If the computed $Q_{i+1}$ is too small to find an EPP feasible allocation, the only possibility to reach schedulability is by removing tasks from the system, as no other EPP allocation strategy would produce a feasible schedule.
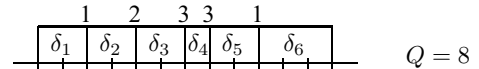
$\text{PPP\_SELECT}(Q, \tau)$
　　Initialize: $Prev_1 \leftarrow \{\delta_1\}$, $B_0 \leftarrow 0$
1 　**for** $(k : 1 \leq k \leq N)$
2 　　　Remove from $Prev_k$ all $\delta_j$ violating (6)
3 　　　**if** $(Prev_k = \emptyset)$
4 　　　　　**return** (Infeasible)
5 　　　Compute $B_k$ using Equation (8)
6 　　　Store $\delta_{Prev}(\delta_k)$
7 　　　$Prev_{k+1} \leftarrow Prev_k \cup \{\delta_k\}$
　　**endfor**
8 　$\delta_j \leftarrow \delta_{Prev}(\delta_N)$
9 　**while** $(\delta_j \neq \emptyset)$
10 　　　Select the PPP at the end of $\delta_{Prev}(\delta_j)$
11 　　　$\delta_j \leftarrow \delta_{Prev}(\delta_j)$
　　**endwhile**
12 　**return** (Feasible)

Fig. 3. Algorithm for the optimal selection of PPPs of a task.

### 5.1 Example

To better clarify how the proposed algorithm works, we illustrate an example of EPP selection using the task previously described in Figure 2. The execution steps of the algorithm are reported in Figure 4.



| $k$ | $Prev_k$ | $\delta_{Prev}(\delta_k)$ | $B_k$ |
|---|---|---|---|
| 0 | | | 0 |
| 1 | $\{\delta_1\}$ | $\emptyset$ | 2 |
| 2 | $\{\delta_1, \delta_2\}$ | $\emptyset$ | 4 |
| 3 | $\{\delta_1, \delta_2, \delta_3\}$ | $\emptyset$ | 6 |
| 4 | $\{\delta_1, \delta_2, \delta_3, \delta_4\}$ | $\emptyset$ | 7 |
| 5 | $\{\delta_2, \delta_3, \delta_4, \delta_5\}$ | $\delta_1$ | 10 |
| 6 | $\{\delta_5, \delta_6\}$ | $\delta_5$ | 14 |

Place an EPP at the end of $\delta_1$ and $\delta_5$

Fig. 4. Sample execution of procedure $\text{PPP\_SELECT}(Q, \tau)$.

At the beginning, $B_0 = 0$, by definition, and $Prev_1$ is initialized to the first BB $\delta_1$. For the first 4 BBs $\delta_1, \ldots, \delta_4$, it is possible to accommodate the overall WCET from the start of the program without any preemption. In fact, $\xi_0 + b_1 + b_2 + b_3 + b_4 = 0 + 2 + 2 + 2 + 1 = 7 \leq Q = 8$, without violating the $Q$ bound. This means that $\forall k, 1 \leq k \leq 4$ the

smallest $B_k$ is obtained with a NPR spanning $\delta_1, \ldots, \delta_k$. Since no preemption penalty is paid at the beginning of the execution ($\xi_0 = 0$), Equation (8) gives $B_k = \sum_{\ell=1}^{k} b_\ell, \forall k, 1 \leq k \leq 4$, as shown in the figure. Moreover, $\delta^*(\delta_k) = \delta_1$ by Equation (9), so that $\forall k, 1 \leq k \leq 4$ there is no BB preceding $\delta^*(\delta_k)$, and $\delta_{Prev}(\delta_k) = \emptyset$.

For $k = 5$, the start of the task is no more in the $Q$ window ($\delta_1 \notin Prev_5$), because the non-preemptive execution time from the start to the fifth PPP is $\sum_{\ell=1}^{5} b_\ell = 9 > Q$. This means that at least one EPP should be inserted between $\delta_1$ and $\delta_5$. To decide which PPP to activate, the minimum $B_5$ is computed with Equation (8), evaluating all $\delta_k \in Prev_5$ and exploiting the values $B_1, \ldots, B_4$ derived at the previous steps. Note that $Prev_5 = \{\delta_2, \delta_3, \delta_4.\delta_5\}$, since $\forall k, 2 \leq k \leq 5, \xi_{k-1} + \sum_{\ell=k}^{5} b_\ell \leq Q$. The minimum $B_5$ is obtained with $\delta^*(\delta_5) = \delta_2$, giving $B_5 = 10$ and $\delta_{Prev}(\delta_5) = \delta_1$. This means that the minimum $B_5$ is obtained by placing an EPP at the end of $\delta_1$. Note that *no EPP is inserted at this point of the program*. The decision on the EPPs will be taken only at the end of the procedure, when the final BB $\delta_N$ is reached. What has been decided at this step is just that *if an EPP is placed at the end of $\delta_5$, then the overall WCET is minimized by placing another EPP after $\delta_1$*.

For $k = 6 = N$, the last BB $\delta_6$ has been reached. $Prev_6$ is derived by adding $\delta_6$ to $Prev_5$, and removing all BBs that cannot be in the same NPR of $\delta_6$. Note that $\delta_2$ is too far, even without considering the preemption overhead: $\sum_{\ell=2}^{6} b_\ell = 2 + 2 + 1 + 2 + 3 = 10 > Q$. For $\delta_3$, $\xi_2 + \sum_{\ell=3}^{6} b_\ell = 2 + 2 + 1 + 2 + 3 = 10 > Q$. For $\delta_4$, $\xi_3 + \sum_{\ell=4}^{6} b_\ell = 3 + 1 + 2 + 3 = 9 > Q$. Therefore, $Prev_6 = \{\delta_5, \delta_6\}$. Using Equation (8), the minimum $B_6$ is obtained by selecting the smallest value between $B_4 + \xi_4 + b_5 + b_6 = 7 + 3 + 2 + 3 = 15$ and $B_5 + \xi_5 + b_6 = 10 + 1 + 3 = 14$. Therefore, $B_6 = 14$, $\delta^*(\delta_6) = \delta_6$ and $\delta_{Prev}(\delta_6) = \delta_5$. Since the end has been reached, $B_6 = 14$ represents the minimum WCET, including preemption overhead, that can be obtained for the considered task without violating the NPR bound of $Q$. This value is obtained by placing an EPP at the end of $\delta_{Prev}(\delta_6) = \delta_5$. Looking up recursively, another EPP is placed at the end of $\delta_{Prev}(\delta_5) = \delta_1$. Since $\delta_{Prev}(\delta_1) = \emptyset$, no other EPP needs to be placed, and PPP_SELECT$(Q, \tau)$ returns a feasible result.

# 6 IMPLEMENTATION CONSIDERATIONS

We now present some considerations about the implementation of procedure PPP_SELECT$(Q, \tau)$ and its complexity.

## 6.1 Memory requirements

For each BB $\delta_k$, the algorithm has to store the $Prev_k$ set. In later steps, only information about elements $\delta_j$ of this set are needed, such as $B_j$, $b_j$ and $\xi_j$. Since the size of this set depends on $Q$, the memory requirements are $O(Q)$. Moreover, it is necessary to maintain a trace of all $\delta_{Prev}(\delta_k), \forall k, 1 \leq k \leq N$, for the final recursive lookup of the PPPs to select. The overall memory requirement is therefore $O(N + Q)$.

## 6.2 Run-time complexity

Regarding the run-time complexity, a naive implementation of the algorithm would search, at each step, within the set $Prev_k$ the element generating the minimum value to compute $\delta_{Prev}(\delta_k)$. This requires $O(Q)$ time for each BB, yielding to a time complexity of $O(N \times Q)$. A smarter implementation could maintain the set $Prev_k$ in an ordered queue, where the element $\delta_j$ generating the minimum $B_{j-1} + \xi_{j-1}$ is always in the head. To maintain this ordered queue, the implementation should, at each step, do the following for the considered BB $\delta_k$:

- Remove infeasible elements from the head, i.e., elements $\delta_j$ that violate Condition (6) for the considered $\delta_k$.
- Compute $B_{k-1} + \xi_{k-1}$ for $\delta_k$.
- Remove from tail all elements $\delta_j$ for which $B_{j-1} + \xi_{j-1} \geq B_{k-1} + \xi_{k-1}$, i.e., elements that cannot minimize $B_k$.
- Insert $\delta_k$ in the tail.

Since, at each step, only elements at the head and tail of the ordered list are considered, the run-time complexity is reduced to $O(N)$.

## 6.3 Implementation of preemption points

One last consideration concerns the implementation of PPPs and EPPs. Since the preemption overhead upper bounds $\xi_k$ are computed a priori by timing analysis tools, it is important for the PPP activation procedure not to modify the code that has been analyzed. Therefore, it is not possible to add a scheduler invocation at a selected EPP, when this system call was not included in the initial code considered by the timing analysis tool. Similarly, it is not possible to remove a scheduler invocation at a PPP that is not selected by procedure PPP_SELECT$(Q, \tau)$, if this call was included in the code analyzed by the tool. These changes in the code would cause a shift in the memory locations that could modify the cache states derived in the timing analysis, potentially invalidating the derived bounds on the preemption overhead. A possibility to sidestep this problem is to add a modified scheduler invocation at each PPP in the code analyzed by the tool. This call contains a boolean parameter that is set if the PPP is selected by procedure PPP_SELECT$(Q, \tau)$. The scheduler can then check this argument to decide whether to return immediately to the calling task, or to take a scheduling decision. Another alternative, that does not require any modification to the scheduler, is to add annotations for the compiler at each PPP, specifying whether the point is selected or not. The compiler will then replace a selected EPP with a call to the scheduler, and an unselected PPP with a NOP instruction.

# 7 EXPERIMENTAL RESULTS

This section presents some experimental results aimed at measuring the preemption cost of real tasks and at evaluating the performance of the proposed algorithm in terms of task set schedulability, as a function of different task set characteristics.

## 7.1 Measuring preemption costs

A first set of experiments has been carried out to measure the overhead costs due to preemption and evaluate the coherence of the task model proposed in Section 3. To deal with realistic cases, we considered two control tasks produced by the Simulink automatic code generator, which intrinsically generates sequences of code blocks:

- the Matlab U.S. Navy F-14 Tomcat aircraft control task [1], which guarantees the aircraft to operate at a high angle of attack with minimal pilot workload, and
- the Matlab automotive task that models an automatic transmission controller [2].

All timing parameters were measured using a cycle-accurate platform emulator [9] of an ARM7TDMI processor running at 200 MHz and equipped with a 4Kb Harvard I/D caches. Measures have been obtained by placing a preemption point in the task code in different positions corresponding to the end of a basic block. Preemption is enforced by creating a higher priority task that evicts all cache lines. Then, preemption cost is computed as the difference between the execution times measured with and without preemptions. Note that measured values include the costs related to cache misses, context switches and pipelines flushes. The results achieved on the two considered tasks are reported in Figure 5. The $x$-axis represents the time at which a preemption takes place, while the $y$-axis represents the increase in the execution time caused by a preemption at that place.

## 7.2 Evaluation of the algorithms

A second set of experiments has been carried out on synthetic task sets, to evaluate the performance of the proposed algorithm against other approaches in improving the schedulability of the system. The following scheduling algorithms have been considered in the comparison, under the Rate Monotonic priority assignment:

- **FuP-nocost**: Fully preemptive scheduler with no cost. Under this scheduler, a task can be preempted everywhere within its code without extra penalty. Although unrealistic, this algorithm has been considered to provide a reference for an ideal behavior. In this case, the task set schedulability is computed using the Response Time Analysis technique.
- **FuP**. Fully preemptive scheduler with preemption cost. The cost accounted for each preemption is the maximum among the costs of all potential preemption points in the task. The schedulability test is performed using the $CRTA$ algorithm proposed by Busquets-Mataix et al. in [16].
- **LiP-naive**. Limited preemption approach with variable cost, using the naive algorithm adopted in [10]. This algorithm activates a PPP after at most $Q$ time-units of execution from the previous one, starting from the beginning of the task.
- **LiP-opt**. Limited preemption approach with variable cost, using the optimal algorithm proposed in this paper.
- **NoP**. Non-preemptive algorithm. For the sake of completeness, we also evaluated the performance of a fully
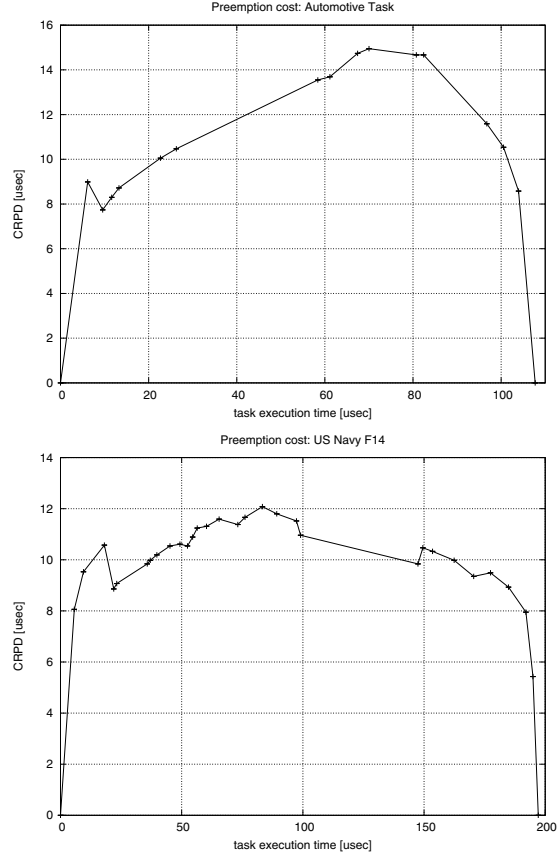


Fig. 5. Preemption costs for two real control tasks.

non preemptive case, which provides a lower bound for the task set schedulability.

Task parameters were randomly generated to produce both the basic blocks execution times and the preemption cost for each preemption point. The number of basic blocks was randomly generated in an interval $[20, 200]$ with uniform distribution, while their WCET was generated according to a Gaussian distribution with mean equal to 4000 time units and variance of 3000 time units. Notice that in the ARM7 architecture used in the previous experiments, a time unit corresponds to 5 nanoseconds. The utilization of each task has been generated using the approach proposed in [12]. The task periods were then computed dividing the WCET by the utilization of each task. Preemption costs were randomly generated using the following function, to achieve a realistic distribution similar to the one shown in Figure 5:

$$\xi_i = \xi_{i-1} + \Delta\xi_i$$

where $\Delta\xi_i = \text{gaus}(m_i, \sigma)$, and

$$m_{i+1} = \begin{cases} -M & \text{if } \xi_i > \xi_{max} \\ +M & \text{if } \xi_i < \xi_{min} \\ \text{sgn}(\Delta\xi_i)M & \text{otherwise} \end{cases}$$

The variance $\sigma$ determines the degree of variability of the preemption overhead between any two consecutive PPPs. An

example of values generated by this function is shown in Figure 6 for different values of the variance $\sigma$, setting $M = 20$, $\xi_{min} = 1000$, $\xi_{max} = 55000$. Note that the requirement for positive CRPD values introduces an asymmetry that makes them to increase with the variance $\sigma$. In the following experiments, we adopt a variance $\sigma = 3000$.
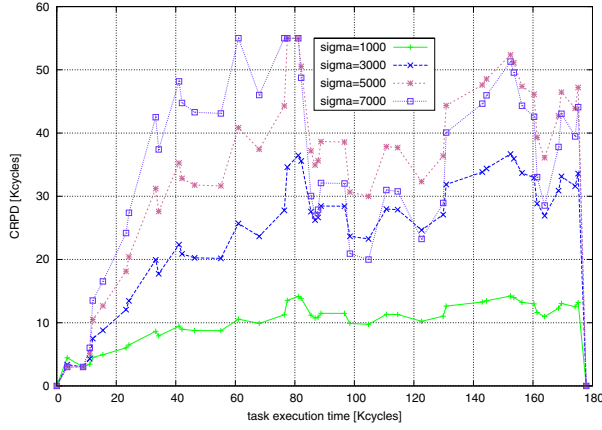


Fig. 6. CRPD of the same task as a function of the CRPD variance.

In a first test, we monitored the percentage of schedulable task sets for every algorithm as a function of the task set utilization. In Figure 7, we report the achieved results for the case with $n = 7$ tasks. Each point in all the graphs has been obtained as the average upon 2000 simulation runs.
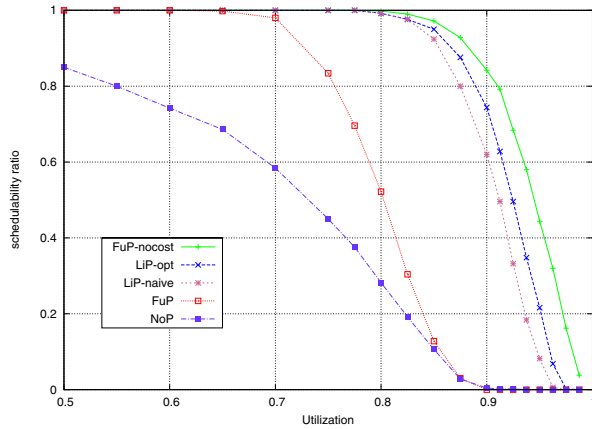


Fig. 7. Percentage of schedulable task sets as a function of the utilization.

The non preemptive algorithm exhibits the worst performance for utilizations smaller than 0.9, while the fully preemptive scheduler without preemption cost outperforms all the others. Note that both limited preemptive approaches have a performance close to the one of the ideal fully preemptive scheduler, and much better than the one shown by the fully preemptive algorithm with preemption cost. Between the partially preemptive approaches, the one proposed in this paper outperforms the other.

Figure 8 illustrates the results of another test, where the percentage of schedulable task sets is shown as a function of the number of tasks. In this setup, the task set utilization was fixed at $U = 0.9$. Note that the fully preemptive approach presents a very poor performance, because the cost of such a high number of preemptions leads to a total computation time higher than the one achieved using limited preemptive algorithms. The graph shows that the impact of the preemption cost awareness is higher for a small number of tasks, because, under this condition, each task presents more effective preemption points, amplifying the advantages of the proposed selection algorithm.
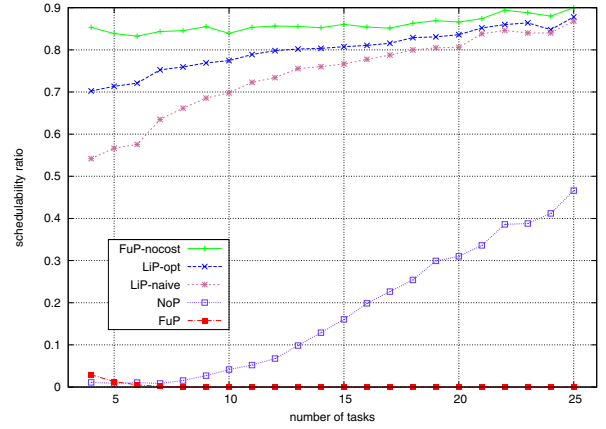


Fig. 8. Percentage of schedulable task sets as a function of the number of tasks.

The next experiment is aimed at showing the effects of the maximum preemption cost $\xi_{max}$ on the task set schedulability. We show here the case with $n = 7$ and total utilization $U = 0.95$, in order to have a large number of generated task sets that are close to the schedulability border. Figure 9 shows that even the Fully Preemptive algorithm without preemption cost is able to schedule less than one half of the generated task sets. The performance of the LiP-opt algorithm are significantly better than the LiP-naive algorithm, which has a performance degradation that is twice as fast. The FuP algorithm, which takes preemption costs into account, performs very poorly, while the non-preemptive algorithm is never able to produce a feasible schedule with the generated tasks.

Such a big difference between the two limited preemption algorithms can be explained by considering a cascade effect. In fact, selecting the EPPs with smaller cost reduces the global WCET of the task. This leads to a reduced task utilization, but also to a larger value of $Q$ for tasks with lower priority, which causes a reduction of preemption points to achieve schedulability. Such a cumulative effect is significant and can be appreciated even with a small task set, like the one used in the experiment.

Increasing the variability of preemptions costs produces the results reported in Figure 10, which shows that the advantage of the proposed algorithm with respect to the others is always significant and improves along with the variance $\sigma$. Note that a large $\sigma$ implies a larger average preemption overhead, as
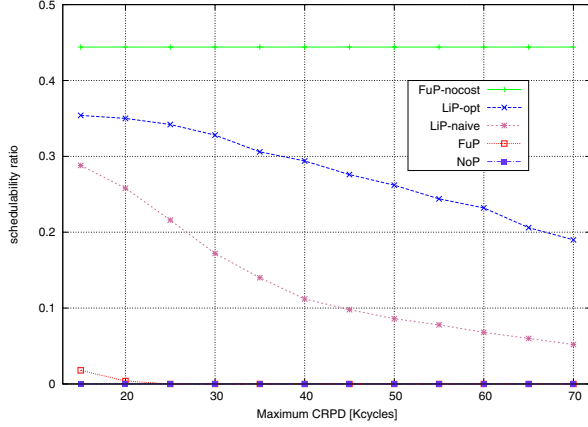
Fig. 9. Percentage of schedulable task sets as a function of the maximum CRPD.

we showed in Figure 6. However, it also implies a higher probability of having a PPP with low overhead that will be selected by the proposed algorithm. While the other algorithms suffer the increment of the average preemption cost, our approach compensates it because of its capability of finding a PPP with lower cost, significantly increasing the number of schedulable task sets.
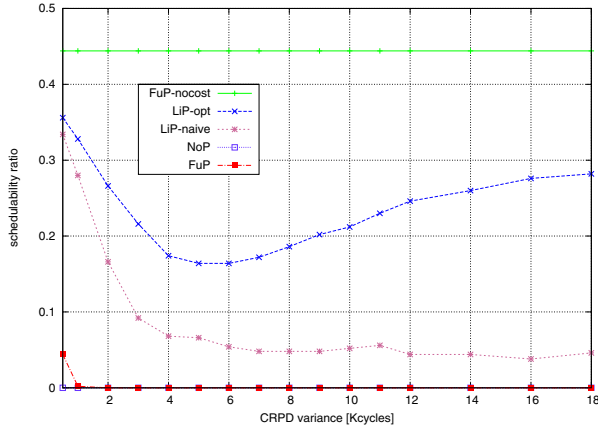


Fig. 10. Percentage of schedulable task sets as a function of the variability of the CRPD (variance $\sigma$).

## 8 OPEN PROBLEMS

The optimality of the proposed algorithm for the activation of the preemption points depends on assumptions A1. and A2. These assumptions have been introduced in order to simplify the analysis, ignoring the dependency of preemption overhead on the preempting task, and allowing the WCET of a task $\tau_i$ to be expressed using Equation (1). A more detailed analysis is possible if the preemption overhead of $\tau_i$ is considered as a function of the preempting task. For instance, tasks that have a small cache footprint will not be able to invalidate all the useful blocks that $\tau_i$ loaded in the

cache before being preempted. As a consequence, relaxing Assumption A1., the preemption overhead of $\tau_i$ at the $k$-th PPP could be expressed as an array of $i - 1$ values $\xi_{i,k}^j$, one for each potential preempting task $\tau_j$. How to derive an optimal placement algorithm for such an improved model is an interesting problem that we leave as a future work.

Another interesting open problem is finding an optimal EPP selection method relaxing Assumption A2., i.e., considering which of the selected EPPs will possibly lead to a preemption. For example, there can be cases in which many EPPs need to be activated for a task $\tau_i$ in order to avoid an excessive blocking to higher priority tasks, but only a few of them could effectively lead to a preemption. Consider the following example.

**Example 1.** *A task set is composed of two tasks $\tau_1$ and $\tau_2$ to be scheduled with limited preemption EDF or FP. Task $\tau_1$ has deadline $D_1 = 10$, period $T_1 = 100$, and is composed of a single basic block of length $C_1 = 1$. Task $\tau_2$ has deadline $D_2 = 16$, period $T_2 = 100$, and is composed of four basic blocks of length $4, 4, 2, 2$, respectively. The three PPPs of $\tau_2$ have a preemption overhead of $3, 5, 3$, respectively.*

*Note that, according to Theorem 1, the maximum allowed NPR of $\tau_2$ is $D_1 - C_1 = 9$. Possible EPP selections are (i) activating the first and third PPP, with a total overhead of 6 units, or (ii) activating only the second PPP, with a total overhead of 5 units. All other possible choices are either infeasible or redundant. Applying procedure PPP_SELECT to $\tau_2$ with $Q_2 = 9$ activates only the second PPP, leading to a WCET $C_2 = 12 + 5 = 17$, and the task set is clearly not schedulable. However, considering that task $\tau_1$ has a period of 100, it can preempt each job of $\tau_2$ at most once. Therefore, an EPP selection that activates only the first and third PPP of $\tau_2$ leads to a WCET $C_2$ of at most $12 + 3 = 15$. Considering the additional interference of 1 unit by $\tau_1$, the task set is schedulable.*

In the above example, both the first and the third PPP have to be activated in order to avoid an excessive blocking to $\tau_1$. However, only one of them will possibly lead to a preemption. Note that procedure PPP_SELECT is optimal only under the assumption that all EPPs lead to a preemption. When this assumption is relaxed, the EPPs activated by PPP_SELECT are not guaranteed to minimize the overall WCET. This is true even for implicit deadline task sets, as could be proved with simple examples. We leave the problem of finding an optimal EPP activation relaxing Assumption A2. as a future work.

One last open problem that is worth mentioning is considering applications that are not modeled as a sequential flow of basic blocks. Typical applications are composed of many conditional branches and loops, which could cover significant portions of the task code. Requiring all loops and branches to be contained within one BB could be very constraining, limiting the applicability of the proposed approach. Therefore, it would be particularly interesting to extend the model, considering applications that can have a PPP even inside a conditional branch or a loop. Under this model, it is difficult to understand what is the best combination of

EPPs that guarantees the WCET of a task to be minimized, independently of the particular branch of basic blocks that will be conditionally executed. We believe that optimal solutions with linear complexity are unlikely for this problem, due to the cross dependencies that might arise between different conditional structures of the same application. Designing an algorithm that is able to solve this problem with a reasonable complexity would be indeed an interesting achievement.

# 9 CONCLUSIONS

This paper presented an algorithm for automatically setting a number of preemption points within the code of each task, to minimize the overall preemption cost under schedulability constraints. Preemption points are selected among a larger predefined set of potential preemption points, defined by the programmer at design time. The adopted task model fits well with tasks consisting of a sequence of basic function blocks, as those produced by code generators running in standard design tools, like Simulink.

The algorithm was proved to have a linear complexity and to be optimal, in the sense that, if a feasible schedule is not found by the proposed method, then no other strategy can lead to a feasible solution. Extensive experiments demonstrated the effectiveness of the proposed approach in increasing the system schedulability with respect to other algorithms.

# REFERENCES

[1] Designing an f-14 high angle of attack pitch mode control. Technical report. Available on line at: http://www.mathworks.com.

[2] Modeling an automatic transmission controller. Technical report. Available on line at: http://www.mathworks.com.

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

[4] Sebastian Altmeyer and Claire Burguière. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS '09)*, Dublin, Ireland, July 2009.

[5] Sebastian Altmeyer, Claire Burguière, and Reinhard Wilhelm. Computing the maximum blocking time for scheduling with deferred preemption. In *Workshop on Software Technologies for Future Dependable Distributed Systems*, 2009.

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

[7] 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.

[8] Sanjoy Baruah, Aloysius K. Mok, and Louis E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th Real-Time Systems Symposium*, Orlando, Florida, 1990. IEEE Computer Society Press.

[9] Luca Benini, Davide Bertozzi, Alessandro Bogliolo, Francesco Menichelli, and Mauro Olivieri. Mparm: Exploring the multi-processor soc design space with systemc. *The Journal of VLSI Signal Processing*, 41:169–182, 2005.

[10] M. Bertogna, G. Buttazzo, M. Marinoni, G. Yao, F. Esposito, and M. Caccamo. Preemption points placement for sporadic task sets. In *ECRTS*, Bruxelles, Belgium, June 2010.

[11] Marko Bertogna and Sanjoy Baruah. Tests for global EDF schedulability analysis. *IEEE Transactions on Industrial Informatics*, 2010. Accepted for publication. Available on the first author's web page.

[12] Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Syst.*, 30(1-2):129–154, 2005.

[13] 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.

[14] R.J. Bril, J.J. Lukkien, and W.F.J. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption. *Real-Time Systems: The International Journal of Time-Critical Computing*, 42(1-3):63–119, 2009.

[15] Alan Burns. Preemptive priority-based scheduling: an appropriate engineering approach. In Sang H. Son, editor, *Advances in real-time systems*, pages 225–248. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[16] 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.

[17] M. Destelle and J.-L. Dufour. Deterministic scheduling reconciles cache with preemption for WCET estimation. In $ERTS^2$, Toulouse, France, May 2010.

[18] 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.

[19] 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.

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

[21] Daniel Kästner et al. Timing validation of automotive software. In *Leveraging Applications of Formal Methods, Verification and Validation*, volume 17 of *Communications in Computer and Information Science*, pages 93–107. Springer Berlin Heidelberg, 2009.

[22] U. Keskin, R.J. Bril, and J.J. Lukkien. Exact response-time analysis for fixed-priority preemption-threshold scheduling. In *Work-in-Progress Session (WiP) of the 15th International Conference on Emerging Technologies and Factory Automation (ETFA)*, Bilbao, Spain, September 2010.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

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

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

[33] 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.

[34] 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.

[35] 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.