Tightening the Bounds on Cache-Related Preemption Delay in Fixed Preemption Point Scheduling*

Filip Marković[1], Jan Carlson[2], and Radu Dobrin[3]

1 **Mälardalen Real-Time Research Center, Mälardalen University, Västerås, Sweden**
`filip.markovic@mdh.se`
2 **Mälardalen Real-Time Research Center, Mälardalen University, Västerås, Sweden**
`jan.carlson@mdh.se`
3 **Mälardalen Real-Time Research Center, Mälardalen University, Västerås, Sweden**
`radu.dobrin@mdh.se`

Abstract

Limited Preemptive Fixed Preemption Point scheduling (LP-FPP) has the ability to decrease and control the preemption-related overheads in the real-time task systems, compared to other limited or fully preemptive scheduling approaches. However, existing methods for computing the preemption overheads in LP-FPP systems rely on over-approximation of the evicting cache blocks (ECB) calculations, potentially leading to pessimistic schedulability analysis.

In this paper, we propose a novel method for preemption cost calculation that exploits the benefits of the LP-FPP task model both at the scheduling and cache analysis level. The method identifies certain infeasible preemption combinations, based on analysis on the scheduling level, and combines it with cache analysis information into a constraint problem from which less pessimistic upper bounds on cache-related preemption delays (CRPD) can be derived.

The evaluation results indicate that our proposed method has the potential to significantly reduce the upper bound on CRPD, by up to 50% in our experiments, compared to the existing over-approximating calculations of the eviction scenarios.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases Real-time systems, CRPD Analysis, WCET analysis, Limited Preemptive Scheduling, Fixed Preemption Point Approach

Digital Object Identifier 10.4230/OASIcs.WCET.2017.4

1 Introduction

Preemption-related overheads are of significant importance in real-time scheduling, as they may have a decisive impact on the overall system schedulability. It has been shown that, in some cases, the cumulative preemption overhead may increase a task's execution time up to 33% [11].

In this context, the Limited Preemptive Scheduling (LPS) paradigm has emerged as a valuable scheduling approach in order to reduce the overall preemption overhead of such

* We want to express our gratitude to the EUROWEB+ project that partially funded this work.

systems, and, consequently, to increase their schedulability. In this paper, within LPS we consider Fixed Preemption Points (LP-FPP) because it provides more predictability with respect to the calculation of the preemption-related overhead, compared to Preemption Thresholds Scheduling and Deferred Preemption Scheduling. The dominance considering the overhead calculation predictability in LP-FPP comes from the fact that the preemption points are selected off-line, during the design phase of the system, unlike the other two LPS approaches where the preemption points are known only at runtime.

In order to calculate the preemption-related overhead we need to consider different types of costs such as: scheduling cost, pipeline cost, bus-related cost, et cetera. However, the cache-related preemption delay (CRPD) is often the largest part of the preemption-related overhead [3], and it denotes the time that is needed for reloading the cache lines evicted by the preempting task.

The basic CRPD calculation needs to consider the following factors: (1) *The point $\mathcal{P}$ in the code of the preempted task where the preemption occurs*; (2) *Cache blocks used until $\mathcal{P}$ that may be evicted by one or more higher priority (preempting) tasks and reused afterwards in the remaining execution of the preempted task*; and finally (3) *The evicting cache blocks of the preempting tasks.* Considering this knowledge, we can see that the major difference between fully preemptive and LP-FPP scheduling comes from the fact that in LP-FPP a preemption point selection defines the non-preemptive regions between consecutive points and consequently affects the set of useful cache blocks for each selected preemption point. Contrary, in a fully preemptive scheduling we cannot control the number of preemption points and useful cache blocks per preemption point, therefore the CRPD value is defined per task. Despite the significant research done for the CRPD computation under fully preemptive scheduling, e.g., [1, 2, 10], relatively little work is done considering the LP-FPP approach. This led to the bifurcation in the research between scheduling and CRPD analysis, that is visible in the fact that a majority of the papers considering LP-FPP schedulability tests e.g., [4, 12] assumed that the upper bound CRPD values are provided for each preemption point. Such model for CRPDs introduces a considerable overestimation of a cumulative CRPD value per task, especially in the sporadic task model where we need to assume that the preempting tasks may be released at any time instant after the minimum inter-arrival time from their previous deadline. Consequently, we need to assume that each preemption point may experience the worst case eviction scenarios which is often not feasible for all preemption points of a task.

Recently, Cavicchio et al. [8] proposed the CRPD computation for the LP-FPP approach that considers the CRPD for each pair of adjacent preemption points, and it significantly improves the CRPD estimation under LP-FPP. However, the assumption about the evicting cache blocks from preempting tasks still remains an overestimation as it assumes that each pair of adjacent preemption points is affected by all higher priority tasks.

In this paper, we propose the methodology for tightening the upper bounds on the CPRD values in sporadic task sets, scheduled under the Fixed Priority LP-FPP paradigm, by using a less pessimistic calculation of the evicting cache blocks of the preempting tasks. In our proposed approach, we calculate a single CRPD value per preempted task by integrating the scheduling- and cache level analysis to identify infeasible preemptions and further formulate a constraint satisfaction models. The preliminary evaluation results indicate that our methodology can significantly reduce the upper bound of CRPD values up to 50% compared to the existing methods for ECB calculation.

The remainder of the paper is organised as follows: In Section 2 we introduce the system model, terminology and notations used in the paper which describe the task and cache model.

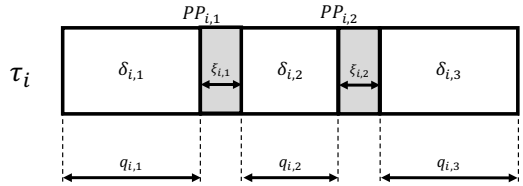

Figure 1 Task model with two selected preemption points ($PP_{i,1}$ and $PP_{i,2}$ with respective CRPD overheads $\xi_{i,1}$ and $\xi_{i,2}$) which form three non-preemptive regions ($\delta_{i,1}$, $\delta_{i,2}$ and $\delta_{i,3}$ with respective worst case execution times $q_{i,1}$, $q_{i,2}$ and $q_{i,3}$)

In Section 3 we describe the related work and problem formulation of the paper. In Section 4 we describe the proposed methodology for tightening the upper CRPD bound in the LP-FPP task model. In Section 5 we discuss the preliminary evaluation of the methodology and Section 6 concludes the paper.

2 System Model

We consider a real-time sporadic task model Γ composed of n tasks τ_i $(1 \leq i \leq n)$ scheduled under the Fixed Priority (FP) paradigm on a single processor. Tasks are ordered in a decreasing priority order (and each task τ_i generates an infinite number of task instances $\tau_{i,j}$). Each task is described with a tuple $\{P_i, C_i^{np}, D_i, T_i\}$ where P_i denotes the task's priority, C_i^{np} denotes the non-preemptive worst case execution time of $\tau_{i,j}$. Considering each task instance, D_i denotes a relative deadline, and the minimum inter-arrival time between two consecutive instances of τ_i is denoted with T_i.

We also consider the LP-FPP approach and therefore assume the sequential task splitting model such that each task is divided by d selected preemption points $PP_{i,k}$ $(1 \leq k \leq d)$ with an estimated preemption-related overhead value $\xi_{i,k}$. Furthermore, selection points form $d+1$ non-preemptive regions $\delta_{i,k}$ with worst case execution times $q_{i,k}$ such that $\sum_{k=1}^{d+1} q_{i,k} = C_i^{np}$ (see Figure 1).

We furthermore extend the system model in order to consider preemption-related overhead calculation assuming a direct-mapped cache.

As previously mentioned, the preemption-related overhead is primarily caused by the cache-related preemption delay (CRPD) compared to the other cost types such as scheduling cost, pipeline cost etc., which we consider as constants that are already included in the non-preemptive execution time C_i^{np} of a task τ_i. We denote the upper bound on CRPD of a task τ_i with γ_i and furthermore we denote with C_i^{γ} the worst case execution time considering preemptions on τ_i such that $C_i^{\gamma} = C_i^{np} + \gamma_i$.

As γ_i represents the time needed for reloading the cache blocks of the preempted task τ_i that are evicted by the preempting tasks τ_h, it is bounded by the following equation $\gamma_i = g \times BRT$ where g is the upper bound on the number of needed cache block reloads caused by preemptions, precisely evictions of cache blocks belonging to τ_i, and BRT is a cache block reload time. Furthermore, in order to calculate g we define two concepts of cache blocks considering the preemptions:

Useful cache block (UCB) – As proposed by Lee et al. [10], UCB at preemption point $PP_{i,k}$ is a memory block m of the preempted task such that: a) m may be cached at $PP_{i,k}$, and b) m may be reached from $PP_{i,k}$ and reused at program point $\mathcal{P}$ that is reachable from $PP_{i,k}$ without eviction of m on this path. If the preemption occurs at $PP_{i,k}$ we need to address only the memory blocks that are cached and may be reused. More formally,

considering a non-preemptive region $\delta_{i,k}$ we define a set of useful cache blocks $UCB_{i,k}$ such that $m \in UCB_{i,k}$ if and only if non-preemptive region $\delta_{i,k}$ has m as a useful cache block at point $PP_{i,k}$. Furthermore, we define the set of useful cache blocks UCB_i per τ_i: $UCB_i = \bigcup_{\delta_{i,k} \in \tau_i} UCB_{i,k}$ $(1 \leq k \leq d)$. Notice that we do not consider the useful cache blocks of the last non-preemptive region $\delta_{i,d+1}$ as it cannot be preempted.

Evicting cache block (*ECB*) – defines a memory block m of the preempting task that may be accessed during the execution of the preempting task.

In this paper we also consider the evicting cache set ECB_i of a task τ_i such that a memory block $m \in ECB_i$ if and only if τ_i may evict cache block m.

Finally, in some cases we consider the upper bound on the single preemption cost $\xi_{i,k}$ at preemption point $PP_{i,k}$ which is computed considering the useful memory blocks $UCB_{i,k}$ and the union of the evicting cache blocks ECB_h from all possibly preempting tasks:

$$\xi_{i,k} = \left| UCB_{i,k} \cap \left(\bigcup_{h \in hp(i)} ECB_h \right) \right| \times BRT \,.$$

3 Related Work

In the seminal paper that considered preemption cost aware schedulabity tests, Busquets et al. [6] considered an upper bound on the preemption cost of a single job of a preempting task τ_h that executes during the response time of a preempted task τ_i, which is defined by a term $\gamma_{i,h}$. This value is calculated for each higher priority job that may be released during the response time. Later, Petters et al. [13] proposed the more precise analysis considering the upper bound on the preemption cost of all jobs of τ_h executing during the response time of τ_i. By using this approach they were able to reduce the pessimism from the previously proposed method that considered the upper bound on preemption cost of each τ_h's job separately. Considering exact $\gamma_{i,h}$ computation there are two dominant approaches: UCB-union and ECB-union, and both of them assume that the evicting cache blocks of the preempting jobs are given and will definitely evict intersected useful cache blocks of the preempted task. However, those approaches do not account for the fact that the additional preemptions may result in a smaller preemption costs than the prior accounted ones. This fact was accounted by introducing the multiset-based computation of CRPD, proposed by Staschulat et al. [15] with a drawback that it over-estimates the number of preemptions that have an impact on the response time and does not account for the variability in the number of possible intermediate preemptions. Therefore, Altmeyer et al. [1] proposed ECB/UCB-Union Multiset approaches, which account for the precise number of possible intermediate preemptions.

Considering periodic task systems, Ramaprasad and Miller [14] analysed the feasible preemptions by analysing each job of a preempted task throughout the hyper-period considering the worst-case placement of preemptions of preempting tasks. In this paper, they considered that not every preempting job of τ_h can cause a preemption on the preempted task τ_i.

In the LP-FPP approach, the majority of the papers defining the schedulability tests e.g., [4, 7, 12] assume the upper bound on preemption cost values for each preemption point. In sporadic task systems this approach can lead to a huge overestimation of the preemption cost because the release time of the higher priority job is not definitely known, thus we would need to assume that each point suffers from the worst case eviction scenario. Improving the assumption of per point preemption cost, Cavicchio et al. [8] proposed the CRPD computation for the LP-FPP approach that considers the CRPD for each pair of adjacent preemption points which significantly improved the CRPD estimation under LP-FPP. However, the assumption about the evicting cache blocks from preempting tasks still remains

Data: Task set Γ
Result: Set of tightened preemption overhead values for each task from Γ

1 **for** $i \leftarrow 2$ **to** n **do**
2 $V \leftarrow$ Generate a set of variables that represent a preemption affection by τ_h at $PP_{i,k}$ of τ_i
3 $C \leftarrow$ Generate a set of constraints that define infeasible preemption combinations
4 $G \leftarrow$ Generate a goal function for the given constraint problem
5 $\gamma_i \leftarrow$ compute CRPD of τ_i by solving the constraint problem defined by V, C and G
6 $C_i^{\gamma} \leftarrow C_i^{np} + \gamma_i$
7 **end**

Algorithm 1: Algorithm for tightening the upper bound of CRPD in a taskset.

an overestimation as it assumes that each pair of adjacent preemption points is affected by all higher priority tasks.

Contrasting these methods, we propose an improved method for tightening the bounds on the preemption cost calculations in sporadic LP-FPP task scheduling, by identifying infeasible eviction scenarios.

4 Computing CRPD bounds

In order to reduce the CRPD overestimation we propose a method that for each preempted task τ_i in a taskset first identifies infeasible preemption combinations, and then calculates the maximum CRPD considering the remaining preemption combinations. Concretely, the basic idea of the method is to identify cases where two instances of a higher priority task τ_h cannot affect the preemption costs at two preemption points of the same τ_i instance.

Considering the taskset Γ, we propose an algorithm that computes a CRPD upper bound γ_i for each task in a decreasing priority order (see Algorithm 1).

For each task, we first (line 2) generate a set of variables, each representing the case when a certain higher priority task τ_h affects the preemption cost of a certain preemption point $PP_{i,k}$ of the considered task τ_i. Next (line 3), we generate a set of constraints that capture identified infeasible preemption combinations, and a goal function (line 4) representing the preemption cost associated with the different preemption scenarios. Furthermore (line 5), we compute the upper-bounded cache-related cost γ_i of the preempting task τ_i by solving the constraint problem defined by V, C and G. Next, we compute the execution time of τ_i considering the CRPD upper bound C_i^{γ} (line 6), to be used in the following iterations. This is done because the preemptive execution time of the higher priority task may impact the CRPD computation of the lower priority task and because we need to address the case of nested preemptions.

We describe in detail each of the computations in the following subsections. To illustrate the approach, we will use the tasks depicted in Figure 2 as a running example, focusing on the second iteration of the algorithm, meaning that the preempted task τ_3 is being considered.

In the example, we also consider the preempting tasks (τ_1 and τ_2). Furthermore, τ_1 has the worst case execution time $C_1 = 30$ and the minimum inter-arrival time $T_1 = 100$. Its set of evicting cache blocks ECB_1 consists of two cache blocks presented by integers 1 and 2. The preempted task τ_3 is divided by three selected preemption points ($PP_{3,1}$, $PP_{3,2}$ and $PP_{3,2}$) and it consists of four non-preemptive regions with given WCET values ($q_{3,1} = 20$, $q_{3,2} = 10$, etc.). Set of useful cache blocks, e.g., $UCB_{3,1}$, is defined for each preemption point.

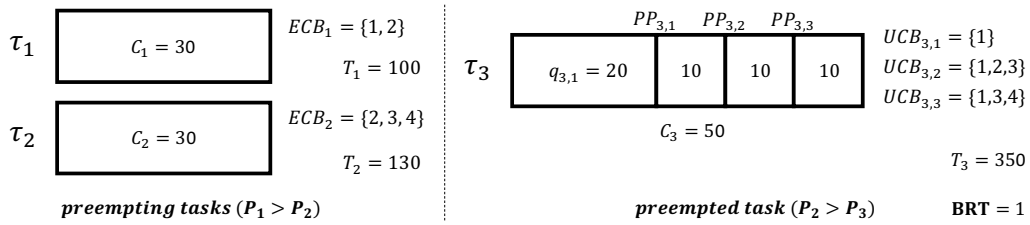

Figure 2 Running example: Two higher priority tasks τ_1 and τ_2 and the preempted task τ_3.

4.1 Variable declaration

Considering a preemption point $PP_{i,k}$ of a preempted task τ_i and a single preempting task τ_h, we declare a boolean variable $X_{h,k}$ which represents the case when τ_i is preempted at $PP_{i,k}$ and the associated preemption cost $\xi_{i,k}$ is affected by an instance of τ_h. Consequently, for τ_i we generate $d \times (i-1)$ boolean variables $X_{h,k}$ where d is the number of preemption points of τ_i, since $(i-1)$ is the number of tasks with a higher priority than τ_i. Formally, the set V of generated variable declarations is defined as:

$$V = \left\{ X_{h,k} \in \{0,1\} \;\middle|\; (1 \leq h < i) \wedge (1 \leq k \leq d) \right\}.$$

The set of variable declarations for τ_3 in the running example is:

$$V = \left\{ X_{1,1} \in \{0,1\},\ X_{1,2} \in \{0,1\},\ X_{1,3} \in \{0,1\},\ X_{2,1} \in \{0,1\},\ X_{2,2} \in \{0,1\},\ X_{2,3} \in \{0,1\} \right\}.$$

4.2 Constraint formulation

Considering the constraints, we are interested in identifying infeasible preemption combinations, focusing on preemption scenarios where two instances of a preempting task τ_h can directly affect the cost of only one of the two preemption points $PP_{i,k}$ and $PP_{i,l}$ $(k < l \leq d)$ of a preempted task τ_i, but not both.

To formally define the constraint generation, we first define the computation of the maximum time interval $I_i^{k,l}$ from the start time of $\delta_{i,k}$ until the start time of $\delta_{i,l+1}$. Informally, $I_i^{k,l}$ represents the time interval during which the two instances of τ_h must be released in order to affect both preemption points.

The definition of $I_i^{k,l}$ is based on the traditional response time analysis, but considering only a subset of the task, from $\delta_{i,k}$ to $\delta_{i,l}$. The longest interval is found by assuming that the higher priority tasks arrive as early and as often as possible in the considered interval. Since there is no possibility of blocking by lower priority tasks in this case, we only sum the execution time of the non-preemptive regions with respective upper-bounded preemption costs (from k to l) and the interference from the higher priority tasks. Each upper-bounded preemption cost $\xi_{i,w}$ is added to the respective execution time $q_{i,w}$ of the non-preemptive region $\delta_{i,w}$ in order to account for the impact of the CRPD on the length of the non-preemptive region. The iterative computation of $I_i^{k,l}$ is defined as follows:

$$\begin{cases} I_i^{k,l}(0) = \sum_{w=k}^{l} (q_{i,w} + \xi_{i,w}) + \sum_{h \in hp(\tau_i)} C_h^{\gamma}, \\ I_i^{k,l}(z) = \sum_{w=k}^{l} (q_{i,w} + \xi_{i,w}) + \sum_{h \in hp(\tau_i)} \left(\left\lfloor \frac{I_i^{k,l}(z-1)}{T_h} \right\rfloor + 1 \right) C_h^{\gamma}. \end{cases}$$

Finally, we define $I_i^{k,l}$ to be the least fixed point of the recursion, i.e., $I_i^{k,l} = I_i^{k,l}(z)$ where z is the lowest value for which $I_i^{k,l}(z) = I_i^{k,l}(z+1)$.

Next, we show how the time interval $I_i^{k,l}$ can be used to identify infeasible preemption combinations.

▶ **Lemma 1.** *If task τ_h affects the preemption cost at $PP_{i,k}$, then an instance of τ_h was released between the start of $\delta_{i,k}$ and the start of $\delta_{i,k+1}$.*

Proof. An instance of τ_h released before the start of $\delta_{i,k}$ will either not preempt τ_i at all, or execute during a preemption before the start of $\delta_{i,k}$. An instance released after the start of $\delta_{i,k+1}$ clearly did not affect a preemption at $PP_{i,k}$. ◀

Note that an instance released after the end of $\delta_{i,k}$ cannot *cause* a preemption at $PP_{i,k}$, but it can still affect the preemption overhead if it arrives during the execution of some other task that caused the preemption.

▶ **Corollary 2.** *If task τ_h affects the preemptions at both preemption points $PP_{i,k}$ and $PP_{i,l}$ (where $k < l$) in the same instance of τ_i, then two instances of τ_h were released between the start of $\delta_{i,k}$ and the start of $\delta_{i,l+1}$.*

Proof. A single instance of τ_h cannot affect both preemptions, and the interval in which they were released is given by Lemma 1. ◀

▶ **Proposition 3.** *If $I_i^{k,l} \leq T_h$ then task τ_h cannot affect one instance of τ_i at both preemption points $PP_{i,k}$ and $PP_{i,l}$.*

Proof. Proof by contradiction: Assume that $I_i^{k,l} \leq T_h$ and that τ_h affects one instance of τ_i at both preemption points $PP_{i,k}$ and $PP_{i,l}$. Then, by Corollary 2, two instances of τ_h were released between the start of $\delta_{i,k}$ and the start of $\delta_{i,l+1}$, and thus within an interval of length $I_i^{k,l}$. This contradicts the minimum inter-arrival time T_h. ◀

Finally, per τ_i we generate a set C of constraints stating that at most one of $X_{h,k}$ and $X_{h,l}$ be true at the same time, for all cases where the inequality $I_i^{k,l} \leq T_h$ holds:

$$C = \left\{ X_{h,k} + X_{h,l} \leq 1 \;\middle|\; (1 \leq h < i) \wedge (k+1 \leq l \leq d) \wedge (I_i^{k,l} \leq T_h) \right\}.$$

Continuing the running example from Figure 2, we first compute the following $I_i^{k,l}$ values: $I_3^{1,2} = 94$, $I_3^{2,3} = 83$ and $I_3^{1,3} = 167$. In order to generate constraints, we check the proposed inequality by comparing the derived values with $T_1 = 100$ and $T_2 = 130$ and e.g, get that: $I_3^{1,2} \leq 100$, therefore, we generate the constraint $X_{1,1} + X_{1,2} \leq 1$ denoting that τ_1 cannot directly affect the preemption cost of both preemption points: $PP_{3,1}$ and $PP_{3,2}$. The complete set of constraints for τ_3 of the running example is:

$$C = \Big\{ \underset{\uparrow}{X_{1,1} + X_{1,2} \leq 1}; \quad \underset{\uparrow}{X_{1,2} + X_{1,3} \leq 1}; \quad \underset{\uparrow}{X_{2,1} + X_{2,2} \leq 1}; \quad \underset{\uparrow}{X_{2,2} + X_{2,3} \leq 1} \Big\}.$$

$$I_3^{1,2} \leq 100 \qquad I_3^{2,3} \leq 100 \qquad I_3^{1,2} \leq 130 \qquad I_3^{2,3} \leq 130$$

4.3 Goal function formulation

For the final part of the constraint problem, we define the following goal function:

$$G = \mathit{Maximize} : \sum_{PP_{i,k} \in \tau_i} \sum_{m \in UCB_{i,k}} \min\left(1, \sum \{X_{h,k} \mid \tau_h \in hp(\tau_i) \wedge m \in ECB_h\}\right) \times BRT$$

The minimum function stands for the calculation of the impact of single memory block m that may be in the ECB set of many higher priority tasks but still only contributes at most once to the preemption cost of one preemption. During the generation, *min* functions that only contain a single $X_{h,k}$ variable (or none) can be simplified.

The optimized goal function for τ_3 of the running example is:

$$
\begin{array}{lllllllll}
G = \mathit{Maximize} : BRT \times \big(& & & & & & & & \\
X_{1,1} & + & & & & & & & \leftarrow k = 1 \\
X_{1,2} & + & \min(1, X_{1,2} + X_{2,2}) & + & X_{2,2} & + & & & \leftarrow k = 2 \\
 & & X_{1,3} & + & X_{2,3} & + & X_{2,3} \big) & & \leftarrow k = 3 \\
 & & & & & & & & \\
\uparrow & & \uparrow & & \uparrow & & \uparrow & & \\
m = 1 & & m = 2 & & m = 3 & & m = 4 & &
\end{array}
$$

The m and k values indicate the different parts of the nested sum of the goal function by denoting the actual memory block m and the preemption point k that are considered by the specified element of the sum. The empty positions in the array come from the definition of the second sum over $m \in UCB_{i,k}$. For example, since the useful cache block set of the first preemption point of τ_3 is $UCB_{3,1} = \{1\}$, the first row ($k = 1$) intersects only with the $m = 1$ column. In this example, only one *min* function remained after the simplification, since memory block $m = 2$ is the only block that is in the ECB sets of both preempting tasks τ_1 and τ_2.

The final result from the solver for the running example corresponds to the preempting scenario where $PP_{3,2}$ is affected by τ_1 and $PP_{3,3}$ is affected by τ_2. Concretely, τ_1 evicts the useful cache blocks 2 and 3 from $UCB_{3,1}$, and τ_2 evicts useful cache blocks 3 and 4 from $UCB_{3,3}$. Since the block reload time is $BRT = 1$, the finally computed upper-bounded preemption cost of the task τ_3 is $\gamma_3 = 4$.

5 Experimental Results

We conducted two experiments, using the open-source Java constraint programming library Choco [9], in order to investigate the possibility of tightening the CRPD per taskset using the proposed method. Each point in the experiment represents an average of applying the algorithm to 2000 randomly generated tasksets.

In each experimental setup we generate tasksets consisting of the defined number of tasks. Each taskset is generated with the utilization level of 0.8 and the tasks' individual utilizations are generated using the U-unifast algorithm [5]. Then, the minimum inter-arrival time for each task is generated using a uniform distribution from the range $[5ms, 5s]$, thus reasonably corresponding to real systems. Next, we calculate the execution times such that $C_i^{np} = U_i \times T_i$ and assign priorities in a rate monotonic order. The number of resulting non-preemptive regions are generated from the uniform distribution in range $[1, 10]$.

Regarding the cache simulation we represent evicting and useful cache block sets using the integer values from 0 to 256, corresponding to the maximum cache set size $CS = 256$. Following the setup used by Altmeyer et al. [3], we use the following values:

- $BRT = 8\mu s$.
- ECB_i for each task is generated using the U-unifast algorithm such that the total cache utilization $CU = \sum_1^n |ECB_i|/CS$. If the utilization of the evicting set is above 1, it means that it uses all cache sets.
- UCB_i for each task is generated from the ECB_i using the reload factor RF which is uniformly generated from the range $[0, 0.3]$, thus $UCB_i = RF \times |ECB_i|$ where $|ECB_i|$ is

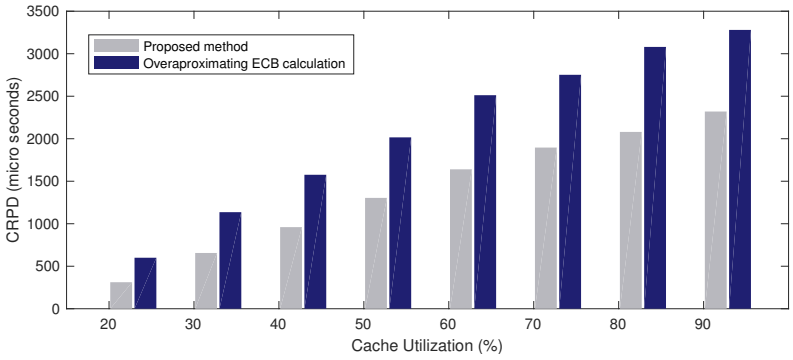

Figure 3 CRPD estimation per taskset for different levels of cache utilization, calculated as the average over the 2000 generated tasksets.

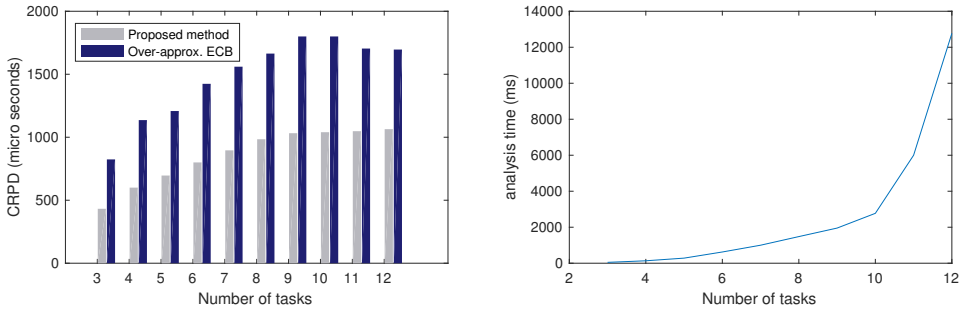

Figure 4 Left: CRPD estimation per taskset, for different taskset sizes, calculated as the average over the 2000 generated tasksets. Right: Analysis time of the proposed method per taskset, for different taskset sizes.

the number of cache sets in the ECB_i. The reload factor is used to adapt the assumed reuse factor, thus reflecting systems with a low reuse up to control-based applications with heavy reuse (up to 0.3).

- Each $UCB_{i,k}$ is generated using the uniform distribution from $[0, 100 \times |UCB_i|]$.

In the first experiment, we evaluated the CRPD estimation varying the total cache utilization from 20% to 90% (see Figure 3). Taskset utilization was fixed to 0.8 and the number of tasks was fixed to 10. We show the CRPD values computed by the proposed method, compared to the method where the worst case eviction scenario is assumed for each preemption point. As expected, the proposed method succeeds in tightening the CRPD value by identifying infeasible eviction cases. The increase of cache utilization is followed by an increase of the computed CRPD values. However, we see that the CRPD reduction ratio is decreasing by the increase of the cache utilization (when $CU = 20\%$, the reduction ratio is 49% and when $CU = 90\%$ it is less than 30%). This is the case because of the fact that with high cache utilization each higher priority task evicts a significant part of the total cache. Therefore, the benefit of identifying infeasible eviction cases is reduced since the remaining cases still evict much of the total cache in each point.

In the second experiment, we evaluated the CRPD estimation varying the taskset size from 3 to 12 (see Figure 4). Taskset utilization was fixed to 0.8 as well as the total cache utilization, $CU = 40\%$. From 3 to 9 tasks, the ratio between the CRPD estimated by the proposed method and the over-approximating ECB computation remains roughly the same

(48% for 3 tasks in a taskset, and 42% for 9 tasks). From this point, the over-approximated CRPD values drop because the evicting cache blocks of the tasks are significantly reduced as we fix the total cache utilization per taskset. Therefore, considering 12 tasks in a taskset the CRPD reduction ratio is 35%. To investigate scalability of the approach, we also measured the analysis time as the number of tasks increases. We can see that on average the method needs approximately 12 seconds for tasksets consisting of 12 tasks, and only 48 *ms* for tasksets consisting of 3 tasks. Note that the analysis time varies a lot for different tasksets, and in this experiment there are some cases for which the constraint solver needs several minutes to solve the given problem. Therefore we used a time limit of 40 seconds, and if the method failed to provide a value within this time, it instead reported the over-approximated CRPD value.

6 Conclusions

In this paper we proposed a novel method for computing the cache-related preemption delay (CRPD) in sporadic task model scheduled under the Fixed Preemption Point approach. The method identifies infeasible preemption combinations for a given preempted task based on scheduling level analysis and then calculates the maximum CRPD value considering the remaining preemption combinations and cache level analysis information. Furthermore, we defined an algorithm that iteratively estimates the upper bound on CRPD values in a taskset, using the derived upper bounds for computing the tighter CRPD estimates for lower priority tasks. In the evaluation of the proposed method, we showed that it can significantly reduce the CRPD of a preempted task and moreover all tasks in a taskset, up to 50% as shown in some cases.

For future work, we envision a preemption point selection algorithm that will exploit the proposed method for CRPD computation to achieve CRPD minimisation by the appropriate preemption point placement, benefiting from the information of the infeasible evicting scenarios defined in this paper. We also plan to investigate a way to manage the scalability limitations by restricting the algorithm to consider only a selected subset of tasks. In particular, the higher priority tasks since any tightening of their CRPD impacts the response times of all lower priority tasks, but there might be also other indicators to be included in the analysis. Another way to manage scalability limitations would be to combine the multiset-based CRPD estimation approaches with the analysis of infeasible preemption combinations in order to derive safe but to some extent pessimistic CRPD upper-bounds, compared to those computed with the constraint solving approach.

Finally, future work will also focus on developing an analogical method for tightening the CRPD bounds in fully preemptive systems. In this context, we envision a method that would first virtually divide the preempted task into subsections such that it accounts for the variability of useful cache blocks throughout the execution of the preempted task. By analysing the infeasible preemption scenarios among those subsections we would be able to furthermore tighten the CRPD bounds in such systems as well.

Acknowledgements. We are very grateful to Peter Puschner and TACLe summer school for the inspiration and provided knowledge which helped us to pursue this line of research. We would also like to thank the anonymous reviewers for their comments on an earlier draft of this paper and the suggestions for the continuation of the work.

References

1. Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems*, 48(5):499–526, 2012.
2. Sebastian Altmeyer, Claire Maiza, and Jan Reineke. Resilience analysis: tightening the CRPD bound for set-associative caches. *ACM Sigplan Notices*, 45(4):153–162, 2010.
3. Sebastian Altmeyer, Douma Roeland, Will Lunniss, and Robert I. Davis. Evaluation of cache partitioning for hard real-time systems. In *Proceedings Euromicro Conference on Real-Time Systems (ECRTS)*, pages 15–26, 2014. doi:10.1109/ECRTS.2014.11.
4. Marko Bertogna, Orges Xhani, Mauro Marinoni, Francesco Esposito, and Giorgio Buttazzo. Optimal selection of preemption points to minimize preemption overhead. In *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*, pages 217–227. IEEE, 2011.
5. Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.
6. José V. Busquets-Mataix, Juan José Serrano, Rafael Ors, Pedro Gil, and Andy Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Real-Time Technology and Applications Symposium, 1996. Proceedings.*, pages 204–212. IEEE, 1996.
7. Giorgio C. Buttazzo, Marko Bertogna, and Gang Yao. Limited preemptive scheduling for real-time systems. A survey. *industrial Informatics, IEEE Transactions on*, 9(1):3–15, 2013.
8. John Cavicchio, Corey Tessler, and Nathan Fisher. Minimizing cache overhead via loaded cache blocks and preemption placement. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 163–173. IEEE, 2015.
9. CHOCO: Open Source Java Library for Constraint Programming. http://www.choco-solver.org/. Accessed: 2017-04-13.
10. Chang-Gun Lee, Joosun Han, Yang-Min Seo, Sang Luyl Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sam Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.
11. Rodolfo Pellizzoni, Bach D. Bui, Marco Caccamo, and Lui Sha. Coscheduling of CPU and I/O transactions in cost-based embedded systems. In *Real-Time Systems Symposium, 2008*, pages 221–231. IEEE, 2008.
12. Bo Peng, Nathan Fisher, and Marko Bertogna. Explicit preemption placement for real-time conditional code. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, pages 177–188. IEEE, 2014.
13. Stefan M Petters and Georg Färber. Scheduling analysis with respect to hardware related preemption delay. In *Workshop on Real-Time Embedded Systems, London, UK*, 2001.
14. Harini Ramaprasad and Frank Mueller. Tightening the bounds on feasible preemptions. *ACM Transactions on Embedded Computing Systems (TECS)*, 10(2):27, 2010.
15. Jan Staschulat, Simon Schliecker, and Rolf Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Real-Time Systems, 2005.(ECRTS 2005). Proceedings. 17th Euromicro Conference on*, pages 41–48. IEEE, 2005.