Improving CRPD Analysis for EDF Scheduling: Trading Speed for Precision

Giuseppe Lipari, Fabien Bouquillon, Smail Niar

Improving CRPD Analysis for EDF Scheduling: Trading Speed for Precision*

Fabien Bouquillon[1], Giuseppe Lipari[2], and Smail Niar[3]

[1]Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France , Univ. Polytechnique Hauts-de-France, CNRS, UMR 8201 - LAMIH, F-59313 Valenciennes, France
[2]Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
[3]Univ. Polytechnique Hauts-de-France, CNRS, UMR 8201 - LAMIH, F-59313 Valenciennes, France

January 18, 2022

Abstract

Cache Related Preemption Delay (CRPD) analysis is a methodology for bounding the cost of cache reloads due to preemptions. Many techniques have been proposed to estimate upper bounds to the CRPD for Fixed Priority (FP) and Earliest Deadline First (EDF) scheduling. Given the complexity of the problem, existing methods make simplifying assumptions to speed up the analysis, but they also introduce large amounts of pessimism.

In this paper we present two contributions for reducing the pessimism in the CRPD analysis of real-time systems that use set-associative cache memories. First, we present a new way to compute a bound on the number of preemptions on a task under EDF scheduling. Second, we propose a novel algorithm that trades off speed for precision in state-of-the-art analysis in the literature. We show improvements in the schedulability ratio on classical benchmarks in comparison with the state-of-the-art.

1 Introduction

Hard real-time systems are critical systems that require a careful analysis of their temporal behavior to ensure the respect of the timing constraints. For this reason, an accurate model of the system is necessary. Unfortunately, modern COTS hardware platforms present complex features that are difficult to model and to analyze accurately. For example, modern single processor micro-controllers feature cache memory, shared bus, DMA, etc. Such features contribute to improve the average performance of the system, but are difficult to model, and many pessimistic assumptions and large approximations are introduced in the real-time scheduling analysis.

The scheduling analysis of a real-time system requires an estimation of the *Worst-Case Execution Times* (WCETs) of all the tasks, and an analysis of their interactions. The WCET is usually calculated by analyzing the task as if it executes *in isolation* on the target platform. However, even when tasks are considered to be functionally independent, they can indirectly interfere with each other due to shared hardware resources, like cache memory, bus, DMA, etc. Concerning the interference due to shared caches, a higher preemption-level task that preempts a lower preemption-level task can evict some of the cache blocks of the latter; when the lower preemption-level task resumes execution, it may need to reload the cache blocks, thus increasing its execution time.

*This is an early version of the paper accepted at the SAC'22 conference.

Several solutions to this problem have been proposed in the literature, from non-preemptive or limited-preemptive scheduling strategies, to fully preemptive systems with *Cache Related Preemption Delay* (CRPD) analysis. In this paper we focus on improving the CRPD analysis of single-processor fully preemptive real-time systems with one level of cache.

CRPD analysis works as follows: first, a WCET is computed for each task, assuming the task is executed alone on the processor. The computed WCET includes the effect of cache misses caused by the task itself (*intra-task* cache misses). A static analysis also produces one or more lists of cache blocks used by the task.

The CRPD analysis computes an upper bound to the number of cache misses caused by preemption to consider in the scheduling analysis. In Section 2 we briefly describe the current state of the art CRPD analyzes.

One limitation of existing CRPD analyzes with Earliest Deadline First scheduling policy is the computation of the number of preemptions on a given task. Since most of the existing analyzes were originally designed for Fixed Priority, they consider a *preemption interval* (that is the interval of time where a task may be preempted) to the task worst-case response time. However, computing the task response time for EDF is computationally expensive (and very complex). For this reason, existing CRPD analysis consider a preemption interval size that is equal to the relative deadline of the task, thus introducing extra pessimism.

Another source of pessimism is the computation of the set of cache blocks (the *useful cache block set*, or UCB) that may be evicted by a preemption. Existing analyzes use a single set for every possible preemption point, thus simplifying the analysis by introducing extra pessimism. We propose to consider a number $M > 1$ of UCB sets per each task, thus increasing precision without exploding the complexity.

Summarizing, in this paper we present two improvements over the state of the art CRPD analysis:

- a simple algorithm for computing a more precise *preemption interval* of a task scheduled by EDF on a single processor; by reducing the preemption interval, we tighten the upper bound on the number of preemptions a task can suffer;
- A simple algorithm to reduce the number of UCB sets per tasks to a given constant $M > 1$.

The paper is organized as follows. First we briefly discuss some related works. Then we introduce the model and the notation in Section 3. We remind the methods presented in the literature in Section 4. In Sections 5 and 6, we present our original contributions. We discuss the complexity of the preemption interval length computation in Section 7. We evaluate our methods against the state of the art [17, 3], and we present the results of the evaluation in Section 8.

2 Related Works

Many methods have been proposed in the literature to address the problem of the inter-task interference due to cache memory. We can classify them into two different approaches.

The first approach consists in limiting task preemption. However, non-preemptive systems may suffer from long blocking times: a long low preemption-level task may block an urgent high preemption-level task for the duration of its execution, causing unwanted deadline misses.

A second approach is to reduce the pessimism of the CRPD analysis. Lee et al. [16] introduced the notion of Useful Cache Block (UCB) as a block that is used by the task at some point in the code, and that will be reused by the same task later in the code. They proposed a static analysis of the binary code of the task to compute a set of UCBs for each preemption point. They also propose to compute the cost of n preemptions for a task as the size of the union of the n largest UCB lists. However, this technique suffers from a large pessimism because it analyzes each task in isolation. In particular, it does not consider the set of cache blocks of the preempting tasks.

Tan et al. [22] take into account both the preempting and the preempted tasks. In order to simplify the analysis and avoid combinatorial explosion, Tan et al. use a single set of UCB per task, obtained as the union of all UCB set for every preemption point.

In Lunniss et al. [17] and Altmeyer et al. [3], the authors propose two approaches called *UCB-union multiset* and *ECB-union multiset*. The first one consists in computing a global cost in

an interval of length t by combining the UCBs of all instances of lower preemption-level tasks in the interval, and intersecting the result with the *Evicting Cache blocks* (ECB) of the higher preemption-level task which causes the preemption. The number of elements in the resulting set is an upper bound to the number of evicted cache blocks. In the *ECB-union multiset* approach, all ECBs of higher preemption-level tasks are merged together and intersected with the UCB of the preempted task to obtain a cost of preemption for the lower preemption-level task. Since none of two methods dominates the other, the authors propose a *combined approach* to improve the precision. Their framework has been initially proposed for Fixed priority scheduling systems. It has been extended later to Earliest Deadlines First scheduling systems [17].

Shah et al. [21] reproduced CRPD analysis methods from Altmeyer et al. [3]. They show that the generation of synthetic task sets used in [3] is unrealistic, and propose a different way to generate task sets based on low-level analysis with LLVMTA. They also conclude that block reload time has a low impact on the schedulability, something that we could not confirm in our experiments. We will discuss this discrepancy in Section 8.2.

Previous papers mostly consider direct-mapped caches. Concerning the cache model and the LRU replacement policy, Burguière et al. [12] present how to adapt existing CRPD analysis from direct mapped caches to set-associative caches. They show that only adaptation for LRU replacement policy is possible, since PLRU and FIFO cannot be bounded using the number of ways.

Altmeyer et al. [4] propose to use the ages of the UCBs and the number of reloading of ECBs to eliminate the UCBs that cannot be evicted from the analysis. They demonstrate their algorithm on the Papabench benchmark suite [20] where they compare against the UCB-union approach with a cache of 8 ways and 32 cache rows. However, the number of tasks which can be involved in the same preemption is smaller than the number of cache ways, therefore the results cannot be generalized to an arbitrary cache setting.

Marković et al. [19] recently proposed two novel methodologies for Fixed priority fully preemptive scheduling which improve the CRPD cost estimation of [3]. However the performances between our approach and [19] cannot be compared due to the difference of scheduler as shown by Lunniss et al. [18].

3 System model

3.1 Task Model

We consider a system composed of N independent real-time sporadic tasks, denoted as $\mathcal{T} = \{\tau_1, \cdots, \tau_n\}$.

A task τ_i is described by the classic tuple (C_i, D_i, T_i), with C_i the worst case execution time, D_i the relative deadline, and T_i the minimum interarrival time. In this work we consider constrained deadline tasks, i.e. each task τ_i has $D_i \leq T_i$. The worst-case execution time C_i is estimated through a static analysis tool like OTAWA [6] assuming the task executing alone on the processor. We consider sporadic tasks, i.e. the exact arrivals times of the instances of the tasks are not known at analysis time, however the minimum distance between two consecutive instances of the same task is T_i.

We assume that the tasks are scheduled by Earliest Deadline First. The *preemption-level* of task τ_i is defined as $\pi_i = 1/D_i$. Baker [5] proved that, in EDF scheduling, a task τ_j may preempt a task τ_i only if $\pi_j > \pi_i$. We assume that tasks are sorted in non-ascending order of preemption-level: for any two tasks τ_j and τ_i, $\pi_j > \pi_i$ implies $j < i$.

3.2 Cache model

We assume that tasks are executed on a single processor with a L1 cache memory. We consider a **set-associative**, virtually-indexed, physically-tagged instruction cache. For simplicity, in this paper we only consider the instruction cache (see the discussion in Section 9 for data caches).

A **N-way set-associative** cache memory can be described as a matrix of **cache lines**, the basic memory block unit of a cache memory; cache lines are regrouped into **cache sets** of N elements, also called *ways*.

In case of a virtually-indexed physically-tagged memory cache, the virtual memory is divided in **cache blocks**, which are memory blocks of the size of a cache line. Each of these cache blocks

is assigned an *index* computed as the modulo between the virtual address of the cache block and the number of cache sets, which corresponds to the cache set where the cache block may be stored. As each task has its virtual memory, cache blocks from different tasks may have the same index: to differentiate them, each on is assigned a *tag* computed from their physical address.

When the processor needs to access an instruction, the index and the tag are computed from the virtual address. The tag is compared with all the cache blocks contained in the corresponding cache set: if a match is found (*hit*), then its *age* in the cache set is updated according to the replacement policy; if the tag is not found (*miss*), the cache block is loaded from the main memory and stored in the cache set according to the replacement policy.

We assume a cache memory with the *Least Recently Used* replacement policy (LRU). Each cache way in a cache set is assigned an *age*: at each access, ages are updated so that a cache block with a more recent access than another has a lower age. When a cache block has to be loaded in a cache set that is full, the least recently used cache block is evicted and the all the ages are updated accordingly.

3.3 Useful Cache Blocks and Evicting Cache Blocks

To compute the Cache Related Preemption Delay, the notions of *Useful Cache Block* (UCB) and *Evicting Cache Block* (ECB) have been defined in the literature. We recall the definitions here and introduce the notation that will be useful in the rest of the paper.

3.3.1 ECBs

The evicting cache blocks of a task consist of all the cache blocks of this task that can evict cache blocks of another task during a preemption. Most papers in the literature assume direct-mapped caches (only one cache way in the cache memory), therefore the ECBs of a task are modeled as a single set of cache set indexes. Indeed, in the case of a direct-mapped cache memory only one cache block per set can be evicted.

In the case of a set-associative cache memory with the LRU replacement policy, a preempting task that accesses a given cache block can replace all the cache blocks present in the corresponding set in a *chain reaction* [12].

Therefore, in this paper we represent the ECBs of a task as a *multiset*: for each evicting cache block index, the multiset contains W instances of the index, where W is the number of ways in the cache set. From now on, we will use the symbol $\mathcal{E}_i$ for describing the multiset representing the *ECB*s of a task τ_i.

For example, consider a task τ_i that accesses the cache blocks with index $3, 6, 7, 10$ in a cache set with 2 ways. Then $\mathcal{E}_i = \{3, 3, 6, 6, 7, 7, 10, 10\}$.

3.3.2 Cache analysis

The WCET analysis builds a *Control Flow Graph* from the binary code, where each node is a *basic block* of code, and edges are possible execution paths. A basic block is a segment of sequential instructions in the binary code of the task (no jumps). A basic block can span several cache blocks, and a cache block can contain instructions belonging to several basic blocks. Therefore, we introduce the notion of *line blocks*, which are (portions of) basic blocks that are contained in single cache blocks.

The analysis classifies the line blocks according to their behavior in the cache, as to compute their impact on the execution time. In the literature [2] the *must* cache analysis is used to classify the line blocks. In the *must* analysis, there are 4 categories for a line block: *always miss*, *first miss*, *always hit* and *first hit*.

3.3.3 An example

Figure 1 represents the relation between the cache blocks and the basic blocks of an example task. The cache memory is composed of two cache sets and it is depicted on the left. At its right, the binary code of the task is composed of 4 cache blocks, their arrows in direction of the cache memory point towards their respective cache set. Continuing to the right, we show the basic blocks and

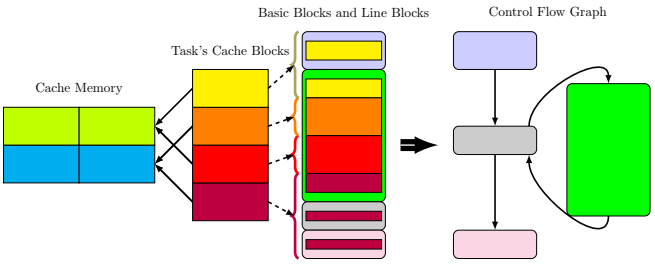


Figure 1: Relation between Cache Blocks and Basic Blocks

line blocks as computed by the WCET analysis: for example, the yellow cache block is composed of two line blocks, one in the purple basic block and the other in the green basic block.

On the right, we depict the CFG built from the binary during the WCET analysis. At the beginning of the execution of the task the yellow cache block will be stored in one of the green cache lines. As the yellow cache block is not present in the cache before the execution of the purple basic block, the yellow line block will be classified as *always miss*. During execution, the program will first move to the grey basic block, and the dark red cache block will be loaded into the blue cache set. However, if the task jumps into the green basic block and returns to the grey one later on, the dark red cache block will still be present in the cache. Therefore, it is classified as *first miss*.

3.3.4 UCBs

Given a preemption point p in the code of a task, the set of UCBs represents all cache blocks that are present in the cache *and* may be reused sometime later in the code. In fact, in case of preemption at point p, in the worst case all the evicted cache blocks that are UCBs will be reloaded later on, increasing the WCET of the task. Thus, only the cache blocks that can be classified as *first miss*, *always hit* and *first hit* will be considered as UCBs. In our model, we suppose that the line block that is preempted by another task is always evicted.

Therefore, each preemption point in the task is associated with a (possibly different) set of UCBs. To reduce the complexity of the analysis, recent approaches in the literature [3, 17, 1, 19, 18, 11, 12] use a single set of UCB cache set index for the entire task: the set is computed as the *fusion* of all UCBs for the different preemption points (see Section 3.4 for a definition of the fusion operation).

In this paper, we use the more complete model of one UCB set per preemption point. Therefore, in our model each task is characterized by a *set of multisets* (SOM) of UCBs. We mitigate the combinatorial explosion of the analysis by using an approximation function that trades off complexity against precision (described in Algorithm 1).

The multiset representing the *UCBs* of a task for a given preemption point p is denoted as $\mathcal{U}^p$ and the *SOM* representing the set of $\mathcal{U}$ for the task is denoted by $\overline{\mathcal{U}}$. For example, consider a task τ_i that has two preemption points, where for the first preemption points 4 UCBs are present in the cache with index $3, 6, 6, 7$ and 2 cache blocks with index $7, 7$ for the second preemption point. Then $\overline{\mathcal{U}}_i = \{\{3, 6, 6, 7\}, \{7, 7\}\}$. As several useful cache blocks may be stored at same time in the same cache set, a $\mathcal{U}^p$ may have multiple instance of the same index.

3.4 Operations on multisets

In this section we formally define multisets and *SOM*, and we define their operations.

Definition 1. *A* multiset *is a set that may contain more than one instance of the same element. Given an element $a \in \mathsf{A}$, we denote as $\mathsf{rep}_{\mathsf{A}}(a)$ the number of instances of a in A: for all $x \notin \mathsf{A}$, $\mathsf{rep}_{\mathsf{A}}(x) = 0$.*

We define the following operations on multisets:

- The size of a multiset, denoted by $|\mathsf{A}|$ is the number of elements in the multiset, including repetitions.

- The *multiset union* between multisets A and B is a multiset, it is denoted by $\mathsf{A} \uplus \mathsf{B}$, and it is defined as:
$$\forall x \quad \mathsf{rep}_{\mathsf{A} \uplus \mathsf{B}}(x) = \mathsf{rep}_{\mathsf{A}}(x) + \mathsf{rep}_{\mathsf{B}}(x)$$
- The *multiset fusion* between multisets A and B is a multiset, it is denoted by $\mathsf{A} \bigtriangledown \mathsf{B}$, and it is defined as:
$$\forall x \quad \mathsf{rep}_{\mathsf{A} \bigtriangledown \mathsf{B}}(x) = \max(\mathsf{rep}_{\mathsf{A}}(x), \mathsf{rep}_{\mathsf{B}}(x))$$
- The *multiset intersection* between multisets A and B is a multiset, it is denoted by $\mathsf{A} \sqcap \mathsf{B}$, and it is defined as:
$$\forall x \quad \mathsf{rep}_{A \sqcap B}(x) = \min(\mathsf{rep}_{\mathsf{A}}(x), \mathsf{rep}_{\mathsf{B}}(x))$$

Definition 2. *$\overline{\mathsf{A}}$ denotes a* SOM*, that is a set whose elements are multisets. The largest size of any multiset in a* SOM *$\overline{\mathsf{A}}$ is denoted as:*

$$\mathsf{MS}(\overline{\mathsf{A}}) = \max_{\mathsf{A} \in \overline{\mathsf{A}}}(|\mathsf{A}|)$$

Since $\overline{\mathsf{A}}$ is a set, the usual operations of set union, set intersection and set difference apply, respectively, with the usual symbols.

We extend the operations of *multiset union* and *multiset intersection* to *SOM* in the most natural way: the multiset union (resp. intersection) between $\overline{\mathsf{A}}$ and $\overline{\mathsf{B}}$ is the *SOM* obtained by applying the multiset union (resp. intersection) to every pair of elements of $\overline{\mathsf{A}}$ and $\overline{\mathsf{B}}$. We define the intersection between a multiset $\mathcal{E}$ and *SOM* $\overline{\mathsf{A}}$ as the *SOM* obtained by applying the multiset intersection of every element of $\overline{\mathsf{A}}$ and the multiset $\mathcal{E}$.

4 Reminder on CRPD Analysis

4.1 EDF analysis with CRPD

We consider systems scheduled with the Earliest Deadline First policy. In this work, we use the processor demand bound analysis, first proposed by Baruah et al. [8]. We compute the processor demand of the task set at each deadline in the interval $[0, L]$, where L is an estimated upper bound on the first idle time. The demand bound function [8] of a task τ_i in the interval $[0, t]$ can be computed as:

$$\mathsf{dbf}_i(t) = \eta(i, t) \cdot C_i \tag{1}$$

With $\eta(i, t)$ an upper bound to the number of instances of the sporadic task τ_i that have arrival and deadline in interval $[0, t]$:

$$\eta(i, t) = \max\left(0, \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) \tag{2}$$

The system is schedulable if and only if:

$$\forall t \leq L, \quad \mathsf{dbf}(t) = \sum_{i}^{N} \mathsf{dbf}_i(t) \leq t \tag{3}$$

In order to include the CRPD in the analysis, Lunniss et al. [17] redefined Condition 3 as:

$$\forall t \leq \mathrm{hp}(\tau), \quad \mathsf{dbf}(t) = \sum_{i}^{N} \mathsf{dbf}_i(t) + \gamma(t) \leq t \tag{4}$$

where $\mathrm{hp}(\tau)$ is the hyperperiod of task set τ and $\gamma(t)$ is an upper bound to the total CRPD in interval $[0, t]$.

To compute $\gamma(t)$, we use the *Combined* approach of Lunniss et al. [17]. This approach is the combination of two methods, the *UCB-union multiset* and the *ECB-union multiset*.

As our task model differs from the state of the art (we use a *SOM* to represent the UCBs of the task instead of a simple multiset), we denote by *ECB-union* SOM the modified version of

the second approach based on our task model, and *Combined* SOM will be the combination of *UCB-union multiset* and *ECB-union* SOM. In order to present the following equations in short form, we use $(\mathcal{U}_k)^x$ as a short form for the multiset union of $\mathcal{U}_\mathsf{k}$ with itself x times.

Given a *SOM* of UCBs $\overline{\mathcal{U}}_k$ for all the possible preemption points of task τ_k, a single multiset UCB can be computed as:

$$\mathcal{U}_k = \bigtriangledown_{\forall \mathcal{U}_k^P \in \overline{\mathcal{U}}_k} \mathcal{U}_k^P$$

4.2 UCB-union multiset

First, the maximum number of preemptions by task τ_j in interval $[0, D_i]$ is computed as:

$$\mathbf{Pr}_j(D_i) = max\left(0, \left\lceil \frac{D_i - D_j}{T_j} \right\rceil\right) \tag{5}$$

Then, the method constructs the multisets of UCBs and ECBs for the interval $[0, t]$:

$$M_{t,j}^{\mathbf{ucb}} = \biguplus_{\forall k, t \geq D_k > D_j} (\mathcal{U}_k)^{\mathbf{Pr}_j(D_k)\cdot\eta(k,t)} \tag{6}$$

$$M_{t,j}^{\mathbf{ecb}} = (\mathcal{E}_j)^{\eta(j,t)} \tag{7}$$

where $(\mathcal{E}_j)^x$ is a short form for the multiset union of $\mathcal{E}_\mathsf{j}$ with itself x times. Finally, the overall preemption cost is computed as the intersection of the ECB and UCB multisets obtained above:

$$\gamma_j^{\mathbf{ucbm}}(t) = \mathbf{BRT} \cdot (|M_{t,j}^{\mathbf{ucb}} \sqcap M_{t,j}^{\mathbf{ecb}}| + Y(j,t)) \tag{8}$$

$$Y(j,t) = min\left(\sum_{\forall k, t \geq D_k > D_j} \Big(\mathbf{Pr}_j(D_k) \cdot \eta(k,t)\Big), \eta(j,t)\right) \tag{9}$$

with **BRT** the block reload time of a cache block from the main memory. As we consider the preempted line block cache block evicted for each preemption, we add $Y(j,t)$ to the cost to consider this assumption.

4.3 ECB-union *SOM*

This approach is the version of ECB-union multiset modified by using a *SOM* to represent the UCBs of a task for the different preemption points. To obtain the same result as ECB-union multiset, $\overline{\mathcal{U}}'_k$ can be used instead of $\overline{\mathcal{U}}_k$ with $\overline{\mathcal{U}}'_k = \{\mathcal{U}_k\}$.

The first step consists in taking the *SOM* of UCBs for the preempted task, and compute the size of the worst-case intersection with higher preemption-level tasks' ECBs:

$$Q_j^{\mathbf{max}}(\overline{\mathcal{U}}_k) = \max_{\forall \mathcal{U} \in \overline{\mathcal{U}}_k} \left(\left|\mathcal{U} \sqcap \mathcal{E}'_j\right|\right) + 1 \tag{10}$$

With $\mathcal{E}'_j$ defined as:

$$\mathcal{E}'_j = \Big(\biguplus_{\forall h, D_h < D_j} \mathcal{E}_h\Big) \uplus \mathcal{E}_j$$

Notice that we add 1 to the cost to consider the preempted line block cache block as evicted. Then, the algorithm computes a multiset as the union of all costs of the multisets computed for the preemptions that occur in interval $[0, t]$:

$$Q_{t,j} = \biguplus_{\forall k, t \geq D_k > D_j} \Big((\{Q_j^{\mathbf{max}}(\overline{\mathcal{U}}_k)\})^{\mathbf{Pr}_j(D_k)\cdot\eta(k,t)}\Big) \tag{11}$$

Finally:

$$\gamma_j^{\mathbf{ecb}\boldsymbol{SOM}}(t) = \mathbf{BRT} \cdot \mathsf{sum\_max\_elems}(Q_{t,j}, \eta(j,t)) \tag{12}$$

where function $\mathsf{sum\_max\_elems}(\mathsf{A}, n)$ computes the sum of the n greatest elements in multiset A.

Notice that we slightly changed the algorithm of [17], in particular we modified Equation (10): instead of using a multiset which represents the UCBs of all the tasks, we changed the Equation to use only the multiset of UCBs at the point p of the task which is involved in the worst CRPD cost. This modification actually improves the performance of *ECB-union multiset* as it reduces the pessimism of considering additional cache blocks.

The combined *SOM* approach is the minimum between the demand bound of the task set computed with the previous approaches.

$$dbf^{combinedSOM}(t) = min(dbf^{\mathbf{ucbm}}(t), dbf^{\mathbf{ecb}\boldsymbol{SOM}}(t)). \tag{13}$$

5 Preemption Interval

In [17], in the case of EDF scheduling the number of preemptions on a job of task τ_i is computed using the relative deadline D_i. In particular, all instances of jobs with arrival and deadline in $[0, D_i]$ may preempt τ_i. This is a pessimistic assumption because jobs that arrive after the completion of τ_i are counted as preempting jobs. A less pessimistic assumption would be to consider only jobs that arrive before the worst-case response time of τ_i. However, computing an upper bound to the response time of a task in EDF is complex.

In this paper, we propose to use the concept of *preemption interval*: an upper bound I_i to the length of interval of time where a job of task τ_i may be preempted. Therefore I_i is a bound on the length of the interval between the start of the execution of any job of τ_i and its completion. Indeed, a task can be preempted only after it has started its execution, and before it completes.

Please notice that I_i is different from the worst-case response time R_i: the latter corresponds to the size of an interval starting from the job's arrival until its completion. As the start time of a job is always greater than or equal to its arrival time, it follows that $R_i \geq I_i$ by definition.

Let $\eta'_i(j,t)$ be the number of jobs of task τ_j that can preempt τ_i on an interval of length t, with $t \leq D_i$.

Lemma 1. *The number of preemptions by higher preemption-level task τ_j on task τ_i on any contiguous interval of size t inside $[0, D_i]$ (with $t \leq D_i$) is:*

$$\eta'_i(j,t) = min\left(\left\lceil \frac{t}{T_j} \right\rceil, \boldsymbol{Pr}_j(D_i)\right).$$

Proof. $\eta'_i(j,t)$ cannot be larger than $\mathbf{Pr}_j(D_i)$, that is the maximum number of preemptions of τ_j on τ_i. Also, since any job of τ_j must arrive inside the interval, $\eta'_i(j,t)$ cannot be larger than the number of jobs of τ_j arriving in any contiguous interval of size t. The latter can be computed as $\left\lceil \frac{t}{T_j} \right\rceil$. Hence the lemma is proved. □

Preempting jobs, that arrive during the preemption interval I_i of a task τ_i, contribute to increasing the length of the interval: in fact, we must account for their execution and for their cache impact.

Thus, to compute I_i we use an iterative formula starting with the WCET of the task. Since the preemption interval can only increase with the preempting jobs and cannot be greater than the relative deadline, we can use $\eta'_i(j,t)$ to count the number of instances of τ_j that can increase the preemption interval. The WCET of the preempting jobs and the CRPD provoked by them must be added to the WCET of the task to obtain the new preemption interval. The iterative procedure is guaranteed to stop, since $\eta'_i(j,t)$ is bounded by a constant $\mathbf{Pr}_j(D_i)$.

Theorem 1. *The following iterative equation gives an upper bound to the preemption interval of task τ_i:*

$$\begin{cases} I_i^{(0)} = C_i \\ I_i^{(k)} = C_i + \gamma'(I_i^{(k-1)}) + \displaystyle\sum_{j|D_j<D_i} C_j \cdot \eta'_i(j, I_i^{(k-1)}) \end{cases} \tag{14}$$

where $\gamma'(I_i)$ is the CRPD provoked by the preempting jobs during interval I_i. The iteration stops when $I_i^{(k)} > D_i$ (unschedulable) or $I_i^{(k-1)} = I_i^{(k)}$.

Proof. Only higher preemption-level tasks can interfere with the execution of a given task. In our model, we only consider two resources, the processor and the cache memory, thus the possible sources of interference are the suspension of the execution by a higher preemption-level task, and its CRPD. Lemma 1 provides an upper bound on the number of preemptions in interval I_i. Therefore, the value obtained in the last iteration represents an upper bound to the length of the preemption interval, or the fact that the task is not schedulable. □

Computing the CRPD during the Preemption Interval. Equations 6, 7, 9, 11, 12 use $\mathbf{Pr}_j(D_i)$ to count the number of preemptions by task τ_j on a job of τ_i.

We assume to compute the preemption interval in the task order from τ_1 (the task with the highest preemption level) to τ_n (the task with the lowest preemption-level). Of course, $I_1 = C_1$. We define $\mathbf{Pr}'_j(D_i)$ as:

$$\mathbf{Pr}'_j(D_i) = min\left(\left\lceil \frac{I_i}{T_j} \right\rceil, \left\lceil \frac{D_i - D_j}{T_j} \right\rceil\right) \tag{15}$$

and we use $\mathbf{Pr}'_j(D_i)$ instead of $\mathbf{Pr}_j(D_i)$ in the above equations. Also, to properly bound the number of preempting jobs from higher preemption-level tasks during the computation of a preemption interval I_i, we use $\eta'_i(j, I_i)$, which represents the number of jobs arriving in the preemption interval I_i and with relative deadline less than D_i, instead of $\eta(j, I_i)$.

6 Reduce the number of UCBs

As we use a multiset to represent the UCBs for each preemption point in the task, the size of the obtained *SOM* may be too large. Indeed the complexity of the *Combined* SOM approach depends on the size of the *SOM* of each preempted task.

Algorithm 1 describes the *reduce* operation on a *SOM* $\overline{\mathsf{A}}$. The algorithm removes the multiset X with the minimum size from the input vector (lines 3 and 4); then it performs a *multiset fusion* of $\overline{\mathsf{X}}$ with every other multiset of $\overline{\mathsf{R}}$, and selects the one that gives the smallest result (lines 6-11); then, it removes this multiset and adds the result of the fusion (line 12). It repeats the operation as long as the size of $\overline{\mathsf{R}}$ is greater than M (while loop at line 2).

The resulting *SOM* contains at most M multisets. We highlight that the *reduce* operation may increase the size of the multisets, thus introducing some pessimism; however, by selecting the merged multiset with smaller sizes, the pessimism is kept in check.

Algorithm 1 reduce()

Require: A vector of multisets $\overline{\mathsf{A}}$
Require: Max size M
Ensure: A vector of multisets $\overline{\mathsf{R}}$
1: $\overline{\mathsf{R}} \leftarrow \overline{\mathsf{A}}$
2: **while** $|\overline{\mathsf{R}}| > M$ **do**
3: $\mathsf{X} \leftarrow$ multiset with minimum size in $\overline{\mathsf{R}}$
4: $\overline{\mathsf{R}} \leftarrow \overline{\mathsf{R}} \setminus \{\mathsf{X}\}$
5: $\mathsf{L} \leftarrow \mathsf{X} \bigtriangledown \mathsf{Y}$, with any $\mathsf{Y} \in \overline{\mathsf{R}}$
6: **for** $\mathsf{Z} \in \overline{\mathsf{R}}$ **do**
7: **if** $|\mathsf{X} \bigtriangledown \mathsf{Z}| < |\mathsf{L}|$ **then**
8: $\mathsf{L} \leftarrow \mathsf{X} \bigtriangledown \mathsf{Z}$
9: $\mathsf{Y} \leftarrow \mathsf{Z}$
10: **end if**
11: **end for**
12: $\overline{\mathsf{R}} \leftarrow \overline{\mathsf{R}} \cup \{\mathsf{L}\} \setminus \{\mathsf{Y}\}$
13: **end while**
14: **return** $\overline{\mathsf{R}}$

7 Complexity

We denote by ω an upper bound to the maximum number of instances of any task contained in the hyperperiod H:

$$\omega = \max_{\forall \tau_i \in \tau} (\lfloor \frac{H - D_i}{T_i} \rfloor + 1)$$

We denote by ψ an upper bound of the maximum number of preemptions by any task on any job from another task as follows:

$$\psi = \max_{\forall \tau_j, \tau_i \in \tau; j<i} (\mathbf{Pr}_j(D_i))$$

The complexity of any multiset operation (union, intersection, etc.) is bounded by $\mathcal{O}(Z^2)$ with Z the number of cache sets (we represent a multiset as a list of pair (value, repetition factor)). The complexity of any operation between a multiset and a *SOM* is $\mathcal{O}(MZ^2)$, with M the number of element in the *SOM*.

7.1 Complexity of UCB multiset

In the UCB multiset union approach, we need to perform the union of the UCBs of lower preemption level tasks. In the worst-case, the complexity is $\mathcal{O}(NZ^2)$. As a task τ_i can be preempted at most $\omega\psi$ times by another task τ_j, the complexity of the union between all the lower preemption-level tasks is $\mathcal{O}(\omega\psi NZ^2)$. The intersection with the ECB has complexity $\mathcal{O}(Z^2)$, thus the complexity for a preempting task is again $\mathcal{O}(\omega\psi NZ^2)$, and the total complexity is $\mathcal{O}(\omega\psi N^2Z^2)$.

7.2 Complexity of ECB-union *SOM*

The ECB-union *SOM* approach can be split in three steps: the computation for the preempting task, building the list of costs, and summing the largest costs. The first step is a union between all the preempting tasks ECBs, thus its complexity is $\mathcal{O}(NZ^2)$. The second step consists of listing all the preemption costs, its complexity is $\mathcal{O}(N\omega\psi MZ^2)$, with M the maximum number of multisets that a *SOM* can contain.

The last step consists sorting and summing the ω greatest values. The number of element in the list is bounded by $N\omega\psi$, thus the complexity is $\mathcal{O}(\omega^2 N\psi \log(N\omega\psi))$. The complexity for the three steps is $\mathcal{O}(N\omega\psi(\omega \log(N\omega\psi) + MZ^2))$. Since we do this for N tasks, the final complexity can be bound by $\mathcal{O}(N^2\omega\psi(\omega \log(N\omega\psi) + MZ^2))$.

Since ω and ψ are pseudo-polynomial in the size of the input (they depend on periods and relative deadlines), we can state that the overall complexity is pseudo-polynomial, hence in the same complexity class as the original demand bound function.

The iterative formula for computing the preemption interval has the same structure of the response time analysis in Fixed Priority analysis, hence it is also pseudo-polynomial in the input.

8 Evaluation

8.1 Experiments Setup

To evaluate the impact of the preemption interval on the schedulability. We consider a single core ARM7 processor with one level of set-associative cache memory. We use OTAWA [6] to compute the WCET of the tasks and the list of UCBs and ECBs. Blocks are collected based on a *must* analysis of cache memory [1]. Tasks are chosen from the Malärdalen benchmark suite [14] and from TACLeBench [13].

The analysis has been repeated multiple times with different cache configurations: 2KB, 4KB, with 2 and 4 ways. The size of a cache line has been fixed to 32 bytes, as it is a standard in ARM processors, like the MCore A7.

Following of [10], we set the Block Reload Time $\mathbf{BRT} = 50, 100, 200$. In all configurations we considered the LRU replacement policy. In Table 1 we report the value of the WCET (in processor cycles) and the size of the ECB multiset for the corresponding benchmarks. The number reported for the UCB is the size of the multiset obtained by merging all the UCB multisets for

Tasks	Bench	WCET	Nb ECB	Nb UCB
fibcall	Malärdalen	5235	14	7
lcdnum	Malärdalen	9132	16	8
duff	Malärdalen	12034	20	10
binarysearch	TACLeBench	21511	42	21
insertsort	TACLeBench	26086	58	29
iir	TACLeBench	31439	62	31
complex_updates	TACLeBench	62040	64	34
ns	Malärdalen	88058	32	16
cnt	Malärdalen	112574	54	27
ud	Malärdalen	132411	64	36
fir2dim	TACLeBench	228105	64	36
ludcmp	TACLeBench	340181	64	37
crc	Malärdalen	442317	64	32
expint	Malärdalen	554166	46	25
nsichneu	Malärdalen	569174	64	32

Table 1: List of tasks used in the experiments: Cache size of 2KB, 2 ways, **BRT**= 200 cycles.

each preemption point using the *fusion* operation. Please notice that the WCET and the number of UCBs and ECBs depends on the cache configuration.

To adhere to a more realistic setting in a real compiler and linker, once the tasks have been selected to build a task set of 6,8,10 or 12 tasks, their memory locations are randomly assigned by adding a random offset to the cache sets index of their cache block addresses such that: $\mathcal{U}_i'^k = (\mathcal{U}_i'^K + \phi) mod \Lambda$ with Λ the number of cache sets in the cache, and ϕ a random number in $[0, \Lambda - 1]$.

We generate a set of utilizations using the UUnifast algorithm [9], and we compute a tentative period $T_i^{prime} = \frac{C_i}{U_i}$. In order to generate realistic workloads, and to avoid an excessive length of the hyperperiod of the task sets, we approximate the value of T_i' to a value T_i taken from a list of periods (expressed in thousands of "ticks"): $\{10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000\}$, taken from [15]. Then, we assign each task a relative deadline uniformly chosen in interval $[\max(C_i, 0.9T_i), T_i]$. Finally, all schedulability analysis were performed with the data computed in the previous step.

All the algorithms have been implemented in an open-source C++ program that will be made available to reviewers upon request (due to double blind review rules).

8.2 Results

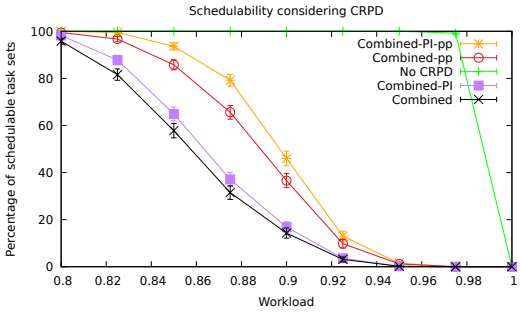
All experiments have been conducted using the tasks of Table 1, taken from [14] and [13], and applying all algorithms on 1000 randomly generated task sets per utilization point.

In all the figures the *No-CRPD* curves represent the results of the dbf analysis *without* considering the CRPD cost, and it is reported as a reference for the other algorithms. We decide to use the combined approach from Lunniss et al. [17] to see the impact of the preemption interval. Our approach is labeled as *Combined-PI* and *Combined-pp-PI* for *Combined preemption interval* and *Combined preemption points and preemption interval*. The original version from the literature are denoted as *Combined* and *Combined-pp*. To the best of our knowledge, the combined approach is the best approach so far for EDF scheduling. For both *Combined-PI* and *Combined* we use the reduce operation 1 to consider only one multiset of UCBs per task. For each point, we also report the 95% confidence interval, computed as:

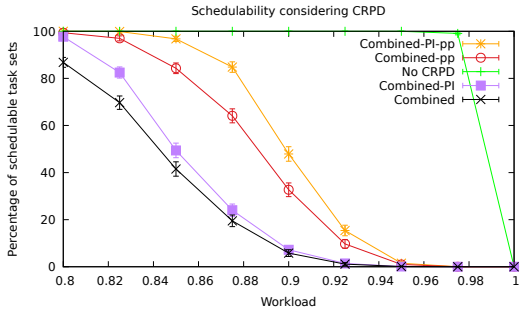
$$CI = \hat{p} \pm Z^* \sqrt{\frac{\hat{p} \cdot (1 - \hat{p})}{n}}$$

where $\hat{p}$ is the ratio of schedulable task sets for a given workload, n is the sample size (here 1000), and 1.96 is the value for Z^*.

Impact of the cache size The variation of the cache size has a strong impact on the Combined and Combined-PI-pp methods. Consider the experiments in Figure 2a and Figure 2b, whose only

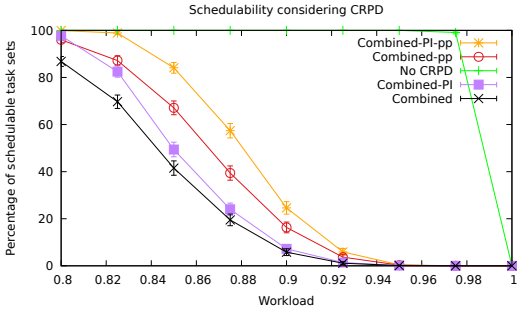


(a) Schedulability of $n_{tasks} = 12$ with cache of 2KB, 2 ways, **BRT** $= 200$ and $M_{reduce} = 4$

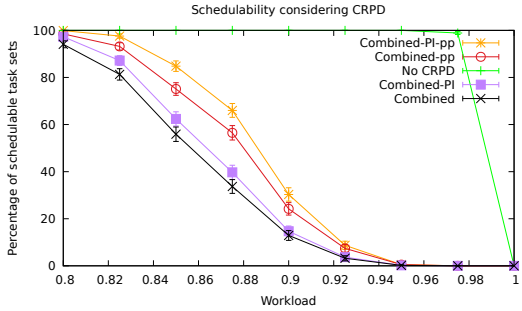


(b) Schedulability of $n_{tasks} = 12$ with cache of 4KB, 2 ways, **BRT** $= 200$ and $M_{reduce} = 4$

Figure 2: Impact of cache size.

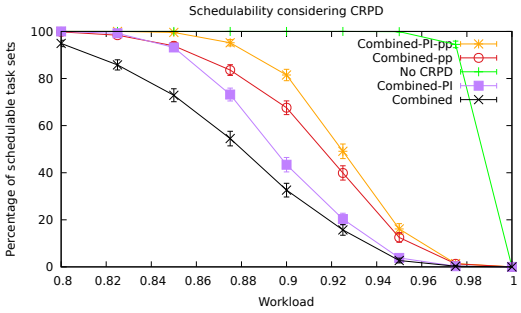


(a) Schedulability of $n_{tasks} = 12$ with cache memory of 4KB, 2 ways, **BRT** $= 200$ and $M_{reduce} = 2$

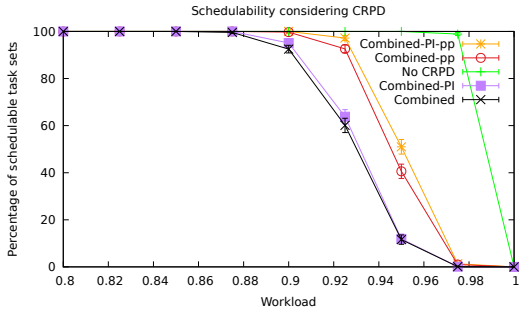


(b) Schedulability of $n_{tasks} = 12$ with cache memory of 4KB, 4 ways, **BRT** $= 200$ and $M_{reduce} = 2$

Figure 3: Impact of number of ways.



(a) Schedulability of $n_{tasks} = 6$ with cache memory of 4KB, 2 ways, **BRT** $= 200$ and $M_{reduce} = 4$



(b) Schedulability of $n_{tasks} = 12$ with cache memory of 4KB, 2 ways, **BRT** $= 50$ and $M_{reduce} = 4$

Figure 4: Impact of number of tasks and **BRT**.

difference is the size of the cache (2KB and 4KB, respectively). The Combined analysis loses 10% of schedulable task sets already at $U = 0.8$ as we increase the size of the cache, whereas Combined-PI-pp increases its performances for all values of the workload. In particular, at $U = 0.875$ Combined-PI-pp increases its performance by more than 10%. We explain this difference with the fact that using only one UCB set per task produces larger UCB set as the cache size increases, thus increasing pessimism.

Impact of the number of cache ways and cache sets. Figures 3a and 3b show two different scenarios that use a 4KB cache, the first with 2 ways and the second with 4 ways. As you can see, the second scenario provides better performance than the first scenario while the performance of different versions of the combined approach are closer. We explain this by observing that our algorithms is more effective in reducing the pessimism as the scenario becomes more competitive (less number of ways).

Impact of the number of tasks. The number of tasks has a strong impact on the schedulability ratio. Comparing Figure 2b (12 tasks) with Figure 4a (6 tasks), we notice two things: first, by increasing the number of tasks, the performance of Combined-PI decreases to become closer to combined; Second, by increasing the number of tasks, the performance of Combined-PI-pp decreases slower than the performance of Combined-pp, increasing the gap between the two approaches. This means that the combination of the Preemption Interval and the use of several UCB multisets to describe preemption points is more useful with a high number of tasks.

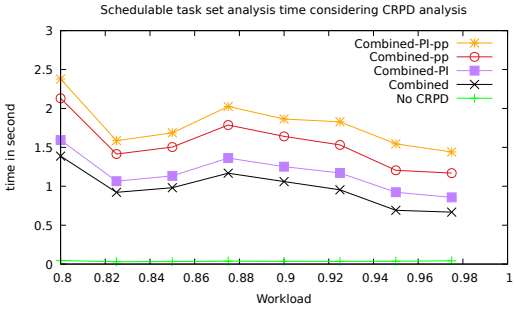
You may notice that overall the schedulability ratio is lower in the case of a task set with 6 tasks compared to a task set of 12 tasks at $U = 0.975$. This is due to the algorithm we use for generating the task set: in fact, in order to obtain the same workload, a set of 6 tasks must have a higher average workload per task $\frac{C_i}{D_i}$ than a set of 12 tasks. Therefore, a set of 6 tasks is marginally less schedulable than a set of 12 tasks.

Impact of the block reload time. By comparing Figure 2b and Figure 4b, we conclude that the block reload time has a strong impact on schedulability, since 90% of task sets are schedulable with combined-PI-pp and a $\mathbf{BRT} = 50$ compared to 50% in the case of $\mathbf{BRT} = 200$ at $U = 0.9$. With a $\mathbf{BRT} = 50$ the gap between Combined-PI-pp and Combined-pp is smaller and we observe the same between Combined-PI and Combined. However, there is an augmentation of 40% in the schedulable task sets between Combined and Combined-PI. This gap shows the gain in precision when using the reduce operation. As you will see later in the section, using the *reduce* heuristic allow us to find a compromise between the accuracy of the analysis and its complexity.

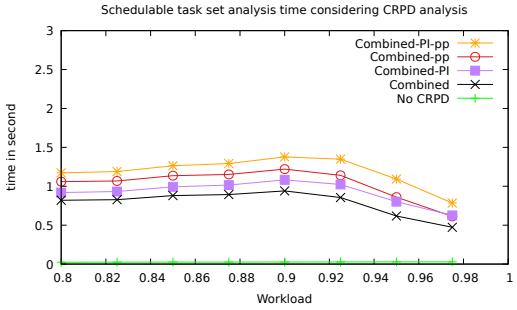
The impact of the **BRT** on the schedulability contradicts the conclusions of Shah et al. [21]. However, we observe that in the experiments of [21] the task sets are generated differently: the WCET of all tasks is almost proportional to the **BRT** (probably due to the different WCET analysis tool), and the periods are computed as $T_i = \frac{C_i}{U_i} \approx \frac{k_i \mathbf{BRT}}{U_i}$, with U_i randomly generated by UUnifast. Therefore, the dominant factor in their experiments is the number of preemptions, and not the value of **BRT**. In our case, 1) the WCETs and the UCBs and ECBs have been computed by OTAWA, and the WCETs are not proportional to the **BRT**; 2) periods are selected from a list. We believe that our experimental setting presents a better approximation of a real application.

Analysis time. We can see in Figure 6 the analysis time of schedulable task sets for the different approaches with 4 configurations of task set sizes: 6, 8, 10 and 12 tasks with a processor load of 0.85. The other parameters are a reduce operation parameters M_{reduce} equal to 4 with $\mathbf{BRT} = 200$ and a cache memory of 4KB and 2 ways. Between the first and the last case (6 and 12 tasks, respectively), we observe a multiplying factor of 10 for the analysis time of the combined approach with *SOM* as model for the UCBs.

The reduce operation has a strong impact on the analysis time of schedulable task sets as you can see with Figure 5a and Figure 5b. Both experiments use the same cache configuration, a 2KB with 4w cache memory with a $\mathbf{BRT} = 50$. The task set is of size 12 for both experiments and the reduce operation parameter M_{reduce} is set to 4 in the first configuration and 2 in the second. As you can see, multiplying by two M_{reduce} doubles the gap between combined-PI and combined-PI-pp time analysis. The same observation is also true between Combined and Combined-pp. We think



(a) Analysis time for $n_{tasks} = 12$ with cache memory of 2KB, 4 ways, **BRT** $= 50$ and $M_{reduce} = 4$



(b) Analysis time for $n_{tasks} = 12$ with cache memory of 2KB, 4 ways, **BRT** $= 50$ and $M_{reduce} = 2$

Figure 5: Impact of their reduce operation on analysis time.

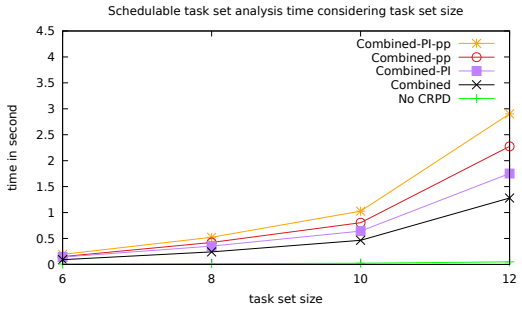


Figure 6: Analysis time for a cache memory of 4KB, 2 ways, $U = 0.85$, **BRT** $= 200$ and $M_{reduce} = 4$

that using all the UCB multiset obtained by the WCET analysis for each task is not scalable. It also shows the importance of heuristics such as the reduce operation to keep the complexity in check, to increase the number of schedulable task sets that the analysis can verify.

9 Conclusions and future works

In this paper we improve the precision of CRPD analysis while proposing a compromise with the complexity of the analysis. We target fully preemptive real-time tasks scheduled by EDF on single processor systems with set-associative caches. In particular, we propose two original contributions:

- A method to precisely compute the Preemption Interval length of each task (Section 5), thus the number of preemptions that a task can undergo;
- A heuristic to keep the complexity of analysis in check (Algorithm 1).

In this paper we worked on the basic techniques behind the CRPD analysis. For simplicity, we decided to focus on single-processor architectures with one single level of cache and we only consider instruction caches. While this is a very limited setting compared to modern architectures, we believe that our propositions represent a building brick that can be reused to improve CRPD analysis in more complex settings like multilevel caches and multicore systems with private and shared caches.

Regarding possible extensions for data caches, we believe that the problem lies mostly in the correct computation of the UCBs and ECBs for data, especially in the presence of pointers in the code. We are currently investigating the application of the static analysis techniques described in [7], based on abstract interpretation using polyhedra, to detect UCB and ECB for data caches.

10 Acknowledgments

This work is partially funded by the French National Research Agency, Corteva project (ANR-17-CE25-0003).

References

[1] S. Altmeyer and C. Burguière. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *21st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 109–118, Los Alamitos, CA, USA, jul 2009. IEEE Computer Society.

[2] Sebastian Altmeyer and Claire Maiza Burguière. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture*, 57(7):707–719, 2011.

[3] 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.

[4] 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.

[5] T. Baker. A stack-based resource allocation policy for realtime processes. In *Proceedings 11th Real-Time Systems Symposium*, pages 191,192,193,194,195,196,197,198,199,200, Los Alamitos, CA, USA, dec 1990. IEEE Computer Society.

[6] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Otawa: An open toolbox for adaptive wcet analysis. In Sang Lyul Min, Robert Pettit, Peter Puschner, and Theo Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems*, pages 35–46, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[7] Clément Ballabriga, Julien Forget, Laure Gonnord, Giuseppe Lipari, and Jordy Ruiz. Static analysis of binary code with memory indirections using polyhedra. In Constantin Enea and Ruzica Piskac, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 114–135, Cham, 2019. Springer International Publishing.

[8] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings 11th Real-Time Systems Symposium*, pages 182,183,184,185,186,187,188,189,190, Los Alamitos, CA, USA, dec 1990. IEEE Computer Society.

[9] E. Bini and G. C. Buttazzo. Biasing effects in schedulability measures. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 196–203, Los Alamitos, CA, USA, jul 2004. IEEE Computer Society.

[10] Roman Bourgade, Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Accurate analysis of memory latencies for WCET estimation. In Giorgio Buttazzo and Pascale Minet, editors, *16th International Conference on Real-Time and Network Systems (RTNS 2008)*, Rennes, France, October 2008. Isabelle Puaut.

[11] R. J. Bril, S. Altmeyer, M. Heuvel, R. I. Davis, and M. Behnam. Integrating cache-related pre-emption delays into analysis of fixed priority scheduling with pre-emption thresholds. In *2014 IEEE Real-Time Systems Symposium (RTSS)*, pages 161–172, Los Alamitos, CA, USA, dec 2014. IEEE Computer Society.

[12] Claire Burguière, Jan Reineke, and Sebastian Altmeyer. Cache-Related Preemption Delay Computation for Set-Associative Caches - Pitfalls and Solutions. In Niklas Holsti, editor, *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*, volume 10 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–11, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-252-6.

[13] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sorensen, Peter Wägemann, and Simon Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:10, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[14] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 136–146, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7.

[15] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

[16] Chang-Gun Lee, Hoosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE transactions on computers*, 47(6):700–713, 1998.

[17] W. Lunniss, S. Altmeyer, C. Maiza, and R. I. Davis. Integrating cache related pre-emption delay analysis into edf scheduling. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 75–84, Los Alamitos, CA, USA, apr 2013. IEEE Computer Society.

[18] Will Lunniss, Sebastian Altmeyer, and Robert Davis. A comparison between fixed priority and edf scheduling accounting for cache related pre-emption delays. *Leibniz Transactions on Embedded Systems*, 1(1):01–1–01:24, 2014.

[19] Filip Marković, Jan Carlson, Sebastian Altmeyer, and Radu Dobrin. Improving the Accuracy of Cache-Aware Response Time Analysis Using Preemption Partitioning. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:23, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[20] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel. PapaBench: a Free Real-Time Benchmark. In Frank Mueller, editor, *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2006. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[21] Darshit Shah, Sebastian Hahn, and Jan Reineke. Experimental Evaluation of Cache-Related Preemption Delay Aware Timing Analysis. In Florian Brandner, editor, *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*, volume 63 of *OpenAccess Series in Informatics (OASIcs)*, pages 7:1–7:11, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[22] Yudong Tan and Vincent Mooney. Timing analysis for preemptive multitasking real-time systems with caches. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(1):7, 2007.