Resilience Analysis: Tightening the CRPD bound for set-associative caches *

Sebastian Altmeyer Claire Maiza (Burguière) Jan Reineke[1]
Saarland University, Saarbrücken
{altmeyer, maiza, reineke}@cs.uni-saarland.de

Abstract

In preemptive real-time systems, scheduling analyses need—in addition to the worst-case execution time—the context-switch cost. In case of preemption, the preempted and the preempting task may interfere on the cache memory. This interference leads to additional cache misses in the preempted task. The delay due to these cache misses is referred to as the cache-related preemption delay (CRPD), which constitutes the major part of the context-switch cost.

In this paper, we present a new approach to compute tight bounds on the CRPD for LRU set-associative caches, based on analyses of both the preempted and the preempting task. Previous approaches analyzing both the preempted and the preempting task were either imprecise or unsound. As the basis of our approach we introduce the notion of *resilience*: The resilience of a memory block of the preempted task is the maximal number of memory accesses a preempting task could perform without causing an additional miss to this block. By computing lower bounds on the resilience of blocks and an upper bound on the number of accesses by a preempting task, one can guarantee that some blocks may not contribute to the CRPD. The CRPD analysis based on resilience considerably outperforms previous approaches.

1. Introduction

In hard real-time systems, one needs to prove that all time-critical tasks meet their deadlines. Many task sets are only schedulable in a *preemptive* scheduling regime. For instance, high priority tasks with short deadlines are often unschedulable in non-preemptive regimes.

However, in modern hardware architectures, preemption does not come for free. The preempting task may "disturb" the state of performance-enhancing features like caches, pipelines, etc. This disturbance may significantly increase the execution time of the preempted task once it is resumed. The additional execution time compared with non-preempted execution, including the additional execution time of the preempted task and the execution time of the scheduler in the operating system, is referred to as the context-switch cost. Schedulability analyses for preemptive systems require bounds on the worst-case execution times (WCET)[2] of tasks as well as on these context-switch costs.

In *cached* systems, the major part of the context-switch cost is due to additional cache misses in the preempted task: Memory accesses of the preempting task change the cache contents. As a consequence, accesses in the preempted task that would have been cache hits without preemption turn out to be misses. This part of the context-switch cost is commonly referred to as the *cache-related preemption delay* (CRPD). There are two main approaches to statically bound the CRPD:

- By analyzing the *preempted* task [8, 10, 17, 18]:
  Additional misses can only occur for memory blocks that are *useful* without preemption. A *useful cache block* (UCB) is a block that may be cached and that may be reused later, resulting in a cache hit. The number of such useful cache blocks is a bound on the number of additional cache misses due to preemption and can therefore be used to bound the CRPD independently of the preempting task.
- By analyzing the *preempting* task [10, 17–19]:
  The preempting task may only cause additional misses in the cache sets modified during its execution. The number of cache sets that memory blocks of the preempting task map to can therefore be used to bound the CRPD independently of the preempted task. However, for set-associative caches, the latter approaches have either been imprecise [6] or unsound [18] as shown in [6].

In this paper, we present a new approach to *precisely* and *soundly* bound the CRPD for set-associative caches, taking into account both the preempted and the preempting task. To this end, we introduce the notion of *resilience*, and a corresponding *resilience analysis* that determines how much "disturbance" by the preempting task a useful cache block may endure before becoming unuseful for the preempted task. The results of this analysis can then be combined with those of a simple analysis of the preempting task to determine a set of useful cache blocks that are guaranteed to *remain useful* after the preemption. Only cache blocks that are useful before preemption but that are not guaranteed to remain useful may contribute to the CRPD.

In our evaluation, resilience analysis improves on previous approaches by at least 28% and by 64% on average. Our new analysis is particularly useful in case of frequent preemptions by small tasks. Interrupt routines are typical representatives of this class.

* This work was supported by ICT project PREDATOR in the European Community's Seventh Framework Programme under grant agreement no. 216008, by Transregional Collaborative Research Center AVACS of the German Research Council (DFG) and by ARTIST DESIGN NoE.

[1] Current address: Department of EECS, University of California, Berkeley, CA 94720, eMail: reineke@eecs.berkeley.edu

LCTES'10, April 13–15, 2010, Stockholm, Sweden.

[2] Considering non-preempted execution.

2. Background & Related Work

In this section we introduce caches and the notions of useful cache blocks and evicting cache blocks. Then, we show that previous work on the computation of a bound on the CRPD by combining UCB and ECB for set-associative caches is either unsound or imprecise.

2.1 Caches

We will investigate CRPD computation for set-associative caches in the context of the Least-Recently-Used (LRU) policy, which is used, for instance, in the INTEL PENTIUM I and the MIPS 24K/34K. In the following description of LRU and later, we will use k for the associativity of the cache and c for the number of cache sets. The LRU policy replaces the least-recently-used element on a cache miss. It conceptually maintains a queue of length k for each cache set. Elements of the set are ordered from the most-recently-used to the least-recently used. In LRU each memory block can be associated with an age: the age of a memory block m is the number of other memory blocks that have been accessed since the last access to block m. In the case of LRU and associativity 4, $[b, c, e, d]$ denotes a cache set, where elements are ordered from most- to least-recently-used: the most-recently used element has age 0 (here: b) and the least-recently-used has age $k-1$ (here: d). If an element is accessed that is not yet in the cache (a miss), it is placed at the front of the queue. The last element of the queue, i.e., the least-recently-used, is then removed if the set is full. In our example, an access to f would thus result in $[f, b, c, e]$. The least-recently-used element d is replaced. Each element is aged by one and the age of element f is 0. Upon a cache hit, the accessed element is moved from its position in the queue to the front, in this respect treating hits and misses equally. Accessing c in $[f, b, c, e]$ results in $[c, f, b, e]$: the age of the elements that were younger than the accessed element (f and b) is incremented by one, the age of the elements that were older (element e) is not changed and the age of the accessed element c becomes 0.

2.2 Bounding the CRPD

The cache-related preemption delay denotes the additional execution time due to cache misses caused by preemption. Such cache misses occur, if the cache accesses of the preempting task cause eviction[3] of cache blocks of the preempted task that otherwise would be reused later. Therefore upper bounds on the CRPD can be derived from two directions: bounding the effect on the preempted task or bounding the impact of the preempting task.

Effect on the preempted task: For the analysis of the effect on the preempted task, Lee et al. introduced the notion of a *useful cache block* [8]:

Definition 1 (Useful Cache Block (UCB))**.** *A memory block m is called a* useful cache block *at program point P, if*

a) m may be cached at P and
b) m may be reused at program point Q that may be reached from P without eviction of m on this path.

In the case of preemption at program point P, only the memory blocks that a) are cached and b) will be reused, may cause additional reloads. Hence, the number of UCBs at program point P gives an upper bound on the number of additional reloads due to a preemption at P. A global bound on the CRPD of the whole task is determined by the program point with the highest number of UCBs.

[3] Such eviction does not necessarily happen during the execution of the preempting task. They can also occur during the execution of the preempted task but as a consequence of the preemption (see Figure 1(a)).

However, due to imprecision in the UCB analysis, the number of UCBs per set may exceed the associativity of the cache. Still, the number of additional misses per set is limited to k:

$$\text{CRPD}_{\text{UCB}} = \sum_{s=1}^{c} \text{CRPD}^{s}_{\text{UCB}} \tag{1}$$

where

$$\text{CRPD}^{s}_{\text{UCB}} = \text{CRT} \cdot min\{|\text{UCB}^{s}|, k\} \tag{2}$$

where UCB^{s} denotes the set of UCBs mapping to cache set s and c is the number of sets. For a preemption at a specific program point, the whole CRPD is bounded by the sum of the CRPDs of all cache sets. For each set, the CRPD is bounded by the cache reload time (CRT), i.e., the time needed to load a cache block, times the minimum of the number of UCBs and the associativity (see [6]).

Note that the CRPD bounds denote the additional delay for one preemption. In case of several preemptions, the CRPD bound must be accounted for as often as preemption might occur.

Recently, Altmeyer and Burguiere introduced a new analysis of the effect on the preempted task: As some cache accesses are taken into account as misses as part of the WCET analysis, these accesses do not have to be accounted for a second time as part of the CRPD [4]. At a program point P a UCB is a *definitely-cached UCB* if it must be cached at P and along the path to its reuse. Using the notion of definitely-cached UCBs (DC-UCB), one computes the number of additional cache misses due to preemption that are not already taken into account as a miss by the WCET analysis. This number does not bound the CRPD but the part of the CRPD that is not already accounted for by the WCET analysis in the WCET bound. Thereby, the global bound on the WCET+CRPD can be significantly refined. The analysis presented in this paper can be used in both contexts—CRPD computed separately (based on the UCB notion introduced by Lee et al. [8]) or in combination with the WCET (DC-UCB).

Effect of the preempting task: The worst-case impact of a preempting task is given by the number of cache blocks this task may evict. Such evictions may occur during and after the preemption: accessing a cache set may have a deferred impact in case of set-associative caches, as we will illustrate shortly.

To analyse the effect of the preempting task, Tomiyama and Dutt introduced the concept of an *evicting cache block* [19]:

Definition 2 (Evicting Cache Blocks (ECB))**.** *A memory block of the preempting task is called an* evicting cache block*, if it may be accessed during the execution of the preempting task.*

As part of their CRPD computation, Tan et al. [18] use the number of ECBs as an upper-bound on the number of reloads:

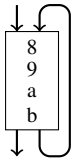
$$\text{CRPD}_{\text{MIN\_ECB}} = \sum_{s=1}^{c} \text{CRPD}^{s}_{\text{MIN\_ECB}} \tag{3}$$

where

$$\text{CRPD}^{s}_{\text{MIN\_ECB}} = \text{CRT} \cdot \min\{|\text{ECB}^{s}|, k\} \tag{4}$$

where ECB^{s} denotes the set of ECBs mapping to cache set s.

However, this function may underestimate the number of misses in several cases. Consider the CFG of Figure 1(a). Assume, all memory blocks map to the same cache set. Then, at the end of the execution of this basic block, the content of the 4-way set is given by $[b, a, 9, 8]$. Assume, furthermore, a preemption between two iterations of the loop and one block of the preempting task maps to this set: Using the formula presented above, only one additional miss is taken into account for this memory set ($min(1, 4) = 1$). However, the number of additional misses, four, is greater than the number of ECBs, one: All useful cache blocks are evicted and need



Without preemption:

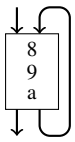
$[b, a, 9, 8] \xrightarrow{8} [8, b, a, 9] \xrightarrow{9} [9, 8, b, a] \xrightarrow{a} [a, 9, 8, b] \xrightarrow{b} [b, a, 9, 8]$ 0 misses

ECBs $= \{\mathbf{e}\}$

With preemption:

$[\mathbf{e}, b, a, 9] \xrightarrow{8^*} [8, \mathbf{e}, b, a] \xrightarrow{9^*} [9, 8, \mathbf{e}, b] \xrightarrow{a^*} [a, 9, 8, \mathbf{e}] \xrightarrow{b^*} [b, a, 9, 8]$ 4 misses

(a) Sequence of accesses where $\text{CRPD}_{\text{MIN\_ECB}}$ ($= 1 \cdot \text{CRT}$) underestimates the CRPD ($= 4 \cdot \text{CRT}$)



Without preemption:

$[a, 9, 8, 7] \xrightarrow{8} [8, a, 9, 7] \xrightarrow{9} [9, 8, a, 7] \xrightarrow{a} [a, 9, 8, 7]$ 0 misses

ECBs $= \{\mathbf{e}\}$

With preemption:

$[\mathbf{e}, a, 9, 8] \xrightarrow{8} [8, \mathbf{e}, a, 9] \xrightarrow{9} [9, 8, \mathbf{e}, a] \xrightarrow{a} [a, 9, 8, \mathbf{e}]$ 0 misses

(b) Sequence of accesses where $\text{CRPD}_{\text{UCB\&ECB}}$ ($= 3 \cdot \text{CRT}$) roughly overestimates the CRPD ($= 0$). Note that CRPD_{TAN} ($= 1 \cdot \text{CRT}$) overestimates, too.

Figure 1. Evolution of the cache contents for LRU replacement. The first row shows the evolution of the cache contents for one iteration of the loop without preemption. The second row shows the evolution of the cache contents on the same sequence with preemption. The preempting task accesses block e. A * as in a^* indicates a miss.

to be reloaded. The cache blocks are not evicted during the execution of the preempting task but after the preemption during the execution of the preempted task. In this example, a valid upper bound is given, for instance, by the associativity whereas, the minimum between the number of ECBs and the associativity gives an underestimation of the CRPD.

Hence, instead of using the formula by Tan et al., a sound upper bound on the CRPD using ECB is given by:

$$\text{CRPD}_{\text{ECB}} = \sum_{s=1}^{c} \text{CRPD}^s_{\text{ECB}} \tag{5}$$

where

$$\text{CRPD}^s_{\text{ECB}} = \begin{cases} 0 & \text{if } \text{ECB}^s = \emptyset \\ \text{CRT} \cdot k & \textit{otherwise} \end{cases} \tag{6}$$

where ECB^s denotes the set of ECBs mapping to cache set s. The CRPD is bounded by the cache reload time times the associativity (k) of the cache in case at least one ECB maps to set s [6]. Note that in case of nested preemption the set of ECBs in the formula is the union of all ECB sets of the preempting tasks [17, 19].

In this paper, we focus on the CRPD computation for set-associative caches with LRU replacement using UCBs and ECBs. As shown in [6], for FIFO and PLRU replacement strategies, the CRPD cannot be bounded directly using UCB and ECB analyses. In the rest of the paper, we will investigate the CRPD computation for LRU only. In [6], Burguiere et al. sketched how to use the UCB analysis for LRU to bound the number of misses in case of preemption for FIFO and PLRU, by relative competitiveness [14]. The next subsection presents related work focusing on set-associative caches and the computation of a bound on the CRPD by combining UCB and ECB analyses.

Effect of the preempting task on the preempted task The results from the CRPD computation via UCB and via ECB can be combined by taking into account the minimum between the effect on the preempted task and the effect of the preempting task [6, 18]:

$$\text{CRPD}_{\text{UCB\&ECB}} = \sum_{s=1}^{c} min(\text{CRPD}^s_{\text{UCB}}, \text{CRPD}^s_{\text{ECB}}) \tag{7}$$

The CRPD computed in [18] takes into account Equation 4:

$$\text{CRPD}_{\text{TAN}} = \sum_{s=1}^{c} min(\text{CRPD}^s_{\text{UCB}}, \text{CRPD}^s_{\text{MIN\_ECB}}) \tag{8}$$

Due to its use of $\text{CRPD}^s_{\text{MIN\_ECB}}$ this formula is unsound. However, we list it in order to later compare our results to the ones obtained with it.

Equation 7 gives a bound on the CRPD that is sound but imprecise. Consider the example of Figure 1(b). As there is one ECB, the number of additional misses is bounded by the number of UCBs (three), which is lower than the number of ways (four). However, there are no additional misses due to this preemption: using Equation 7, the CRPD is overestimated; the formula is imprecise. Not every UCB may be evicted by a single ECB. Some UCBs *remain useful* under preemption. In the case of Figure 1(b), blocks $8, 9$, and a remain useful under preemption. On the other hand, we strongly believe that Equation 7 is the best bound we can obtain by using *only* the numbers of UCBs and ECBs. A new analysis is necessary to combine the results of the UCB and ECB analyses considering the blocks that remain useful under preemption. For this purpose we introduce the notion of *resilience* in the following section.

Remark & notation Note that the computation of a bound on the CRPD for a whole set-associative cache is done by adding the bound on the CRPD for each set. For the sake of simplicity, in Section 3 and 4, we assume the cache to be fully-associative ($c = 1$). The extension to set-associative caches is then discussed in Section 5.

Table 2.2 presents the notation used in this paper. Note that, as we consider all types of caches (data and instruction), the set of UCBs can be different before and after a program point (see e.g. [4]).

$\mathbb{M}$	set of memory blocks
$\mathbb{P}$	set of all program points
k	associativity
c	number of sets
UCB^b_P	set of UCBs before program point P
UCB^a_P	set of UCBs after program point P
ECB	set of ECBs for a given preempting task

Table 1. Notation used within this paper.

3. The Notion of Resilience

The aim of our analysis is to derive a subset of the set of UCBs that cannot contribute to the cache-related preemption delay. To this end, we need to argue about the amount of "disturbance" caused by a preempting task and the "resilience" of the useful cache

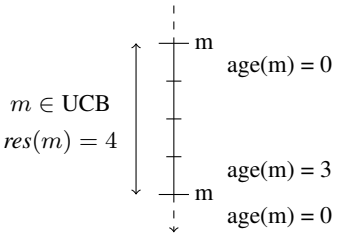


Figure 2. Illustration of a UCB m, with $res(m) = 4$. The cache associativity is 8. The dashes denote memory accesses.

blocks. The amount of "disturbance" is given by the set of evicting cache blocks as presented in Section 2. The resilience $res(m)$ of a useful cache block m is the maximal disturbance, i.e., the maximal number of additional cache accesses (to different memory blocks) by the preempting task, such that m is still cached after preemption and remains cached until its next access. Hence, if the disturbance of the preempting task is less than or equal to the resilience, UCB m will remain useful after preemption and will not lead to an additional cache miss. *Resilience* is formally defined as follows:

Definition 3 (Resilience)**.** *The resilience $res_P(m)$ of memory block m at program point P is the greatest l, such that all possible next accesses to m,*

a) that would be hits without preemption,
b) would still be hits in case of a preemption with l accesses at P.

The resilience of UCB m, $res_P(m)$ is l, if a preemption with up to l ECBs does not cause an additional miss to m: an access that would be a hit without preemption, would still be a hit under such a preemption. The resilience depends on the age of block m right before its next access. Consider the example in Figure 2 in which we assume the associativity to be 8. There are three memory accesses between the two accesses to m: before its reuse the age of m is 3. In an LRU-controlled cache with associativity 8, exactly those memory blocks with ages between 0 and 7 are cached. Thus, in the example, m could age by up to 4 ($3 + 4 = 7$) due to a preemption and still be cached at its reuse (see Figure 3). Between the two accesses to m, $res(m) = 4$.

So, at a program point P, the resilience of a block m is determined by the maximal age $max\text{-}age_P(m)$ of m at program points accessing m that can be reached from P without eviction of m:

$$res_P(m) = (k-1) - max\text{-}age_P(m). \tag{9}$$

Consider Figure 4(a). The associativity is 8. The control flow first joins and then splits. The maximal age of 7 is obtained along the longest path, i.e., from the upper-right access to m to the lower-left access to m. Along this path, there are 7 different accesses between the two accesses to m. Thus, at all points along this path, $res(m) = 0$. The longest path starting from the upper-left access to m only contains 6 different accesses. Therefore from the upper-left access to the control flow join, the maximal age is 6. Similarly, the longest path to the lower-right access contains 5 different accesses.

If a memory block is not accessed along a path, then this path does not influence its resilience (Figure 4(b)) because of condition (a) of Definition 3. Similarly, if a memory access definitely leads to a cache miss, then this access does not influence the memory block's resilience either (Figure 4(c)).

Using bounds on the resilience, we compute a set of useful cache blocks that must remain useful under preemption with a given number of ECBs. Those blocks will not lead to additional misses due to preemption. The upper-bound on the CRPD can be

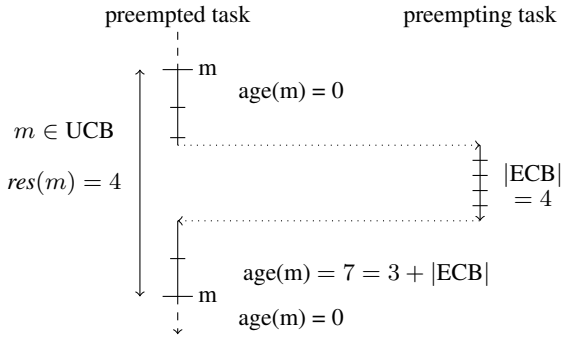


Figure 3. Illustration of a UCB m with $res(m) = 4$ remaining useful after a preemption with 4 ECBs (associativity $k = 8$).

refined by considering as additional misses under preemption only the UCBs that cannot be guaranteed to remain useful:

$$\text{CRPD-RES} = \text{CRT} \times |\underbrace{\text{UCB}}_{\text{may be useful}} \setminus \overbrace{\underbrace{\{m \mid res(m) \geq l\}}_{\text{must remain useful}}}^{\text{blocks that may have to be reloaded}}| \tag{10}$$

In order to ensure the soundness of our results, we have to overapproximate the set of UCBs and to bound the number of ECBs from above, but underapproximate the resilience of the useful cache blocks. Only cache blocks, for which we can guarantee the "survival", can be safely excluded from the CRPD bound.

We can similarly refine the CRPD analysis results based on Definitely-Cached UCBs (see Section 2 and [4]). We just need to replace UCB by DC-UCB in Equation 10.

4. Resilience Analysis

How is the resilience computed? The resilience of a UCB m is determined by the maximal age $max\text{-}age(m)$ at all next hits to m without prior eviction. At each program point P, this maximal age can be split into two parts:

i) $ca_P^{\leftarrow}$, the maximal number of accesses from the last use of m to program point P, and

ii) $ca_P^{\rightarrow}$, the maximal number of accesses from program point P to the next hit to m,

both under the constraint that m is not evicted before its next reuse. We denote the two parts as the constrained ages $ca_P^{\leftarrow}$ and $ca_P^{\rightarrow}$ of m at P and employ two symmetric data-flow analyses on the control flow graph of the analyzed program to bound both parts from above: a forward analysis for the first (i) and a backward for the second part (ii). The maximal age is then bounded by the sum of both.

For the analysis[4] of the constrained age, ca, of a memory block, we only take those paths into account on which the constraint is satisfied[5]. Thus, ca of m does not necessarily overapproximate the actual age of m. To be able to correctly update the constrained age ca, upon an access to a memory block, we also need to maintain an unconstrained bound on the age of the block. We denote this upper bound on the actual age by ua. To derive ua, we employ a *must* cache analysis [7].

[4] The analysis of $ca^{\leftarrow}$ and $ca^{\rightarrow}$ only differ by the direction. Hence, we omit the direction in the following explanation.

[5] m is not evicted before its next reuse.

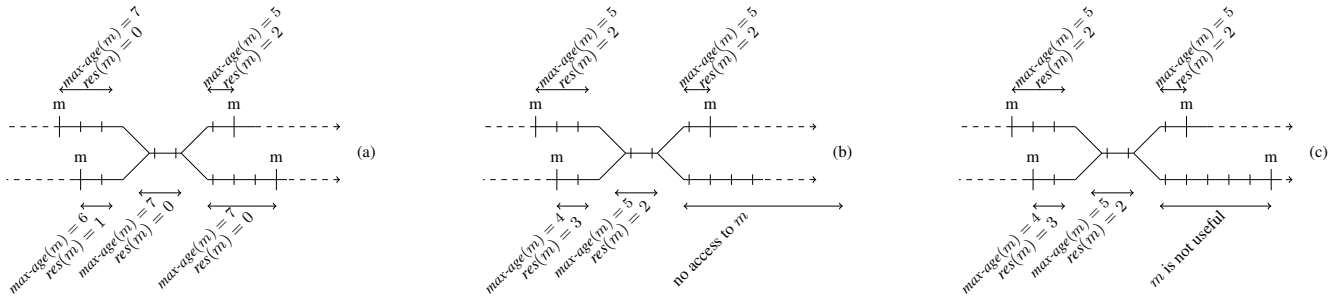


Figure 4. Resilience under several possible next accesses.

Remember that we assume a fully-associative cache for this section. In case of a set-associative cache, the analysis is performed for each cache set.

The domain of the analysis is a tuple of two functions. The first function assigns each memory block a bound on its constrained age ca, and the second assigns a bound on its unconstrained age ua.

$$\mathbb{D} : \mathbb{D}_{ca} \times \mathbb{D}_{ua} \tag{11}$$

with

$$\mathbb{D}_{ca} : \mathbb{M} \to \{0, \ldots, k-1\} \tag{12}$$

and

$$\mathbb{D}_{ua} : \mathbb{M} \to \{0, \ldots, k-1, \infty\} \tag{13}$$

The age of a memory block m whose next access can still result in a hit is bounded by $k-1$. Older memory blocks are not cached. The next access to such blocks must be a miss. Hence, due to the constraint, the constrained age $ca(m)$ of a memory block m is at most $k-1$. Without the constraint, blocks may reach arbitrary ages $\geq k$, which we do not need to distinguish. In $\mathbb{D}_{ua}$, ages $\geq k$ are represented by ∞.

The join operator of the domain, invoked to combine flow information from different paths, is defined using a separate join-function for each element of the tuple:

$$\bigsqcup : \mathcal{P}(\mathbb{D}) \to \mathbb{D}$$

$$\bigsqcup D := (\bigsqcup_{ca} CA, \bigsqcup_{ua} UA) \tag{14}$$

where $CA := \{ca \mid (ca, ua) \in D\}$ and $UA := \{ua \mid (ca, ua) \in D\}$.

The join operator for the ca and ua are defined as follows:

$$\bigsqcup_{ca} : \mathcal{P}(\mathbb{D}_{ca}) \to \mathbb{D}_{ca}$$

$$\bigsqcup_{ca} CA := \lambda m. \max_{ca \in CA} ca(m) \tag{15}$$

$$\bigsqcup_{ua} : \mathcal{P}(\mathbb{D}_{ua}) \to \mathbb{D}_{ua}$$

$$\bigsqcup_{ua} UA := \lambda m. \max_{ua \in UA} ua(m) \tag{16}$$

In both cases, we bound the ages from above. Hence, we need to take the maximum of the bounds of all incoming data-flow values.

The transfer function, computing the update on the bounds, is defined using the following auxiliary functions t_{ca} and t_{ua}.

$$t_{ua} : \mathbb{D}_{ua} \times \mathbb{M} \to \mathbb{D}_{ua}$$

$$t_{ua}(ua, m) := \lambda m'. \begin{cases} 0 & m' = m \\ ua(m') & ua(m') \geq ua(m) \\ ua(m') + 1 & ua(m') < ua(m) \wedge ua(m') < k-1 \\ \infty & \textit{otherwise} \end{cases} \tag{17}$$

The inputs to t_{ua} are the upper bound on the age ua and the accessed memory block m. The age of the currently accessed memory block is zero. Older elements or elements of the same age as the accessed one remain unchanged. Only younger elements are aged by one. Note that $k-1$ is the maximal age of cached memory blocks. If $ua(m) = \infty$, m is not guaranteed to be cached.

The auxiliary function t_{ca} is similar to t_{ua}, yet it also takes unconstrained ages as inputs:

$$t_{ca} : \mathbb{D}_{ca} \times \mathbb{D}_{ua} \times 2^{\mathbb{M}} \times \mathbb{M} \to \mathbb{D}_{ca}$$

$$t_{ca}(ca, ua, \text{UCB}, m) := \lambda m'. \begin{cases} 0 & m' = m \vee m' \notin \text{UCB} \\ ca(m') & ca(m') \geq ua(m) \vee ca(m') = k-1 \\ ca(m') + 1 & ca(m') < ua(m) \end{cases} \tag{18}$$

The inputs to t_{ca} are the bound on the constrained age ca, the bound on the unconstrained age ua, the set of UCBs at the specific program point and the accessed memory block m. Function t_{ca} defines the change of ca upon access to m given a specific set of UCBs. The update of ca is similar to the update of ua. In the first case, $ca(m') = 0$, if $m' = m$ or $m' \notin$ UCB. If $m' \notin$ UCB the constraint is not satisfied, i.e., the next access to m' may not be a hit. Then any bound would be correct and $ca(m') = 0$ is the best possible. Note that the bound on the constrained age ca may underapproximate the actual age. Hence, we need to consider $ua(m)$ as the upper bound on the age of the accessed element m. Under the constraint, that the next access to m' is a hit, it can never obtain an age greater than $k-1$, this explains the additional condition $ca(m') = k-1$ compared with the other transfer function.

The transfer function invokes t_{ca} and t_{ua} depending on the program point and the direction of the analysis. Remember that the forward analysis ($\rightarrow$) bounds the number of accesses from the last access to a memory block m to the current program point and the backward analysis ($\leftarrow$) from the current program point to the next access to m. The forward analysis invokes t_{ca} with incoming data-flow value ca, the memory block m_P accessed at P and the set of UCBs before P, UCB^b_P. The backward analysis considers the set of UCBs after the program point, UCB^a_P. In both cases, t_{ua} is invoked with ua and m_P.

$$T : \mathbb{D} \times \mathbb{P} \to \mathbb{D}$$

$$T^{\rightarrow}((ca, ua), P) := (t_{ca}(ca, ua, \text{UCB}^b_P, m_P), t_{ua}(ua, m_P)) \tag{19}$$

$$T^{\leftarrow}((ca, ua), P) := (t_{ca}(ca, ua, \text{UCB}^a_P, m_P), t_{ua}(ua, m_P)) \tag{20}$$

In case of data caches, the address of a memory access may be unknown statically. For instance, consider an array access within a loop. In such a case, only a non-singleton set of possible addresses for each memory access can be derived. The transfer function $\widehat{T}$ (for both, forward and backward analysis) is then defined by the join of the transfer function t applied to each element in the set of possible memory accesses ($\mathbb{M}_P$ is the set of memory blocks possibly accessed at P).

$$\widehat{T} : \mathbb{D} \times \mathbb{P} \to \mathbb{D}$$

$$\widehat{T}^{\rightarrow}((ca, ua), P) := \bigsqcup_{m \in \mathbb{M}_P} \{(t_{ca}(ca, ua, \text{UCB}^b_P, m), t_{ua}(ua, m))\} \tag{21}$$

$$\widehat{T}^{\leftarrow}((ca, ua), P) := \bigsqcup_{m \in \mathbb{M}_P} \{(t_{ca}(ca, ua, \text{UCB}^a_P, m), t_{ua}(ua, m))\} \tag{22}$$

Let $ca^{\rightarrow}_P$ be the result of the forward analysis and $ca^{\leftarrow}_P$ the result of the backward analysis before program point P. The maximal age *max-age* is then bounded by the sum of both bounds limited to $(k-1)$:

$$\textit{max-age}_P(m) \leq \min\{ca^{\rightarrow}_P(m) + ca^{\leftarrow}_P(m), k-1\} \tag{23}$$

Note that $k-1$ is the maximal age such that m is still cached at the next access. The resilience $\textit{res}_P(m)$ of a memory block m before P is then given by Equation 9:

$$\textit{res}_P(m) := (k-1) - \textit{max-age}_P(m).$$

5. From Resilience to CRPD

The CRPD computation, based on the notion of resilience, has shortly been sketched in Section 3 for fully-associative caches and for a single program point only. In this section, we present the computation of the CRPD bound of a whole task, the extension to set-associative caches (with more than one set), an efficient approach to compute the CRPDs for given task sets, and the extension to multiple preemptions by multiple tasks.

In case of set-associative caches, the CRPD at program point P is given by the sum of the CRPDs of each set s:

$$\text{CRPD-RES}_P(\text{ECB}) = \sum_s \text{CRPD-RES}^s_P(\text{ECB}^s) \tag{24}$$

$$\text{CRPD-RES}^s_P(\text{ECB}_s) = \text{CRT} \times |\text{UCB}^s_P \setminus \{m \mid \textit{res}_P(m) \geq |\text{ECB}^s|\}| \tag{25}$$

where ECB^s and UCB^s_P denote the sets of elements from ECB/UCB mapping to cache set s.

The bound on the CRPD of the whole task is then determined by the maximum CRPD of all program points.

$$\text{CRPD-RES}(\text{ECB}) = \max\{\text{CRPD-RES}_P(\text{ECB}) \mid P \in \mathbb{P}\} \tag{26}$$

5.1 Computation of a sufficient set of preemption points

The drawback of this computation is that the set $(\text{UCB}^s_P \setminus \{m \mid \textit{res}_P(m) \geq |\text{ECB}^s|\})$ has to be evaluated for each program point each time the CRPD is computed for another preempting task. Depending on the task size and the set of preempting tasks, this computation can be very time-consuming. Therefore, we present a precomputation to speed up the instantiation by reducing the number of program points that need to be considered. To this end, we use a partial order on the program points, such that the CRPD only needs to be derived for a subset of all program points.

The partial order on CRPD-RES^s_P is defined by the point-wise comparison on the set of $(\text{UCB}^s_P \setminus \{m \mid \textit{res}_P(m) \geq |\text{ECB}^s|\})$ for each possible input.

$$\text{CRPD-RES}^s_{P_1} \leq \text{CRPD-RES}^s_{P_2} \text{ iff } \forall l \in [0; k-1] : |\text{UCB}^s_{P_1} \setminus \{m \mid \textit{res}_{P_1}(m) \geq l\}| \leq |\text{UCB}^s_{P_2} \setminus \{m \mid \textit{res}_{P_2}(m) \geq l\}| \tag{27}$$

The partial order on CRPD-RES_P is then defined by the point-wise comparison on CRPD-RES^s_P for each cache-set s.

$$\text{CRPD-RES}_{P_1} \leq \text{CRPD-RES}_{P_2} \text{ iff } \forall s : \text{CRPD-RES}^s_{P_1} \leq \text{CRPD-RES}^s_{P_2} \tag{28}$$

The global CRPD bound must be assumed at one of the minimal program points in the partial order:

$$\mathbb{P}_{\textit{Max}} = \{P \mid \neg\exists P' : \text{CRPD-RES}_P < \text{CRPD-RES}_{P'}\} \tag{29}$$

$$\text{CRPD-RES}(\text{ECB}) = \max\{\text{CRPD-RES}_P(\text{ECB}) \mid P \in \mathbb{P}_{\textit{Max}}\} \tag{30}$$

5.2 Multiple preemptions

So far, we have presented the computation of bounds on the CRPD for a single preemption by a single task. However, schedulability analyses usually have to take into account multiple and nested preemptions. This can be preemptions by a single task or even by multiple different tasks. Nested preemptions can be easily handled by taking the union of the ECBs of all preempting tasks. In this section, we discuss the challenges that arise when bounding the CRPD for multiple preemptions and present an approach for our resilience analysis.

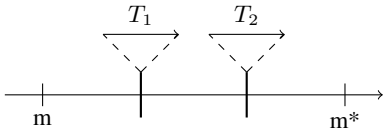
Why are multiple preemptions a challenge? Multiple preemptions, in particular by multiple preempting tasks, may "interact" to cause more additional misses than they would in "isolation". We say that two preemptions *interact* if there is a memory block m, s.t. there are two consecutive accesses to m that enclose the two preemptions. See Figures 5(a) and 5(b) for examples of interacting tasks T_1 and T_2 due to accesses to memory block m. We call preemptions that do not interact with any other preemption *isolated*. For direct-mapped (dm) caches, multiple—possibly interacting—preemptions do not pose additional problems. In direct-mapped caches, for all useful cache blocks m, $\textit{res}(m) = 0$, i.e., each useful cache block is evicted by a single ECB. If two preempting tasks interact, they may only cause less misses than the sum of the misses of isolated preemptions by the two tasks. If the two preempting tasks access the same cache set (of associativity one) that contains a useful cache block they may cause at most one additional miss, while two isolated preemptions could cause up to two misses.

So, the CRPD caused by such preemptions is always bounded by the sum of the CRPDs the preemptions would cause in isolation. Let *Tasks* be the set of preempting tasks, $\text{CRPD}^{dm}(T)$ the cost of a single preemption by task $T \in \textit{Tasks}$, and $\#p(T)$ the number of preemptions by task T. Then the total $\text{CRPD}^{dm}(\textit{Tasks}, \#p)$ in a direct-mapped (dm) cache is bounded by:

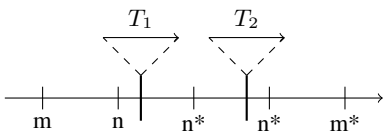
$$\text{CRPD}^{dm}(\textit{Tasks}, \#p) \leq \sum_{T \in \textit{Tasks}} \#p(T) \cdot \text{CRPD}^{dm}(T). \tag{31}$$

This property is used in the analysis of multiple preemptions for direct-mapped caches [8, 16, 18].

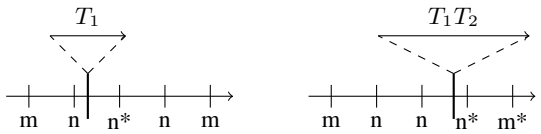
However, this property does not hold for set-associative caches. The total CRPD caused by such preemptions may be higher than



(a) Interacting preemptions by T_1 and T_2 between two subsequent accesses to m.



(b) Interacting preemptions by T_1 and T_2 between two subsequent accesses to m and between different accesses to n.



(c) Isolated preemptions by T_1 and T_1T_2. The sum of the number of additional misses, $1 + 2 = 3$, bounds the number of additional misses in case of interacting preemptions, 3, by T_1 and T_2 as depicted in (b).

Figure 5. Different preemption scenarios with interacting and isolated preemptions. m and n denote memory accesses. A $*$ as in m^* indicates an additional miss due to one or more preemptions. In all scenarios, $|\text{ECB}(T_1)| = |\text{ECB}(T_2)| = 2$, $|\text{ECB}(T_1) \cup \text{ECB}(T_2)| = |\text{ECB}(T_1T_2)| = 4$, and $res_P(m) = 3, res_P(n) = 1$ for all program points P.

the sum of the costs of preemptions by single tasks. Consider a useful cache block m that can be evicted by four evicting cache blocks but not by three. In our terms, $res(m) = 3$. If tasks T_1 and T_2 both access at most 2 memory blocks in the same set ($|\text{ECB}(T_1)| = |\text{ECB}(T_2)| = 2$), a preemption by T_1 alone or T_2 alone will not cause eviction of block m. However, if the task accessing m is preempted by both T_1 and T_2 between two accesses to m, i.e., the two preemptions interact, m may be evicted. Figure 5(a) illustrates this. Only upon the next use of m does the memory block "regain" its resilience. Even multiple preemptions by a single task may interact in such a way. While T_1 accesses at most 2 memory blocks in the same set, it may access different blocks in different executions. So several preemptions by T_1 may access more than 3 different blocks in the cache set of m and evict it.

In general, it is therefore not sufficient to simply sum up the CRPD bounds obtained for individual isolated preempting tasks. Whether and to what extent this is possible depends on the CRPD analysis for the individual preemptions. Taking into account the $\text{CRPD}_{\text{UCB\&ECB}}$ as described in Section 2, one can compute a CRPD bound for set-associative caches as in Equation 31. This is because, $\text{CRPD}_{\text{UCB\&ECB}}$ does not tightly couple the analysis of the preempted and the preempting task. It does not take into account that certain blocks are too resilient to be evicted due to a particular preempting task. It assumes that blocks are evicted even if a single ECB maps to the same set, resembling the direct-mapped cache case. For CRPD bounds based on resilience summing up the CRPD bounds as in Equation 31 is not correct in general.

Bounds on the CRPD for multiple preemptions by a single task Let the set of $\text{ECB}(T)$ overapproximate the set of memory blocks accessed by task T in *any* execution. Even if certain blocks may not be accessed in the same execution of a task do they appear in $\text{ECB}(T)$. Using $\text{ECB}(T)$ in $\text{CRPD-RES}(\text{ECB}(T))$ we already account for misses that may only occur due to several executions of T. Therefore, we can bound the total CRPD in set-associative (*sa*) caches caused by multiple preemptions by a single task in the following way:

$$\text{CRPD}^{sa}(\{T\}, \#p) \leq \#p(T) \cdot \text{CRPD-RES}(\text{ECB}(T)). \quad (32)$$

However, for multiple preemptions by multiple different tasks, this does not work as discussed in the previous section and as illustrated in Figure 5(a).

Bounds on the CRPD for multiple preemptions by multiple tasks Our approach is to incrementally compute bounds on the CRPD for growing sets of preempting tasks. We start out with a single preempting task. Then, we keep adding preempting tasks, one at a time, maintaining an upper bound on the CRPD due to preemptions of the tasks that have been added so far.

In principle, any order of inserting the preempting tasks can be processed. However, the order has an influence on the precision of the resulting CRPD bound. Adding tasks in nonincreasing order of the number of preemptions simplifies the algorithm and promises to yield low bounds. In the following, we will assume w.l.o.g. that the preempting tasks $\mathit{Tasks} = \{T_1, T_2, \ldots\}$ are ordered in this way, i.e., $\#p(T_i) \geq \#p(T_{i+1})$ for all i.

Our algorithm is based on a slight generalization of the following insight:

$$\begin{aligned}\text{CRPD}^{sa}(\{T_1, T_2\}, \#p) \;\leq\; & \text{CRPD}^{sa}(\{T_1\}, \#p) \\ & + \text{CRPD}^{sa}(\{T_1T_2\}, \#p')\end{aligned} \quad (33)$$

where T_1T_2 denotes a task that is the sequential composition of T_1 and T_2 and $\#p'(T_1T_2) = \#p(T_2)$.

Equation 33 says that we can compute a bound on the CRPD due to preemptions by T_1 and T_2 by adding up

1. a bound on the cost of the preemptions of T_1 in isolation, and
2. the cost of $\#p'(T_1T_2) = \#p(T_2)$ preemptions by T_1T_2.

Intuitively, the preemptions of T_2 may only interact $\#p(T_2)$ $(= \min\{\#p(T_1), \#p(T_2)\}$[6]$)$ times with the preemptions of T_1. If a memory block may be evicted by an interaction of T_1 and T_2, it may also be evicted by the sequential composition T_1T_2 of the two tasks. Figure 5(c) illustrates this.

Generalizing Equation 33 from pairs of preempting tasks to sets of preempting tasks yields the following:

$$\begin{aligned}\text{CRPD}^{sa}(\{T_1, \ldots, T_i\}, \#p) \leq\; & \text{CRPD}^{sa}(\{T_1, \ldots, T_{i-1}\}, \#p) \\ & + \text{CRPD}^{sa}(\{T_1T_2 \ldots T_i\}, \#p')\end{aligned} \quad (34)$$

where $T_1T_2 \ldots T_i$ denotes a task that is the sequential composition of $T_1, T_2, \ldots, T_i$ and $\#p'(T_1T_2 \ldots T_i) = \#p(T_i)$.

Intuitively, the preemptions of T_i may only interact $\#p(T_i)$ $(= \min\{\#p(T_1), \ldots, \#p(T_i)\})$ times with the preemptions of $T_1, \ldots, T_{i-1}$. If a memory block may be evicted by an interaction of $T_1, T_2, \ldots, T_i$, it may also be evicted by the sequential composition $T_1T_2 \ldots T_i$ of the set of tasks.

As the set of evicting cache blocks by $T_1T_2 \ldots T_i$ is simply $\bigcup_{j=1\ldots i} \text{ECB}(T_j)$ we can use Equation 32 to bound the second summand of Equation 34 by $\#p(T_i) \cdot \text{CRPD-RES}(\bigcup_{j=1..i} \text{ECB}(T_j))$:

$$\begin{aligned}\text{CRPD}^{sa}(\{T_1, \ldots, T_i\}, \#p) \leq\; & \text{CRPD}^{sa}(\{T_1, \ldots, T_{i-1}\}, \#p) \\ & + \#p(T_i) \cdot \text{CRPD-RES}\Big(\bigcup_{j=1..i} \text{ECB}(T_j)\Big)\end{aligned} \quad (35)$$

[6] Due to our assumption of a nonincreasing $\#p$-function.

Plugging Equation 35 recursively into the first summand of itself yields Equation 36:

$$\text{CRPD}^{sa}(\{T_1,\ldots,T_k\},\#p) \leq \sum_{i=1\ldots k} \#p(T_i)\cdot \text{CRPD-RES}(\bigcup_{j=1..i}\text{ECB}(T_j)) \quad (36)$$

Based on this, we can compute bounds on the CRPD for arbitrary sets of preempting tasks. In fact, given *any* method to compute bounds on the CRPD for multiple preemptions by a single task, Equation 34 enables to incrementally compute bounds on the CRPD for multiple preemptions by multiple tasks.

6. Evaluation

In this section, we evaluate the resilience analysis. The implementation is based on the aiT Timing Analyzer [7]. We compare the bound on the CRPD we obtain with the results of the former approaches ($\text{CRPD}_{\text{UCB\&ECB}}$, CRPD_{TAN} and CRPD_{UCB}, see Section 2). We have selected two different benchmark suites to show the improvements for a single preemption (Mälardalen WCET Benchmark Suite [1]) and the improvements on a realistic example with multiple preemptions (Papabench [2]). In both cases, the ARM7 processor[8] is our target architecture. The runtime for both benchmarks was less than 30 minutes on an AMD Quadcore with 2500 Mhz.

6.1 Single Preemption

Task	Code Size	Cache Util.	|UCB|
minmax	608B	7.4%	4
insertsort	384B	4.7%	5
fibcall	256B	3.1%	5
fac	256B	3.1%	6
bs	320B	3.9%	8
bsort100	544B	6.6%	10
ns	576B	7%	11
matmult	864B	10.5%	12
fir	928B	11.3%	22
crc	1216B	14.8%	35
select	1280B	15.6%	37
qsort-exam	1440B	17.6%	42
sqrt	3680B	44.9%	101
qurt	4160B	50.8%	118

Table 2. Mälardalen benchmark suite: Code sizes, cache utilization and number of UCBs.

The Mälardalen Benchmark Suite offers a wide range of different test programs. These programs, however, do not form an application. They are merely a set of small, independent tasks. However, they are the standard benchmarks for timing analysis. Instead of assuming an artificial schedule, we used this suite to show the improvement of our analysis for single preemptions. We selected a cache of size 8KB with 8 ways, 32 sets and a line-size of 32 bytes. The cache utilization and the maximal number of useful cache blocks of the different tasks are given in Table 2. Note that although PLRU replacement policy is often used for caches with an associativities higher than 4, an efficient LRU implementation is also feasible as Ackland et al. [3] have shown.

As the preempting tasks, we select *fibcall* (smallest number of ECBs) and *qurt* (highest number of ECBs) and bound the number of additional misses using the different approaches discussed in this paper (Resilience, Tan, UCB & ECB, UCB) see Figure 6. Preempted tasks are ordered according to the number of UCBs. Note that the scale is logarithmic. Results show that the resilience

[7] `http://www.absint.de/ait/`

[8] `http://www.arm.com/products/CPUs/families/ARM7Family.html`

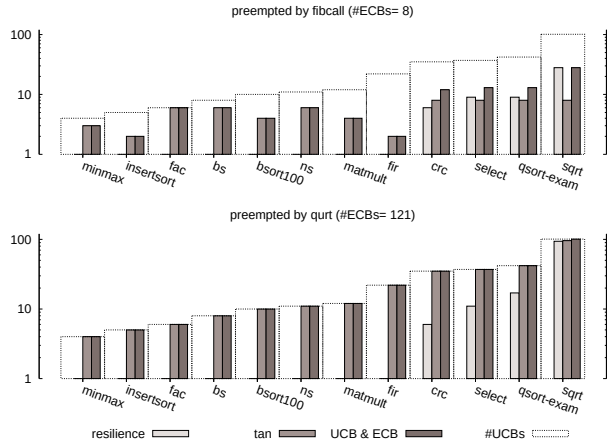


Figure 6. Number of additional misses due to preemption by *fibcall* and *qurt* when the CRPD is computed with Equations 10 (resilience), 7 (UCB & ECB), 8 (Tan) and 1 (#UCBs).

analysis classifies all UCBs to remain useful (no CRPD cost) if the preempted task is small ($|\text{UCB}| < 20$). In such cases, the useful cache blocks are very resilient and are thus proven to remain useful under preemption. In these cases, the improvement over previous approaches is therefore 100%. Even for larger tasks, resilience usually outperforms former approaches. Only for *select, qsort-exam* and *sqrt* preempted by *fibcall*, Equation 8 computes a smaller, but potentially unsafe bound.

6.2 Multiple Preemptions/Complete Scenario

Papabench is a free benchmark suite based on a realistic real-time application, the control software of an unmanned aircraft vehicle. It offers two different sets of tasks executed on two processors with two different modes, *manual* and *automatic* flight control. For our evaluation, we chose the larger task set, which is executed on processor *MCU0* with periodic tasks and preemptive rate-monotonic scheduling (see configuration 3 in [11]). Tasks with the same period cannot preempt each other in this configuration. In contrast to other benchmarks, we can derive the periods and priorities of all tasks from the specification of the system. See Table 3 for an overview of the task set. Hence, we can evaluate the approaches discussed in this paper on a realistic and complete example. To this end, we bounded the preemption costs for one instance of a task. In contrast to the evaluation for single preemptions, we omit the unsound CRPD-bound of Equation 8. There is simply no way to combine unsound intermediate results to a safe bound in case of multiple preemptions. For the other two approaches, we computed the CRPD bound for multiple preemptions in the following way:

UCB & ECB the CRPD bound for multiple preemptions is the sum of all CRPD bounds for each preempting task T_i weighted by an upper bound on the number of preemption due to T_i.

Resilience the CRPD bound for multiple preemptions is computed using Equation 36.

The intermediate results for these computations for T10 in automatic and manual mode are shown in Figure 7 and 8. Figure 7 shows the individual costs for the preempting tasks. The resilience analysis has to take potential interaction with already considered tasks into account. Hence, the cost for the current preempting task T_i is determined by the union of the ECBs sets of T_i and all preempting tasks that have been previously accounted for. In case of UCB & ECB, the cost for preemption by T_i is only determined by

Task	Description	Period	Modes	# preempt	
				A	M
T5	altitude_contro	250ms	A,M	13	18
T6	climb_control	250ms	A,M	13	18
T7	link_fbw_send	250ms	A	13	-
T8	navigation	250ms	M	-	18
T9	radio_control	25ms	M	-	0
T10	receive_gps_data	250ms	A,M	13	18
T11	repeating_task	100ms	A	4	-
T12	stabilisation	50ms	A	0	-
I4	interrupt_modem	100ms	A,M	4	6
I5	interrupt_spi_1	50ms	A,M	13	18
I6	interrupt_spi_2	50ms	A,M	0	2

Table 3. Papabench benchmark suite: Task set for processor *MCU0*, (A)utomatic and (M)anual modes. Description, period, modes in which tasks are active and upper bounds on the number of preemptions by higher priority tasks in the different modes.

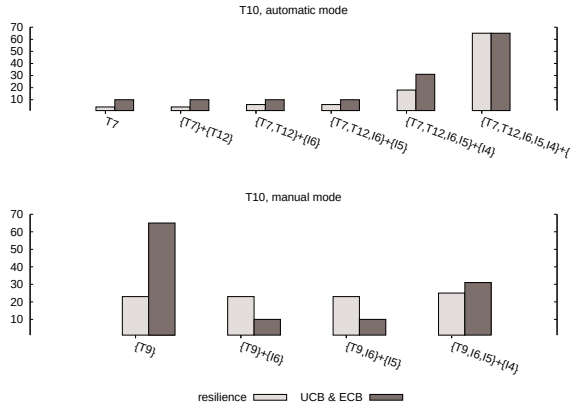


Figure 7. Bounds on the number of additional misses of task T10 due to a single preemption 1) by a set of tasks (resilience), e.g. {T9, I6, I5} and 2) by a single task (UCB&ECB), e.g I5.

the ECB set of T_i. Figure 8 shows the next step for the computation of the CRPD bounds for multiple preemptions. The costs for the individual tasks are weighted by the number of preemptions and summed up.

Due to space limitations, we only present the final results for the tasks with the lowest priority that are used in both automatic and manual mode (T5, T6, T10). These tasks are preempted most often. The CRPD bounds for multiple preemptions are shown in Figure 9.

The possible pessismism introduced by Equation 36 does not consume the precision gain of the resilience analysis. In our benchmarks, the new resilience analysis always outperforms the previous approach by at least 28% and by 64% on average. In manual mode, the bound on the overall number of additional misses in T6 could even be reduced from 256 to 0. The analysis of the second benchmark suite, Papabench, to evaluate the precision in case of multiple preemptions, leads to very similar results as the analysis of the Mälardalen benchmarks for single preemptions.

7. Further Applications of Resilience

So far, we have shown how resilience analysis can be used to improve estimation of the CRPD. Yet, there are other problems in timing analysis that may benefit from resilience analysis. In this section, we will point out two such opportunities and sketch how CRPD analyses and in particular resilience analysis can be applied.

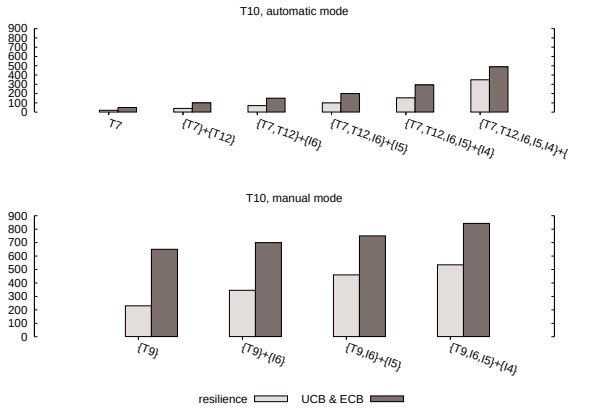


Figure 8. Bounds on the number of additional misses of task T10 due to preemptions by sets of tasks, derived by Resilience and UCB&ECB. These are weighted sums of the results for single preemptions which are shown in Figure 7.

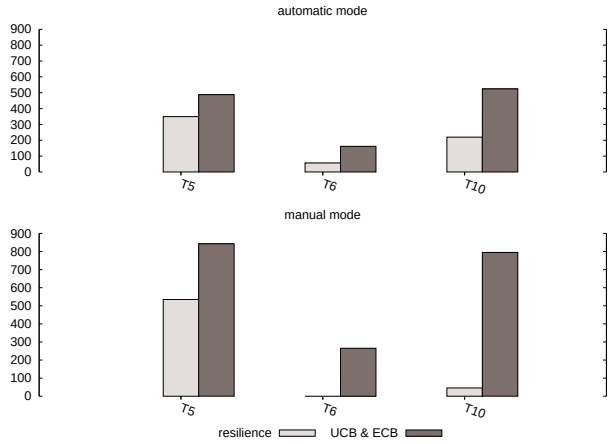


Figure 9. Overall number of additional misses for lowest priority tasks due to preemptions by higher priority tasks in case of automatic and manual mode.

7.1 Inter-Task Cache Analysis

At program start, current WCET analyses either assume nothing about the cache contents, or an empty cache, which may be unsound in case of timing anomalies [9, 15]. These WCET analyses predict cache hits due to *intra-task* reuse of instructions or data. Yet, such analyses do not predict any cache hits due to *inter-task* reuse, due to their assumptions about the cache contents at program start. In small applications inter-task reuse may be substantial.

Except for the first instance of a task, there might be cache hits that result from inter-task reuse, i.e., reuse across different instances of the task. It would be beneficial to account for such hits in a timing analysis. The only paper we are aware of that deals with precisely this problem is [12]. In [12], the result of a pessimistic analysis[9] is enhanced by computing a set of memory accesses that must result in cache hits due to previous instances of a task. We propose to start out optimistically, assuming that no other tasks are executed between two instances of a task. Executions of other

[9] Pessimistic in the sense that it assumes no knowledge of the cache contents at program start.

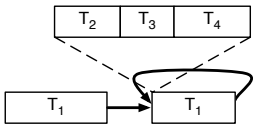


Figure 10. The execution of T_2, T_3, T_4 between the periodic execution of T_1 seen as a "preemption" of a cyclic T_1 task.

tasks between two instances of the same task can then be seen as preemptions at fixed program points. Additional misses due to such "preemptions" can then be accounted for as described in the previous sections. This idea is illustrated in Figure 10. It remains to evaluate the precision of this approach compared with that of [12].

7.2 Component-Wise Cache Analysis

Component-wise cache analysis [5, 13] tries to increase the scalability of cache analysis by analyzing components of a system independently. Instead of analyzing a function at every call site, it is only analyzed once, generating a summary of the "damage" on the cache state caused by the function. This summary is then applied to the cache states at every call site. Following such an approach one has to analyze library functions only once. Ballabriga et al. [5] recently increased the precision of component-wise cache analysis by deriving more precise summaries than Rakib et al. [13] who introduced the idea. Both [13] and [5] still have some limitations:

- They require the knowledge of relative positions of functions in the cache. This knowledge is only available after linking. If the relative positions are unknown, the two approaches would have to account for the *sum* of the "damages" caused by all different relative positions in the cache.
- If function pointers are employed it is not always statically known which function is called. In such cases, the above analyses again have to conservatively account for the *sum* of the "damages" done by all of the possible functions. If for instance, one function accesses cache sets 1 to 4 and the other function accesses the cache sets 5 to 8, the above analyses would have to account for "damage" in cache sets 1 to 8.

How can resilience analysis improve component-wise cache analysis? Instead of directly accounting for the damage done by the function that is called, one could first optimistically analyze a program ignoring the function calls. In a second step, one could then use the results of a resilience analysis to determine the maximal number of additional misses due to the function calls. If the relative positions of the called function and the caller are unknown, one could conservatively account for the maximal number of additional misses. This should be considerably more precise than accounting for the *sum* of the damages resulting from all possible relative positions. Similarly, for unresolved function calls due to function pointers.

8. Conclusions

Preemptive scheduling often offers increased schedulability at the cost of higher analysis complexity. Apart from schedulability analysis, timing analysis has to be extended to derive the context switch costs, and especially the cache-related preemption delay. The corresponding analyses for direct-mapped caches can be considered rather mature and precise. For set-associative caches, however, former analyses are either imprecise or unsound.

In this paper, we introduced a new CRPD analysis for LRU caches based on the notion of resilience. The resilience denotes how much disturbance by the preempting task, a cache block of the preempted one may endure, without eviction before its reuse. We presented a data-flow analysis to derive the resilience of a cache block as well as the computation of CRPD bounds for single and multiple preemption based on this notion. The evaluation has shown that our analysis improves on former approaches by at least 28% and by 64% on average. Especially, for preemption due to small, but frequent tasks, such as interrupts, a strong improvement can be achieved.

As future work, we plan to evaluate our method for other cache configurations and in the presence of an RTOS.

References

[1] Mälardalen WCET benchmark suite. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

[2] Papabench: a free real-time benchmark. `http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=97`.

[3] B. Ackland, D. Anesko, D. Brinthaupt, S. J. Daubert, A. Kalavade, J. Knoblock, E. Micca, M. Moturi, C. J. Nicol, J. H. O'Neill, J. Othmer, E. Sackinger, K. J. Singh, J. Sweet, C. J. Terman, and J. Williams. A single-chip, 1.6 billion, 16-b mac/s multiprocessor dsp,. *IEEE Journal of Solid-state circuits*, 35(3):412–423, 2000.

[4] S. Altmeyer and C. Burguière. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *ECRTS '09*, pages 109–118. IEEE Computer Society, 2009.

[5] C. Ballabriga, H. Cassé, and P. Sainrat. An improved approach for set-associative instruction cache partial analysis. In *SAC '08*, pages 360–367, 2008.

[6] C. Burguière, J. Reineke, and S. Altmeyer. Cache-related preemption delay computation for set-associative caches: Pitfalls and solutions. In *WCET '09*, 2009.

[7] C. Ferdinand and R. Wilhelm. Fast and efficient cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2/3):131–181, 1999.

[8] C.-G. Lee, J. Hahn, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *RTSS'96*, page 264. IEEE Computer Society, 1996.

[9] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '99*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.

[10] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS'03*. ACM, 2003.

[11] F. Nemer, H. Cassé, P. Sainrat, J. P. Bahsoun, and M. D. Michiel. Papabench: a free real-time benchmark. In *WCET '06*, Dagstuhl, Germany, 2006.

[12] F. Nemer, H. Cassé, P. Sainrat, and J. P. Bahsoun. Inter-task WCET computation for a-way instruction caches. In *SIES '08*, pages 169–176, 2008.

[13] A. Rakib, O. Parshin, S. Thesing, and R. Wilhelm. Component-wise i-cache behavior prediction. In *ATVA '04*, pages 211–229, 2004.

[14] J. Reineke and D. Grund. Relative competitive analysis of cache replacement policies. In *LCTES'08*, pages 51–60. ACM, June 2008.

[15] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. In *WCET '06*, Dagstuhl, Germany, 2006.

[16] J. Staschulat and R. Ernst. Multiple process execution in cache related preemption delay analysis. In *EMSOFT '04*, pages 278–286. ACM, 2004.

[17] J. Staschulat and R. Ernst. Scalable precision cache analysis for real-time software. *ACM TECS*, 6(4):25, 2007. ISSN 1539-9087.

[18] Y. Tan and V. Mooney. Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems. In *SCOPES'04*, pages 182–199, 2004.

[19] H. Tomiyama and N. D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *CODES '00*. ACM, 2000.