

Cache-Related Preemption Delay Analysis for FIFO Caches

Clément Ballabriga Lee Kee Chong Abhik Roychoudhury

National University of Singapore
{clementb,cleek,abhik}@comp.nus.edu.sg

Abstract

Hard real-time systems are typically composed of multiple tasks, subjected to timing constraints. To guarantee that these constraints will be respected, the Worst-Case Response Time (WCRT) of each task is needed. In the presence of systems supporting preemptible tasks, we need to take into account the time lost due to task preemption. A major part of this delay is the Cache-Related Preemption Delay (CRPD), which represents the penalties due to cache block evictions by preempting tasks. Previous works on CRPD have focused on caches with Least Recently used (LRU) replacement policy. However, for many real-world processors such as ARM9 or ARM11, the use of First-in-first-out (FIFO) cache replacement policy is common.

In this paper, we propose an approach to compute CRPD in the presence of instruction caches with FIFO replacement policy. We use the result of a FIFO instruction cache categorization analysis to account for single-task cache misses, and we model as an Integer Linear Programming (ILP) system the additional preemption-related cache misses. We study the effect of cache related timing anomalies, our work is the first to deal with the effect of timing anomalies in CRPD computation. We also present a WCRT computation method that takes advantage of the fact that our computed CRPD does not increase linearly with respect to the preemption count. We evaluated our method by computing the CRPD with realistic benchmarks (*e.g.* drone control application, robot controller application), under various cache configuration parameters. The experimentation shows that our method is able to compute tight CRPD bound for benchmark tasks.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification

Keywords CRPD, FIFO caches, WCRT, timing anomalies

1. Introduction

In real time systems, the execution time of a real time program must be *predictable* and *consistent* to ensure reliability and safety of the system. The Worst Case Execution Time (WCET) of a program is a bound on the maximum execution time of the program over all possible executions. WCET analysis is used to bound the WCET of a program to verify that all required timing constraints

are met. The micro-architecture on which the program executes is a significant factor in WCET analysis, as a program can have different execution time depending on the underlying hardware. Thus, the timing bound obtained from WCET analysis is significantly improved by modeling the underlying micro-architecture components such as caches, pipeline and branch predictor.

Conventionally WCET analysis assumes an *uninterrupted* sequential execution of the program being analyzed. However, in reality many real time systems run in a *multi-tasking* environment, in which different programs (or tasks) are scheduled to run concurrently. For system with *pre-emptive priority scheduling*, a task can be interrupted by another task which has a higher priority to run. Therefore, it is impractical to assume that a program is always allowed to run uninterrupted. There could be an additional delay imposed on a running task due to interruption by another task. This delay is caused by changes to the micro-architectural states of the system by the interrupting task. For example, the interrupting task may replace some cache blocks in the caches. Caches are small but fast memories, used to bridge the performance gap between a processor and the main memory. Set-associative caches are divided into fixed-size *sets*. Each set can hold up to A different blocks from the main memory (A is called the cache associativity level). When a block needs to be added to a cache set that is already full, a *replacement policy* is used to determine the evicted block. Caches are at least an order of magnitude faster than the main memory, thus the changes to cache content due to an interrupting task is a major factor in the delay caused by the interruption. This delay is known in the literature as *Cache Related Preemption Delay* (CRPD). CRPD analysis techniques have been proposed to put a bound on CRPD.

In this paper we will concentrate on obtaining the bound on CRPD for set-associative caches with *First-In First-Out* (FIFO) replacement policy. Traditionally, CRPD analysis bounds the additional cache misses introduced by preemptions through the following two factors: (i) number of cache blocks introduced by the preempting task (*i.e.* Evicting Cache Block or ECB), and (ii) number of cache blocks that may be reused by the preempted task after preemption (*i.e.* Useful Cache Block or UCB). Existing work on CRPD analysis focus on caches with Least Recently Used (LRU) replacement policy. A study [4] shows that these factors cannot safely bound CRPD cost for FIFO caches, due to the presence of *unbounded timing effect* for FIFO caches. A single evicted memory block from cache due to preemption can cause unbounded number of additional cache misses after the preemption. Thus, the concepts of ECB and UCB cannot be used to safely bound CRPD cost for FIFO caches. These concepts do not work for FIFO caches because they try to bound the number of additional concrete cache misses. Instead, our CRPD analysis relies on information from the underlying cache analysis for computing WCET. We utilize *static phase detection* [7] technique to obtain the set of memory blocks that are categorized as *always hit* in the cache assuming no preemption. We solve the maximum number of additional cache misses introduced by the *always hit* blocks as an *integer linear programming* (ILP)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

LCDES '14, June 12–13, 2014, Edinburgh, United Kingdom.
Copyright © 2014 ACM 978-1-4503-2877-7/14/06...\$15.00.
<http://dx.doi.org/10.1145/2597809.2597814>

problem, given a bound on the total number of preemptions. Our analysis is safe as we conservatively introduce miss penalty for all memory blocks not classified as *always hit*.

We also studied the possible effects of *timing anomalies* on CRPD analysis in general. Existing CRPD analysis techniques assume an underlying micro-architecture model that is free from timing anomalies. However, such assumption may render the CRPD analysis unsound, as the worst-case delay is underestimated. In this work, we studied three types of timing anomalies exhibited by out-of-order processors or FIFO caches [13]. We take these timing anomalies into consideration in our CRPD analysis and show that our analysis is safe in the presence of these timing anomalies. To the best of our knowledge, ours is the first work on CRPD analysis that explicitly handles architectures exhibiting timing anomalies.

We implemented our FIFO CRPD analysis in Chronos [11], an open source WCET analysis tool. We have tested our analysis method on several subject programs, and compare the results with a state-of-the-art approach. The state-of-the-art approach to handle CRPD analysis with FIFO instruction caches is by computing the CRPD assuming LRU replacement policy, and bound the value for FIFO policy using the concept of *relative competitiveness* [16]. Our experimental results show that, when compared to current CRPD analysis technique for FIFO caches, we are able to compute significantly tighter bound on CRPD cost for all subject programs.

2. Background

FIFO caches CPU caches are generally small in size and they can be filled up with memory blocks rapidly and frequently. There needs to be a *replacement policy* to decide the exact memory blocks that should be discarded (*i.e.* replaced by newly inserted memory blocks) when a cache set is full. *First-In First-Out* or FIFO is one such replacement policies. FIFO caches are used in CPU architectures such as ARM9 and ARM11.

In general, FIFO caches always replace a memory block that has been in the cache set the longest, as shown in Figure 2 (this figure assumes that all represented blocks are mapped to the same set).

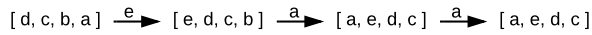


Figure 2: Effect of a memory sequence *eea* on a cache set for FIFO policy. First memory access *e* causes a cache miss, and block *a* is evicted since it is the first memory block inserted in the cache set. Similarly for the second memory access *a*, block *b* is replaced. There is no change to the cache state when there is a cache hit (as shown by the third memory access).

Timing anomalies In general, timing anomaly describes a counter-intuitive observation, in which a local worst case timing behaviour does not entail a global worst case timing behaviour. Certain micro-architectural features exhibit timing anomalies. For example, a cache hit can cause a higher execution time than a cache miss in a processor with out-of-order execution [13]. This may affect the soundness of WCET analysis that models the underlying hardware.

Work in [12] and [5] shows that FIFO caches exhibit timing anomaly due to *unbounded timing effect*. With FIFO caches, if some memory block access in a program results in a cache miss, then the cache miss outcome will not necessarily lead to the maximal cache miss count for the overall program. An example is shown in Figure 1. Let us consider that we have a 2-way FIFO instruction cache. Here, the first access to *c* is a cache miss in the general case (*i.e.* the initial cache state does not contain any block from the example). In this case, the overall cache miss count is 5. However, if we alter the initial cache state to make sure that the first access to *c* is a cache hit, the overall cache miss count is increased to 6.

3. Methodology

In this section, we present our method for CRPD computation in the presence of FIFO caches. Our method seeks to return a bound on the number of additional cache misses for a task T_0 , which is preempted by another task up to *MPC* (Maximum Preemption Count) times. We assume that the preempting task shares cache sets with T_0 . The CRPD of T_0 is simply the maximum number of additional cache misses (caused by preemption) multiplied by the cache miss penalty. Our method uses concepts from *static phase detection* [7], which is an approach to statically categorize FIFO instruction cache accesses without considering preemption. We shall briefly describe static phase detection, then we shall explain how our analysis can use the information produced by static phase detection to compute CRPD.

3.1 Static phase detection

Static phase detection is a method to statically categorize each instruction as *always hit*, *always miss*, or *not classified*, in the presence of an instruction cache with FIFO replacement policy. The analysis works by detecting *phases* in instruction cache accesses.

Let B be a set of memory blocks that are mapped to the same cache set, and let $|B|$ be the number of pairwise different blocks in B . A *B-phase* is an access sequence such that (i) $|B| \leq A$ (A being the cache associativity) and (ii) all blocks in B and only the blocks from B are accessed (*i.e.* a block can be accessed more than once in the phase, as long as all blocks from B are accessed). After exactly $|B|$ *B-phases*, it is guaranteed that all the blocks from B are loaded in the cache. Therefore, subsequent accesses to blocks in B are guaranteed to be cache hits.

This allows us to categorize some instructions in a program as *always hit*, if those instructions will always be cache hits during runtime. Let I be an instruction in the program. For all paths in the control flow graph (CFG) leading to I , if the cache accesses immediately preceding I can be partitioned into $|B|$ *B-phases* where B contains I 's cache block, then I is classified as *always hit*. Conversely, if there exists a path leading to I that cannot be partitioned into $|B|$ *B-phases* where B contains I 's cache block, then execution along this path can lead to a cache miss when I 's cache block is accessed. In this case, I is categorized as *not classified*.

3.2 Phase content

Static phase detection does not consider the effect of task preemption. Any instruction that is classified as *always hit* by static phase detection may cause cache misses in case of preemption. The goal of our analysis is to bound the number of cache misses occurring in *always hit* instructions due to preemption. We do not take into account cache misses occurring in instructions classified as *always miss* or *not classified* under the assumption that those cache misses will already be taken into account in the computation of WCET.

Since an *always hit* instruction has, on all incoming paths, an access sequence that can be partitioned into $|B|$ *B-phases*, cache misses can occur if a preemption disrupt these access sequences. Thus, the first step of our analysis is to compute the *phase content* for all *always hit* instructions.

DEFINITION 3.1. (Phase content). *The phase content of an instruction I , denoted as $PC(I)$, is the minimal set of instructions such that for any path p from the program entry point to I , there exists a sub-path p' leading to I that contains only instructions from $PC(I)$, and whose access sequence can be partitioned into $|B|$ *B-phases* where B contains I 's cache block.*

If a preemption occurs at any instruction in $PC(I)$, the preemption may disrupt the access sequence leading to instruction I in a way that causes cache misses for I . An example is shown in Figure 3. We consider a FIFO cache with an associativity of 2. In the

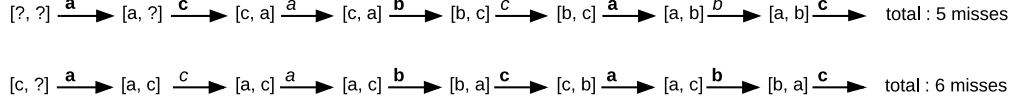


Figure 1: FIFO cache timing anomaly example. For the memory access sequence at the bottom, the entry cache state causes a cache hit for the first access to c , but this causes the overall cache miss count to be greater than the memory access sequence on top.

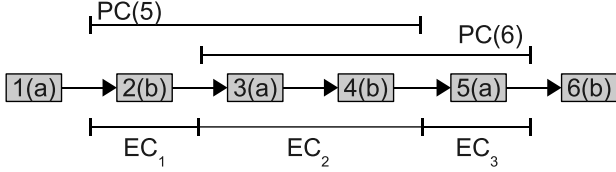


Figure 3: Phase contents and equivalence classes

figure, the nodes of the CFG are l -blocks. An l -block is a maximal sequence of instructions such that each instruction is in the same basic block, and mapped to the same cache block. As with basic blocks, it is possible to connect l -blocks using edges to represent the control flow. Each l -block in the figure is labeled with $x(y)$, where x refers to the node number and y refers to the mapped cache block set. l -blocks 5 and 6 are classified as *always hit*, because they are executed after two phases containing cache blocks $[a, b]$. The figure shows the phase contents for l -blocks 5 and 6. If a preemption occurs in the code delimited by $PC(5)$ (resp. $PC(6)$), an additional cache miss may occur in l -block 5 (resp. l -block 6).

3.3 Equivalence classes

Let us define PC^{-1} as follows:

$$I' \in PC^{-1}(I) \Leftrightarrow I \in PC(I')$$

In other words, if a preemption occurs at instruction I , it may cause additional misses for instructions in the set $PC^{-1}(I)$. Let us define the equivalence relation \sim , as follows:

$$I \sim I' \Leftrightarrow PC^{-1}(I) = PC^{-1}(I')$$

Let SI be the set of all instructions in a program, and let $(SI/\sim) = \{EC_1, \dots, EC_N\}$ be the set of N equivalence classes defined by the equivalence relation \sim . We also define $PC^{-1}(EC_k)$ as the result of applying $PC^{-1}()$ function to any instruction in EC_k .

To bound the maximum number of additional cache misses due to preemption, we first need to bound, for each equivalence class $EC_k \in (SI/\sim)$, the number of preemptions occurring inside it (i.e. occurring at an instruction in that equivalence class). We only need to bound the preemption count for each equivalence class, and not for each individual instruction. This is based on the observation that if two instructions I_1 and I_2 are in the same equivalence class EC_1 , then $PC^{-1}(EC_1) = PC^{-1}(I_1) = PC^{-1}(I_2)$. This means that a preemption occurring at either I_1 or I_2 will have the same effect on the additional cache misses. Then, the bound for additional cache misses of each *always hit* instruction can be expressed as a function of the preemption count for each equivalence class.

Figure 3 shows the partitioning of the CFG according to the equivalence classes. In this example, the equivalence classes $\{EC_1, EC_2, EC_3\}$ are computed based on the phase contents for l -blocks 5 and 6 only. A preemption occurring in EC_1 (resp. EC_3) may add one cache miss for l -block 5 (resp. l -block 6), and a preemption occurring in EC_2 may add one cache miss for both l -blocks 5 and 6.

3.4 ILP formulation

We bound the additional cache miss count using an ILP system. We make use of the ILP system computed by the main WCET analysis (containing, for example, the structural constraints derived from the program CFG), and we add additional CRPD-related constraints to it. We first describe how to bound the preemption count for each equivalence class, and then we show how to express the bounds on the number of cache misses for each *always hit* instruction.

For each equivalence class $EC_k \in (SI/\sim)$, an ILP variable ec_k is created. This variable represents the number of times a preemption occurs at an instruction contained in the equivalence class EC_k . The ec_k variables are bounded as follows:

$$\sum_{k \in [1;n]} ec_k \leq MPC$$

MPC is the maximum preemption count of the preempting task. MPC is a parameter of the analysis, and is considered a constant from the point of view of the ILP system.

For each *always hit* instruction I_k , an ILP variable xpm_k is created to represent the number of cache misses (due to preemption) for I_k . xpm_k is created only if the cache set containing I_k is also used by the preempting task (otherwise the preempting task cannot evict the cache block containing I_k). Instruction I_k may cause cache miss only if a preemption occurs at an instruction from an equivalence class EC_k such that $I_k \in PC^{-1}(EC_k)$. Thus, the number of cache misses for I_k can be bounded using the bounds on the maximum preemption count in each equivalence class:

$$xpm_k \leq \sum_{\forall j/I_k \in PC^{-1}(EC_j)} ec_j$$

Each xpm_k variable must also be bounded by the execution count of the basic block containing I_k . This can be achieved by the constraint $xpm_k \leq x_k$, where x_k represents the execution count of the basic block. We assume that x_k already exists in the ILP system computed by the main WCET analysis.

Finally, the maximum number of additional cache misses due to preemption is found by solving the following objective function:

$$\text{maximize } \sum_{\forall k/I_k \in SI} xpm_k$$

We will show a simple example using the CFG in Figure 3. The additional cache misses due to preemption for l -blocks 5 and 6 are represented by ILP variables xpm_5 and xpm_6 respectively. Equivalence classes ec_1 and ec_2 contribute additional cache misses for l -block 5, while equivalence classes ec_2 and ec_3 contribute additional cache misses for l -block 6. The total preemption count in all equivalence classes is bounded by the maximum preemption count, MPC (set to 1 in this case). Thus, we have the following ILP constraints :

$$\begin{aligned}
xpm_5 &\leq ec_1 + ec_2 \\
xpm_6 &\leq ec_2 + ec_3 \\
ec_1 + ec_2 + ec_3 &\leq 1
\end{aligned}$$

Maximizing the number of additional cache misses ($xpm_5 + xpm_6$) with an ILP solver yields the following result:

$$ec_1 = 0, ec_2 = 1, ec_3 = 0, xpm_5 = 1, xpm_6 = 1$$

It shows that the preemption should occur in equivalence class EC_2 to cause maximum additional cache miss count (two additional misses; one for l -block 5 and one for l -block 6).

3.5 Computing the WCRT

In this section, we will show how to get the Worst-Case Response Time (WCRT) of a task, based on our CRPD computation method. We assume a *fixed-priority preemptive scheduling* of a set of periodic tasks with possibility of nested preemptions. The ILP system constructed in section 3.4 can be used to compute the CRPD between two tasks for any number of preemptions, based on the MPC parameter. Let us define:

$$crpd(T_i, T_j, n) = \text{admiss}(T_i, T_j, n) \times \text{penalty}$$

$crpd(T_i, T_j, n)$ is the estimated CRPD when task T_j preempts task T_i for n preemptions (the n parameter is optional, and defaults to 1 if omitted). $\text{admiss}(T_i, T_j, n)$ is the bound on the additional cache misses for n preemptions of T_i by T_j (as computed using the ILP system, while setting MPC to n). penalty bounds the increase in execution time when a cache miss occurs.

Traditionally, CRPD is computed for one preemption, and then this result is multiplied with the preemption count in the WCRT computation formula. However, that may be pessimistic, since although we have $n \times crpd(T_i, T_j) \geq crpd(T_i, T_j, n)$, in the general case we do not have $n \times crpd(T_i, T_j) = crpd(T_i, T_j, n)$. This is because there is a finite number of program points where a preemption can cause a large number of cache misses. Once these program points are taken by preemptions, additional preemptions cannot contribute as much to the CRPD.

In this paper, we discuss on the WCRT computation of a task using two approaches - *Fixed CRPD approach* and *Iterative approach*. In the former approach, we will have to run the CRPD computation only once (for one preemption). In the latter approach, we have to iteratively re-compute the CRPD each time the maximum preemption count is updated, during the fixed-point calculation. Each approach has its advantages and drawbacks, as discussed in the next subsections.

3.5.1 Fixed CRPD approach

In this approach, we compute the CRPD for one preemption, and use the result as it is done traditionally to compute the WCRT. The following equation computes the WCRT of task T_i , $WCRT_{T_i}$ until a fixed-point is reached:

$$WCRT_{T_i} = WCET_{T_i} + \sum_{\forall j \in hp(i)} \left\lceil \frac{WCRT_{T_i}}{PERIOD_{T_j}} \right\rceil (WCET_{T_j} + \gamma_{T_i, T_j}) \quad (1)$$

In Equation (1), $WCET_{T_i}$ is the computed WCET of task T_i . $\left\lceil \frac{WCRT_{T_i}}{PERIOD_{T_j}} \right\rceil$ bounds the number of preemptions by task T_j on task T_i , where $PERIOD_{T_j}$ is the period of task T_j . $hp(i)$ contains the set of tasks with higher priority than task T_i . Without considering nested preemption, we can simply define $\gamma_{T_i, T_j} = crpd(T_i, T_j)$. If nested preemptions are possible, then this is incorrect, because if task T_j preempts a task T_k which in turn preempts task T_i , then the CRPD of T_j preempting T_k is not taken into account. To solve this problem, we define γ function as such :

$$\gamma_{T_i, T_j} = crpd(T_i, T_j) + \sum_{\forall k/k \in hp(i) \wedge j \in hp(k)} crpd(T_k, T_j)$$

This approach has the advantage of being fast (we have to perform the CRPD computation only once), and easily adaptable to existing WCRT computation formula. Its main drawback is the introduction of pessimism.

3.5.2 Iterative approach

It is possible to take into account the real preemption count at each step of the fixed-point WCRT computation. With an iterative approach, the WCRT of each task is computed in order of decreasing priority. For each task, the maximum preemption count (MPC) by higher-priority tasks is computed, and is fed to the ILP system. This (intermediate) CRPD enables us to refine the maximum preemption count, and this process is repeated until a fixed-point is reached. Thus, this method gives a tighter result.

The main drawback of this approach is the analysis time, because we have to solve an ILP system at each step of the fixed-point WCRT computation. To mitigate this issue while still maintaining a relatively tight CRPD bound, it is possible to use a *hybrid approach*. The main idea is to use the iterative approach until some arbitrary time limit is reached. This will produce a temporary (underestimated) CRPD, since the analysis is not finished. Then, from this temporary value, we can compute the final (safe) CRPD bound in a non-costly way.

To describe this last step, we make the following observation: an increase in the maximum preemption count, $MPC_{increase}$, leads to an increase in the CRPD, $CRPD_{increase}$. As the maximum preemption counts get higher, the ratio $\frac{CRPD_{increase}}{MPC_{increase}}$ decreases (or remains equal). Because of this, the following property holds for all M (where M is the preemption count):

$$crpd(T_0, T_1, M + 1) - crpd(T_0, T_1, M) \leq crpd(T_0, T_1, M) - crpd(T_0, T_1, M - 1)$$

Let M_1 be the maximum preemption count obtained when the time limit is reached. Based on the observation made above, for any M_2 greater than M_1 , the following property holds :

$$crpd(T_0, T_1, M_2) - crpd(T_0, T_1, M_1) \leq (crpd(T_0, T_1, M_1 + 1) - crpd(T_0, T_1, M_1)) \times (M_2 - M_1)$$

Therefore, once the iterative analysis has reached the time limit, it is possible to bound $crpd(T_0, T_1, M_2)$ for any M_2 greater than M_1 , by the following value:

$$crpd(T_0, T_1, M_1) + (crpd(T_0, T_1, M_1 + 1) - crpd(T_0, T_1, M_1)) \times (M_2 - M_1)$$

This allows us to compute the final WCRT by fixed-point iteration without having to solve a costly ILP system at each step. The computed WCRT will be tighter than the one computed with fixed CRPD approach, but less tight than the one computed with fully iterative approach. This hybrid approach is described in Algorithm 1.

3.6 Scalability

The ILP constraint generation described in section 3.4 produces a large number of constraints and variables, which causes a large ILP computation time. Ideally we want to decrease the number of ILP constraints and variables. The following observation can be made: for any l -block (recall that an l -block is a maximal sequence of instructions such that each instruction is in the same basic block, and in the same cache block) containing the sequence of instructions (I_1, I_2, \dots, I_n) , the effect on cache misses will be the same if a preemption occurs in $[I_2; I_n]$, independently of the exact instruction. The reason for this is illustrated in Figure 4. In l -block 2, the instructions I_2 through I_5 are guaranteed to be cache hits, since the cache block is loaded in I_1 . If a preemption occurs at any of these instructions and evicts the cache block, it will result in one

Algorithm 1 Iterative hybrid computation

```

1:  $TL \leftarrow$  task list ordered by decreasing priority
2: for  $i \in TL$  do
3:    $WCRT_i \leftarrow WCET_i$ 
4:    $change \leftarrow true$ 
5:    $flag \leftarrow false$ 
6:   while  $change \wedge (WCRT_i \leq deadline_i)$  do
7:     for  $j \in hp(i)$  do
8:        $MPC_{i,j} \leftarrow \lceil \frac{WCRT_i}{PERIOD_j} \rceil$ 
9:       if  $flag$  then
10:         $\gamma_{i,j} \leftarrow \gamma_{0_{i,j}} + R \times (MPC_{i,j} - MPC_{0_{i,j}})$ 
11:       else
12:         $\gamma_{i,j} \leftarrow crpd(i, j, MPC_{i,j})$ 
13:       end if
14:     end for
15:     if  $(time\ is\ up) \wedge \neg flag$  then
16:       for  $j \in hp(i)$  do
17:         $\gamma_{0_{i,j}} \leftarrow \gamma_{i,j}$ 
18:         $MPC_{0_{i,j}} \leftarrow MPC_{i,j}$ 
19:         $R \leftarrow crpd(i, j, MPC_{i,j} + 1) - \gamma_{0_{i,j}}$ 
20:       end for
21:        $flag \leftarrow true$ 
22:     end if
23:     for  $j \in hp(i)$  do
24:        $S_\gamma \leftarrow \sum_{\forall k/k \in hp(i) \wedge j \in hp(k)} \gamma_{k,j}$ 
25:        $\gamma'_{i,j} \leftarrow MPC_{i,j} \times WCET_j + S_\gamma$ 
26:        $WCRT'_i \leftarrow WCET_i + \sum_{\forall j \in hp(i)} \gamma'_{i,j}$ 
27:     end for
28:      $change \leftarrow (WCRT_i = WCRT'_i)$ 
29:      $WCRT_i \leftarrow WCRT'_i$ 
30:   end while
31: end for

```

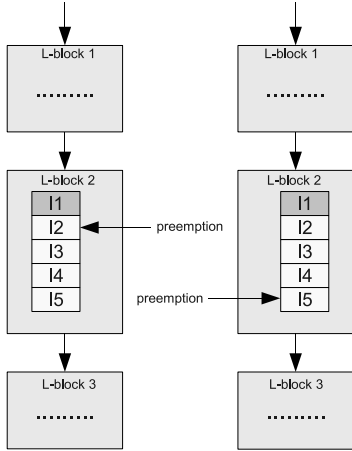


Figure 4: Preemption point equivalence

preemption-related miss, regardless of the specific instruction that is executing when the preemption occurs.

Based on this observation, it is possible to generate, for each l-block L_k , only two xpm variables: variable xpm_1 for the first instruction I_1 of the l-block, and variable xpm_2 for the second instruction I_2 . Both variables are constrained by the expression $xpm_I \leq \sum_{\forall k/I \in PC^{-1}(EC_k)} ec_k$ as described in Section 3.4. Ad-

ditionally, variable xpm_1 is bounded by $xpm_1 \leq x_k$ (where x_k represents the execution count of l-block L_k , which is equal to the execution count of the basic block containing L_k). On the other hand, variable xpm_2 is bounded by $xpm_2 \leq x_k \times |instr(k) - 1|$, where $instr(k)$ is the set of instructions in the l-block L_k . This constraint represents the fact that variable xpm_2 counts additional cache misses not only for instruction I_2 , but also for all instructions in the l-block except for I_1 . This amounts to $|instr(k) - 1|$ instructions. As such, the maximum additional misses for these instructions is $|instr(k) - 1|$ each time l-block L_k is executed.

3.7 Handling timing anomalies

In this section, we will discuss three timing anomalies related to caches that may affect the safety of CRPD analysis in general. We shall refer to these timing anomalies as *Anomaly 1*, *Anomaly 2* and *Anomaly 3*. *Anomaly 1* and *Anomaly 2*, as mentioned in [13], may occur in the presence of an out-of-order processor. Mainly, a cache hit or miss may cause unexpected timing delay in the execution of instructions in the pipeline. *Anomaly 3* came from work in [3], which show that FIFO caches exhibit *domino effect*, in which a change in the cache state could potentially cause an unbounded timing delay. Thus, if an additional cache hit or miss is introduced due to preemption, a safe CRPD analysis should consider these unexpected timing delays. We also propose some solutions to handle the identified anomalies. It should be noted that the solutions are in general applicable to any CRPD analysis, with certain assumptions on the WCET analysis technique that is being used. We first state the necessary assumptions.

Assumptions about the WCET analysis Let us assume that a task T is defined as its control flow graph, $G_T = (B, E)$, where $B = \{LB_1, \dots, LB_n\}$ is the set of l-blocks in task T , and E is the set of edges representing control flow between two l-blocks. Task T is then represented by an ILP system having a variable c_n (execution count) for each l-block $LB_n \in B$. For each l-block, two variables exists: th_n and tm_n , representing respectively the maximum execution time of that l-block in case of cache hit or cache miss. The ILP variables ch_n and cm_n represent the number of cache hits and misses, respectively, for l-block n . For each l-block n , an ILP constraint $ch_n + cm_n = c_n$ is generated. The $WCET$ and MC functions, to compute respectively the WCET of the task, and its total cache miss count (preemption-related or not), are defined as follows:

DEFINITION 3.2. (WCET). The result $WCET(T)$ is defined as the maximized objective function $\sum_{\forall k \leq n} th_k \times ch_k + tm_k \times cm_k$, for the ILP system generated for task T .

DEFINITION 3.3. (MC). The miss count, noted $MC(T)$ is defined as the maximized objective function $\sum_{\forall k \leq n} cm_k$, for the ILP system generated for task T .

For each of the three timing anomalies, we first give an example that illustrates the anomalous behaviour, then we proceed to propose a solution to handle the anomaly.

Anomaly 1: Miss penalties can be higher than expected The cache analysis used in WCET computation usually take cache misses into account by adding a fixed miss penalty for each miss. Simply making this miss penalty equal to the memory latency behind the cache can lead to WCET underestimation. Lundqvist et al. proved in [13] that in some cases, replacing a cache hit by a cache miss can increase the execution time by an amount greater than the memory latency.

Figure 5 shows an example, assuming a processor similar to the one used in the example in Figure 6. The cache miss while fetching instruction A causes instruction B to be scheduled later, after

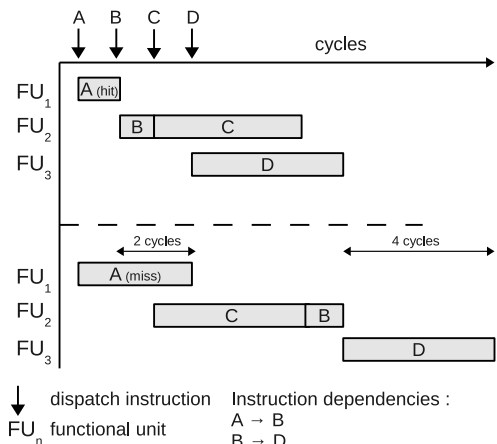


Figure 5: Miss penalty greater than memory latency

instruction C . Since instruction D depends on B , the execution is delayed by 4 cycles (compared to the cache hit scenario), while the memory latency is only 2 cycles.

In order to avoid WCET underestimation due to this anomaly, we need to compute correctly the miss penalty for each potential cache miss. A sound way to do that is to compute the th_n and tm_n values for each l -block using a pipeline analysis approach (e.g. the *execution graph* method [10]). Then, the penalty is computed as shown in Definition 3.4 to avoid the problem described above. This penalty should be a sound over-approximation, since computing the exact th_n and tm_n values is generally infeasible.

DEFINITION 3.4. (*penalty*). The miss penalty, $penalty(T)$ is defined for a specific task T , as:

$$penalty(T) \geq \max(tm_n - th_n | n \in [1; N])$$

Anomaly 2: Cache hits can result in worst case timing The majority of cache-related analysis for WCET computation assume that, if the hit/miss classification of a memory access cannot be determined, the case leading to the WCET is the cache miss. Unfortunately, it has been shown by Lundqvist et al. [13] that it is not always true. In some cases, specifically in the presence of processors with out-of-order execution, replacing a cache miss by a cache hit can increase the execution time of an instruction sequence.

An example is shown in Figure 6, assuming an out-of-order execution processor, and an instruction sequence using three functional units. In this example, a cache miss while fetching the instruction A (shown in the lower half of the figure) causes instruction C to be scheduled earlier. Since instruction D depends on C , this causes the execution of the instruction sequence to finish one cycle earlier compared to the cache hit case. This effect can lead to unsafe CRPD analysis. For example, let us consider a preempted task, containing a l -block LB_k classified as *always miss*. A preemption can load the cache block of LB_k into the cache, causing cache hit for LB_k . Traditionally, CRPD analysis attempt to bound additional cache misses, but does not consider any additional cache hits due to preemption. If the execution time for LB_k is greater in case of cache hits, this effect will not be captured by the CRPD analysis, potentially leading to an unsafe WCET.

There is a trivial way to prevent WCET underestimation in this case, and another, more sophisticated way. The trivial way is to consider all *always miss* as *not classified* in the WCET analysis prior to the CRPD computation. This allows us to modify (without

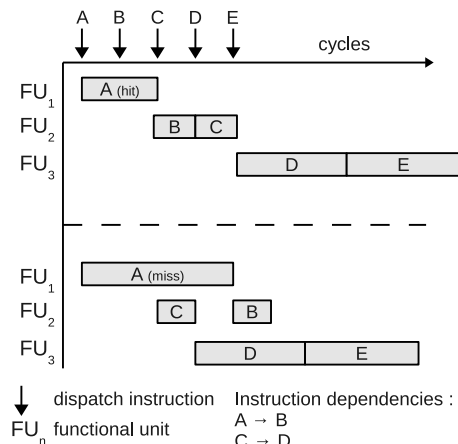


Figure 6: Hit resulting in longer time

any impact on the computed WCET) tm_n for each l -block as such:

$$tm_n \leftarrow \max(tm_n, th_n) \quad (2)$$

By doing this, we guarantee that tm_n is the worst possible time for the l -block n (including the scenario presented in Figure 6).

The other way is to include lost time due to additional cache hits in the CRPD computation. To do that, we define the *hit penalty* as $\max(0, tm_n - th_n)$, and compute the maximum additional cache hits in the same way we computed the bound on the additional cache misses. This is done by modifying our CRPD analysis so that the phase content, $PC(I)$ is computed for each *always miss* instruction I (instead of *always hit*), and if a preemption occurs at an instruction in $PC(I)$, then I can cause cache hits. We did not implement the latter method as it makes a difference only for l -blocks showing a greater execution time for cache hits. That is quite rare in our observation, so the increase in precision is negligible.

Anomaly 3: Impact on WCET may not be bounded (Domino effect) In LRU caches, the effect of a change in cache state is *bounded*, because after any sequence of (at least) A different blocks mapping to the same cache set (on a A -way cache), the whole set is filled with blocks belonging to this sequence. As shown by C. Berg et al. [3], this is not true with FIFO caches. With FIFO caches, a cache state alteration can have unbounded repercussions in subsequent accesses.

Figure 7 shows an example with a 2-way set-associative FIFO cache. The edges are labeled with the concrete cache states at that program point, with the most recently loaded block located on the left. For the CFG on the left side of the figure, for even iteration numbers, accesses to a and c are hits; while for odd iteration numbers, accesses to b are hits, so there is 1.5 cache misses on average per loop iteration. For the CFG on the right, an access to block x is added. Each cache block access in the loop is now a cache miss. The additional access to block x adds, on average, 1.5 cache misses per loop iteration, and this effect is unbounded (except, of course, by the maximum loop iteration count).

This problem does not occur in our CRPD analysis, since the cache blocks involved in the domino effect would be categorized as *not classified* by the static phase detection step. However, this observation is not sufficient to ensure that this effect can be safely ignored in the general case. To handle the problem in the general case, we make the following observation: when dealing with WCET computation, we are assuming that no infinite path exists in the program. This is guaranteed by additional flow constraints, such as loop bounds. When computing the effect of a preemption on the

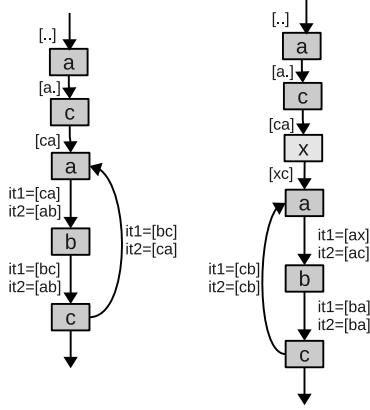


Figure 7: FIFO cache domino effect

miss count, we must take these constraints into account, in order to get a bounded result. The effect of a preemption on the miss count is captured by $addmiss$ (defined in Definition 3.7), which allows us to ensure a bounded effect.

CRPD soundness We proceed to present a proof to guarantee the safety of our CRPD analysis. We have defined $WCET(T)$ in Definition 3.2. Let us further introduce $WCETPR(T_0, T_1, p)$ as the WCET of task T_0 subject to cache interference due to preemption by task T_1 at program point p , as defined in Definition 3.5. Let us also define $addmiss(T_0, T_1)$ in Definition 3.7, as the bound on the additional cache misses due to preemption of T_0 by T_1 . Our concept of program path in a CFG is defined in Definition 3.6.

DEFINITION 3.5. (WCETPR and MCPR). $WCETPR(T_0, T_1, p)$ is the computed WCET of task T_0 , such that the cache classification analysis is performed on the CFG $G_{T_0, T_1, p}$. This CFG results from the merging of the CFGs of T_0 (preempted task) and T_1 (preempting task), connected with call/return edges at program point p . Only the cache classification analysis is computed on $G_{T_0, T_1, p}$. The main WCET analysis is performed on T_0 alone. The function $MCPR(T_0, T_1, p)$ is defined in a similar way for the miss count (recall that $MC(T)$ is defined in Definition 3.3).

DEFINITION 3.6. (path). A program path in a task T is defined as a function associating an execution count to each node in the CFG of T . The functions $WCET$, $WCETPR$, MC , and $MCPR$ are enhanced to accept a path as the last (optional) parameter. The effect of this optional path parameter is the creation of ILP constraints $\forall n, path(LB_n) = c_n$. We also note $path \in T$ if $path$ is a valid path for the task T .

DEFINITION 3.7. (addmiss). $addmiss(T_0, T_1)$ is defined as:

$$\max(MCPR(T_0, T_1, p, path) - MC(T_0, path)) | path \in T_0$$

In other words, it is the maximum additional cache miss counted by the ILP solution, for any possible path through task T_0 , and for any possible program point for preemption by task T_1 . This does not represent the maximum number of additional concrete misses, but instead it is the difference of miss count as determined by the analysis. Since $addmiss()$ is computed from the result of the ILP system, it takes into account the various flow control constraints that are needed to ensure that no infinite path exists within the program. Therefore, an implementation of $addmiss()$ compatible with this definition will always yield a finite and safe bound on the number of additional cache misses, even in the presence of unbounded domino effect.

The following lemma states that the CRPD is sound if a preemption does not change the worst-case path for the preempted task.

LEMMA 3.1. Let T_0 be a preempted task, and T_1 a preempting task. Then $\forall path \in T_0$:

$$\forall p, WCETPR(T_0, T_1, p, path) \leq WCET(T_0, path) + addmiss(T_0, T_1) \times penalty(T_0)$$

PROOF. Let $\sum_{\forall k \leq n} th_k \times ch'_k + tm_k \times cm'_k$ be the objective function for computing $WCETPR(T_0, T_1, p, path)$, and let $\sum_{\forall k \leq n} th_k \times ch_k + tm_k \times cm_k$ be the objective function for computing $WCET(T_0, path)$. ch'_k and cm'_k represent the number of cache hits and misses, respectively, for a l-block k in case of preemption. Therefore, $WCETPR(T_0, T_1, p, path) - WCET(T_0, path)$ is equal to the difference of the maximized objective functions:

$$\sum_{\forall k \leq n} th_k \times (ch'_k - ch_k) + tm_k \times (cm'_k - cm_k)$$

Since the worst-case path is unchanged by the preemption, we have $\forall k, c'_k = c_k$, and so the above can be rewritten as:

$$\sum_{\forall k \leq n} (cm'_k - cm_k) \times (tm_k - th_k)$$

We need to make the following assumption :

$$\forall n, th_n \leq tm_n \quad (3)$$

Therefore, we have :

$$\sum_{\forall k \leq n} (cm'_k - cm_k) \times (tm_k - th_k) \leq \sum_{\forall k \leq n} (cm'_k - cm_k) \times penalty(T_0)$$

$$\sum_{\forall k \leq n} (cm'_k - cm_k) \times penalty(T_0) \leq addmiss(T_0, T_1) \times penalty(T_0)$$

Recall that $penalty(T_0)$ is defined in Definition 3.4.

The following theorem states that the CRPD is sound even if the preemption changes the worst-case path.

THEOREM 3.2. Let T_0 be a preempted task, and T_1 a preempting task. Then:

$$\forall p, WCETPR(T_0, T_1, p) \leq WCET(T_0) + addmiss(T_0, T_1) \times penalty(T_0)$$

PROOF. Let p be any program point in T_0 , and let $wcpath$ be the worst-case path for $WCETPR(T_0, T_1, p)$. Then, the following properties are true:

1. $WCET(T_0, wcpath) \leq WCET(T_0)$, since adding a path constraints to an ILP system can never increase the result.
2. $\forall p, WCETPR(T_0, T_1, p) \leq WCETPR(T_0, T_1, p, wcpath)$, since $wcpath$ is the path with the highest WCET
3. $\forall p, WCETPR(T_0, T_1, p, wcpath) \leq WCET(T_0, wcpath) + addmiss(T_0, T_1) \times penalty(T_0)$, from Lemma 3.1

Therefore from (1) and (2), we have $\forall p, WCETPR(T_0, T_1, p) \leq WCET(T_0) + addmiss(T_0, T_1) \times penalty(T_0)$.

This result enables us to guarantee that a CRPD analysis will be safe even in the presence of timing anomalies, provided that some conditions are respected. Indeed, it is not tied to any specific CRPD computation method, and can be applied to any existing CRPD analysis, as long as these propositions are true:

1. The CRPD computed for the preemption of the task T_0 by T_1 is a bound on $addmiss(T_0, T_1) \times penalty(T_0)$.

Task	Size	Task	Size
senddataautopilot	300	altitudecontrol	1496
checkfailsafe	1116	climbcontrol	6104
checkmega128value	648	stabilisation	3600
testppm	7876	radiocontrol	3600
Fly-by-wire		Autopilot	

Table 1: Code size (in bytes) of tasks in Papabench *fly-by-wire* and *autopilot* modules

- penalty* and *admiss* are compatible with the definitions found in Definition 3.4 and Definition 3.7.
- Assumption (3) must be true. This is easily accomplished by disabling the *may* cache analysis, and altering tm_n values as described in Equation 2.

Timing anomalies with our FIFO CRPD analysis In our approach, we do not do the *may* cache analysis, and our *penalty* is computed as defined in Definition 3.4. Furthermore, since our FIFO CRPD analysis uses an ILP system to bound the number of additional cache misses due to preemption for any possible path and preemption point, this result bounds $admiss(T_0, T_1)$, therefore the proof described above applies to our analysis as well.

4. Experimental results

In this section, we give experimental results for our method by analyzing a set of representative benchmarks. We implemented our CRPD analysis framework on top of Chronos [11], an open source WCET analysis tool. We extended Chronos to support ARM architecture and all of our chosen benchmarks are compiled as ARM binaries. In our analysis, we model a single ARM926EJ-S processor core with a level 1 instruction cache that supports FIFO replacement policy. We run our analysis for different instruction cache configurations (associativity level, number of sets, and cache block size). We use three types of benchmarks: *PapaBench* [15], *Mälardalen* benchmarks [8], and a robot control application [6].

For each benchmark, we compute the bound on the additional cache misses due to a single preemption for each *task set* consisting of two tasks: a low priority task, and a high priority task. We assume a *fixed-priority preemptive scheduling* of tasks. The list of the analyzed task sets is defined in Table 3. Both the tasks in a task set will run in the same processor core. We compute the preemption cost (in term of additional cache misses) with our method, and compare it to the preemption cost computed with the *relative competitiveness* method [16]. We also plot the average number of additional cache misses against the number of preemptions, to attempt to determine the advantage of using the iterative *CRPD* computation, as opposed to computing the *CRPD* for one preemption and multiplying it by the preemption count.

4.1 Benchmarks

PapaBench PapaBench is a real-time benchmark based on the control application of a drone called Paparazzi. It has two modules: *fly-by-wire*, and *auto-pilot*. Each module contains several tasks, which are large enough for the needs of our experiments.

Mälardalen benchmarks The Mälardalen benchmarks are a set of programs designed to evaluate WCET analysis methods. Most Mälardalen programs are too small to be interesting for our experiments, so we used two of the largest programs in the Mälardalen benchmarks, *compress* and *adpcm*.

Robot Control Application This benchmark is a real-life robot controller application. This software contains several tasks, such

Task	Size	Task	Size
encode (adpcm)	5716	remote	944
decode (adpcm)	5240	balance	27580
reset (adpcm)	1104	trackandmove	6704
clblock (compress)	2016	Robot control	
output (compress)	1372	Mälardalen benchmarks	

Table 2: Code size (in bytes) of tasks in Mälardalen benchmarks and robot control application

Set	Low-priority	High-priority
1	senddataautopilot	checkfailsafe
2	senddataautopilot	checkmega128values
3	senddataautopilot	testppm
4	testppm	checkfailsafe
5	testppm	checkmega128values
6	testppm	senddataautopilot

(a) PapaBench (fly-by-wire)

Set	Low-priority	High-priority
7	altitudecontrol	climbcontrol
8	altitudecontrol	radiocontrol
9	altitudecontrol	stabilisation
10	climbcontrol	altitudecontrol
11	climbcontrol	radiocontrol
12	climbcontrol	stabilisation

(b) PapaBench (auto-pilot)

Set	Low-priority	High-priority
13	encode (adpcm)	decode
14	reset (adpcm)	encode
15	clblock (compress)	output

(c) Mälardalen benchmarks

Set	Low-priority	High-priority
16	remote	balance
17	remote	trackandmove
18	trackandmove	balance

(d) Robot control application

Table 3: Task sets definition

as *navigation* task, and *balance* task (to ensure that the robot does not fall). The tasks are preemptible (*balance* task has the highest priority), and sufficiently large for our experiments.

Table 1 shows the code size (in bytes) for the PapaBench tasks. Table 2 shows the code size (in bytes) for the tasks from the *Mälardalen* benchmarks and the robot control application.

4.2 Results

For each task set, the bound on the number of additional cache misses (the misses already present without preemption are not counted) for a single preemption is computed for each cache configuration. The cache configuration parameters include the associativity level (from 1 to 4), the cache block size (from 16 to 32) and the set count (from 16 to 64). The results are shown for PapaBench in Figure 8 and Figure 9 (for *fly-by-wire* and *autopilot* modules respectively). The results for the Mälardalen benchmarks and the robot control application are shown in Figure 10 and Figure 11 respectively. The task sets referenced are defined in Table 3.

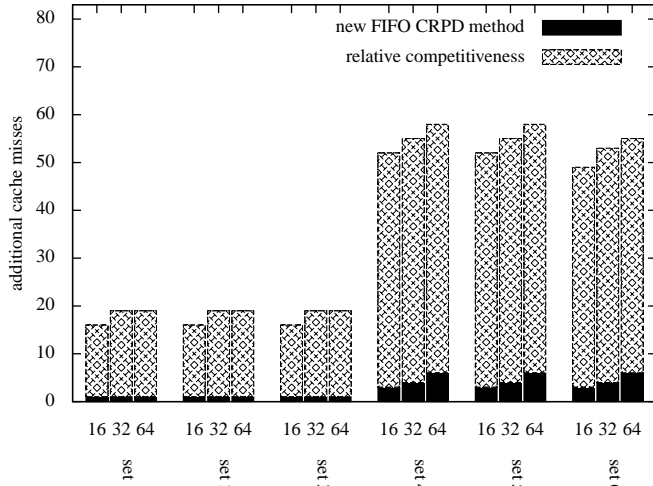


Figure 8: Papabench (fly-by-wire) experimental results

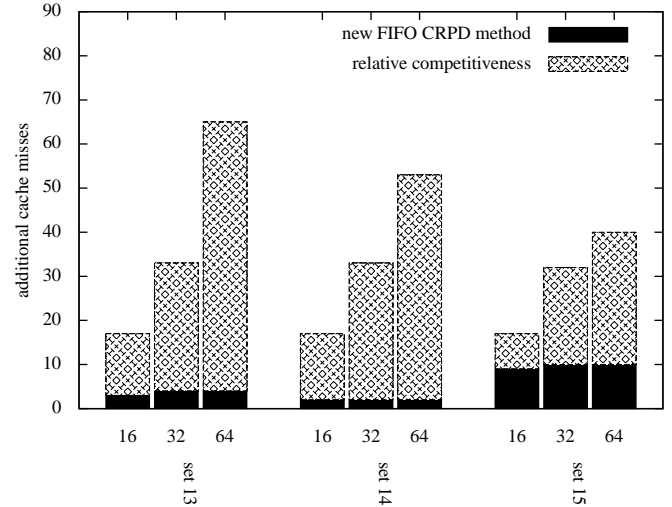


Figure 10: Mälardalen experimental results

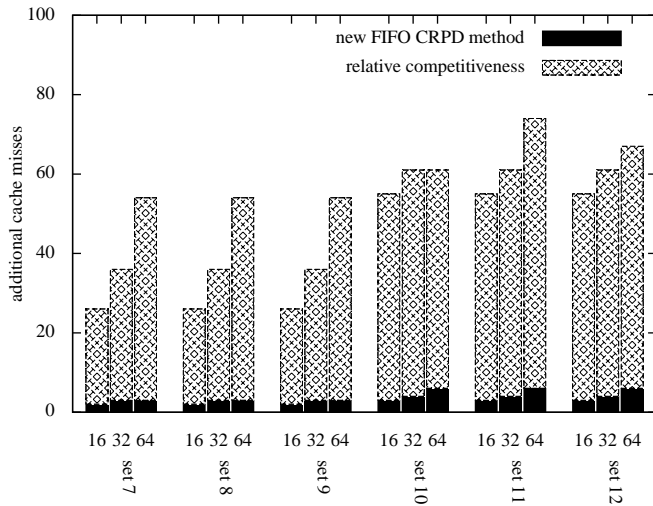


Figure 9: Papabench (auto-pilot) experimental results

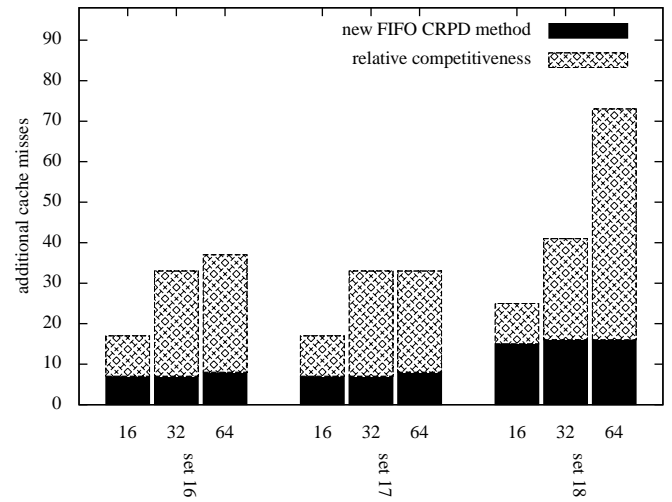


Figure 11: Robot experimental results

We display results only for each different set count, while averaging the results over the other parameters (cache block size, and associativity level), because we observe that the results are primarily influenced by the cache set count. The associativity level has little effect on the results, since increasing the number of ways for FIFO caches only increases the maximum length of phases to detect in the static phase detection analysis, however those longer phases happen rarely in programs. The number of additional cache misses increases with cache set count, as more cache sets allow for more blocks to be in the cache at the same time, which are potentially evicted by a preemption. The cache block size has little effect on the additional cache miss count on average, because while it allows for a greater amount of data in the cache, it does not affect the maximum count of blocks that can be in the cache.

The results show that the new approach introduces far less pessimism, compared with the approach based on relative competitiveness. This gap between the two methods can be attributed to two main causes. First, since the relative competitiveness based approach handles FIFO caches by assuming a LRU cache with a lower associativity level, and that the resulting miss count must be multiplied (by a factor depending on the associativity level), it is reasonable to expect a high miss count. Additionally, our approach

counts additional cache misses due to preemption only for blocks that were previously *always hit*, thus limiting the double-counting of cache miss significantly.

Sensitivity to number of preemptions As mentioned previously in Section 3, our method can be used either at each iteration in the WCRT computation (in order to compute the exact number of added cache misses for each preemption count), or it can be used to compute the additional misses for one preemption, and multiply that number by the preemption count at each iteration of the WCRT computation. The second method is faster (because we do not need to repeat the computation at each step of the iteration), but it is also more pessimistic: while the number of preemption increases, less and less additional cache misses are caused by each preemption.

This effect is shown in Figure 12. We see that for a low number of preemptions (*i.e.* less than 200), there is not much difference in tightness between the two approaches (note that the preemption count is the maximum number of preemptions each time the preempted task is activated, not the total preemption count). As the preemption count goes up, the difference between the two methods increases. This threshold increases with the preempted task size (since it increases the number of program points where a preemption could generate a lots of additional misses).

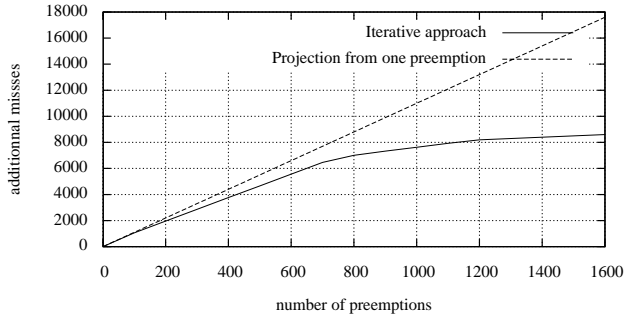


Figure 12: Iterative approach vs. standalone CRPD

5. Related work

There has been extensive research focusing on CRPD analysis on LRU caches. Traditionally, CRPD is computed by analyzing (i) the preempted task [1, 9], (ii) the preempting task [19], or (iii) both the preempted and preempting tasks [2, 14, 17, 18]. The concept of *useful cache block* (UCB) is introduced by [9] for analyzing the preempted task. UCB is computed for the preempted task, and represents the cache blocks that are in use by the preempted task that could cause cache misses if evicted by the preempting task. On the other hand, the set of *evicting cache block* (ECB) is computed for the preempting task, and represents the cache blocks that may be evicted by the preempting task. The notion is that a cache set unused by the preempting task will not cause any eviction of cache blocks used by the preempted task in the same cache set. Several approaches combine ECB and UCB to compute the CRPD. An article by S. Altmeyer et al. [2] gives an overview of these methods, and shows the strengths and weaknesses of each approach.

C. Burguière et al. [4] have performed a study of other replacement policies (including Pseudo-LRU and FIFO), in the context of the CRPD. This study finds that in FIFO caches, the bound on the number of additional misses due to a preemption cannot be expressed in terms of UCB and ECB. C. Berg [3] shows that some non-LRU cache replacement policies (including FIFO, among others) exhibit *domino effect*. This means that for FIFO instruction caches, an additional cache miss somewhere in a program can modify the cache state, thus causing further misses later in the program, which in turn can cause even further misses, and so on. As shown in [3], this effect has a potentially *unbounded* length (except, of course, by the length of the program execution). This fact largely contributes to the infeasibility of the CRPD computation on FIFO caches using UCB and ECB, as presented in [4].

A work-around for this problem has been proposed in [16] by using the concept of *relative competitiveness*. *Relative competitiveness* of cache replacement policies allows us to express the bound on the number of cache misses for a given access sequence and replacement policy, in terms of the bound under a different replacement policy. Applied to the CRPD computation for FIFO caches, computation on relative competitiveness enables us to express the CRPD for a FIFO cache as a function of the CRPD for a LRU cache of lower associativity level, and so we can apply the methods described previously (*i.e.* UCB/ECB analysis) to compute the CRPD for FIFO caches. The main drawback with this method is the high over-estimation of cache miss count.

6. Conclusion

We have proposed an approach to handle CRPD in the presence of an instruction cache with a FIFO replacement policy. Our analysis computes the additional cache misses due to task preemption in a safe way, while avoiding the double-counting of cache misses

already taken into account by the underlying FIFO cache categorization method. We have also proposed a set of properties that a CRPD analysis must satisfy in order to be safe in presence of timing anomalies, and we show that our analysis does satisfy these properties. Finally, we presented a method to use our CRPD analysis in the context of WCRT computation, in a way that handles diminishing CRPD contributions as the preemption count increases, providing tight bounds. We evaluated our analysis on realistic benchmarks by modeling a real-life ARM processor. The results show that our approach provides tight CRPD bounds in comparison to the state-of-the-art approach.

Acknowledgments

This work was partially supported by A*STAR Public Sector Funding Project Number 1121202007 - "Scalable Timing Analysis Methods for Embedded Software".

References

- [1] S. Altmeyer and C. Burguière. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *ECRTS*, 2009.
- [2] S. Altmeyer, R. I. Davis, and C. Maiza. Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. In *RTSS*, 2011.
- [3] C. Berg. Plru cache domino effects. In *WCET*, 2006.
- [4] C. Burguière, J. Reineke, and S. Altmeyer. Cache-related preemption delay computation for set-associative caches - pitfalls and solutions. In *WCET*, 2009.
- [5] F. Cassez, R. R. Hansen, and M. C. Olesen. What is a timing anomaly? In *WCET*, 2012.
- [6] L. K. Chong, C. Ballabriga, V.-T. Pham, S. Chattopadhyay, and A. Roychoudhury. Integrated timing analysis of application and operating systems code. In *RTSS*, 2013.
- [7] D. Grund and J. Reineke. Precise and efficient FIFO-replacement analysis based on static phase detection. In *ECRTS*, 2010.
- [8] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The mälardalen wcet benchmarks: Past, present and future. In *WCET*, 2010.
- [9] C.-G. Lee, H. Hahn, Y.-M. Seo, 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. *IEEE Trans. Comput.*, 47(6), 1998.
- [10] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for wcet analysis. *Real-Time Systems*, 34(3), 2006.
- [11] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 2007.
- [12] T. Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Chalmers University of Technology, 2002.
- [13] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS*, 1999.
- [14] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS*, 2003.
- [15] F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, and M. De Michiel. Papabench: a free real-time benchmark. In *WCET*, 2006.
- [16] J. Reineke and D. Grund. Relative competitive analysis of cache replacement policies. In *LCTES*, 2008.
- [17] J. Staschulat and R. Ernst. Scalable precision cache analysis for real-time software. *ACM TECS*, 6(4), 2007.
- [18] Y. Tan and V. J. Mooney. Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems. In *SCOPES*, 2004.
- [19] H. Tomiyama and N. D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *CODES*, 2000.