Towards Analysing Cache-Related Preemption Delay in Non-Inclusive Cache Hierarchies

THILO LEON FISCHER, Institute of Embedded Systems, Hamburg University of Technology, Hamburg, Germany
HEIKO FALK, Institute of Embedded Systems, Hamburg University of Technology, Hamburg, Germany

The impact of preemptions has to be considered when determining the schedulability of a task set in a preemptively scheduled system. In particular, the contents of caches can be disturbed by a preemption, thus creating context-switching costs. These context-switching costs occur when a preempted task needs to reload data from memory after a preemption. The additional delay created by this effect is termed *cache-related preemption delay* (CRPD). The analysis of CRPD has been extensively studied for single-level caches in the past. However, for two-level caches, the analysis of CRPD is still an emerging area of research. In contrast to a single-level cache, which is only affected by direct preemption effects, the second-level cache in a two-level hierarchy can be subject to *indirect interference* after a preemption. Accesses that could be served from the L1 cache in the absence of preemptions, may be forwarded to the L2 cache, as the relevant data was evicted by a preemption. These accesses create the *indirect interference* in the L2 cache and can cause further evictions. Recently, a CRPD analysis for two-level non-inclusive cache hierarchies was proposed. In this article, we show that this state-of-the-art analysis is unsafe as it potentially underestimates the CRPD. Furthermore, we show that the analysis is pessimistic and can overestimate the indirect preemption effects. To address these issues, we propose a novel analysis approach for the CRPD in a two-level non-inclusive cache hierarchy. We prove the correctness of the presented approach based on the set of feasible program execution traces. We implemented the presented approach in a *worst-case execution time* (WCET) analysis tool and compared the performance to existing analysis methods. Our evaluation shows that the presented analysis increases task set schedulability by up to 14 percentage points compared with the state-of-the-art analysis.

CCS Concepts: • **Computer systems organization** → **Real-time systems**; *Embedded software;*

Additional Key Words and Phrases: WCET analysis, multi-level caches, context-switching costs, cache-related preemption delay, preemptive scheduling

ACM Reference Format:
Thilo Leon Fischer and Heiko Falk. 2024. Towards Analysing Cache-Related Preemption Delay in Non-Inclusive Cache Hierarchies. *ACM Trans. Embedd. Comput. Syst.* 24, 1, Article 8 (October 2024), 37 pages. https://doi.org/10.1145/3695768

1 Introduction

In hard real-time systems, every task has an associated deadline. For the system to operate properly, not only the functional correctness, i.e., the correct result of a computation, but also the timeliness

Authors' Contact Information: Thilo Leon Fischer, Institute of Embedded Systems, Hamburg University of Technology, Hamburg, Germany; e-mail: thilo.leon.fischer@tuhh.de; Heiko Falk, Institute of Embedded Systems, Hamburg University of Technology, Hamburg, Germany; e-mail: Heiko.Falk@tuhh.de.

ACM 1539-9087/2024/10-ART8
https://doi.org/10.1145/3695768

of the computation has to be ensured. To verify that a system will keep all deadlines even in the worst-case scenario, it is necessary to provide an upper bound on the execution time of each task. This upper bound is called the ***worst-case execution time*** (**WCET**). An estimate for the WCET can be computed by analysing the **control-flow graph** (**CFG**) of a task and determining the path that maximizes the execution time [25].

The WCET is, however, not sufficient to decide whether all instances of a task, called jobs, will finish in time, if preemptions occur. During a preemption, the execution of a job is paused in favor of processing another job. A preemption does not only delay the execution of the preempted job but also causes further context-switching costs for the preempted job after resuming execution. This additional delay arises as the preemption may change the state of the hardware. In particular, the state of the caches can be modified by the preempting job; evicting data that is required by the preempted job. When resuming the execution of the preempted job, this data has to be reloaded into the caches, which causes the additional delay. This delay is named ***cache-related preemption delay*** (**CRPD**) [24]. The problem of CRPD has been extensively studied for single-level caches in the past [2, 5, 24, 31].

The CRPD analysis for multi-level caches is fundamentally more challenging than for a single-level cache. This is due to the so-called *indirect effect of preemptions* in a multi-level cache hierarchy [10]. The indirect effect describes the increased intra-task interference in the L2 cache after a preemption has happened. A preemption may evict useful data from the L1 cache. Thus, a cache block that would have been reused from the L1 cache without any preemptions has to be reloaded from the L2 cache due to the preemption. This additional access to the L2 cache creates intra-task interference and can potentially evict data from the L2 cache, which in turn causes further delays. Capturing the aging effects in the L2 cache caused by the indirect effect is crucial to determine a safe bound on the CRPD. We call the indirect effect of preemption *indirect interference* in this article.

Only few approaches have been proposed for the problem of CRPD in multi-level cache hierarchies. Chattopadhyay and Roychoudhury created the first analysis considering the indirect preemption effects for non-inclusive two-level cache hierarchies [10]. Two years later, Zhang and Koutsoukos analyzed CRPD in inclusive cache hierarchies [47]. Recently, Rashid et al. proposed an improvement for the analysis of non-inclusive caches [36]. In their work, they expanded the concept of **useful cache blocks** (**UCB**) to two-level caches by differentiating between L1-useful and L2-useful blocks. To the best of our knowledge, no further improvements for non-inclusive cache hierarchies have been proposed, which is why we refer to [36] as the state-of-the-art for non-inclusive cache hierarchies in the rest of the article.

In this article, we demonstrate several shortcomings of the state-of-the-art approach. We show multiple issues that may lead to the underestimation of the CRPD and a source of overestimation that causes pessimism in the analysis. To fix these issues, we present a novel analysis approach for the CRPD in a two-level non-inclusive instruction cache hierarchy. We prove the correctness of the presented analysis at the level of task execution traces, i.e., sequences of program locations and concrete system states. We integrated the presented approach in a WCET-aware compiler [13] and evaluated its performance for different task sets and system configurations. Our evaluations show that the presented approach improves the task set schedulability by up to 14 percentage points over the previous state-of-the-art.

Over the past decade, the state-of-the-art WCET research has been lagging behind the developments of commercial processor architectures. For this reason, a gap has developed between the capabilities of static worst-case analysis methods and **commercial-of-the-shelf** (**COTS**) processors.

We consider this article to be a first step towards enabling the analysis of modern hardware architectures featuring multi-level caches. Further research is needed to close the gap between

static WCET analysis methods and modern processor architectures. This is the case as the presented analysis poses strict requirements on the hardware, such as separate instruction and data caches as well as **least-recently-used** (**LRU**) replacement in both cache levels. These restrictions need to be addressed in future work.

The main contributions of this article are:

— Definition of the concrete semantics of CRPD in a two-level non-inclusive cache hierarchy to prove the correctness of the presented analysis.
— A novel analysis approach for CRPD in non-inclusive cache hierarchies, which fixes unsafe estimations and eliminates pessimism present in the state-of-the-art analysis.
— An evaluation showing significant improvements in task set schedulability using the presented CRPD analysis.

The rest of this article is structured as follows: Related work is discussed in Section 2. In Section 3, we discuss the assumptions and requirements on the system and the analyzed tasks. We present the background and state-of-the-art in Section 4. The pessimism and unsafe formulations of the state-of-the-art analysis are demonstrated in Section 5. In Section 6, we introduce the concrete semantics for a two-level non-inclusive cache hierarchy to precisely define the required concepts. We analyze the CRPD stemming from accesses that result in an L1 hit without preemptions in Section 7. In Section 8, the CRPD from accesses considered L2 hits without preemptions is analyzed. We evaluate the performance of the presented approach in Section 9. Finally, we conclude the article in Section 10.

2 Related Work

Static analysis of cache behavior is an active research field. For over two decades, many different facets of cache behavior and cache-related delays have been analyzed.

In Reference [14], Ferdinand and Wilhelm presented *may* and *must* analyses. The concrete states of the system are abstracted using *abstract interpretation* [11] to the minimal and maximal age of a cache block. Twenty years later, Touzeau et al. [41] presented an abstraction based on zero-suppressed binary decision diagrams that allows for a precise analysis of may/must information.

The may and must analyses are instances of *qualitative* analyses. Each individual access to the cache can be analyzed and classified as either a hit, a miss, or unknown. In contrast, *quantitative* analyses provide an upper bound on the total delay due to some cache behavior. The *persistence* analysis [14, 38] is a quantitative analysis that determines whether a set of accesses can result in at most one cache miss. Stock et al. [40] presented a precise persistence analysis.

If a cache is shared between multiple cores, the interference between different cores has to be considered. Hardy and Puaut [21], as well as Liang et al. [26] analyzed the interference between tasks in a shared cache by counting the number of potential conflicting cache blocks. Nagar [33] determined the worst-case placement of interfering accesses in the CFG for a quantitative analysis of shared cache interference. Zhang et al. [46] eliminated infeasible access combinations in the interference computation by considering the order of accesses. Fischer and Falk [15] presented a qualitative analysis for shared caches by bounding the time between conflicting accesses. Xiao et al. [45] performed a schedulability analysis in a non-preemptive system with a shared cache.

CRPD occurs when multiple tasks are assigned to a single core and preemptions are enabled. Busquets-Mataix et al. [9] showed how the **worst-case response time** (**WCRT**) of a task can be determined in the presence of preemption delays. Lee et al. [24] analyzed the CRPD based on the number of UCBs. Altmeyer and Burguiere [3, 5] refined the definition of a UCB. A block is only considered to be useful if it is definitely cached, as otherwise the timing analysis will already

account for the potential cache miss. Altmeyer et al. [4] introduced the concept of *resilience* to analyze set-associative LRU caches.

Cache blocks may persist in the cache between two executions of the same task. This fact can be used to reduce the WCRT estimate. Rashid et al. [35] analyzed *persistence reload* overheads that occur when persistent blocks are evicted by a preempting task.

The CRPD analyses listed above have focused on architectures with a single level cache. The analysis of CRPD in two-level caches is still an emerging area of research. Chattopadhyay and Roychoudhury [10] presented the first CRPD analysis for a two-level cache hierarchy, focusing on non-inclusive cache hierarchies.

Zhang and Koutsoukos [47] developed an analysis for inclusive cache hierarchies.

The work of Rashid et al. [36] introduced the concept of L1-UCBs and L2-UCBs and represents the state-of-the-art for CRPD analysis of non-inclusive two-level caches. The analysis poses an additional requirement, compared with Reference [10]: the L2 cache needs a higher or equal number of cache sets and associativity than the L1 cache. While [36] reports increased analysis precision compared with Reference [10], we will show in this article that the analysis contains flaws, which may cause underestimation of the actual preemption penalty.

All three approaches for multi-level CRPD analysis assume LRU replacement policy at both cache levels and instruction-only caches. Furthermore, the accesses which will be issued by the analyzed tasks must be known by the analysis. Otherwise, it is impossible to determine the possible cache states and the resulting preemption delays. This requires the architecture to have predictable fetching behavior.

The architecture requirements of the presented approach are identical to the requirements of the state-of-the-art analysis [36]. We discuss these requirements and the reasoning behind them in the following section.

3 System Model

The architectural and task model used in the presented analysis are discussed in this section. First, in Section 3.1, we focus on the hardware requirements and their impact on the applicability of the analysis to commercial processors. The properties of the executed tasks are treated in Section 3.2.

3.1 Architectural Model

We describe the target architecture to which the presented analysis can be applied in Section 3.1.1 and discuss the applicability to commercial processors in Section 3.1.2.

3.1.1 Target Architecture. We analyze an architecture that consists of a single core with a two-level instruction cache hierarchy. When processing a memory access, the L1 cache is always accessed; the L2 cache is only accessed if the requested information is not contained in the L1 cache. Both cache levels utilize the LRU replacement policy, as it offers high predictability [39] and is recommended for real-time systems in Reference [43].

As is the case in previous work on multi-level CRPD analysis [10, 36, 47], we focus on instruction caches in this article. We do not model interference from data accesses in the caches, as data accesses frequently target uncertain addresses. Thus, it is required that the L1 and L2 caches are instruction only caches, or that the caches are partitioned in such a way that the data accesses do not interfere with instructions stored in the caches.

The mapping from instructions to the corresponding cache set needs to be known at analysis time to determine which cache blocks conflict in the cache at runtime. This is the case if no virtual memory is used, or when using virtual memory one of the following is true: (1) the caches are physically indexed, or (2) the index derived from the virtual address is identical to the index derived

from the physical address. The second condition holds when the index bits are contained in the page offset of the address, or the page is colored so that the index bits match the physical address.

There are three different approaches to manage cache inclusivity in multi-level caches [6]. In an *inclusive* hierarchy, the L1 cache is enforced to contain a subset of blocks from the L2 cache. In an *exclusive* hierarchy, a cache block may only be contained in a single cache level at a time. No such condition is imposed in a *non-inclusive* hierarchy. A cache block may be contained only in the L1 cache, only in the L2 cache, or in both cache levels at the same time. We focus on *non-inclusive* cache hierarchies in this article.

As is the case in Reference [36], we impose the restriction that the number of ways in the L1 cache W_1 is less or equal to the number of ways in the L2 cache W_2. The same restriction is made for the number of sets in the L1 and L2 caches. These restrictions are not significant for real world systems, as the L2 cache is typically much larger than the L1 cache and easily satisfies these conditions.

Furthermore, we assume that the system is timing compositional [20]. This means that the WCRT, which describes the longest duration from the release of a job to its completion, may be computed using the WCETs and an additional penalty to account for the CRPD.

All memory accesses that are performed need to be known at analysis time in order to detect potential sources of CRPD. This means that the fetching behavior has to be deterministic; features like prefetching additional code on a cache miss and speculative execution are not considered in the analysis.

Branch predictors predict whether a conditional branch is taken or not. There are two main categories of branch prediction: *static* and *dynamic*. In static branch prediction, the prediction is made based on a simple rule. For example, conditional branches are assumed to be always taken or never taken. These branch prediction modes are supported, for example, by the Arm Cortex-R4 [28]. Another architecture implementing static branch prediction is the Infineon TC1.6E. 16 bit branches and 32 bit branches with backward displacement are predicted as taken, whereas 32 bit branches with forward displacement are predicted as not taken [23].

The presented analysis supports static branch prediction as the predictions made during runtime are known. Thus, it is possible to incorporate the static branch predictions into the CFG of the analyzed task by annotating the predicted instruction fetches. Dynamic branch prediction gathers information during runtime and performs the target prediction based on this information. As the prediction made during runtime is not known to the analysis, both scenarios, taking the branch and not taking the branch, need to be considered.

Another source of unpredictability is out-of-order execution. The order of accesses needs to be known to determine which conflicts occur in the caches. If the order of two accesses is not fixed, it is not possible to decide whether a potential conflict actually occurs during the analysis. Considering all possible scenarios creates high uncertainty in the analysis. For this reason, we consider an in-order pipeline.

3.1.2 Applicability to Commercial Processors. Modern architectures have evolved to contain an increasing number of features to improve the average case performance at the cost of predictability. Static worst-case analyses have failed to keep pace with these developments. This gap between the hardware architecture and static analyses have been noted already in the literature [12, 18, 32, 44]. There are two components to solving this problem: (1) ensuring predictability during the hardware design process by avoiding unpredictable features, and (2) improving the capabilities of static analyses to support more complex hardware architectures.

The former component has been addressed by Wilhelm et al. in Reference [43]. Recommendations for future architectures are made in order to enable precise and efficient analysis of the

system properties. For example, caches should use LRU replacement and be separated into instruction and data caches. Similarly, Reineke showed in Reference [37] that, for real-time systems, the LRU replacement policy is preferable over randomized replacement, which is used in the real-time focused Arm Cortex-R4, Cortex-R5, and Cortex-R7 processors [29].

Over the past years, the open-source RISC-V instruction set [42] has become increasingly popular for use in real-time systems [7, 17, 34]. The open-source nature of these architectures facilitates WCET analysis: System designers can incorporate predictability requirements into the architectural design, and researchers have access to specification details, eliminating the need for reverse-engineering of undocumented features. This trend is expected to continue, potentially increasing availability of timing-predictable processors in the future.

In this article, we focus on the latter component and aim at increasing the capabilities of static analysis methods to hardware architectures with multi-level cache hierarchies. We argue that, despite the strict requirements imposed on the architecture, the algorithms and proofs presented in this article are a first step to close the gap and advance the capabilities of static worst-case analysis.

3.2 Task Model

The tasks in the system are contained in a set T. Each task has three properties: its WCET, the period, which describes how often a new job of the task is released, and a relative deadline, before which each job has to be finished. We assume that tasks are scheduled according to a fixed priority. The set $hp(\tau) \subset T$ contains all tasks that may preempt $\tau \in T$. To denote the task currently being analyzed, we use the symbol τ; preempting tasks are denoted by φ or ψ.

The set $\mathbb{P}$ contains all program locations of τ. The CFG $(\mathbb{P}, E)$ of τ is given by the set of program locations and edges E that connect the locations. We associate every memory reference in the code to a location $p \in \mathbb{P}$. The reference associated to a location p causes a memory access when the control transfers to a new location p' with $(p, p') \in E$.

In the following, the set of all memory blocks is denoted by $\mathbb{M}$. The function $set_l : \mathbb{M} \to \mathbb{N}$ denotes the cache set for a block $m \in \mathbb{M}$ at the level $l \in \{1, 2\}$. The cache miss timing penalty of the L1 / L2 cache is written as d_{L1} / d_{L2}.

We assume that tasks do not share code. Sharing code between tasks is problematic when analyzing CRPD as the access classification for the second level cache can be invalidated by a preempting task. An access that is an L1 cache miss, without any preemptions, will reach the L2 cache and promote the targeted cache block to the youngest position in the L2 cache. We can thus be sure that, without preemptions, the target cache block is refreshed in the L2 cache. However, when this block is also used by another task, it can be loaded into the L1 cache during a preemption. As a result, the L2 cache may not be accessed for that particular request. It is no longer possible to make statements on the age of the shared cache block in the L2 cache. We have excluded code sharing for this reason in the presented analysis.

When determining the schedulability of a task set T, we assume that the timing overhead of the scheduler is negligible and the state of the caches are only affected by the preempting task and not the scheduler itself.

4 Background

In this section, we establish the background for cache analysis in the context of CRPD and give an overview of the current state-of-the-art analysis for non-inclusive cache hierarchies.

To determine whether a cache block is definitely cached at a particular location in the program, a *must* analysis is performed [14]. The *must* analysis determines an upper bound on the LRU age of each cache block for every program location. If the maximal age is less than the associativity of the cache, the block is guaranteed to reside in the cache. We denote the result of the *must* analysis

for the cache level $l \in \{1, 2\}$, at location p, for the cache block m by $MustAge_l^p(m)$. Note, that for LRU caches, the age of a cache block can be represented using the set $\{0, \ldots, W-1\} \cup \{\infty\}$, where W is the associativity of the cache and ∞ signifies that the block is not contained in the cache. When computing an upper bound on the cache age, we regard values larger than $W-1$ to be equivalent to ∞, as all values larger than $W-1$ show that the block is not definitely contained in the cache. The complementary analysis to the *must* analysis is the *may* analysis. It keeps track of the minimal age of cache blocks and is used to decide whether a cache block may be contained in the cache. The correctness of the *may* and *must* analyses are based on the theory of abstract interpretation [11].

Together, the results of the *may*/*must* analyses for the first-level cache can be used to determine the so-called ***cache-access-classification*** (**CAC**) for the second level cache [22]. The CAC of an access can take three values: *always* (A): the access will always reach the L2, i.e., it is a definite L1 cache miss; *never* (N): the access will never reach the L2, i.e., it is a definite L1 cache hit; *unknown* (U): the access may reach the L2. Note that the first-level cache is always accessed and that these classifications refer to an execution without any preemptions.

The analysis of CRPD is commonly based on the notion of *useful* and ***evicting*** **cache blocks** (**ECB**) [31]. To capture the effects of a preempting task, the cache blocks that may be stored in the cache by the preempting task are determined. These cache blocks are called ECB.

Definition 4.1 (Evicting Cache Block [4]). An evicting cache block is a cache block that may be accessed by a task. The set of ECB of task φ, cache level $l \in \{1, 2\}$, and cache set s is denoted by $ECB_l^{\varphi}(s)$.

When τ is preempted by φ, φ may also be preempted by all tasks $\psi \in hp(\varphi)$. Thus, the ECB of φ and $\psi \in hp(\varphi)$ may work together to evict data from τ. Such nested preemptions can be modeled by considering the union of all ECBs of potentially preempting tasks [4]. We introduce the following notation for the ECBs considering nested preemptions:

$$\widehat{ECB}_l^m = \left| \bigcup_{\psi \in \{\varphi\} \cup hp(\varphi)} ECB_l^{\psi}(set_l(m)) \right|. \tag{1}$$

The value $\widehat{ECB}_l^m$ is the number of cache blocks that preempting tasks may store in the same cache set as $m \in \mathbb{M}$ at the cache level $l \in \{1, 2\}$.

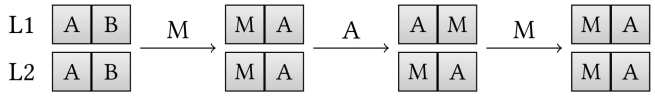
The ECB of a preempting task can evict data from the preempted task. To determine whether the evicted data contributes to the preemption penalty, cache blocks of the preempted task have been classified as UCB.

Definition 4.2 (Useful Cache Block [2]). A cache block $m \in \mathbb{M}$ is useful at a program location $p \in \mathbb{P}$ iff it is definitely cached at p and may be reused without eviction at a location reachable from p.

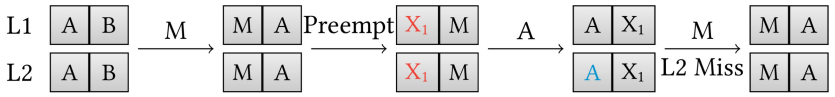
Definition 4.2 has been utilized for analyzing single-level caches. However, this definition is unsuited for multiple cache levels as it does not differentiate between the cache levels. For this reason, Rashid et al. [36] introduced the notion of *L1-useful* and *L2-useful* cache blocks.

Definition 4.3 (L1-Useful Cache Block [36]). A cache block $m \in \mathbb{M}$ is L1-useful at a program location $p \in \mathbb{P}$ iff it is definitely cached in L1 at p and may be reused without eviction from the L1 cache at a location reachable from p.

Definition 4.4 (L2-Useful Cache Block [36]). A cache block $m \in \mathbb{M}$ is L2-useful at a program location $p \in \mathbb{P}$ iff it is definitely cached in L2 at p and may be reused without eviction from the L2 cache at a location reachable from p. Furthermore, m must not be a L1-UCB at p.



(a) Unpreempted execution. The access to A and the second access to M result in an L1 hit.



(b) Execution with a preemption before the access to A. The preemption loads an ECB X_1 into the caches. The creation of indirect interference from A is highlighted in blue. The second access to M results in an L2 miss.

Fig. 1. Example of indirect interference. The access sequence M, A, M is shown: (a) without preemption and (b) with preemption before the access to A.

We denote the set of L1-useful / L2-useful blocks at location p by UCB_1^p / UCB_2^p. Using the Definitions 4.3 and 4.4, we can determine from which cache level a block may be reused and how high the penalty will be if it has to be reloaded after a preemption. In the remainder of this section, we will explain how a bound on the CRPD is computed using the state-of-the-art [36] approach.

An important concept in [36] is the so-called *maximal LRU age* of the block m. Note that, even though [10] and [36] use the name *CU* for similar concepts, the definition of this value is slightly different in [10]. Following the definition from [36], the value $CU_l^p(m)$ is the maximal age of $m \in \mathbb{M}$ in cache level l at the first reference, reachable from p, that accesses m. If there are multiple references that could result in the next access to m with different cache ages for m, the maximal age that is smaller than W_l is taken. This value essentially captures the minimal resilience of the cache block m to preemption effects to create an increase in the execution time (cf. [4]).

For example, consider a cache hierarchy with $W_1 = 2$ and $W_2 = 4$. Suppose a cache block m has the values $CU_1^p(m) = 1$ and $CU_2^p(m) = 2$ for some location $p \in \mathbb{P}$. This shows us that there exists a path from p to an access to m that hits in the L1 cache and the L1 age of m at that access is 1. It would require a single evicting cache block to evict m from the L1 cache on that path.

Furthermore, we know that there exists a path to an access to m where m has age 2 in the L2 cache. It would require 2 distinct interferences, either from ECB or indirect interference, to evict m from the L2 on this path. Thus, after a preemption that loads one ECB in the L1 cache, m can contribute one L1 miss to the CRPD. If the preemption loads two or more ECBs into the L2 cache, an L2 access to m could result in an L2 cache miss. Note that the paths that realize the maximal LRU ages for the L1 and L2 may be different. Thus, this is an approximation as both cache levels are considered in isolation for the CU_l^p value.

Using these definitions, we can now focus on the indirect effect of preemption in two-level caches.

Definition 4.5 (Indirect Effect of Preemption [10]). The indirect effect of preemption, called *indirect interference* in this article, is defined as the additional L2 cache conflicts that are caused by L1 cache misses due to preemption.

Figure 1 shows an example for indirect interference and its consequences. The analyzed caches are 2-way associative. We utilize capital letters to denote cache blocks. Initially, the L1 and L2 cache contain the cache blocks A, B; ordered in ascending LRU age. We analyze the access sequence M, A, M. The unpreempted scenario is shown in Figure 1(a). The access to A and the second access to M both result in an L1 cache hit.

ALGORITHM 1: Indirect Interference $Ind^{p}_{m_y}$ due to Preemption at Location $p \in \mathbb{P}$ [36]

Result: The indirect effect suffered by every $m_y \in \mathbb{M}$

1 **for** $m_y \in \mathbb{M}$ **do**
2 $\quad Ind^{p}_{m_y} \leftarrow \emptyset$
3 **end**
4 **for** $m_x \in UCB^{p}_{1}$ **do**
5 $\quad$ **if** $CU^{p}_{1}(m_x) + \widehat{ECB}^{m_x}_{1} \geq W_1$ **then**
6 $\quad\quad FA^{p}_{m_x} \leftarrow GetFirstAccess(m_x, p)$
7 $\quad\quad$ **for** $m_y \in \mathbb{M} \setminus \{m_x\}$ **do**
8 $\quad\quad\quad$ **if** $MustAge_2^{FA^{p}_{m_x}}(m_y) \neq \infty \wedge set_2(m_y) = set_2(m_x) \wedge$
9 $\quad\quad\quad MustAge_2^{FA^{p}_{m_x}}(m_x) > MustAge_2^{FA^{p}_{m_x}}(m_y)$ **then**
10 $\quad\quad\quad\quad Ind^{p}_{m_y} \leftarrow Ind^{p}_{m_y} \cup \{m_x\}$
11 $\quad\quad\quad$ **end**
12 $\quad\quad$ **end**
13 $\quad$ **end**
14 **end**

Now consider the preempted scenario from Figure 1(b). In this example, the block A causes indirect interference to M. The preemption occurs after the first access to M, before the access to A. The preempting task stores an ECB X_1 in the caches. The ECB is marked in red in Figure 1(b). Due to the preemption, the access to A results in an L1 cache miss. The block A is thus accessed in the L2 cache. This ages M in the L2 cache and causes its eviction from the L2 cache. Thus, A causes indirect interference to M, as in the scenario without preemption, A does not interfere with M in the L2 cache. This indirect interference is highlighted in blue in Figure 1(b). In consequence, the final access to M results in an L2 cache miss, even though it was not directly evicted by the ECB from the preemption.

Such indirect interference has to be considered in the CRPD estimation to arrive at a safe upper bound. We will now give an overview of the state-of-the-art approach [36] to analyze indirect interference and compute the CRPD in two-level non-inclusive caches.

The state-of-the-art method utilizes Algorithm 1 to determine the potential indirect interference, which causes useful blocks to age in the L2 cache, due to a preemption at the program location $p \in \mathbb{P}$. The cache blocks causing indirect interference for the block m_y due to a preemption at p are stored in the set $Ind^{p}_{m_y} \subset \mathbb{M}$.

The $Ind^{p}_{m_y}$ values are initialized as the empty set (lines 1–3). All L1-UCBs blocks at p, $m_x \in UCB^{p}_{1}$, are considered as potential sources of indirect interference (line 4). This is done because, in the absence of preemptions, these blocks can be served from the L1 cache without accessing the L2 cache. If they are evicted and have to be reloaded from L2 or memory, they create additional intra-task interference in the L2 cache.

Line 5 of Algorithm 1 checks whether the block m_x may be evicted from the L1 cache due to the ECBs of the preempting tasks. This is a necessary condition for m_x to contribute to the indirect interference for another block $m_y \neq m_x$. In Line 6, the function $GetFirstAccess(m_x, p)$ is used. This function "determine[s] the first reachable reference" [36] to m_x from P. No formal definition is given for this function in Reference [36]. As we will see in Section 5, the ambiguity inherent in this function description causes the analysis to compute unsafe bounds on the CRPD.

It is then determined for which $m_y \neq m_x$ the reloading of m_x causes indirect interference in the L2 cache (lines 8–11). This condition consists of three parts: (1) on the first access to m_x, m_y must

ALGORITHM 2: Indirect Interference $ColInd^p_{m_y}$ due to Multiple Preemptions, the First Preemption Occuring at $p \in \mathbb{P}$ [36]

Result: The indirect interference suffered by $m_y \in \mathbb{M}$ when multiple preemptions collaborate.

1 **for** $m_y \in \mathbb{M}$ **do**
2 $\quad ColInd^p_{m_y} \leftarrow \emptyset$
3 $\quad FA^p_{m_y} \leftarrow GetFirstAccess(m_y, p)$
4 $\quad PP \leftarrow GetProgramPoints(p, FA^p_{m_y})$
5 $\quad$ **for** $p' \in PP$ **do**
6 $\quad\quad ColInd^p_{m_y} \leftarrow ColInd^p_{m_y} \cup Ind^{p'}_{m_y}$
7 $\quad$ **end**
8 **end**

be cached in the L2, (2) m_x and m_y must be mapped to the same cache set, and (3) m_x must be older than m_y in the L2 cache. (This includes m_x not being cached in the L2.) If these conditions are true, the block m_x potentially causes indirect interference to m_y and is added to $Ind^p_{m_y}$ in line 10.

The authors of [36] showed that multiple preemptions may collaborate to create a higher CRPD than each preemption could in isolation. To consider this fact in the analysis, Algorithm 1 is augmented to check all program locations, starting from p and leading to the first access of the analyzed block m_y. The pseudocode for this algorithm is shown in Algorithm 2. The set of all program locations contained in paths starting from p and ending in $FA^p_{m_y}$ is given by the function $GetProgramPoints(p, FA^p_{m_y})$. The cache blocks causing indirect interference are accumulated for all of these program locations to yield an upper bound on the indirect aging events $ColInd^p_{m_y}$. This upper bound captures the potential indirect interferences from multiple preemptions between p and $FA^p_{m_y}$.

Using this upper bound, the CRPD from L1 hits being degraded due to preemptions can be bounded. An L1 hit may either degrade to an L2 hit or an L2 miss. The former case is accounted for by γ^p_{L1} in Equation (2), while the latter penalty is accumulated by γ^p_{L1+L2} in Equation (3). For an access to contribute in one of these two ways, the targeted cache block m_y has to be an L1-UCB. To result in an L1 miss, the ECB have to be able to evict the block from the L1 cache, i.e., $CU^p_1(m_y) + \widehat{ECB}^{m_y}_1 \geq W_1$. This condition is the same for both Equations (2) and (3). The difference between an L2 hit and miss is whether the L2 ECBs and the indirect interference can evict the block also from the L2 cache. If the interference due to ECBs and indirect interference causes the maximal LRU age to be greater or equal to W_2, m_y may also be evicted from the L2 cache.

$$\gamma^p_{L1} = d_{L1} \cdot \left| \left\{ m_y \in UCB^p_1 \;\middle|\; CU^p_1(m_y) + \widehat{ECB}^{m_y}_1 \geq W_1 \wedge CU^p_2(m_y) + \widehat{ECB}^{m_y}_2 + |ColInd^p_{m_y}| < W_2 \right\} \right| , \tag{2}$$

$$\gamma^p_{L1+L2} = (d_{L1} + d_{L2}) \cdot \left| \left\{ m_y \in UCB^p_1 \;\middle|\; CU^p_1(m_y) + \widehat{ECB}^{m_y}_1 \geq W_1 \wedge CU^p_2(m_y) + \widehat{ECB}^{m_y}_2 + |ColInd^p_{m_y}| \geq W_2 \right\} \right| . \tag{3}$$

For each block contributing in this way to the CRPD, a cache miss penalty of d_{L1} cycles is added for each L1 cache miss and d_{L2} is added for each L2 cache miss. The value $\gamma^p_{L1} + \gamma^p_{L1+L2}$ thus gives an upper bound on the delay caused by reloading UCB into the L1 cache.

There is a third type of preemption delay in a two-level cache hierarchy that is not covered by Equations (2) and (3). References that result in an L1 miss but L2 hit may be degraded to an L2

ALGORITHM 3: CRPD from L2 Cache Misses [36]

Result: The delay suffered from L2 hits that are degraded to an L2 miss due to preemption.

1 $\gamma^p_{L2} \leftarrow 0$
2 **for** $m_y \in \widehat{UCB}^p_2$ **do**
3 $\quad \gamma_{m_y} \leftarrow 0$
4 $\quad \mathbb{R} \leftarrow GetHitLocations^p(m_y)$
5 $\quad$ **if** $MustAge^{R^1}_2(m_y) + \widehat{ECB}^{m_y}_2 + \left|ColInd^p_{m_y}\right| \geq W_2$ **then**
6 $\quad\quad \gamma_{m_y} \leftarrow \gamma_{m_y} + 1$
7 $\quad$ **end**
8 $\quad$ **if** R^1 *is in loop* **then**
9 $\quad\quad$ **if** $MustAge^{R^1}_2(m_y) + \left|ColInd^p_{m_y}\right| \geq W_2$ **then**
10 $\quad\quad\quad \gamma_{m_y} \leftarrow \gamma_{m_y} + 1$
11 $\quad\quad$ **end**
12 $\quad$ **end**
13 $\quad$ **for** $1 < n \leq |\mathbb{R}|$ **do**
14 $\quad\quad$ **if** $MustAge^{R^n}_2(m_y) + \left|ColInd^p_{m_y}\right| \geq W_2$ **then**
15 $\quad\quad\quad \gamma_{m_y} \leftarrow \gamma_{m_y} + 1$
16 $\quad\quad$ **end**
17 $\quad$ **end**
18 $\quad \gamma_{m_y} \leftarrow \min\{\gamma_{m_y}, |\mathbb{R}|, |ColInd^p_{m_y}| + 1\}$
19 $\quad \gamma^p_{L2} \leftarrow \gamma^p_{L2} + (\gamma_{m_y} \cdot d_{L2})$
20 **end**

miss due to a preemption. In the state-of-the-art method, this component of the CRPD is computed using Algorithm 3.

The set $\widehat{UCB}^p_2$ contains all memory blocks that are an L2-UCB in any location reachable from p. We have to consider all blocks that may become L2-UCBs even after the preemption because the indirect effect of a previous preemption can still cause these blocks to miss in the L2 cache. In contrast, we only had to consider the first access to an L1-UCB after the preemption, as after the first access the block will be the youngest block in the L1 cache and does not experience further interference from the preemption in the L1 cache.

For each memory block $m_y \in \widehat{UCB}^p_2$, the locations where an L2 hit takes place is stored in $\mathbb{R}$ by the function $GetHitLocations^p$ (line 4). The symbol $\mathbb{R}$ represents a sequence of references $(R^1, R^2, \ldots)$ that target the block m_y and result in an L2 hit. Note that no formal definition of this function is given in Reference [36].

In the following lines 5–17, it is checked whether these L2 hits may degrade to an L2 miss due to the preemption effects. The algorithm differentiates between the first L2 hit R^1 after a preemption and later accesses. The first reference to m_y after preemption may also experience direct interference from the ECBs of the preempting task. The potential eviction of the first reference is handled in lines 5–7. If the first reference may be reached again due to a loop, it can also contribute to the CRPD solely due to indirect interference. This is covered in lines 8–12. For all later accesses, only the indirect interference plays a role in the eviction of the target block (lines 13 — 17). The total contribution of m_y to this component of the CRPD is limited by $\min\{|\mathbb{R}|, |ColInd^p_{m_y}| + 1\}$ [36] (line 18). The total delay from L2-UCBs is accumulated in γ^p_{L2} (line 19).

According to Reference [36], Equations (2),(3), and Algorithm 3, which computes γ^p_{L2}, allow us to compute all three components of the preemption delay. The total CRPD γ is computed as the maximal sum of the three components γ^p_{L1}, γ^p_{L1+L2}, and γ^p_{L2} for any program location $p \in \mathbb{P}$:

$$\gamma = \max_{p \in \mathbb{P}} \left(\gamma^p_{L1} + \gamma^p_{L1+L2} + \gamma^p_{L2} \right). \tag{4}$$

However, as we will discuss in the following section, this approach contains several issues that may lead to an unsafe CRPD estimate.

5 Safety Issues and Pessimism in the State-Of-The-Art

In this section, we will analyze the state-of-the-art and highlight issues inherent in the approach. This is structured as follows:

First, in Section 5.1, we show that the indirect effect of a preemption may be underestimated in Algorithm 2. The algorithm considers the potential for indirect interference only up to the first reference. However, the algorithm does not consider accesses that have an uncertain cache-access classification, i.e., accesses that may not reach the L2 cache in every circumstance. Considering the CAC of accesses to the L2 cache is crucial in the analysis, as an access that does not reach the L2 cache will not refresh or load the targeted cache block to the youngest LRU position in the L2 cache. The indirect effects may thus still affect future L2 accesses to the same memory block. We show that multiple preemptions can create an L2 cache miss, that is not detected by the state-of-the-art analysis due to this issue. This can violate the safety of the analysis as the CRPD is underestimated.

In Section 5.2, we identify pessimism in the analysis of indirect interference. Algorithm 1 does not consider whether the access creating indirect interference actually occurs before the analyzed access. Thus, cache blocks are considered to create indirect interference even though they will always be accessed *after* the analyzed block. The indirect interference may thus be overestimated, which causes a pessimistic CRPD bound.

Section 5.3 discusses Algorithm 3. The algorithm is incomplete in the sense that it is not able to process all possible CFGs. In particular, the algorithm assumes that there will be one distinct reference to an L2-UCB which will perform the first access to the cache block. However, as the CFG may split into multiple branches, this is not always the case. We demonstrate such a scenario in Section 5.3.1. Finally, in Section 5.3.2, we show that the CAC for accesses to L2-UCBs is not correctly integrated in Algorithm 3. The algorithm thus overlooks potential L2 cache misses, resulting in an unsafe CRPD estimate.

5.1 Unsafe Estimations in Algorithm 2

It is possible for Algorithm 2 to underestimate the indirect interference due to multiple preemptions. This causes the resulting CRPD bound to be unsafe. The issue originates from the use of the $GetFirstAccess(m_y, p)$ function on line 3 in Algorithm 2.

Recall the definition of $GetFirstAccess()$ from [36]: "$GetFirstAccess(m_x, p)$ [...] is used to determine the first reachable reference to every m_x". No formal definition of this function is given. Note in particular that the CAC of the first reference is not considered.

Algorithm 2 checks all locations from the preemption at p to the first reference of the memory block and collects all memory blocks that may create indirect interference. In a two-level cache hierarchy, an access to a memory block does not guarantee to refresh the block at the L2 cache. Thus, indirect interference that has accumulated in the second level cache is not removed if the first access hits in the L1 cache.

An access beyond which indirect interference does not propagate has been termed a *firewall* in [47]. To be a firewall for data in the L2 cache, an access has to have an L2 CAC of *always*. This fact is not considered in Algorithm 2. Thus, there are scenarios in which the state-of-the-art method does not recognize a potential cache miss, leading to an unsafe CRPD bound.

We will now construct such a scenario in Figure 2. Consider the following cache configuration: the L1 and L2 caches are 4-way associative, while the L2 possesses more sets than the L1. We utilize capital letters to represent cache blocks. In the initial state, the L1 cache contains the blocks B, M, D, A, ordered in ascending LRU age. The L2 cache initially contains M, D, A, C. The access sequence to cache blocks is A, M, B, E, F, S, B, M. The critical access we focus on in this example is the final access to the block M.

Two different scenarios, without and with preemptions, are shown in Figure 2(a) and 2(b) respectively. The cache state is depicted by two rows of cache blocks. The first row represents the L1 cache, the second row represents the L2 cache. The blocks are ordered in ascending LRU age; the youngest block is placed at the left most position. We only depict a single set of the L2 for clarity. Note that the block S is mapped to a different L2 set. It thus only appears in the L1 cache set but not in the shown L2 cache set. We can see that during normal execution, the final access to M will result in an L2 hit.

It is possible to place two preemptions in this access sequence to cause the second access to M to miss. In Figure 2(b), a preemption happens directly at the beginning of the sequence and another preemption happens before the second reference to B. Each preemption causes one evicting cache block X to be stored in the L1 cache; it does not affect the shown L2 cache set. This is possible if X is mapped to a different L2 cache set, which is not shown in the figure. In Figure 2(b), the insertion of ECB is marked in red.

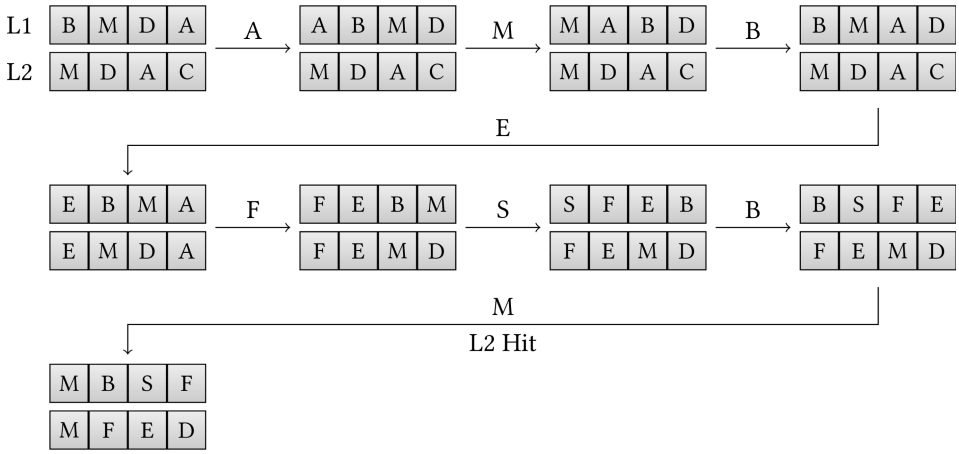
The first preemption causes the block A to be evicted from the L1 cache. Thus, when A is accessed after the preemption, it creates indirect interference for M in the L2 cache. In the figure, the creation of indirect interference is marked in blue. The L2 age of M increases from 0 (as in the unpreempted scenario) to 1. As the first reference to M does not reach the L2 cache, the age of M in the L2 cache remains 1 after the first reference to M. The second preemption evicts B from the L1 cache. B is accessed after the preemption and causes indirect interference to M. The L2 access for B actually causes M to be evicted from the L2 cache, thus incurring an L2 miss for the final access to M.

The final access to M experiences two indirect interferences. The age of M, in the unpreempted case, at the final access is 2. Adding the indirect interference to the cache age shows that M will be evicted as $2 + 2 \geq 4$ (cf. line 14 of Algorithm 3).

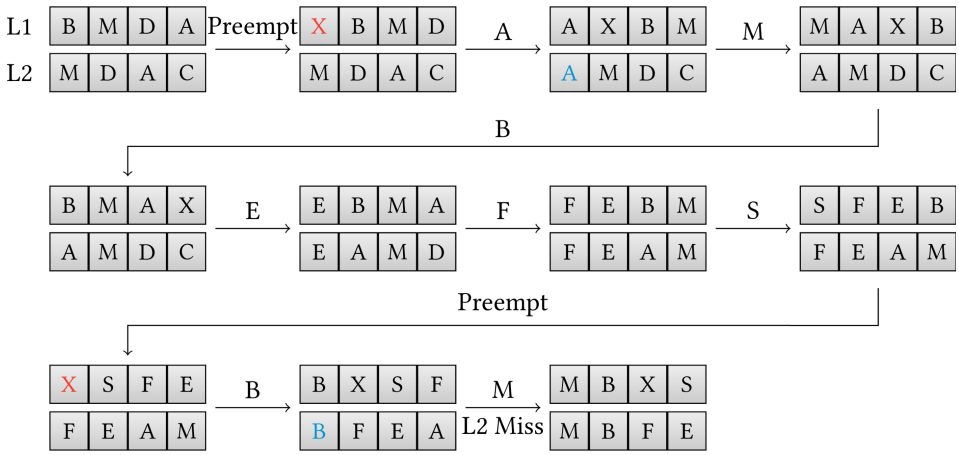
However, this L2 miss is not detected by the state-of-the-art method, as it stops considering the indirect interference after the first access to M (see Algorithm 2 lines 4–7). While the indirect interference of A is detected, the interference from B is missed as B's L1 age is 2 at the first acccess to B (see line 5 in Algorithm 1). B will thus not be evicted by the first preemption and, according to Algorithm 2 not cause indirect interference for later L2 hits for M. Consequently, the state-of-the-art CRPD computation is unsafe in this situation.

5.2 Pessimism in Algorithm 2

In addition to the optimism discussed in the previous section, the state-of-the-art method may also overestimate the indirect interference, leading to an overly pessimistic CRPD estimation. The overestimation originates from the fact that Algorithm 1 does not consider the ordering of accesses. In line 6 of Algorithm 1, the first access to the block m_x is determined. It is then checked whether m_x may cause indirect interference on another block m_y in lines 7–12. However, it is not considered whether an access to m_y will happen before or after the first access to m_x. m_x will only contribute



(a) Simulation of the access sequence A, M, B, E, F, S, B, M without preemptions. The final access to M results in an L2 hit.



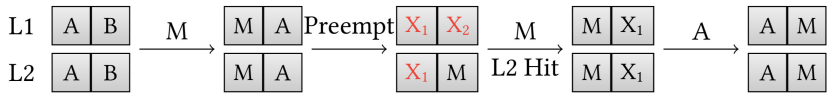
(b) Simulation of the access sequence A, M, B, E, F, S, B, M with two preemptions. The first preemption occurs before the access to A and the second preemption occurs before the second access to B. Both preemptions cause the ECB X to be loaded into the L1 cache. The insertion of ECBs is marked in red, while the creation of indirect interference is marked in blue. The final access to M results in an L2 miss.

Fig. 2. Simulation of the access sequence A, M, B, E, F, S, B, M: (a) without any preemption and (b) with two preemptions.

to the indirect interference, if it happens prior to the access to m_y. Thus, a block m_x may be pessimistically included in $Ind^p_{m_y}$ even though it will not cause m_y to age prior to the next access targeting m_y.

We will now provide an example of such pessimistic behavior in Figure 3. Consider the following cache configuration: the L1 and L2 caches are both 2-way associative, while the L2 cache has more sets than the L1 cache. The access sequence is M, M, A. Both analyzed cache sets of L1 and L2 start with the initial state A, B. The access we focus on is the second access to M. Without any preemptions, the second access to M results in an L1 hit. This behavior is shown in Figure 3(a).

(a) Simulation of the access sequence M, M, A. The second access to M and the access to A result in an L1 hit.



(b) Simulation of the access sequence M, M, A with a preemption before the second access to M. The second access to M results in an L2 hit.

Fig. 3. Simulation of the access sequence M, M, A: (a) without any preemptions and (b) with a preemption. The L1 and L2 caches are 2-way associative. The L2 cache has an additional set, which is not shown here, that X_2 is mapped to.

Let us now assume a preemption occurs after the first access to M. Let p denote that location. The preemption causes two blocks X_1 and X_2 to be loaded into the L1 set. X_1 is mapped to the same L2 set as M and A, while X_2 is mapped to a different L2 set. X_2 is thus not shown in the L2 cache in this depiction. The resulting sequence of cache states is shown in Figure 3(b). After the preemption, the second reference to M will not result in an L1 hit, as it has been evicted by X_1 and X_2. However, it will still result in an L2 hit, as only X_1 caused interference for M in the L2 cache. The preemption causes a delay of one L1 miss penalty due to the second reference to M and a full reload penalty for A.

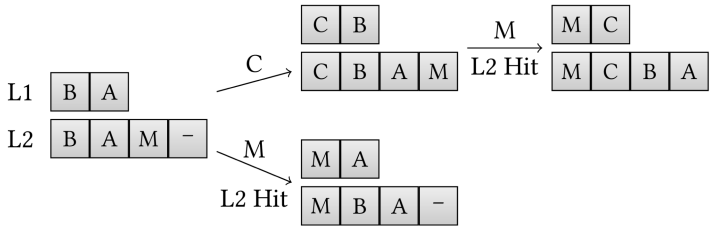
We will now compute the indirect effect on M using the state-of-the-art method. For this, we have to check all L1-UCBs at the site of preemption (see Algorithm 1, line 4). A is definitely cached in the L1 prior to the preemption and will be reused without eviction. Consequently, it is an L1-UCB. Furthermore, its maximal L1 LRU age is 1. The number of L1 ECBs is 2, meaning that A may be evicted due to the preemption from the L1 cache. It is thus a candidate for indirect interference with M. Algorithm 1 checks whether on the next access to A (line 6), the block M is cached in the L2 and has a lower LRU age compared with A (lines 8–9). This is the case in the unpreempted scenario. Hence, Algorithm 1 regards A as a source of indirect aging for M (line 10) and sets $Ind_M^p = \{A\}$. As the preemption happens directly before the next reference to M it follows that $ColInd_M^p = Ind_M^p = \{A\}$.

As M is an L1-UCB, it may contribute to the CRPD in two different ways. It may be degraded to an L2 hit or an L2 miss. The conditions for these penalties are given in Equations (2) and (3), respectively. As is the case for A, M may also be evicted from the L1 cache due to the L1 ECBs. When the execution is preempted, the maximal L2 age of M is 0. Thus, it may not be evicted from the L2 cache solely due to the L2 ECB X_1.

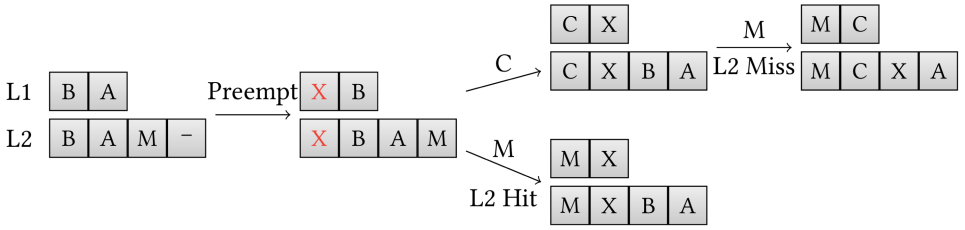
However, due to the potential indirect interference from A, as computed by Algorithm 2, M is considered to be potentially degraded to an L2 miss:

$$CU_2^p(M) + \widehat{ECB}_2^M + \left|ColInd_M^p\right| = 0 + 1 + 1 \geq 2. \tag{5}$$

Consequently, Equation (3) adds a delay of $(d_{L1} + d_{L2})$ cycles to the CRPD value due to a preemption at p, caused by the need to potentially reload M into both cache levels. However, it is clear that A will not create indirect interference for the second reference to M as A will only be accessed after M. In reality, the access to M will not result in an L2 miss but an L2 hit. This can be seen clearly in Figure 3(b). The state-of-the-art method overestimates the CRPD penalty in this situation because it does not consider the ordering of accesses when computing indirect interference.



(a) Analysing both paths without any preemptions shows that both first accesses to M would result in an L2 cache hit.



(b) Considering a preemption at the beginning of the sequence causes different behavior for the upper and lower path. On the upper path, the first access to M results in an L2 miss, while on the lower path the first access to M results in an L2 hit.

Fig. 4. A branch in the control flow creates multiple possible *first accesses* to M. There exist two different paths leading to first accesses of M, with different L2 cache ages. Both paths need to be considered to compute a safe CRPD estimate.

5.3 Analysis of Algorithm 3

The preemption delay caused by L2 hits being degraded to L2 misses is computed using Algorithm 3 in the state-of-the-art method. In this section, we will closely examine this algorithm and highlight two distinct issues.

5.3.1 Incompleteness of Algorithm 3. A key component of Algorithm 3 is $GetHitLocations^{p}(m_y)$, which returns a list of all locations containing a reference to m_y that result in an L2 hit and are reachable from p (line 4).

The first reachable reference is handled differently from the later references, as it may also be subject to direct interference from the loaded ECBs. However, the approach does not consider that there may be multiple references that potentially execute the first access to the block m_y after the preemption. This can happen if there is a branch in the control flow, creating two distinct paths to different first references of the cache block. Thus, there may be multiple execution traces that have a different reference as the first reference to the analyzed cache block.

Such a scenario is shown in Figure 4. In the example, the L1 cache is 2-way associative and the L2 cache has 4-way associativity. The initial state contains B, A in the L1 cache and B, A, M in the L2 cache. The control flow splits into the upper and lower branch. In the upper branch, an additional access to C is performed. Thus, there are two possible paths, which have an access sequence of C, M and M, respectively. If there are no preemptions, the reference to M results in an L2 hit on both paths, as can be seen in Figure 4(a). In Figure 4(b), the same access sequences are analyzed with a preemption happening before the control-flow branch. The preemption causes the block X to be stored in the L1 and L2 cache. This additional interference affects the upper and lower path in different ways. While in the lower path, the reference to M still results in an L2 hit, in the upper path M is evicted prior to the access.

As Algorithm 3 does not account for a situation with multiple possible first accesses, the example is not analyzable with the state-of-the-art method.

5.3.2 Handling of Uncertain L2 Accesses. Algorithm 3 treats the first reference to a memory block after a preemption differently from subsequent references. This distinction is motivated by the observation that after an access to a memory block, it will be younger than the ECB stored in the cache by the preemption. Thus, later accesses will only be subject to indirect interference.

Lemma 2 from Reference [36] tries to apply this observation and states that "[...] it is only the first reference to m_y after P [...] that can be directly impacted due to preemption at P." However, this justification for the structure of Algorithm 3 overlooks the fact that the first reference may not reach the L2 cache and the block m_y might not be refreshed. This situation occurs when the cache access classification of the first reference to m_y after P is *unknown*, i.e., m_y may be contained in the L1, but it is not certain that it is. If this situation occurs, not only the first reference may be impacted by the direct interference in the L2 cache. Also the reference that actually causes the first L2 access to the block m_y can be impacted by direct interference. As Algorithm 3 only checks if m_y may be evicted from the L2 cache at the first reference to m_y, it can underestimate the CRPD.

We will demonstrate this using an example. Consider the access sequence B, A, M, [A, B,] M, C, D, M. The two references [A, B] denote a split in the CFG. These references may be executed, but the control can also skip these accesses, e.g., by a conditional jump. We will now analyze this sequence in the following cache configuration: The caches are 2-way associative; the L1 cache has a single set, while the L2 cache contains two sets.

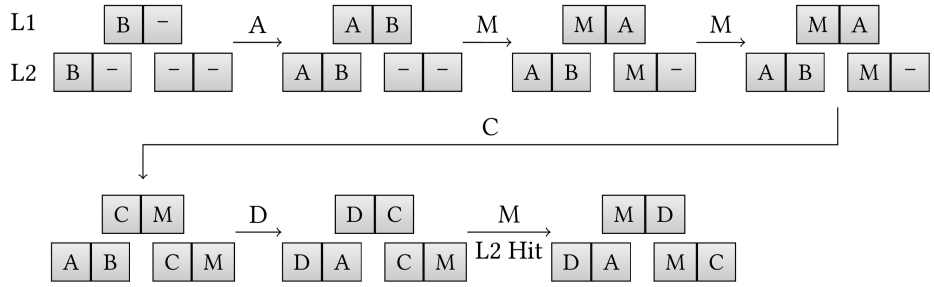
In Figure 5, the cache state simulation for an execution that skips the references [A, B] is shown. In Figure 5(a), the cache state sequence without any preemptions is shown. In between the two references to M, M is contained in the L1 cache, but it is not an L1-UCB. M is not considered to be an L1-UCB as the two optional references [A, B] may evict M from the L1 cache. Directly before the second access to M, M is not definitely cached in the L1 cache, prohibiting its classification as an L1-UCB. In both situations, regardless whether [A, B] is executed or not, M will be in the L2 cache. Thus, M is classified as an L2-UCB (see Definitions 4.3 and 4.4). As M will be in the L2 cache in all cases, the timing analysis will assume that the final reference to M results in an L2 hit. Thus, a potential L2 cache miss must be captured in the CRPD computation.

In Figure 5(b), the effects of a preemption prior to the second reference to M are shown. The preemption stores a single block X in the L1 cache and the second L2 cache set. The preemption does not evict M from the L1 cache. The second reference to M will result in an L1 hit. This situation creates the issue in Algorithm 3. As the first reference to M after preemption resulted in an L1 hit, M is not refreshed in the L2 cache. The direct interference created by X still exists in the L2 cache.

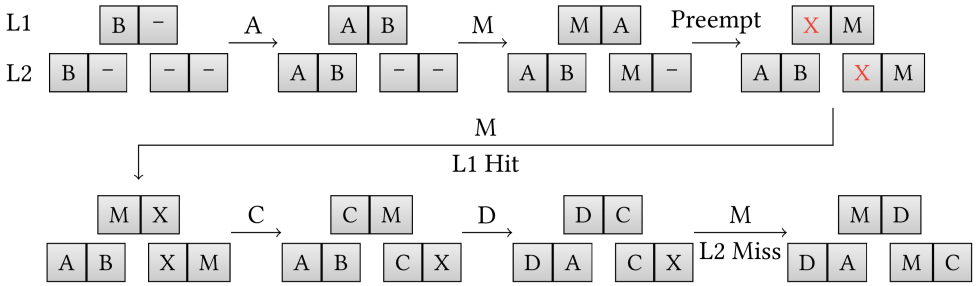
Finally, we can compare the behavior of the third access to M in the unpreempted and preempted scenario. In Figure 5(a), the third reference to M at the end of the sequence results in an L2 hit. However, for the preemption shown in Figure 5(b), as the direct interference still exists, the final access to M will cause an L2 miss. This potential L2 miss is not detected by Algorithm 3, because the direct interference of the preemption is only evaluated for the second reference to M. At that stage, M will not be evicted from the L2 due to the ECBs. However, the direct interference still applies to the third reference to M and causes an L2 miss, which leads to an unsafe CRPD bound for the shown preemption.

6 CRPD in Concrete Semantics

The previous section has demonstrated that the interaction between the L1 and L2 cache after a preemption is non-trivial. In order to be confident that an analysis does not overlook any edge case and thus computes a safe bound on the CRPD, a formal foundation for the key concepts is



(a) Simulation of the access sequence B, A, M, M, C, D, M without any preemptions. The third reference to M results in an L2 hit.



(b) Simulation of the access sequence B, A, M, M, C, D, M, with a preemption between the first and second reference to M. The direct interference by X contributes to the eviction of M prior to the third reference to M, causing an L2 miss.

Fig. 5. Simulation of the access sequence B, A, M, M, C, D, M: in (a) without preemption, in (b) with a preemption. The caches are 2-way associative; the L1 has one set and the L2 has two sets.

required. In this section, we formalize the indirect interference that may occur due to a preemption. To capture all possible interference scenarios, we operate at the level of concrete execution traces in this section. An execution trace is a sequence of program locations and the concrete state of both cache levels at those locations. The formalization established in this section lays the foundation for the safe CRPD analysis we will construct in the following Sections 7 and 8.

In a concrete situation, an LRU cache set contains W_l cache blocks, where l is the level of the cache and W_l is the associativity. The contained blocks are ordered by the LRU property. The cache state can thus be represented as a sequence of W_l blocks. Let C^l be the set that contains all possible concrete cache states:[1]

$$C^l = \left\{ (m_0, \ldots, m_{W_l-1}) \in \mathbb{M}^{W_l} \;\middle|\; i \neq j \implies m_i \neq m_j \right\}. \tag{6}$$

As cache sets operate independently of each other, we formalize the state of the system for a single cache set. The state of the two-level hierarchy can be represented using elements from $C^1 \times C^2$. If an access to the memory block m transforms $(c_1^1, c_1^2) \in C^1 \times C^2$ into $(c_2^1, c_2^2) \in C^1 \times C^2$ according to the semantics of a two-level non-inclusive LRU cache hierarchy, we write $(c_1^1, c_1^2) \rightarrow^m (c_2^1, c_2^2)$. Furthermore, we use $c^l(m)$ as a shorthand for the position of m in the concrete cache state c^l, i.e., the age of m; starting at 0, going up to $W_l - 1$. The value of $c^l(m)$ is ∞ if m is not contained in c^l.

[1]Note that a (partially) empty cache can be modelled by adding "empty" blocks to the set $\mathbb{M}$ to fill up the cache ways.

We represent the concrete state of the analyzed task using the set S:

$$S = \mathbb{P} \times C^1 \times C^2. \tag{7}$$

A tuple $(p, c^1, c^2) \in S$ corresponds to the system state directly before executing the access associated to p. In the state $(p, c^1, c^2) \in S$, the access associated to p, targeting the cache block m_p, results in an L1 hit if $c^1(m_p) < W_1$; an L2 hit if $c^1(m_p) = \infty \wedge c^2(m_p) < W_2$; an L2 miss otherwise.

For $s_1, s_2 \in S$ we write $(p_1, c_1^1, c_1^2) \rightarrow (p_2, c_2^1, c_2^2)$ iff $(p_1, p_2) \in E$ and $(c_1^1, c_1^2) \rightarrow^{m_{p_1}} (c_2^1, c_2^2)$, where E is the set of edges in the CFG. This allows us to create the set of all possible execution traces, starting from the initial program location $p^0 \in \mathbb{P}$ and arbitrary cache states c^1, c^2:

$$\mathcal{S} = \{(s_1, \ldots, s_n) \mid s_1 = (p^0, c^1, c^2) \in S \wedge 1 \leq i < n : s_i \rightarrow s_{i+1}\}. \tag{8}$$

6.1 Analyzing L1 Hits in Concrete Semantics

We can now define the notion of the first access in the concrete semantics.

Definition 6.1 (First Access in Concrete Semantics). In an execution trace $(s_1, \ldots, s_n) \in \mathcal{S}$, the first access to $m \in \mathbb{M}$ after p_i, $s_i = (p_i, c_i^1, c_i^2)$, is defined as the location p_j, $s_j = (p_j, c_j^1, c_j^2)$, with the smallest j, $i \leq j \leq n$ such that the reference associated to p_j targets m.

Using the concept of the first access, we can determine the indirect interference affecting accesses that result in an L1 hit in the absence of preemption.

Lemma 6.2. *The indirect interference in a trace $s = (s_1, \ldots, (p_i, c_i^1, c_i^2), \ldots, (p_k, c_k^1, c_k^2), \ldots, s_n) \in \mathcal{S}$ experienced by an L1 access to m associated to p_k due to a preemption at $p = p_i$ is bounded by $L1I_m^p$.*

$$L1I_m^p(s) = \left\{ m_j \in \mathbb{M} \;\middle|\; \begin{array}{l} p_k \text{ is the first access to } m \text{ after } p_i \text{ in } s \wedge \\ \forall j, i \leq j < k : \\ c_j^1(m_j) < W_1 \wedge c_j^1(m_j) + \widehat{ECB}_1^{m_j} \geq W_1 \wedge \\ c_j^2(m_j) > c_j^2(m_k) \end{array} \right\} \tag{9}$$

Proof. Indirect interference from a preemption at p_i for an access resulting in an L1 hit in the absence of preemptions is limited to the first reference of the memory block [10]. Any accesses happening after the preemption at location p_i and before the first reference to m in p_k may contribute to the indirect interference. For this reason, the index j is limited to $i \leq j < k$.

As noted by Rashid et al. in Reference [36], multiple preemptions prior to p_k may collaborate to increase the indirect interference. Thus, all states in the trace s between p_i and p_k are considered.

The condition for an access to cause indirect interference is that the targeted cache block resides in the L1 cache in the absence of preemptions and may be evicted by the preemptions. Furthermore, its L2 age has to be larger than the age of m_k to cause m_k's age to increase. Thus, $L1I_m^p$ is an upper bound on the indirect interference for the access associated to p_k in the particular trace s. □

We write $L1I_m^p(\mathcal{S})$ for $\bigcup_{s \in \mathcal{S}} L1I_m^p(s)$. We will use Lemma 6.2 to determine an upper bound on the aging from indirect interference over all feasible traces using a **data-flow analysis** (**DFA**) in Section 7.

6.2 Analyzing L2 Hits in Concrete Semantics

Similar to the notion of first access, we define the first L2 access and first L2 hit to a memory block. These definitions are needed to formalize the analysis of accesses that result in an L2 hit in the absence of preemptions.

Definition 6.3 (First L2 Access in Concrete Semantics). In an execution trace $(s_1, \ldots, s_n) \in \mathcal{S}$, the first L2 access to m after p_i, $s_i = (p_i, c_i^1, c_i^2)$, is defined as the location p_j, $s_j = (p_j, c_j^1, c_j^2)$, with the smallest $j, i \leq j \leq n$ such that the reference associated to p_j targets m and $c_j^1(m) = \infty$.

Definition 6.4 (First L2 Hit in Concrete Semantics). In an execution trace $(s_1, \ldots, s_n) \in \mathcal{S}$, the first L2 access to m after p_i, $s_i = (p_i, c_i^1, c_i^2)$, is defined as the location p_j, $s_j = (p_j, c_j^1, c_j^2)$, with the smallest $j, i \leq j \leq n$ such that the reference associated to p_j targets m and $c_j^1(m) = \infty \wedge c_j^2(m) < W_2$.

In order to identify which references will hit in the L2 cache and are possibly affected by direct interference from ECBs, we determine the set of reachable first L2 hits.

Definition 6.5 (Reachable First L2 Hits). For a location $p \in \mathbb{P}$, the set of reachable first L2 hits to block $m \in \mathbb{M}$ is given by

$$F_m^p = \left\{ p_j \in \mathbb{P} \;\middle|\; \begin{array}{l} \exists\, s = (s_1, \ldots, (p_i, c_i^1, c_i^2), \ldots, (p_j, c_j^1, c_j^2), \ldots, s_n) \in \mathcal{S} : \\ p_i = p \wedge \\ p_j \text{ is the first L2 access and first L2 hit for } m \text{ after } p_i \text{ in } s \end{array} \right\}. \tag{10}$$

Lemma 6.6. *The set F_m^p contains all locations that are reachable from $p \in \mathbb{P}$, where an access targeting $m \in \mathbb{M}$ and resulting in an L2 hit may experience direct and indirect interference in the L2 cache due to the preemption at p.*

Proof. There are several conditions in order for an access to be impacted by direct interference. The reference at p_j must reach the L2 cache and m must be contained in the L2 cache at the time of the preemption. Otherwise, m will not age due to the ECBs. m will be cached in the L2 at p_i because p_j performs the first L2 access to m after p_i and this access results in an L2 hit. This can only be the case if m was already cached in L2 at p_i. If p_j does not execute the first L2 access to m in the trace, another prior access will have eliminated the direct interference by refreshing m in the L2.

These conditions are evaluated for every feasible program trace. Thus, the set F_m^p contains all locations where direct and indirect interference may contribute to the eviction of m from the L2 cache. □

Similar to the notion of the reachable first L2 hits are the reachable later L2 hits. These accesses may still experience indirect interference from a preemption but are isolated from the direct effects of the ECBs as they are protected by another access that occurs first.

Definition 6.7 (Reachable Later L2 Hits). For a location $p \in \mathbb{P}$, the set of reachable later L2 hits to block $m \in \mathbb{M}$ is given by

$$L_m^p = \left\{ p_j \in \mathbb{P} \;\middle|\; \begin{array}{l} \exists\, s = (s_1, \ldots, (p_i, c_i^1, c_i^2), \ldots, (p_j, c_j^1, c_j^2), \ldots, s_n) \in \mathcal{S} : \\ p_i = p \wedge \\ p_j \text{ is not the first L2 access to } m \text{ after } p_i \text{ in } s \wedge \\ c_j^1(m) = \infty \wedge c_j^2(m) < W_2 \end{array} \right\}. \tag{11}$$

Lemma 6.8. *The set L_m^p contains all locations reachable from $p \in \mathbb{P}$, where an access targeting $m \in \mathbb{M}$, resulting in an L2 hit, may experience indirect interference without direct interference due to the preemption at p.*

Proof. All feasible program traces in $\mathcal{S}$ are considered to build the set L_m^p. A location p_j is included in the set only if the access associated to that location results in an L1 miss and an L2 hit. As p_j is guaranteed to not issue the first L2 access targeting m after p, m will have been refreshed

in the L2 cache previously. Thus, the access associated to p_j will not experience any direct interference. However, it may still be subject to indirect interference that occured since the last L2 access to m prior to p_j. □

In order to compute the CRPD from accesses classified as an L2 hit by the timing analysis, we have to check every location in F_m^p and L_m^p and observe whether it may result in an L2 miss due to preemption effects. We now bound the indirect effect of preemption for a particular execution trace $s \in \mathcal{S}$.

Lemma 6.9. *The indirect interference in a trace*

$$s = (s_1, \ldots, (p_i, c_i^1, c_i^2), \ldots, (p_j, c_j^1, c_j^2), \ldots, (p', c^1, c^2), \ldots, s_n) \in \mathcal{S}$$

experienced by an L2 access to m at p' due to a preemption at $p = p_i$ is bounded by $L2I_{p'}^p(s)$. Let P be the set of all locations p_j (accessing block m_j) after p_i, $i \leq j$, such that: p' is the first access to m after p_j and p_j is the first access to m_j after p_i.

$$L2I_{p'}^p(s) = \left\{ m_j \;\middle|\; \begin{array}{l} (p_j, c_j^1, c_j^2) \in P \wedge \\ c_j^1(m_j) < W_1 \wedge c_j^1(m_j) + \widehat{ECB}_1^{m_j} \geq W_1 \wedge \\ c_j^2(m_j) > c_j^2(m) \end{array} \right\}. \tag{12}$$

Proof. Indirect interference affecting the access at p' will accumulate only after the last L2 access to m prior to p'. For this reason, the states $(p_j, c_j^1, c_j^2) \in P$ are determined, such that the access from p' will be the first L2 access to m after p_j. Any accesses not associated to states in P will not contribute to the indirect interference. It will occur either after p' or the indirect interference will be eliminated by another access to m prior to p'. Note that an access to a memory block m_j will only contribute to the indirect interference if it results in an L1 hit in the absence of preemptions. Furthermore, m_j is only included in $L2I_{p'}^p$ if its cache age is larger than m in the L2 cache, as otherwise it will not cause m to age in the L2 cache. This condition is only checked for the first access to m_j after p_i, as later accesses will not be affected by the ECBs in the L1 cache. Thus, $L2I_{p'}^p(s)$ is an upper bound on the indirect interference for references resulting in an L2 hit in the absence of preemptions for the particular trace s. □

We write $L2I_{p'}^p(\mathcal{S})$ for $\bigcup_{s \in \mathcal{S}} L2I_{p'}^p(s)$. In Section 8.2, we utilize Lemma 6.9 to compute an upper bound on the aging from indirect interference for L2-UCBs.

7 CRPD Analysis of L1-UCBs

In this section, we will show how an upper bound on the aging from indirect interference for references to L1-UCBs can be computed. The state-of-the-art analysis is pessimistic regarding L1-UCBs, as we have discussed in Section 5.2. We present a novel analysis method for indirect interference on L1-UCBs that eliminates this pessimism. Note that while the previous section operated on concrete cache states and system traces, the analysis uses the efficient abstraction of potential cache states to the maximal block age [14] and thus computes CRPD bounds independent of the particular path taken through the CFG.

To determine the indirect interference for an L1-UCB m at p, we have to determine a safe bound on the aging caused by blocks in $L1I_m^p(\mathcal{S})$. To this end, we construct a DFA [1, p.597ff]. By formulating the problem as a DFA, we can directly solve the overestimation issue of the indirect interference Algorithm 1. The pessimism of the state-of-the-art algorithm originated from the fact that memory references occurring after the analyzed reference are considered to cause indirect

interference. This overestimation is avoided by our approach as the correct ordering of accesses is inherent in the DFA.

We use $\mathcal{P}(\mathbb{M})$ to denote the power set of $\mathbb{M}$. The domain $\mathcal{D}_{Ind}$ of the analysis is the function space of all mappings from a memory block to subsets of memory blocks:

$$\mathcal{D}_{Ind} = \{\mathbb{M} \to \mathcal{P}(\mathbb{M})\}\,. \tag{13}$$

An element $d_1 \in \mathcal{D}_{Ind}$ expresses which blocks may cause indirect interference to m_y on the next access to m_y by the value $d_1(m_y)$. Joining two elements of the domain is performed by taking the union of the interference sets:

$$d_1 \sqcup d_2 = \{(m, d_1(m) \cup d_2(m)) \mid m \in \mathbb{M}\}. \tag{14}$$

The analysis is performed in the backward direction. When formulating the DFA, we use the words *in* (*out*) to refer to *incoming* (*outgoing*) edges of a node in the sense of the regular control-flow.

The data-flow information d_m regarding a block m is transferred over a program location $p \in \mathbb{P}$ using the function $f^p(m, d_m)$, defined in Equation (15), where m_p is the memory block targeted by the reference associated to p.

$$f^p(m, d_m) = \begin{cases} \emptyset & \text{if } m_p = m \\ d_m \cup \{m_p\} & \text{else if } set_2(m_p) = set_2(m) \wedge \\ & \qquad MustAge_1^p(m_p) < W_1 \wedge \\ & \qquad MustAge_1^p(m_p) + \widehat{ECB}_1^{m_p} \geq W_1 \wedge \\ & \qquad MustAge_2^p(m_p) > MustAge_2^p(m) \\ d_m & \text{otherwise} \end{cases} \tag{15}$$

The first case in Equation (15) checks if the location p accesses the analyzed memory block m. If this is the case, we reset the potential for indirect interference to the empty set. This is possible, as the L1 cache is always accessed for every memory access. This means that after an access to the memory block m it is definitely the youngest block in the L1 cache. Thus, further accesses to this block may not result in an L1 miss due to a previous preemption. The indirect interference in the L2 cache need not be transferred for L1-UCBs beyond this point.

The second case checks whether the block m_p may cause indirect interference for m. For m_p to cause indirect interference, it has to be mapped to the same L2 set as m. Directly before the access to m_p it has to be contained in the L1 cache and the timing analysis must consider the access to result in an L1 hit, i.e., the maximal LRU age of m_p must be less than the number of ways in the L1 cache. Additionally, to cause indirect interference, m_p must potentially be evicted from the L1 cache by the ECBs. Finally, the age of m_p must be higher than m in the L2 cache to cause m to age. If these conditions are met, the set d_m is expanded by m_p. Otherwise, d_m is unaffected by the access to m_p.

To propagate information backward over a location p, the function f^p is applied to every memory block, as shown in Equation (16), while the information at an outgoing edge is computed from multiple successor blocks $p' \in succ(p)$ by joining the individual elements, as shown in Equation (17).

$$in_{IndL1}[p] = \{(m, f^p(m, d_m)) \mid (m, d_m) \in out_{IndL1}[p]\} \tag{16}$$

$$out_{IndL1}[p] = \bigsqcup_{p' \in succ(p)} in_{IndL1}[p'] \tag{17}$$

The initial value is the mapping that assigns every block the empty set. The values $in[p]$ and $out[p]$ are computed iteratively for all $p \in \mathbb{P}$ until a fixed point is reached. Termination of the analysis is guaranteed as the transfer function f^p is monotonic and the domain $\mathcal{D}_{Ind}$ is finite. Only a finite number of updates may be performed for each program location, thus ensuring termination.

Theorem 7.1. *Given a preemption that occurs at $p \in \mathbb{P}$: m will be in the L2 cache at the first access to m after p if $CU_2^p(m) + \widehat{ECB}_2^m + |in_{IndL1}[p](m)| < W_2$ holds.*

Proof. The value $CU_2^p(m)$ gives the maximal age of the block m in the L2 cache at the first access to m after p in the absence of preemptions. The direct aging of m due to the preemption at p is limited by $\widehat{ECB}_2^m$. Thus, $CU_2^p(m) + \widehat{ECB}_2^m$ is an upper bound on the L2 age of m including direct preemption effects.

A block m_j may only cause indirect interference to m if $m_j \in L1I_m^p(\mathcal{S})$ (see Lemma 6.2). In case $L1I_m^p(\mathcal{S}) \subseteq in_{IndL1}[p](m)$, the theorem holds due to Lemma 6.2. We show that for $m_j \in L1I_m^p(\mathcal{S}) \setminus in_{IndL1}[p](m)$ the age of m will not increase past $CU_2^p(m)$ due to m_j.

Assume that $\exists\, m_j \in L1I_m^p(\mathcal{S}) \setminus in_{IndL1}[p](m)$. This means that there exists a trace $s \in \mathcal{S}$ where $m_j \in L1I_m^p(s)$, i.e., the conditions listed in Equation (9) hold for m_j on the trace s. Let p_j denote the location where m_j is accessed.

We will now examine the three components of the second case in Equation (15):

(1) $MustAge_1^{p_j}(m_j) < W_1$
(2) $MustAge_1^{p_j}(m_j) + \widehat{ECB}_1^{m_j} \geq W_1$
(3) $MustAge_2^{p_j}(m_j) > MustAge_2^{p_j}(m)$.

As $m_j \notin in_{IndL1}[p](m)$, one of these conditions must be false. If the first part of the condition is false, the access to m_j will not be considered an L1 hit by the cache analysis and already contributes to $CU_2^p(m)$. m_j is thus not a source of indirect interference in this scenario.

The second part of the condition is directly implied by the fact that $m_j \in L1I_m^p(\mathcal{S})$ as $MustAge_1^{p_j}(m_j)$ is an upper bound on the L1 cache age of m_j at p_j on any trace. It will never be false given $m_j \in L1I_m^p(\mathcal{S})$.

If the third part of the condition is false, there exists another trace $s' \in \mathcal{S}, s' \neq s$ that contains the same references to m_j and m as s, where m is older than m_j in the L2 cache. In this situation, an L1 cache miss to m_j will not cause m to age in the L2 cache as m is already older than m_j. The potential interference of m_j toward m is already captured in the value $CU_2^p(m)$.

Hence, the interference from m_j is accounted for either in $CU_2^p(m)$ or by $m_j \in in_{IndL1}[p](m)$ Consequently, there is no memory block $m_j \in L1I_m^p(\mathcal{S}) \setminus in_{IndL1}[p](m)$, that could increase the age of m beyond $CU_2^p(m) + \widehat{ECB}_2^m + |in_{IndL1}[p](m)|$. We conclude that $CU_2^p(m) + \widehat{ECB}_2^m + |in_{IndL1}[p](m)| < W_2$ is a sufficient condition to guarantee that m is contained in the L2 cache at the first reference to m after the preemption at p. □

We modify Equations (2) and (3) to consider the tighter bound on indirect interference as follows. Instead of the value $ColInd_m^p$ given by Algorithm 2, the value $in_{IndL1}[p](m)$ is used to determine whether a memory block may be evicted from the L2 cache.

$$\delta_{L1}^p = d_{L1} \cdot \left| \left\{ m \in UCB_1^p \;\middle|\; CU_1^p(m) + \widehat{ECB}_1^m \geq W_1 \wedge CU_2^p(m) + \widehat{ECB}_2^m + |in_{IndL1}[p](m)| < W_2 \right\} \right|, \tag{18}$$

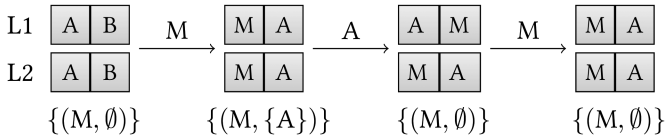


Fig. 6. Example for the indirect interference DFA for L1 cache hits.

$$\delta^p_{L1+L2} = (d_{L1} + d_{L2}) \cdot \left| \left\{ m \in UCB^p_1 \;\middle|\; CU^p_1(m) + \widehat{ECB}^m_1 \geq W_1 \wedge CU^p_2(m) + \widehat{ECB}^m_2 + |in_{IndL1}[p](m)| \geq W_2 \right\} \right|. \tag{19}$$

It follows from Theorem 7.1 that $\delta^p_{L1} + \delta^p_{L1+L2}$, as defined in Equations (18) and (19), is a safe bound on the CRPD resulting from L1 hits being degraded to L2 accesses due to a preemption at p.

Figure 6 illustrates the execution of the DFA on the scenario used in Figure 1, which introduced indirect interference. As before, we consider a preemption with a single interfering cache block. The data-flow information is annotated below the cache states, showing the information concerning the cache block M. The information is propagated backward along the edges, beginning at the rightmost state with the empty set. As it is propagated to the left, the indirect interference information for M remains the empty set. This corresponds to the first case of the transfer Equation (15). In the next propagation step, the set of cache blocks potentially causing indirect interference is updated to {A}. This update occurs because the second case of Equation (15) applies: A is older in the L2 cache than M, and A may be evicted from the L1 cache by a preemption at this stage. The access to A can cause indirect interference and is consequently added to the data-flow information. When propagating the information to the leftmost state, block A is removed again from the indirect interference set, since the access to M refreshes it in the L1 cache. A preemption at this point would not create indirect interference from A toward M, because the first L1 access to M happens before the access to A.

8 CRPD Analysis for L2 UCBs

In this section, we will show how the preemption penalty stemming from L2-UCBs can be computed, even if there are multiple references that potentially cause the first L2 access to a particular cache block after the preemption and accesses may not reach the L2 cache in every situation.

First, we construct a backward DFA that collects all reachable references that result in an L2 hit for every program location $p \in \mathbb{P}$ in Section 8.1. Then a bound on indirect interference for L2-UCBs is introduced in Section 8.2. Finally, in Section 8.3, an algorithm to compute the CRPD from all references to L2-UCBs is presented.

8.1 Determining L2 Access Locations

To analyze how much CRPD can originate from references that are classified as L2 hits in the absence of preemption, we have to determine where these references are located. We solve this problem by creating a backward DFA.

Only the first access to a block in the L2 cache after a preemption experiences the direct interference due to ECBs [36]. However, there may be multiple references that potentially cause the first access to the L2 cache. To safely estimate the CRPD, we have to consider all candidates for the first access. For this reason, we define the set *Loc* in Equation (20).

$$Loc = \{(f, l) \mid f, l \subseteq \mathbb{P}\}. \tag{20}$$

Loc contains pairs of program location subsets. The first element f of a pair $(f, l) \in Loc$ corresponds to the possible first L2 access locations after a preemption, while l contains all locations that may issue the second or later access to the L2 cache. Joining two tuples (f_1, l_1) and (f_2, l_2) is realized by the union of the two sets in both tuples:

$$(f_1, l_1) \sqcup (f_2, l_2) = (f_1 \cup f_2, l_1 \cup l_2). \tag{21}$$

In order to keep track of all L2 hit locations for all memory blocks, we define the DFA domain $\mathcal{D}_{FL}$ as the function space of mappings from every memory block to an element of *Loc*:

$$\mathcal{D}_{FL} = \{\mathbb{M} \rightarrow Loc\}. \tag{22}$$

When a reference resulting in an L2 hit to $m \in \mathbb{M}$ is encountered during the DFA, the mapped element $d(m), d \in \mathcal{D}_{FL}$ is modified. The function $t^p_m((f, l))$, defined in Equation (23), updates the tuple (f, l) according to the access at the location p. We write $CAC(p)$ to denote the CAC of the reference associated to p. A CAC value of A shows that the second level cache is always accessed, i.e., it is an L1 miss; a CAC value of N signifies that the L2 cache is never accessed, while the CAC value U stands for an unknown access behavior.

$$t^p_m((f, l)) = \begin{cases} (f, l) & \text{if } m_p \neq m \vee CAC(p) = N \\ (\emptyset, f \cup l) & \text{else if } CAC(p) = A \wedge MustAge^p_2(m) \geq W_2 \\ (f, f \cup l) & \text{else if } CAC(p) = U \wedge MustAge^p_2(m) \geq W_2 \\ (\{p\}, f \cup l) & \text{else if } CAC(p) = A \wedge MustAge^p_2(m) < W_2 \\ (f \cup \{p\}, f \cup l) & \text{else if } CAC(p) = U \wedge MustAge^p_2(m) < W_2 \end{cases} \tag{23}$$

In the first case, the mapped value is not changed. t^p_m leaves (f, l) unchanged under two conditions: (1) the reference associated to p targets a different memory block than m, or (2) the access never reaches the L2 cache.

The next two cases match if the reference associated to p targets the block m and will not result in a definite L2 hit. The difference between these two cases is whether the access will always reach the L2 cache or whether it might be processed at the L1 cache as a hit. If the access will definitely miss in the L1 cache and always reach the L2 cache, the function t^p_m will set the first access locations to the empty set and add the previous set of first access locations to the set of later accesses. This is done as no L2 hits may be affected by direct interference. The direct interference will be eliminated by the currently analyzed L2 miss.

Otherwise, if the access may hit in the L1 cache, we have to consider both situations. Either the access results in an L1 hit or an L1 miss. For an L1 hit, we leave the sets of first and later L2 hits unchanged (f, l). In the situation of the L1 miss, we get the same result as in the previous case $(\emptyset, f \cup l)$. Joining these two potential results gives us $(f, l) \sqcup (\emptyset, f \cup l) = (f, f \cup l)$.

The fourth case considers accesses that will definitely miss in the L1 cache and hit in the L2 cache. In this situation, the current location p becomes the location of the first L2 hit and all previous first hits are added to the set l, which contains those accesses that may only be affected by indirect interference.

In case the access may not reach the L2 cache but will result in an L2 hit if it does, we have to consider multiple scenarios. The first scenario is that the access reaches the L2 cache, this results in the tuple $(\{p\}, f \cup l)$ as in the previous case. In the second scenario, the access does not reach the L2 cache; the tuple of hit locations is thus not modified and remains (f, l). Joining the results of these scenarios gives us $(f \cup \{p\}, f \cup l)$.

The data-flow information is transferred over a location $p \in \mathbb{P}$ by applying the function t^p_m to every value (m, d_m) contained in the outgoing information:

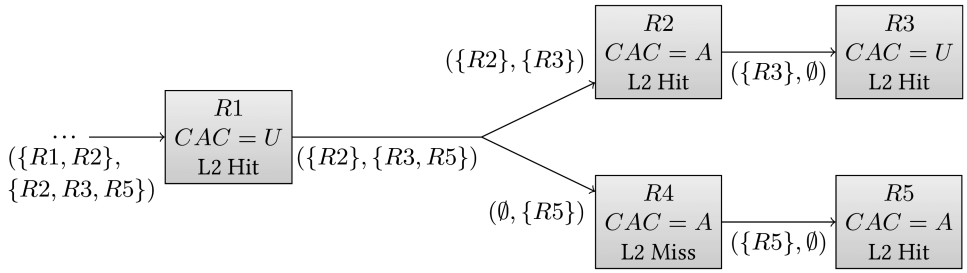


Fig. 7. Example for the first access location DFA on a simplified CFG.

$$in_{FL}[p] = \{(m, t^p_m(d_m)) \mid (m, d_m) \in out_{FL}[p]\}. \tag{24}$$

To compute the data-flow information at the outgoing edge of a node, the information from all successor nodes are joined (25), where two mappings are joined according to Equation (26).

$$out_{FL}[p] = \bigsqcup_{p' \in succ(p)} in_{FL}[p'], \tag{25}$$

$$d_1 \sqcup d_2 = \{(m, d_1(m) \sqcup d_2(m)) \mid m \in \mathbb{M}\}. \tag{26}$$

The initial information for each node is the empty mapping, which assigns each cache block a pair of empty sets. Using this DFA, it is possible to determine for each program location p which references to a memory block are L2 hits and can suffer from direct and/or indirect interference.

Theorem 8.1. *For a memory block m, the least fixed point (f, l) of $in_{FL}[p](m)$, computed by the data-flow analysis, contains all locations with references to m, that may contribute to the CRPD by being degraded from an L2 hit to an L2 miss. References associated to locations $r \in f$, may be subject to direct and indirect interference; references associated to locations $r \in l$ may be subject only to indirect interference.*

Proof. F^p_m contains all potential first L2 accesses to m after p that may be subject to direct and indirect interference (Lemma 6.6). For an access to contribute to the CRPD, it has to be considered a hit in the timing analysis. Thus, only references $r \in F^p_m$ with $MustAge^r_2(m) < W_2$ have to be considered when computing the CRPD.

Assume that $r \in F^p_m$. We know that there exists a trace $s \in \mathcal{S}$ that traverses the state (p, c^1_p, c^2_p) and continues to location (r, c^1_r, c^2_r), where the access results in an L2 hit. Hence, it is true that $c^1_r(m) = \infty \wedge c^2_r(m) < W_2$. If $MustAge^p_2(m)$ were to be greater or equal to W_2, the access would not be considered an L2 hit by the timing analysis and could not contribute to the CRPD. Thus, m is added to the set of first accesses when propagating the data-flow information over (r, c^1_r, c^2_r), as the fourth or fifth case of Equation (23) applies. Furthermore, as the access associated to r is the first L2 access to m, there is no intermediate state (q, c^1_q, c^2_q), that performs an L2 access to m that always reaches the L2 cache. Hence, for all intermediate states the first, third, or fifth case of the propagation function t^p_m applies. It follows that r is not removed from the set of first hit locations. As no elements are removed from the set on a join operation it follows that $r \in f$. The proof for references $r \in L^p_m$ is analogous, using Lemma 6.8 and the addition of an intermediate state that causes r to be added to the set of later L2 hits. □

Thus, based on the value of $in_{FL}[p]$, we can compute a safe estimate of the CRPD for a preemption happening at p.

We demonstrate the functionality of the DFA in Figure 7 on a CFG, which is simplified to only show references to a single cache block. Data-flow information for the targeted cache block is

annotated at the edges. The accesses in the CFG are named $R1$ through $R5$, and their respective access and hit classification are listed in the corresponding node.

The DFA is performed in the backward direction, meaning that the information is propagated from $R3$ and $R5$ toward $R1$. On the edge toward $R3$, $R3$ is the only potential first L2 cache hit for the analyzed cache block, with no additional L2 cache hits reachable from this location. Thus, the data-flow information is equal to $(\{R3\}, \emptyset)$. The same applies for the edge leading towards $R5$.

When the data-flow information is propagated over $R2$, $R3$ is replaced as the first L2 hit and is moved to the set of later L2 hits, while $R2$ is the new first hit: $(\{R2\}, \{R3\})$. This update corresponds to the fourth case in equation (23). Since the access $R4$ can result in a cache miss, it is not considered as a potential source of delay in the CRPD computation. However, the reference $R5$ is moved to the second set, which contains L2 cache hits unaffected by direct interference, as the CAC of $R4$ is A, which means that $R4$ will definitely miss in the L1 cache and refresh the cache block in the L2 cache. A preemption before $R4$ will not thus not create direct interference for $R5$. Case 2 of Equation (23) covers this scenario, resulting in: $(\emptyset, \{R5\})$.

The information of the upper and lower branch are joined by performing the set union, as defined in Equation (21). Finally, the tuple $(\{R2\}, \{R3, R5\})$ is propagated backward over $R1$. As $R1$ will result in an L2 hit, but the CAC is U, the tuple is updated to $(\{R1, R2\}, \{R2, R3, R5\})$, as defined in the fifth case of Equation (23). This means that a preemption at this point can impact $R1$ and $R2$ directly, while $R2$, $R3$, and $R5$ can only be affected by indirect interference.

8.2 Indirect Interference for L2 Hits

The analysis of indirect interference on L1-UCBs as presented in the previous Section 7 is unsuited to compute the indirect interference for references that are considered L2 hits. This results from the fact that it is not sufficient to only check the preemption effects on the first L2 access after the preemption for L2-UCBs. Even after multiple accesses to a memory block m, the indirect preemption effects can evict m from the L2 cache and thus cause a cache miss for the next access. For this reason, we have to compute an upper bound on the indirect interference for every reachable reference targeting m that results in an L2 hit. As we have determined the location of these references in the previous Section 8.1, we can now focus on determining an upper bound on the indirect interference.

Determining such an upper bound can be approached from different perspectives. One approach would be to modify the DFA from Section 7 to operate not on memory blocks but references and adjust the transfer function accordingly. This would allow us to compute an upper bound on the indirect interference for each individual reference. However, this approach scales poorly as the number of reachable L2 hits can be significantly larger than the number of cache blocks (e.g., due to virtual loop unrolling) and the bound has to be computed for every reachable reference that results in an L2 hit and every possible preemption location. The required computational effort of this approach would thus be much larger than for the DFA from Section 7.

We solve this problem by determining an upper bound on the indirect interference for every reference resulting in an L2 hit, irrespective of the preemption location. This approach can be implemented using an algorithm that iterates over every program location $p \in \mathbb{P}$ once and accumulates all potential indirect interference effects for every potential first L2 hit. For each location $p \in \mathbb{P}$, we only have to consider the first L2 hits reachable from p in this algorithm, because for later L2 hits the indirect interference will have been removed by a previous L2 access.

Algorithm 4 computes the indirect interference on L2 hits and starts by setting the indirect interference to the empty set for all references (lines 1–3). Then, for every program location it is determined whether the access on that location can cause indirect interference to another block (lines 5–8). To cause indirect interference, the blocks must be mapped to the same L2 cache set

ALGORITHM 4: Indirect Interference on L2 Hits

Result: The indirect interference for references that result in L2 hits.

1 **for** $p' \in \mathbb{P}$ **do**
2 $\quad IndL2_{p'} \leftarrow \emptyset$
3 **end**
4 **for** $p \in \mathbb{P}$ **do**
5 $\quad$ **for** $m \in \mathbb{M}$ **do**
6 $\quad\quad$ **if** $m \neq m_p \wedge set_2(m) = set_2(m_p) \wedge$
7 $\quad\quad\quad MustAge_1^p(m_p) < W_1 \wedge MustAge_1^p(m_p) + \widehat{ECB}_1^{m_p} \geq W_1 \wedge$
8 $\quad\quad\quad MustAge_2^p(m_p) > MustAge_2^p(m)$ **then**
9 $\quad\quad\quad (f, l) \leftarrow in_{FL}[p](m)$
10 $\quad\quad\quad$ **for** $p' \in f$ **do**
11 $\quad\quad\quad\quad IndL2_{p'} \leftarrow IndL2_{p'} \cup \{m_p\}$
12 $\quad\quad\quad$ **end**
13 $\quad\quad$ **end**
14 $\quad$ **end**
15 **end**

(line 6). Furthermore, it is checked whether m_P is contained in the L1 cache, is considered a definite hit by the *must* age analysis, and may potentially be evicted due to the ECBs (line 7). The block is only considered to cause indirect interference if the L2 *must* age of m_p is greater to that of m (line 8). If these conditions hold, the block m_p is added to the indirect interference estimate for every reachable first L2 hit to m (lines 9–12). The result of the algorithm is the set $IndL2_{p'}$ for each $p' \in \mathbb{P}$ that is considered an L2 hit. It contains all memory blocks that can potentially cause indirect interference for the reference at p'.

Theorem 8.2. *Given a preemption that occurs at $p \in \mathbb{P}$:*

For a reference at location $p' \in f, (f, l) = in_{FL}[p](m)$, m will be contained in the L2 cache at p' if $MustAge_2^{p'}(m) + \widehat{ECB}_2^m + |IndL2_{p'}| < W_2$ holds.

For a reference at location $p' \in l, (f, l) = in_{FL}[p](m)$, m will be contained in the L2 cache at p' if $MustAge_2^{p'}(m) + |IndL2_{p'}| < W_2$ holds.

Proof. The proof is very similar to the proof of Theorem 7.1.

We show that it holds for $p' \in f$. The value $MustAge_2^{p'}(m) + \widehat{ECB}_2^m$ is an upper bound on the cache age of m solely based on the direct effect of preemption. A block m_h may only cause indirect interference to m if $m_h \in L2I_{p'}^p(\mathcal{S})$ (see Lemma 6.9) and the access to m_h is considered an L1 hit in the absence of preemptions. In case $L2I_{p'}^p(\mathcal{S}) \subseteq IndL2_{p'}$, the theorem holds due to Lemma 6.9.

For $m_h \in L2I_{p'}^p(\mathcal{S}) \setminus IndL2_{p'}$, the age of m will not increase over $MustAge_2^{p'}(m)$ due to m_h. For m_h to be contained in $L2I_{p'}^p(\mathcal{S}) \setminus IndL2_{p'}$, there must be a trace s, where on accessing m_h, the L2 age of m_h exceeds the L2 age of m (see the final condition in Equation (12)). m_h will not be added to $IndL2_{p'}$ if at the location p_h where m_h is referenced $MustAge_2^{p_h}(m_h) \leq MustAge_2^{p_h}(m)$ holds. Thus, there exists a different trace $s' \in \mathcal{S}, s' \neq s$, where m is older in the L2 cache than m_h. On the trace s', m_h contributes to the *must* age of m. This interference is captured by $MustAge_2^{p'}(m)$. Consequently, $MustAge_2^{p'}(m) + \widehat{ECB}_2^m + |IndL2_{p'}| < W_2$ is a sufficient condition for the access at p', targeting m, to result in an L2 hit.

The proof for the second part of the theorem for $p' \in l$ is analogous. □

ALGORITHM 5: CRPD from L2 Cache Misses due to a Preemption at p

Result: The delay suffered from L2 hits that are degraded to L2 misses due to preemption.

1 $\delta^p_{L2} \leftarrow 0$
2 **for** $m \in \widehat{UCB}^p_2$ **do**
3 $\quad \delta_m \leftarrow 0$
4 $\quad (f, l) \leftarrow in_{FL}[p](m)$
5 $\quad$ **if** $\exists\, p' \in f : MustAge^{p'}_2(m) + \widehat{ECB}^m_2 + \left|IndL2_{p'}\right| \geq W_2$ **then**
6 $\quad\quad \delta_m \leftarrow \delta_m + 1$
7 $\quad$ **end**
8 $\quad$ **for** $p' \in l$ **do**
9 $\quad\quad$ **if** $MustAge^{p'}_2(m) + \left|IndL2_{p'}\right| \geq W_2$ **then**
10 $\quad\quad\quad \delta_m \leftarrow \delta_m + 1$
11 $\quad\quad$ **end**
12 $\quad$ **end**
13 $\quad \delta^p_{L2} \leftarrow \delta^p_{L2} + (\delta_m \cdot d_{L2})$
14 **end**

8.3 CRPD Computation

The locations of references that may be impacted by direct and indirect effects or only indirect effects can be determined using the DFA from Section 8.1. Using the analysis of indirect interference from the previous Section 8.2, it is now possible to compute a bound on the CRPD. Taking the information from these two components, we construct Algorithm 5 to compute a safe estimate on the CRPD from L2 hits that does not suffer from the safety issues discussed in Section 5. In contrast to Algorithm 3, this Algorithm is capable of dealing with situations where multiple references to the same block may be the first reference to be executed after a preemption. We denote the CRPD contribution of L2 hits being degraded to L2 misses due to a preemption at p by δ^p_{L2}.

Algorithm 5 iterates over all memory blocks that are L2-UCBs or may become L2-UCBs in a location reachable from P (lines 2–14). This is done as these are exactly those blocks that may contribute to the CRPD due to a degradation from an L2 hit to an L2 miss. The CRPD contribution from each $m \in \widehat{UCB}^p_2$ is denoted by δ_m in Algorithm 5.

Instead of determining a singular reference to m that may be executed first after p, we utilize the DFA from Section 8.1, to determine all locations which are reachable from p and access m resulting in an L2 hit (line 4). The set f contains references that may be subject to direct interference from ECB, due to the preemption at p. Notably this includes also those references which occur after another reference to m that has a CAC of U (see Equation (23)). The set l contains references that may be subject to eviction solely due to the indirect effect. Note that f and l are not necessarily disjoint.

In lines 5–7, the algorithm checks whether there is a reference to m in f that may be evicted by the collaboration of direct and indirect preemption effects. m may be evicted from the L2 cache at $p' \in f$ if its maximal LRU age at p' plus the direct interferences of $\widehat{ECB}^m_2$ ECB and the indirect interference is greater or equal to the number of ways in the L2 cache (see Theorem 8.2). As there is at most one reference that can be affected by the direct effects of the preemption, it is sufficient to check for the existence of such a reference. After the first L2 access to m after a preemption, the block m will be refreshed in the L2 cache. Thus, it will be younger than all ECB. Consequently, future accesses to m are no longer affected by those ECBs. If there exists a potential first access that is degraded to an L2 miss due to the preemption, δ_m is increased by 1.

The loop in lines 8–12 checks all references in the set l. The access may result in an L2 miss if the maximal LRU age of m plus the indirect interference exceeds the number of L2 ways. For each potential L2 miss, the variable δ_m is incremented by 1. The total CRPD from L2 cache hits is then given by the sum of $\delta^p_{L2} = \sum_{m \in \widehat{UCB^p_2}} \delta_m \cdot d_{L2}$ (line 13).

An upper bound on the CRPD can thus be computed by taking the maximal sum of $\delta^p_{L1} + \delta^p_{L1+L2} + \delta^p_{L2}$ for any program location $p \in \mathbb{P}$:

$$\delta = \max_{p \in \mathbb{P}} \left(\delta^p_{L1} + \delta^p_{L1+L2} + \delta^p_{L2} \right). \tag{27}$$

As the value δ varies based on the preempted task and preempting tasks, we denote the CRPD of task τ being preempted by φ and all higher priority tasks $\psi \in hp(\varphi)$ as $\delta_{\tau,\varphi}$. The WCRT of a task τ can be computed iteratively using Equation (28) [9]. If the value $WCRT^{i+1}_\tau$ exceeds the deadline of τ, the iteration can be stopped as the task, and by extension the whole system, is not schedulable.

$$WCRT^{i+1}_\tau = WCET_\tau + \sum_{\varphi \in hp(\tau)} \left\lceil \frac{WCRT^i_\tau}{Period(\varphi)} \right\rceil \cdot \left(WCET_\varphi + \delta_{\tau,\varphi} \right). \tag{28}$$

9 Evaluation

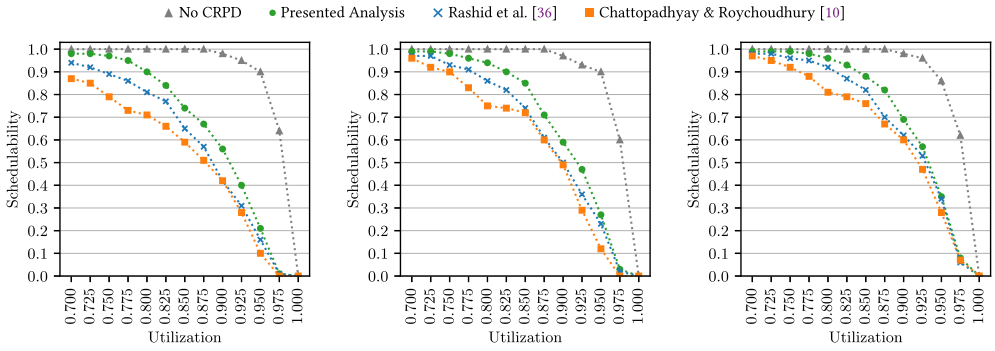
To evaluate the performance of the presented analysis, we integrated it in a WCET-aware C Compiler [13]. We compare the performance of the presented analysis to the two other approaches for non-inclusive caches: Chattopadhyay and Roychoudhury [10], which we refer to as the baseline analysis, as well as the state-of-the-art from Rashid et al. [36].

We modified the state-of-the-art analysis to be able to handle situations with multiple possible first accesses. This was done as every analyzed task set contained at least one situation in which there are multiple possible first references or first L2 accesses. As the state-of-the-art [36] approach does not consider such situations, the analysis failed, preventing us from creating a meaningful comparison to the presented analysis. We used the presented Algorithm 5 instead of Algorithm 3 to compute the contribution of L2 hits to the CRPD. We adapted Algorithm 5 for this application to use the indirect interference values as computed by the state-of-the-art. Furthermore, we modified Algorithm 1 to iterate over all potential first accesses (cf. line 6 in Algorithm 1) and accumulate the results for every potential first access. All other details of the approach were implemented as defined in [36]. These adjustments allowed us to compare the performance of the state-of-the-art approach to the presented analysis, even in the presence of multiple first accesses to a particular block (see Section 5.3.1). Note that even with these modifications, the analysis [36] is not safe as the issue described in Section 5.1 persists: the analysis does not account for the CAC of L2 cache accesses and can underestimate the indirect interference.

The target architecture in our evaluation consists of a single core, with a two-level instruction cache hierarchy. The core features a 3-stage in-order pipeline using the ARMv4T instruction set. Both cache levels use the LRU replacement policy and have a cache block size of 64 bytes. The access timings are 1 cycle for an L1 hit, 10 cycles for an L2 hit and 100 cycles for an L2 miss. The same timings were used in the evaluation of the state-of-the-art approach [36] and are typical timings according to the reference manual of the ARM PL310 cache controller [27]. We abbreviate this timing configuration as the 1/10/100 timing.

To increase the analysis precision, we activated virtual-inlining and virtual-unrolling with a maximal context number of 3. This means, that the analyzer differentiates between the first, second, and all following iterations of a loop.

As the benchmarking programs, we used tasks from the MRTC benchmark suite [19], which are also used in other literature for two-level CRPD analyses [10, 36, 47]. We randomly generated 100



(a) 1-way L1, 2-way L2 (b) 2-way L1, 4-way L2 (c) 4-way L1, 8-way L2

Fig. 8. Schedulability of 100 analyzed systems with 10 tasks per core with 1 KB L1 and 4 KB L2 cache. The y-axis shows the ratio of schedulable systems for different utilization values on the x-axis. (a) shows the results for 1-way L1 and 2-way L2 cache; (b) for 2-way L1 and 4-way L2 cache; (c) for 4-way L1 and 8-way L2 cache.

task sets containing 10 tasks. The task periods were generated using the UUnifast approach [8]. We used *rate-monotonic* scheduling to assign a priority to every task [30].

The performance of the different approaches was measured using the schedulability fraction, which describes how many of the analyzed task sets were schedulable according to Equation (28). We present our results in Figures 8–10. On the x-axis the utilization value u is shown. It is computed as $u = \sum_{\tau \in T}(WCET(\tau)/Period(\tau))$, where $WCET(\tau)$ is the WCET of τ and $Period(\tau)$ is its period. We evaluated utilization values between 0.7 and 1.0. The y-axis shows the fraction of schedulable task sets. The schedulability without considering the CRPD is depicted by gray triangles, the presented analysis as green dots, the state-of-the-art [36] as blue crosses, and the baseline [10] as orange squares.

In Figure 8, the evaluation results for a 1 KB L1 and 4 KB L2 cache are shown. We evaluated three different settings for the cache associativity. Figure 8(a) corresponds to a direct mapped L1 cache and a 2-way set associative L2 cache. In Figure 8(b), the associativity is set to 2-ways for the L1 and 4-ways for the L2 cache. The highest evaluated associativity is shown in Figure 8(c), where the L1 cache is 4-way associative and the L2 cache is 8-way associative. We call these parameter settings the low, medium, and high associativity configuration. The gray markers show the theoretical upper limit for the schedulability, if no CRPD would occur. The baseline analysis [10], shown in orange, produced the lowest schedulability ratio of the three approaches. We use percentage points (pp) to measure the increase in schedulability. For example, increasing the number of schedulable systems from 40 to 50, out of the 100 total systems, corresponds to a 10 percentage point increase.

The state-of-the-art approach [36], shown in blue, improved the average schedulability by 6.2 pp, 4.6 pp, and 4.3 pp for the low, medium, and high associativity setting over the baseline.

Additionally, it can be seen that the presented analysis yielded significant improvements over the state-of-the-art. We observed an average increase in schedulability of 6.9 pp, 5.8 pp, and 3.8 pp for the low, medium, and high associativity setting over the state-of-the-art. The maximal increase in task set schedulability lies between 11 pp and 14 pp depending on the cache associativity. The largest improvement of 14 pp was observed at $u = 0.9$ with low associativity, as seen in Figure 8(a). Using the state-of-the-art analysis, 42 systems are schedulable, while 56 systems are schedulable when computing the CRPD using the presented analysis.

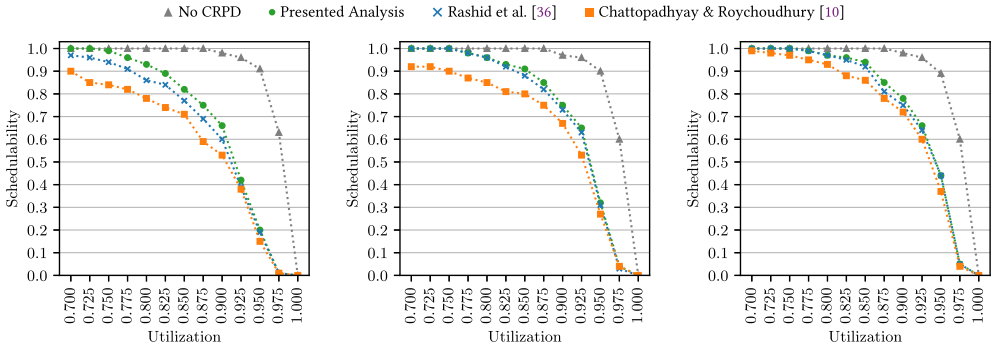
Table 1 shows the decrease of the CRPD values compared with the state-of-the-art. Note in particular that the minimal reduction for the medium associativity is negative. This means that

Table 1. CRPD Reduction Over the State-of-the-Art for 1KB L1 / 4KB L2 Caches

Associativity (L1 / L2)	Maximal Reduction	Average Reduction	Minimal Reduction
1 / 2	75.0%	6.7%	0%
2 / 4	90.9%	6.2%	−5.5%
4 / 8	84.2%	3.4%	0%

Table 2. CRPD Reduction Over the State-of-the-Art for 2KB L1 / 8KB L2 Caches

Associativity (L1 / L2)	Maximal Reduction	Average Reduction	Minimal Reduction
1 / 2	82.8%	2.7%	0%
2 / 4	76.5%	2.3%	−3.9%
4 / 8	71.7%	1.7%	−0.7%



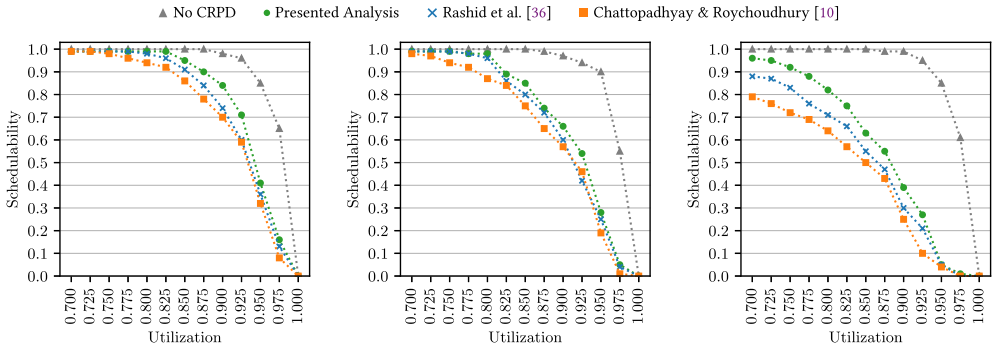
(a) 1-way L1, 2-way L2 (b) 2-way L1, 4-way L2 (c) 4-way L1, 8-way L2

Fig. 9. Schedulability of 100 analyzed systems with 10 tasks per core with 2 KB L1 and 8 KB L2 cache. The y-axis shows the ratio of schedulable systems for different utilization values on the x-axis. (a) shows the results for 1-way L1 and 2-way L2 cache; (b) for 2-way L1 and 4-way L2 cache; (c) for 4-way L1 and 8-way L2 cache.

the CRPD value computed by the presented analysis was 5.5% higher than the value given by the state-of-the-art method. This increase occurred in a system where the task *adpcm* is preempted by the task *bs*. The CRPD increased from 1629 to 1719 cycles. The increased CRPD value could be due to the unsafety of the state-of-the-art, but it may also be the case that the presented analysis is slightly more pessimistic in this particular situation.

In Figure 9, the evaluation results for a 2 KB L1 cache and 8 KB L2 cache are shown. As for the smaller cache size, we analyzed three different associativity settings in Figure 9(a)–9(c). The state-of-the-art yielded on average 6.4 pp / 7.2 pp / 3.5 pp higher schedulability over the baseline for low / medium / high associativity. In contrast to the smaller cache size, the gap between the presented analysis and the state-of-the-art is smaller. An average improvement of 3.8 pp / 1 pp / 0.9 pp for low / medium / high associativity was observed. The maximal improvement of the presented analysis over the state-of-the-art was 7 pp / 3 pp / 4 pp. Hence, we conclude that in this larger cache configuration, the overestimation of the indirect interference in the state-of-the-art is less impactful on the schedulability than for smaller caches.

Table 2 shows the CRPD reduction using the presented analysis over the state-of-the-art for the 1KB L1 / 8 KB L2 cache configuration. The presented analysis achieved an average CRPD reduction of 2.7% / 2.3% / 1.7% over the patched, but unsafe, state-of-the-art analysis. As also visible in Table 1,



(a) 1/10/50 access timing (b) 1/50/100 access timing (c) 1/10/200 access timing

Fig. 10. Schedulability of 100 analyzed systems with 10 tasks per core for different L1 hit / L2 hit / L2 miss timings. (a) shows the results for 1/10/50 cycles; (b) for 1/50/100 cycles; (c) for 1/10/200 cycles.

there are situations in which the presented analysis yields a larger CRPD value compared with the state-of-the-art. The maximal increase, from 4572 to 4752 cycles, was realized in a system where the task *cnt* is preempted by the task *bsort100*.

We note that, for all analyzed system, the presented analysis dominated the state-of-the-art. Even though the CRPD estimate was slightly increased for some task combinations, there was no system which the state-of-the-art classified as schedulable, which was unschedulable using the presented analysis.

Comparing the cache size and associativity configurations using the presented analysis, we observe that the small cache with low associativity yielded the lowest schedulability of an average 63.2%, while the highest average schedulability occurred for the larger cache with high associativity at an average of 74.2%.

We also evaluated a cache size configuration of 4 KB L1 and 16 KB L2 cache. In this configuration, the performance difference between the three approaches diminished. At most, there was a 2 pp improvement of the presented analysis compared with the state-of-the-art. As the cache size increases, the code of the benchmarks fits more easily into the cache causing fewer conflicts. Thus, the precision difference in the analysis methods plays a less important role in system schedulability. On average, the total code size of the task sets was 26.3 KB.

In addition to experiments on the cache size, we explored the impact of the cache access timings. We show the results for a 1 KB L1 and 4 KB L2 cache with 2-way / 4-way associativity in Figure 10. A timing configuration of 10 cycles for an L2 hit delay and 50 cycle L2 miss delay is evaluated in Figure 10(a). Figure 10(b) shows a 50 cycle L2 hit and 100 cycle L2 miss configuration. The final timing evaluation in Figure 10(c) was made with a 10 cycles L2 hit and 200 cycle L2 miss setting.

For these timing configurations, the average improvement in schedulability by the presented analysis over the state-of-the-art was 3.3 pp, 2.6 pp, and 6.8 pp, respectively. Recall, that the improvement for the default 1/10/100 timing was 5.8 pp. For the two timing configurations 1/10/50 and 1/50/100, the difference between an L2 hit and L2 miss is smaller than for the default 1/10/100 timing. It is also for these combinations, that the average improvement was reduced. Whereas, for the slower L2 miss timing of 200 cycles, the average improvement increased from 5.8 pp to 6.8 pp. This observation is congruent with the optimization we made to the state-of-the-art regarding indirect interference. Indirect interference causes L2 hits to be degraded to L2 misses, thus an improvement in the estimation of indirect interferences will scale with the timing ratio between an L2 hit and an L2 miss.

Table 3. Average Runtime of a CRPD Analysis for a System Containing 10 Tasks in Seconds

Size (L1 / L2)	Associativity (L1 / L2)	Baseline [10]	SotA [36]	Presented Analysis
1 KB / 4 KB	1 / 2	12.2 s	70.7 s	7.5 s
	2 / 4	11.2 s	82.0 s	7.7 s
	4 / 8	11.9 s	92.4 s	7.7 s
2 KB / 8 KB	1 / 2	10.7 s	76.0 s	8.2 s
	2 / 4	9.6 s	82.5 s	8.4 s
	4 / 8	9.7 s	90.4 s	8.3 s

To compare the required computational effort, we measured the time required to analyze each system with all three analysis approaches. We performed the evaluations on an Intel Xeon Server with 48 cores running at 3.2 GHz. Every analysis was limited to utilize only a single core. To increase the performance, the analysis could be performed in parallel for every task, as it only requires knowledge of the number of ECBs from preempting tasks for each cache set.

The results are shown in Table 3. Each measurement corresponds to one CRPD analysis of a system independent of the other analyses. The average runtime of the presented analysis is comparable to the baseline approach of Chattopadhyay and Roychoudhury [10]. The baseline analysis required, on average, between 9.6 s and 12.2 s, while the presented analysis took, on average, between 7.5 s and 8.4 s. We call these runtimes comparable, as the relatively small difference in absolute runtime may depend on technical implementation details.

In contrast, the state-of-the-art analysis required substantially more time to complete. The average overhead lies between 9.3× and 12.0× compared with the presented analysis. The main contributors to the runtime of the state-of-the-art analysis were determining the first access to a memory block and collecting all possible locations between the analyzed location and the potential first accesses to that memory block, i.e., the function $GetProgramPoints(P, FA^P_{m_y})$ (see lines 3–4, Algorithm 2). The presented analysis does not contain this bottleneck, as the set returned by $GetProgramPoints(P, FA^P_{m_y})$ is never explicitly determined. Instead, by performing a data-flow analysis to determine potential indirect interference, the relevant locations are considered implicitly by propagating the data-flow information over these locations.

10 Conclusion

In this article, we demonstrate that the state-of-the-art [36] approach for CRPD analysis in two-level non-inclusive caches is flawed. To provide a solid foundation for new analyses, we introduce the concrete semantics of CRPD in two-level non-inclusive caches. Using the concrete semantics, it is possible to argue about the impact of preemption effects at the level of program traces.

Upon this foundation, we construct a novel CRPD analysis for two-level non-inclusive caches and prove its safety. In contrast to the state-of-the-art, we consider the CAC of accesses to the L2 cache and handle scenarios in which multiple references potentially issue the first access to a cache block after a preemption. Furthermore, we eliminated pessimism in the analysis by taking the ordering of accesses into account. Thus, a tighter bound on the indirect preemption effects can be computed. In our evaluations, we observed significant improvements in task set schedulability for small cache configurations. Schedulability was increased by up to 14 percentage points compared with the state-of-the-art.

In the future, it would be interesting to compare the CRPD values computed by the presented analysis with the context-switching costs measured on a real-system or observed in a simulation. This evaluation could show how tight the bound given by the presented analysis is compared with

the actually occurring CRPD and thus show how much room there is for improvements in the analysis precision.

In order for the presented analysis to be applicable to a system it has to satisfy several requirements, as discussed in Section 3. These limitations need to be addressed in future work in order to broaden the applicability of the presented analysis to COTS processors.

This article and all previously published works on two-level CRPD analysis [10, 36, 47] have focused on instruction caches. In the future, data caches need to considered. Data caches are harder to analyze than instruction caches because the target address of memory accesses to data objects may be difficult to determine precisely. This is the case in particular for input dependent data accesses. As memory accesses with uncertain target addresses are currently not supported, the presented analysis requires that both cache levels are instruction-only caches. While current processors commonly feature separated data and instruction caches at the first level, the second level cache is often unified. Thus, further research needs to be performed to handle data caches and consider the impact of data accesses in a unified L2 cache.

The LRU replacement policy has been recommended for real-time systems due to its high predictability [43]. However, commercial processors frequently implement different strategies. An extension of the presented analysis to other replacement policies, such as first-in-first-out replacement, could increase the applicability of the presented analysis.

The presented analysis considers systems with a single processor core. Multi-core systems often share the last-level cache between multiple cores. Sharing the second-level cache causes inter-core interferences, which leads to additional L2 cache misses. Chattopadhyay and Roychoudhury [10] proposed to account for inter-core interference in the CRPD analysis by counting the number of conflicting cache blocks accessed by interfering cores. However, it has been shown that this approach can be overly pessimistic when analyzing shared cache interference [15, 16, 33, 46]. Further research is needed to integrate the precise analysis of CRPD and shared cache interference.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2007. *Compilers: Principles, Techniques & Tools* (2 ed.). Pearson Education.

[2] Sebastian Altmeyer. 2012. *Analysis of Preemptively Scheduled Hard Real-time Systems*. Ph. D. Dissertation. Universität des Saarlandes.

[3] Sebastian Altmeyer and Claire Burguière. 2009. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *21st Euromicro Conference on Real-Time Systems (ECRTS'09)*. 109–118. DOI: https://doi.org/10.1109/ECRTS.2009.21

[4] Sebastian Altmeyer, Claire Maiza, and Jan Reineke. 2010. Resilience analysis: Tightening the CRPD bound for set-associative caches. *ACM SIGPLAN Notices* 45, 4 (2010), 153–162. DOI: https://doi.org/10.1145/1755951.1755911

[5] Sebastian Altmeyer and Claire Maiza Burguière. 2011. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture* 57, 7 (2011), 707–719. DOI: https://doi.org/10.1016/j.sysarc.2010.08.006 Special Issue on Worst-Case Execution-Time Analysis.

[6] Luna Backes and Daniel A. Jiménez. 2019. The impact of cache inclusion policies on cache management techniques. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*. 428–438. DOI: https://doi.org/10.1145/3357526.3357547

[7] Robert Balas and Luca Benini. 2021. RISC-V for real-time MCUs—software optimization and microarchitectural gap analysis. In *2021 Design, Automation and Test in Europe Conference (DATE'21)*. 874–877. DOI: https://doi.org/10.23919/DATE51398.2021.9474114

[8] Enrico Bini and Giorgio C. Buttazzo. 2005. Measuring the performance of schedulability tests. *Real Time Systems* 30, 1–2 (2005), 129–154. DOI: https://doi.org/10.1007/s11241-005-0507-9

[9] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings. 1996. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings Real-Time Technology and Applications*. 204–212. DOI: https://doi.org/10.1109/RTTAS.1996.509537

[10] Sudipta Chattopadhyay and Abhik Roychoudhury. 2014. Cache-related preemption delay analysis for multilevel noninclusive caches. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 5s (2014), 1–29. DOI: https://doi.org/10.1145/2632156

[11] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'77)*. 238–252. DOI: https://doi.org/10.1145/512950.512973

[12] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, and Reinhard Wilhelm. 2010. Predictability considerations in the design of multi-core embedded systems. *Embedded Real Time Software and Systems Conference (ERTS)* 36 (2010), 10. http://web1.see.asso.fr/erts2010/Default.aspx-Id=973-Idd=982.htm

[13] Heiko Falk and Paul Lokuciejewski. 2010. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems* 46, 2 (2010), 251–300. DOI: https://doi.org/10.1007/s11241-010-9101-x

[14] Christian Ferdinand and Reinhard Wilhelm. 1999. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems* 17, 2 (1999), 131–181. DOI: https://doi.org/10.1023/A:1008186323068

[15] Thilo L. Fischer and Heiko Falk. 2023. Analysis of shared cache interference in multi-core systems using event-arrival curves. In *Proceedings of Real-Time Network and Systems (RTNS'23)*. 23–33. DOI: https://doi.org/10.1145/3575757.3593643

[16] Thilo L. Fischer and Heiko Falk. 2024. Shared cache analysis under preemptive scheduling. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE'24)*. DOI: https://doi.org/10.23919/DATE58400.2024.10546581

[17] Alban Gruin, Thomas Carle, Hugues Cassé, and Christine Rochange. 2021. Speculative execution and timing predictability in an open source RISC-V core. In *2021 IEEE Real-Time Systems Symposium (RTSS'21)*. 393–404. DOI: https://doi.org/10.1109/RTSS52674.2021.00043

[18] Daniel Grund, Jan Reineke, and Reinhard Wilhelm. 2011. A template for predictability definitions with supporting evidence. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems (Open Access Series in Informatics (OASIcs), Vol. 18)*, Philipp Lucas and Reinhard Wilhelm (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 22–31. DOI: https://doi.org/10.4230/OASIcs.PPES.2011.22

[19] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET benchmarks–past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET'10)*, Björn Lisper (Ed.). OCG, Brussels, Belgium, 136–146. DOI: https://doi.org/10.4230/OASIcs.WCET.2010.136

[20] Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. 2015. Towards compositionality in execution time analysis: Definition and challenges. *ACM SIGBED Review* 12, 1 (2015), 28–36. DOI: https://doi.org/10.1145/2752801.2752805

[21] Damien Hardy, Thomas Piquet, and Isabelle Puaut. 2009. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *2009 30th IEEE Real-Time Systems Symposium (RTSS'09)*. 68–77. DOI: https://doi.org/10.1109/RTSS.2009.34

[22] Damien Hardy and Isabelle Puaut. 2008. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *2008 Real-Time Systems Symposium (RTSS'08)*. 456–466. DOI: https://doi.org/10.1109/RTSS.2008.10

[23] Infineon. 2014. *AURIX™ TC21x/TC22x/TC23x Family*. Retrieved June 24, 2024 from https://community.infineon.com/gfawx74859/attachments/gfawx74859/AURIX/5399/1/Infineon-TC21x-TC22x-TC23x-UM-v01_01-EN.pdf

[24] Chang-Gun Lee, Hoosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. 1998. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.* 47, 6 (1998), 700–713. DOI: https://doi.org/10.1109/12.689649

[25] Yau-Tsun Steven Li and Sharad Malik. 1997. Performance analysis of embedded software using implicit path enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16, 12 (1997), 1477–1487. DOI: https://doi.org/10.1109/43.664229

[26] Yun Liang, Huping Ding, Tulika Mitra, Abhik Roychoudhury, Yan Li, and Vivy Suhendra. 2012. Timing analysis of concurrent programs running on shared cache multi-cores. *Real-Time Systems* 48, 6 (2012), 638–680. DOI: https://doi.org/10.1007/s11241-012-9160-2

[27] Arm Limited. 2007. *PL310 Cache Controller Technical Reference Manual r0p0*. Retrieved June 5, 2024 from https://developer.arm.com/documentation/ddi0246/a/introduction/about-the-cache-controller

[28] Arm Limited. 2011. *Cortex-R4 and Cortex-R4F Technical Reference Manual r1p4*. Retrieved June 5, 2024 from https://developer.arm.com/documentation/ddi0363/g

[29] Arm Limited. 2024. *ARM Cortex-R Series Programmer's Guide*. Retrieved June 5, 2024 from https://developer.arm.com/documentation/den0042/a/Caches/Cache-policies/Replacement-policy

[30] Chung Laung Liu and James W. Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)* 20, 1 (1973), 46–61. DOI: https://doi.org/10.1145/321738.321743

[31] Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. 2016. A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems* 3, 1 (2016), 05:1–05:48. DOI: https://doi.org/10.4230/LITES-v003-i001-a005

[32] Tulika Mitra. 2019. Time-predictable computing by design: Looking back, looking forward. In *Proceedings of the 56th Annual Design Automation Conference 2019 (DAC'19)*. Article 153, 4 pages. DOI: https://doi.org/10.1145/3316781.3323489

[33] Kartik Nagar. 2016. *Precise analysis of Private and Shared Caches for tight WCET Estimates*. Ph. D. Dissertation. Indian Institute of Science Bangalore.

[34] Michael Platzer and Peter Puschner. 2021. Vicuna: A timing-predictable RISC-V vector coprocessor for scalable parallel computation. In *33rd Euromicro Conference on Real-Time Systems (ECRTS'21) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 196)*, Björn B. Brandenburg (Ed.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1:1–1:18. DOI: https://doi.org/10.4230/LIPIcs.ECRTS.2021.1

[35] Syed Aftab Rashid, Geoffrey Nelissen, and Eduardo Tovar. 2020. Bounding cache persistence reload overheads for set-associative caches. In *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'20)*. 1–10. DOI: https://doi.org/10.1109/RTCSA50079.2020.9203583

[36] Syed Aftab Rashid, Geoffrey Nelissen, and Eduardo Tovar. 2022. Tightening the CRPD bound for multilevel non-inclusive caches. *Journal of Systems Architecture* 122 (2022), 102340. DOI: https://doi.org/10.1016/j.sysarc.2021.102340

[37] Jan Reineke. 2014. Randomized caches considered harmful in hard real-time systems. *Leibniz Transactions on Embedded Systems* 1, 1 (2014), 03:1–03:13. DOI: https://doi.org/10.4230/LITES-v001-i001-a003

[38] Jan Reineke. 2018. The semantic foundations and a landscape of cache-persistence analyses. *Leibniz Transactions on Embedded Systems* 5, 1 (2018), 03:1–03:52. DOI: https://doi.org/10.4230/LITES-v005-i001-a003

[39] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. 2007. Timing predictability of cache replacement policies. *Real Time Systems* 37, 2 (2007), 99–122. DOI: https://doi.org/10.1007/s11241-007-9032-3

[40] Gregory Stock, Sebastian Hahn, and Jan Reineke. 2019. Cache persistence analysis: Finally exact. In *2019 IEEE Real-Time Systems Symposium (RTSS'19)*. 481–494. DOI: https://doi.org/10.1109/RTSS46320.2019.00049

[41] Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. 2019. Fast and exact analysis for LRU caches. *Proceedings of the ACM on Programming Languages (POPL)* 3 (2019), 54:1–54:29. DOI: https://doi.org/10.1145/3290367

[42] Andrew Shell Waterman. 2016. *Design of the RISC-V instruction set architecture*. Ph. D. Dissertation. University of California, Berkeley.

[43] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. 2009. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 7 (2009), 966–978. DOI: https://doi.org/10.1109/TCAD.2009.2013287

[44] Reinhard Wilhelm and Jan Reineke. 2012. Embedded systems: Many cores–Many problems. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*. 176–180. DOI: https://doi.org/10.1109/SIES.2012.6356583

[45] Jun Xiao, Sebastian Altmeyer, and Andy Pimentel. 2017. Schedulability analysis of non-preemptive real-time scheduling for multicore processors with shared caches. In *2017 IEEE Real-Time Systems Symposium (RTSS'17)*. 199–208. DOI: https://doi.org/10.1109/RTSS.2017.00026

[46] Wei Zhang, Mingsong Lv, Wanli Chang, and Lei Ju. 2022. Precise and scalable shared cache contention analysis for WCET estimation. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC'22)*. 1267–1272. DOI: https://doi.org/10.1145/3489517.3530613

[47] Zhenkai Zhang and Xenofon Koutsoukos. 2016. Cache-related preemption delay analysis for multi-level inclusive caches. In *Proceedings of the 13th International Conference on Embedded Software (EMSOFT'16)*. Article 16, 10 pages. DOI: https://doi.org/10.1145/2968478.2968481

Received 31 January 2024; revised 3 July 2024; accepted 1 September 2024