CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

Conference Paper

Bounding Cache Persistence Reload Overheads for Set-Associative Caches

Syed Aftab Rashid*
Geoffrey Nelissen
Eduardo Tovar*

*CISTER Research Centre

CISTER-TR-200716

2020/08/19

Bounding Cache Persistence Reload Overheads for Set-Associative Caches

Syed Aftab Rashid*, Geoffrey Nelissen, Eduardo Tovar*

*CISTER Research Centre

Polytechnic Institute of Porto (ISEP P.Porto)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: syara@isep.ipp.pt, gnn@isep.ipp.pt, emt@isep.ipp.pt

https://www.cister-labs.pt

Abstract

Cache memories have a strong impact on the response time of tasks executed on modern computing platforms.For tasks scheduled under fixed-priority preemptive scheduling(FPPS), the worst-case response time (WCRT) analyses thataccount for cache persistence between jobs along with cacherelated preemption delays (CRPDs) have been shown to dominateanalyses that only consider CRPDs. Yet, the existing approachesthat analyze cache persistence in the context of WCRT analysiscan only support direct-mapped caches.

In this work, we analyze cache persistence in the context ofWCRT analysis for set-associative caches. The main contributionsof this work are: (i) to propose a solution to find persistent cacheblocks (PCBs) of tasks considering set-associative caches, (ii) topresent three different approaches to calculate cache persistencereload overheads (CPROs), i.e., the memory overhead due toeviction of PCBs of tasks, under set-associative caches, and (iii) anexperimental evaluation showing that our proposed approachesresult in up to 22 percentage points higher task set schedulabilitythan the state-of-the-art approaches.

Bounding Cache Persistence Reload Overheads for Set-Associative Caches

Syed Aftab Rashid*, Geoffrey Nelissen†, Eduardo Tovar*
*CISTER, ISEP, Polytechnic Institute of Porto, Portugal, †Technische Universiteit Eindhoven, Eindhoven, The Netherlands

***Abstract*—Cache memories have a strong impact on the response time of tasks executed on modern computing platforms. For tasks scheduled under fixed-priority preemptive scheduling (FPPS), the worst-case response time (WCRT) analyses that account for cache persistence between jobs along with cache related preemption delays (CRPDs) have been shown to dominate analyses that only consider CRPDs. Yet, the existing approaches that analyze cache persistence in the context of WCRT analysis can only support direct-mapped caches.**

In this work, we analyze cache persistence in the context of WCRT analysis for set-associative caches. The main contributions of this work are: (i) to propose a solution to find *persistent cache blocks* (PCBs) of tasks considering set-associative caches, (ii) to present three different approaches to calculate cache persistence reload overheads (CPROs), i.e., the memory overhead due to eviction of PCBs of tasks, under set-associative caches, and (iii) an experimental evaluation showing that our proposed approaches result in up to 22 percentage points higher task set schedulability than the state-of-the-art approaches.

I. Introduction

Caches are smaller faster memories that bridge the speed gap between the main memory and processor by holding frequently required data and instructions closer to the processor. However, due to their limited capacity not all data/instructions of all tasks can simultaneously reside in the cache. Hence, tasks compete for cache space, with one task potentially evicting memory blocks previously loaded into the cache by others. This may result in increasing the worse-case response time (WCRT) of tasks depending on whether the instructions/data needed by the tasks is still available in the cache (i.e., cache hit) or have been evicted by other tasks (i.e., cache miss).

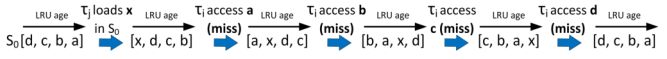
The impact of caches on the WCRT of tasks is more evident under preemptive scheduling. In preemptive scheduling, tasks may suffer additional execution delays due to evictions of their cache content by other tasks, causing for example *Cache Related Preemption Delays* (CRPDs). CRPDs are additional execution delays suffered by preempted tasks when they reload *useful cache blocks* (UCBs) (i.e., memory blocks that would normally remain in cache between two or more successive accesses by the preempted task) that were evicted from the cache during the execution of preempting tasks. Recent works [1], [2] have also shown that under FPPS, the use of caches may result in tighter WCRT of tasks due to cache persistence. Cache persistence refers to cache blocks that are never evicted between any two execution of a task. Those cache blocks are called *persistent cache blocks* (PCBs). Consequently, thanks to PCBs, the memory access demand of subsequent jobs of τ_i can be much lower than its worst-case memory access demand in *isolation*, which results in reducing the WCRT for tasks scheduled using FPPS. By definition, once loaded into cache, PCBs of a task can never be invalidated or evicted by the task itself. However, PCBs may be evicted due to interleaved or preemptive execution of other tasks, leading to *Cache Persistence Reload Overheads* (CPRO).

Considering that CPROs may significantly affect tasks WCRTs, several approaches have been proposed in the state-of-the-art to bound CPROs [1], [2] using the PCBs of the task under analysis and *evicting cache blocks* (ECBs) (all cache block used by the task during its execution) of all other tasks in the system. However, those analyses [1], [2] assumed a *direct-mapped* cache and did not work for processor architectures based on *set-associative* caches. This is mainly because in a direct-mapped cache, each cache set can hold at most one memory block whereas, in a set-associative cache, each cache set may hold as many memory blocks as there are *cache ways* (also referred to as the cache *associativity*). Therefore, in case of a cache conflict between two tasks τ_i and τ_j, each memory access performed during τ_j's execution may evict at most one PCB of τ_i in a direct-mapped cache, while it may lead to multiple evictions in a set-associative cache. This is known as the *cascading* effect. An example of such cascading effect is shown in Fig. 1. We can see that for a direct-mapped cache (see Figure 1a) a single cache conflict between τ_i and τ_j (i.e., due to preemption of τ_i by τ_j) can only cause one cache miss whereas the same cache conflict leads to multiple cache misses for a set-associative cache (see Figure 1b). Therefore, for set-associative caches, a *sound* CPRO estimate can only be obtained by accounting for the cascading effect which is not considered by the existing CPRO analysis that focus on direct-mapped caches.

Note that there exist few analyses [3] in the literature that focus on CRPD computation considering set-associative caches. However, it has been shown in recent works [1], [2] that only considering CRPDs for tasks scheduled under FPPS may result in largely pessimistic WCRT bounds if it does not account for cache persistence. Therefore, in this work, we present different solutions to analyze cache persistence for set-associative caches and integrate those solutions in the WCRT analysis of FPPS. The main contributions of this work are as follow: (i) to propose a solution to find persistent cache blocks (PCBs) of tasks considering set-associative caches; and (ii) to present three different approaches to calculate CPROs on platforms implementing set associative caches. These approaches

978-1-7281-4403-0/20/$31.00 ©2020 IEEE

(a) Direct-mapped Cache



(b) Set-associative Cache

Fig. 1: Example execution of a task τ_i (from left to right) considering (a) a direct-mapped cache with 4 cache sets, i.e., $\{S_0, S_1, S_2, S_3\}$ and (b) a 4-way set-associative cache having one cache set S_0 using a Least-Recently-Used (LRU) cache replacement policy. The LRU age of a block b refers to how many accesses were performed to the cache set in which b is saved since the last access to b.

are (1) the PCB-ECB approach, that uses only the set of PCBs of the task under analysis and the set of ECBs of all other tasks in the system to evaluate the CPRO, (2) the ResilienceP analysis, that removes some of the pessimism in the PCB-ECB approach by considering the *resilience* of PCBs, and (3) the multi-path ResilienceP analysis, that considers the variation in the resilience of PCBs over different executions of a task in order to have an even tighter CPRO bound. The experimental evaluation shows that our proposed approaches result in up to 22 percentage points higher task set schedulability than the state-of-the-art approaches.

Organization. Section II introduces the system model, notations and background. Section III outlines how to determine the PCBs of tasks when considering set-associative caches. We then present the PCB-ECB approach and the ResilienceP analysis in Section IV. Section V details the multi-path ResilienceP analysis and in Section VI we explain how the computed CPRO bounds can be incorporated into the WCRT analysis. An experimental evaluation is presented in Section VII. Lastly, conclusions are drawn in Section IX.

II. System Model, Notations and Background

We focus on single-core processor with a private set-associative instruction cache (also referred to as L1) using the LRU replacement policy, i.e., on a cache miss the least recently used memory block (or equivalently the block with the largest LRU-age) within the targeted cache set is evicted. The number of memory blocks that can be stored in each cache set is referred to as the number of ways or the associativity of the cache and is denoted by W. The set of all cache sets is denoted by $\mathbb{S}$. We use d_{mem} to denote the time needed to load one block from the main memory into the cache.

We consider a task set Γ comprising n sporadic constrained deadline tasks $\{\tau_1, \tau_2, ...\tau_n\}$. Each task $\tau_i \in \Gamma$ is defined by a triplet (C_i, T_i, D_i), where C_i is the worst-case execution time (WCET) of τ_i, T_i is its minimum inter-arrival time and D_i is the relative deadline of each instance or job of τ_i. We assume $D_i \leq T_i$. In addition to the WCET C_i, we use separate terms to measure the worst-case processing demand and memory access demand of each task. PD_i denote the worst-case processing demand of τ_i, i.e., it only accounts for execution requirements of τ_i and does not include the time spent by τ_i to perform memory operations. MD_i denote the worst-case memory access demand of any job of task τ_i, i.e., the maximum time during which any job of τ_i is performing memory operations. Note that the value of C_i, PD_i and MD_i are determined assuming τ_i executes in *isolation*. It obviously hold that $C_i > PD_i$ and $C_i > MD_i$, but it also holds that $C_i \leq PD_i + MD_i$[1] since PD_i and MD_i may result from different execution scenarios of τ_i along different execution paths (e.g., due to different inputs). The WCRT of task τ_i, denoted by R_i, is defined as the longest time between the arrival and the completion of any job of τ_i.

For notational convenience, we use $\mathrm{hp}(i)$ to denote the set of tasks with priorities higher than that of τ_i and $\mathrm{hep}(i)$ to denote the set of tasks with higher than or equal priority to τ_i, including τ_i, i.e., $\mathrm{hep}(i) = \mathrm{hp}(i) \cup \tau_i$. Note that in this work, we only focus on analyzing additional cache overheads due to CRPDs and CPROs. Other overheads that remain constant over the execution of a task, e.g., due to context switches etc., are assumed to be included in the task's WCET.

Evicting and Useful Cache Blocks (ECBs and UCBs). Any cache block m used by a task during its execution is called an evicting cache block (ECB) [6]. A cache block m is also called a UCB at program point P, if m is cached at P and may be reused at program point Q that may be reached from P without eviction of m [7].

Cache Related Preemption Delay (CRPD). When a task τ_i is preempted by a higher priority task τ_j, ECBs of τ_j may evict UCBs of τ_i that are to be reloaded from the main memory after τ_i resumes its execution. The additional execution time incurred by τ_i due to these extra cache reloads is termed as cache related preemption delay (CRPD) and is denoted by $\gamma_{i,j}$.

For set-associative caches, the resilience-analysis [3] dominates all other methods in the state-of-the-art to compute CRPD. It uses the notion of *resilience* defined as follows.

Resilience [3]. The resilience of a memory block m at program point P is the maximum *disturbance* that m can endure before being evicted from the cache. This disturbance represents the number of ECBs of preempting task(s) that may be mapped to the same cache set as m. The Resilience of a cache block m at a program point P is given by

$$res_P(m) = (W - 1) - max\text{-}age_P(m) \tag{1}$$

where $max\text{-}age_P(m)$ is the *maximum* LRU-age of m at program point P, i.e., the maximum number of accesses to the same cache set as m from the last use of m (before or at program point P) to the next access to m after P [3]. For example, assuming memory blocks m_i, m_a, m_b, m_c and m_d in Figure 2 are all accessed by task τ_i and that they are all mapped to the same cache set, the maximum LRU-age of UCB m_i at program point P, i.e., $max\text{-}age_P(m_i)$, is 4 and therefore for $W = 8$, its resilience according to Eq. (1) is $(8-1)-4 = 3$.

[1]It is experimentally confirmed since $C_i \leq PD_i + MD_i$ for several benchmarks from the Mälardalen Benchmark Suite [4] that were analyzed using the Heptane [5] static WCET estimation tool for a MIPS R2000/R3000 architecture.

For every program point P, the maximum LRU-age of a UCB m can be calculated by using a forward analysis to find the maximal number of accesses from the last use of m to program point P and a backward analysis to find the maximum number of accesses from program point P to the next access to m. The maximum LRU-age of m at program point P is then bounded by the sum of the bounds returned by both analyses (see [3] for a detailed description).

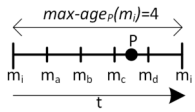


Fig. 2: Illustration of the maximum LRU-age of a UCB m_i. The dashes (from left to right) denote the sequence of memory accesses during the execution of τ_i.

Using the resilience-analysis, the CPRD suffered by a task τ_i due to a single preemption by a higher priority task τ_j in a cache set s is given by $\gamma_{i,j}^{res,s}$, i.e.,

$$\gamma_{i,j}^{res,s} = d_{mem} \times \left| UCB_i^s \setminus \{m_i | res(m_i) \geq |ECB_j^s|\} \right| \quad (2)$$

where $|UCB_i^s|$ and $|ECB_j^s|$ denote the number of UCBs of τ_i and the number of ECBs of τ_j in cache set s, respectively. The total CRPD over all cache sets in $\mathbb{S}$ is then given by $\gamma_{i,j}^{res}$, where

$$\gamma_{i,j}^{res} = \sum_{s \in \mathbb{S}} \gamma_{i,j}^{res,s} \quad (3)$$

Note that since the number of UCBs and their resilience is calculated for each program point, $\gamma_{i,j}^{res}$ is given by the program point that maximizes Eq. (3) over all program points in the task code.

Persistent Cache Block (PCB) [1]. A memory block m_i of task τ_i is a PCB if, once loaded by τ_i, m_i will never be invalidated or evicted from the cache when τ_i executes in isolation. The set of PCBs of a task τ_i is denoted by PCB_i.

If all PCBs of a task τ_i are available in the cache from a previous job execution of τ_i then the memory access demand of subsequent jobs of τ_i will be much lower than the worst-case memory access demand of τ_i in isolation. This reduced memory demand is called the *residual memory access demand* of τ_i and is denoted by MD_i^r.

Residual Memory Demand (MD_i^r) [1]. The residual memory access demand MD_i^r of task τ_i is the worst-case memory access demand of any job of τ_i when all its PCBs are already loaded in the cache memory.

The number of PCBs and the residual memory access demand (MD_i^r) of a task τ_i can be used to bound its total memory access demand $\hat{MD}_i(t)$ in isolation during a time interval of length t:

$$\hat{MD}_i(t) \stackrel{\text{def}}{=} \min \left\{ \left\lceil \frac{t}{T_i} \right\rceil MD_i \; ; \left\lceil \frac{t}{T_i} \right\rceil MD_i^r + \mid PCB_i \mid \times d_{mem} \right\} \quad (4)$$

where $\left\lceil \frac{t}{T_i} \right\rceil$ counts the maximum number of jobs released by τ_i in the interval of length t.

Cache Persistence Reload Overhead (CPRO) [1]. The CPRO suffered by a task τ_j executing during the response time of a task τ_i is denoted by $\rho_{j,i}$ and is formally defined as the additional execution time incurred by task τ_j in reloading from the main memory its PCBs that may have been evicted from the cache due to the executions of tasks in $\text{hep}(i) \setminus \tau_j$.

Disturbance. The disturbance suffered by a task τ_i on a cache set s due to another set of tasks $\mathcal{T}$ is defined as the total number of ECBs of tasks in $\mathcal{T}$ that are mapped to the cache set s. The disturbance due to $\mathcal{T}$ is thus the maximum number of memory blocks of tasks in $\mathcal{T}$ that compete with τ_i for space in cache set s.

III. Finding PCBs for set-associative caches

For direct-mapped caches, determining the set of PCBs, i.e., PCB_i, of a task τ_i is relatively simple. A memory block m_i of task τ_i belongs to PCB_i if it is the only memory block of τ_i mapped to a given cache set. However, under set-associative caches, several memory blocks of τ_i may be mapped to a single cache set and the presence of a memory block in cache depends on the LRU-age of that memory block. A W-way set-associative LRU-cache can hold up to W memory blocks in each cache set and the LRU-age of each memory block can be between 0 and $W-1$ (respectively representing the most-recently and the least-recently accessed memory block in the cache set). Given a program point P, if the LRU-age of a memory block at P is greater than or equal to W then an access to that memory block at P will be a cache miss, i.e., the memory block is not in the cache anymore. We can leverage this information to find PCBs of a task τ_i under a set-associative cache. By definition, once loaded into the cache by task τ_i all PCBs of τ_i will not be evicted or invalidated by τ_i while executing in isolation. Therefore, all memory blocks used by τ_i that have an LRU-age less than or equal to $W-1$ at every program point P in τ_i can be PCBs of τ_i. However, knowing that PCBs are potentially reused by the same but also by every next job executed by task τ_i, using only one job execution of τ_i to bound the maximum LRU-age of PCBs may not be sufficient. To illustrate this last property, consider the control-flow graph (CFG) and mapping of the memory blocks of two successive jobs of task τ_i (i.e., $\tau_{i,1}$ and $\tau_{i,2}$) shown in Fig. 3. In that example, all memory blocks, i.e., m_1, m_2, m_3, and m_4, used by τ_i map to the same cache set s in a 4-way set-associative cache. We can see in Fig. 3a that when considering only one job of τ_i, i.e., $\tau_{i,1}$, the maximum LRU-age of memory blocks m_1, m_2, m_3, and m_4 is 3, 2, 1, and 0 respectively. However, if we consider the execution sequence $\tau_{i,1}$ followed by $\tau_{i,2}$, we can see that the maximum LRU-age of all the memory blocks is 3. Note that underestimating the maximum LRU-age of memory blocks may lead to false positives, i.e., a memory block may be categorized as a PCB (i.e., LRU-age $\leq W-1$ over the execution of one job) while it is not (i.e., LRU-age $> W-1$ over the execution of a sequence of jobs). Therefore, in order to soundly estimate the set of PCBs of a task τ_i, it is important to calculate the maximum LRU-age of all memory blocks used by τ_i after any execution sequence of jobs of τ_i (in isolation). This can be done by assuming that

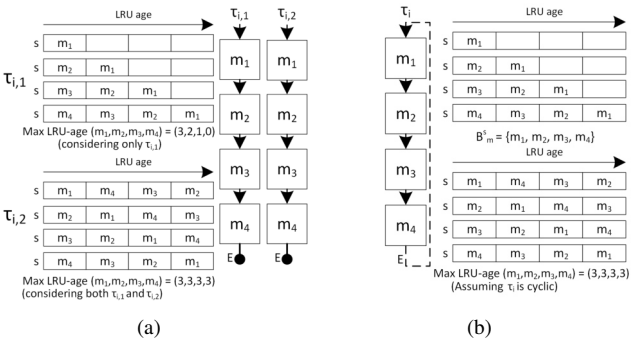


Fig. 3: Maximum LRU-age of memory blocks of task τ_i (a) over the execution of two jobs of τ_i, and (b) under the assumption that τ_i is cyclic

τ_i is cyclic, i.e., a loop is assumed between the end point E and start point S of τ_i (see Fig. 3b). The cyclic assumption ensures that the maximal number of different cache accesses between the last use of a memory block m_i in one job of τ_i and the first access of m_i in the next job of τ_i are considered when determining the maximum LRU-age of m_i.

Formally, the analysis to find PCBs of a task τ_i is performed as follows:

1) Apply the standard *persistence* analysis [8] on the code of task τ_i to determine the set of memory blocks B^s_m that are persistent in each cache set s at the end of τ_i. The Persistence analysis determines if a memory block will not be evicted after it has been loaded in the cache, i.e., the first reference to that memory block may result in a cache miss but all subsequent references to that memory block will be cache hits (see [8] for a detailed description on the persistence analysis). A memory block m_i is persistent in a cache set s if its LRU-age in s is less than or equal to $W-1$ at the end of τ_i's execution.
2) Apply the persistence analysis again (i.e., to account for the cyclic assumption) on τ_i assuming that each cache set s already contains the B^s_m memory blocks at the start of τ_i's execution and that each of those blocks has its maximum LRU-age derived in step 1. All memory blocks in B^s_m that are in every *Abstract Cache States* (ACSs) (i.e., representing all possible mappings of memory blocks to cache lines) of the second persistence analysis (i.e, memory block with LRU-age$\leq W-1$ in all ACSs) are PCBs of τ_i in cache set s and are denoted by PCB^s_i. The final set of PCBs of task τ_i is then given by

$$PCB_i = \bigcup_{s \in \mathbb{S}} PCB^s_i \tag{5}$$

IV. CPRO Analysis for Set-Associative Caches

In this section, we present two approaches for the calculation of the CPRO for set-associative caches, namely, the PCB-ECB approach and the ResilienceP analysis.

A. PCB-ECB Approach

For direct-mapped caches, the CPRO can be computed by using the intersection between the set of PCBs of the task under analysis and the set of ECBs of all other tasks that may evict PCBs of that task. For example, under the CPRO-union approach presented in [1], the CPRO $\rho^{dir}_{j,i}$ one job of a higher priority task $\tau_j \in \mathrm{hp}(i)$ may suffer while executing during the response time of a lower priority task τ_i is computed as follows

$$\rho^{dir}_{j,i} = d_{mem} \times \left| PCB_j \cap \Big(\bigcup_{\forall \tau_k \in \mathrm{hep}(i) \setminus \tau_j} ECB_k \Big) \right| \tag{6}$$

where PCB_j is the set of PCBs of τ_j , $\bigcup_{\forall \tau_k \in \mathrm{hep}(i) \setminus \tau_j} ECB_k$ is the set of ECBs of all tasks in $\mathrm{hep}(i) \setminus \tau_j$ and d_{mem} is the time required to reload one PCB from the main memory (see [1] for a formal proof of Eq. (6)). The bound resulting from Eq. (6) is sound for a direct-mapped cache where each ECB of tasks in $\mathrm{hep}(i) \setminus \tau_j$ can evict at most one PCB of τ_j. However, using Eq. (6) to calculate the CPRO of τ_j under a set-associative cache may result in underestimating the CPRO of τ_j due to the cascading effect mentioned in the introduction, i.e., several/all PCBs of task τ_j may be evicted due to a single ECB of another task mapped to the same cache set (see Fig. 1). Considering that in a set-associative cache, each cache set s can be analyzed independently, a sound estimate of the CPRO suffered by one job of task τ_j due to a cache set $s \in \mathbb{S}$ can be obtained by using the two following properties: (1) PCBs of τ_j mapped in cache set s may be evicted and hence participate to the CPRO of τ_j during the response time of another task τ_i, only if one or more ECB(s) of tasks in $\mathrm{hep}(i) \setminus \tau_j$ are mapped to the same cache set s, i.e., the disturbance τ_j may suffer due to all tasks in $\mathrm{hep}(i) \setminus \tau_j$ in a cache set s must be greater than or equal to one, and (2) the participation of a cache set s to the CPRO of τ_j is upper bounded by the number of PCBs of τ_j in that cache set multiplied by d_{mem}, i.e., $d_{mem} \times |PCB^s_j|$. Therefore, the CPRO one job of task τ_j may suffer due to cache set s is upper bounded by $\rho^{set,s}_{j,i}$, where

$$\rho^{set,s}_{j,i} = d_{mem} \times \begin{cases} |PCB^s_j| & \text{if } D^s_{j,i} \geq 1 \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

where $D^s_{j,i}$ is the disturbance all tasks in $\mathrm{hep}(i) \setminus \tau_j$ may cause on PCBs of τ_j in a cache set s. By definition, the maximum disturbance task τ_j may suffer due to all tasks in $\mathrm{hep}(i) \setminus \tau_j$ is upper bounded by $\sum_{\forall \tau_k \in \mathrm{hep}(i) \setminus \tau_j} |ECB^s_k|$, i.e., the total number of ECBs of tasks in $\mathrm{hep}(i) \setminus \tau_j$ mapped to cache set s. Therefore, $D^s_{j,i} \leq \sum_{\forall \tau_k \in \mathrm{hep}(i) \setminus \tau_j} |ECB^s_k|$. The rationale of using ECBs of all tasks in $\mathrm{hep}(i) \setminus \tau_j$ to bound $D^s_{j,i}$ is to account for nested/multiple execution of tasks in $\mathrm{hep}(i) \setminus \tau_j$ between two jobs of task τ_j. Since, in the worst-case all tasks in $\mathrm{hep}(i) \setminus \tau_j$ may sequentially execute between two jobs of τ_j, the cumulative impact of tasks in $\mathrm{hep}(i) \setminus \tau_j$ on PCBs of τ_j is upper bounded by $\sum_{\forall \tau_k \in \mathrm{hep}(i) \setminus \tau_j} |ECB^s_k|$.

Note that Eq. (7) accounts for the cascading effect by considering that a single ECB of tasks in $\mathrm{hep}(i) \setminus \tau_j$ mapped to a cache set s may evict all PCBs of τ_j in that cache set.

The total CPRO one job of τ_j may suffer during the response time of τ_i is given by

$$\rho^{set}_{j,i} = \sum_{\forall s \in \mathbb{S}} \rho^{set,s}_{j,i} \tag{8}$$

B. ResilienceP Analysis

The PCB-ECB approach presented above assumes that if one ECB of any task in $\text{hep}(i) \setminus \tau_j$ is mapped to a cache set s then all the PCBs of τ_j in s will be evicted. This assumption is safe but very pessimistic. To illustrate, consider the example depicted in Figure 4. It shows a sequence of cache references

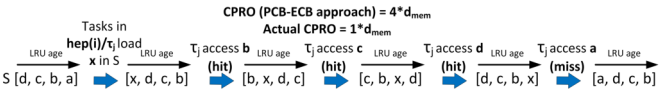


Fig. 4: Highlighting the pessimism in the PCB-ECB approach

during the execution of a task τ_j (from left to right) assuming that τ_j has 4 PCBs in cache set s, i.e., $PCB_j^s = \{a, b, c, d\}$. We also assume that the value of disturbance $D_{j,i}^s$ is equal to 1, i.e., only one ECB of tasks in $\text{hep}(i) \setminus \tau_j$ is mapped to cache set s. Using $|PCB_j^s| = 4$ and $D_{j,i}^s = 1$ in Eq. (7) (i.e., PCB-ECB approach) the CPRO of τ_j due to cache set s is calculated to be $4 \times d_{mem}$. However, we can see in Figure 4 that only one cache reference of τ_j will be a miss after the execution of tasks in $\text{hep}(i) \setminus \tau_j$. Therefore, the actual CPRO of τ_j due to cache set s is only $1 \times d_{mem}$.

The ResilienceP analysis removes excessive pessimism in the PCB-ECB approach by finding PCBs of task τ_j that may remain cached (and therefore does not contribute to the CPRO) even after the execution of tasks in $\text{hep}(i) \setminus \tau_j$ thanks to their resilience. Based on the definition of resilience in Section II, the resilience of a PCB m_j of task τ_j is given by the maximum value of disturbance $D_{j,i}^s$ that m_j can endure before being evicted from the cache due to the execution of tasks in $\text{hep}(i) \setminus \tau_j$.

The resilience-analysis proposed in [3] can be used to determine the resilience of all memory blocks used by a task τ_j, at every program point P during the execution of τ_j. However, using the resilience-analysis [3] to determine the resilience of PCBs may result in overestimating the resilience of PCBs. This is mainly because the resilience-analysis [3] was proposed to calculate the resilience of UCBs instead of PCBs. By definition, UCBs of a task τ_j may only be reused during the same job execution of τ_j and hence it is sufficient to consider the execution of only one job of τ_j when bounding the maximum LRU-age of its UCBs. However, PCBs are different from UCBs considering that PCBs may be reused during the execution of the same job and/or any next job of τ_j. Therefore, to have a sound estimate of the resilience of PCBs of τ_j it is necessary to calculate the maximum LRU-age of PCBs after any execution sequence of jobs of τ_j. See Figure 3 that shows that considering only one job of task τ_i may result in underestimating the maximum LRU-age (i.e., overestimating the resilience) of memory blocks m_2, m_3 and m_4.

The ResilienceP analysis uses the approach described in Section III to determine the resilience of PCBs of each task. Let $PersistentAge_P(m_j)$ denote the LRU-age of a PCB m_j at any program point P during the execution of task τ_j resulting from the analysis detailed in Section III. Then, the maximum LRU-age $max\text{-}age(m_j)$ of m_j is calculated by maximizing $PersistentAge_P(m_j)$ over all program points during the execution of τ_j, i.e.,

$$max\text{-}age(m_j) = \max_{\forall P \in \mathbb{P}} PersistentAge_P(m_j) \tag{9}$$

where $\mathbb{P}$ is the set of all program points. Consequently, the resilience of PCB m_j is then given by $res_{PCB}(m_j) = (k-1) - max\text{-}age(m_j)$.

Therefore, the ResilienceP analysis upper bounds the CPRO that may be suffered by one job of task τ_j due to cache set s by $\rho_{j,i}^{res,s}$, where $\rho_{j,i}^{res,s}$ is computed as follows

$$\rho_{j,i}^{res,s} = d_{mem} \times \left| PCB_j^s \setminus \{m_j | res_{PCB}(m_j) \geq D_{j,i}^s\} \right| \tag{10}$$

where $res_{PCB}(m_j)$ is the resilience of a PCB $m_j \in PCB_j^s$ and $D_{j,i}^s$ is the maximum disturbance all tasks in $\text{hep}(i) \setminus \tau_j$ may cause to a cache set s.

Note that Eq. (10) excludes PCBs of τ_j that remain cached after the execution of tasks in $\text{hep}(i) \setminus \tau_j$ (i.e., those for which $res_{PCB}(m_j) \geq D_{j,i}^s$) from the CPRO. Therefore, it provides a tighter bound on the CPRO than the PCB-ECB approach.

The total CPRO of one job of task τ_j executing during the response time of another task τ_i is thus bounded by

$$\rho_{j,i}^{res} = \sum_{\forall s \in \mathbb{S}} \rho_{i,j}^{res,s} \tag{11}$$

Finally, knowing from [1] that in a time interval of length t at most $\left\lceil \frac{t}{T_j} \right\rceil - 1$ jobs of τ_j may suffer CPRO, the total CPRO of task τ_j in a time interval of length t is bounded by

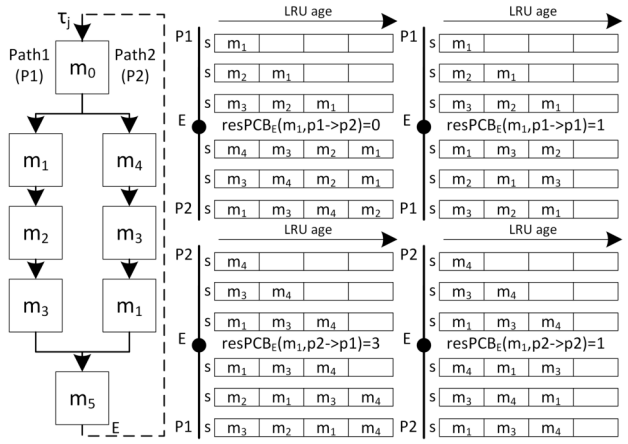
$$\hat{\rho}_{j,i}(t) = \left(\left\lceil \frac{t}{T_j} \right\rceil - 1 \right) \times \rho_{j,i} \tag{12}$$

where $\rho_{j,i}$ can be calculated either using the PCB-ECB approach or the ResilienceP analysis (i.e., by using Eq. (8) or Eq. (11)).

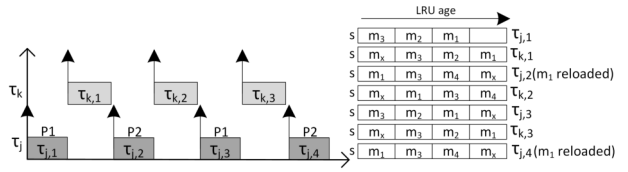
V. Multi-path ResilienceP Analysis

The ResilienceP analysis always considers the worst-case (i.e., minimum) resilience for every PCB and for every job of τ_j that may execute in a time interval of length t. This assumption is exact in the case where τ_j has only a single execution path as shown in Fig. 3b. However, if τ_j has multiple execution paths, the resilience of PCBs may vary depending on the actual execution paths taken by two successive jobs of τ_j. Therefore, always considering the minimum resilience of PCBs over all job executions of τ_j may overestimate the total CPRO τ_j may suffer. To illustrate this, Fig. 5a shows the CFG of a task τ_j with two execution paths and four possible execution flows between two jobs of τ_j, i.e., $p1 \rightarrow p2$, $p2 \rightarrow p1$, $p1 \rightarrow p1$ and $p2 \rightarrow p2$. The cache content along each execution flow is also shown in Fig. 5a. We assume that all memory blocks of τ_j except m_0 and m_5 map to the same cache set s of a 4-way set-associative cache. For simplicity, we only focus on PCB m_1.

We can see in Fig. 5a that the resilience of m_1 is minimum, i.e., $res_{PCB}(m_1) = 0$, if the first job of τ_j follows path $p1$ and the next job follows path $p2$. Now consider the example schedule shown in Fig. 5b showing four jobs of τ_j executed



(a) Variation in the resilience of PCBs of task τ_j



(b) Different job executions of τ_j and τ_k

Fig. 5: Highlighting the pessimism in the ResilienceP analysis

together with three jobs of a task $\tau_k \in \text{hep}(i) \setminus \tau_j$ such that $ECB_k^s = \{m_x\}$, i.e, $D_{j,k}^s = 1$. Fig. 5b shows the contents of cache set s after the execution of every job of τ_j and τ_k.

Since the minimum resilience of m_1 is 0 and $D_{j,k}^s > res_{PCB}(m_1)$, the ResilienceP analysis (i.e., Eq. (10)) implies that every time τ_k preempts τ_j or executes between two subsequent jobs of τ_j, m_1 will be evicted. This results in a CPRO equal to $3 \times d_{mem}$. However, we can see in Fig. 5b that this is not true. In fact even when we maximize the number of jobs of τ_j following the execution flow with the minimum resilience (i.e., $p_1 \to p_2$), m_2 is evicted and reloaded only two times resulting in a CPRO of $2 \times d_{mem}$. The reason behind this result is that if the first two jobs of τ_j execute according to the execution flow $p_1 \to p_2$, then the second and third jobs of τ_j can either follow the execution flow $p_2 \to p_1$ or $p_2 \to p_2$ In both cases, the actual resilience of m_1 is equal to 1 (instead of 0 as assumed by the ResilienceP analysis).

The multi-path ResilienceP analysis reduces the pessimism of the ResilienceP analysis by considering the variation in the resilience of PCBs across different job execution flows of a same task τ_j. It computes the total CPRO task τ_j may suffer in a time interval of length t by first creating a CPRO-table (See Table I) for all PCBs of τ_j in each cache set. The CPRO-table determines how many times each PCB $m_j \in PCB_j^s$ can be evicted in an interval of length t considering a given disturbance D and the maximum number of jobs J released by τ_j in the interval of length t. Given the values of D and J, one entry in Table I tells us how many times PCB m_j may be evicted and must therefore be reloaded.

TABLE I: CPRO-table for every PCB m_j of task τ_j

		Number of jobs of τ_j (J)			
		2	3	...	$\left\lceil \frac{t}{T_j} \right\rceil$
Disturbance D	1	$\rho_{j,m_j}(1,2)$	$\rho_{j,m_j}(1,3)$	...	$\rho_{j,m_j}(1, \left\lceil \frac{t}{T_j} \right\rceil)$
	2	$\rho_{j,m_j}(2,2)$	$\rho_{i,j}(2,3)$	...	$\rho_{j,m_j}(2, \left\lceil \frac{t}{T_j} \right\rceil)$
	...	...	...	...	...
	$\geq W$	1	2	...	$\left\lceil \frac{t}{T_j} \right\rceil - 1$

Algorithm 1 Building the CPRO-table for PCB m_j of task τ_j

Input: Interval length t; PCB m_j; Set of all possible execution paths EP_j of τ_j.
Output: All $\rho_{j,m_j}(D,J)$ entries in Table I.

1: $Flows$:=PathsPermutations($\tau_j, |EP_j|$)
2: **for** $D := 1$ to $W-1$ **do**
3: $PossibleFlows$:=FindPathsCombinations(m_j, $Flows$, D)
4: L :=FindLongestFlow($PossiblePaths$)
5: **for** $J := 2$ to $\left\lceil \frac{t}{T_j} \right\rceil$ **do**
6: **if** $|PossibleFlows| = 0$ **then**
7: $\rho_{j,m_j}(D,J) := 0$;
8: **else**
9: $MaxCPRO := J - 1$
10: **if** $L \geq MaxCPRO$ **then**
11: $\rho_{j,m_j}(D,J) := MaxCPRO$;
12: **else**
13: $\rho_{j,m_j}(D,J) := MaxCPRO - \left\lfloor \frac{J}{L+1} \right\rfloor$;
14: **end if**
15: **end if**
16: **end for**
17: **end for**

A. Building the CPRO-table

In this subsection we discuss how a CPRO-table can be built for PCBs.

First, we make use of a couple of simple properties to bound the size of that table:

1) It is proved in [1] that if a task τ_j releases J jobs in a time interval of length t, the maximum number of times each PCB $m_j \in PCB_j^s$ can be evicted is upper bounded by $J-1$.
2) If the disturbance D suffered by a PCB m_j is greater than or equal to the number of ways W in the cache (i.e., $D \geq W$) then the entire cache set s will be filled by the ECBs of disturbing tasks and PCB m_j will be evicted after every job execution i.e., it will be evicted $J-1$ times.

We use that information to fill all the CPRO-table entries such that $D \geq W$ (see Table I).

The remaining entries (noted $\rho_{j,m_j}(D,J)$) are calculated using Algorithm 1. Algorithm 1 uses the set of all possible execution paths EP_j of task τ_j and the maximum length t of the interval in which its PCB m_j may be evicted as input. It then fills row-wise all entries in Table I. The function `PathsPermutations(·)` at line 1 returns a set that contains all possible executions paths combinations between two jobs of τ_j. Given that task τ_j has $|EP_j|$ possible execution paths, the size of $Flows$ is $2^{|EP_j|}$.

The external loop (lines 2 to 17) is used to iterate over all disturbance values D between 1 and $W-1$ (all table entries for $D \geq W$ are already filled). As previously dis-

cussed, the resilience of a PCB m_j may vary depending on the specific combination of execution paths taken by two successive jobs of τ_j (see Fig. 5a). Therefore, the function `FindPathsCombinations(·)` (line 3) returns the set of paths combinations of two successive jobs of τ_j for which the resilience of m_j is less than D. By the definition of resilience, the memory block m_j may be evicted only for those paths combinations. Function `FindLongestFlow(·)` then generates (at line 4) the longest execution flow composed of path combinations in $PossibleFlows$. For example, assuming $PossibleFlows$ contains the three paths combinations $p_1 \rightarrow p_2$, $p_3 \rightarrow p_1$ and $p_3 \rightarrow p_2$. The longest execution flow that may be generated by `FindLongestFlow(·)` is $p_3 \rightarrow p_1 \rightarrow p_2$. The function thus returns 2 as the length L of that flow. Note that by definition of $PossibleFlows$, there exists a (possibly different) program point P in each paths composing that execution flow for which the resilience of m_j is less than D. Therefore, if the maximum disturbance D is applied at each of those program points, then m_j will be evicted L times. Moreover, since `FindLongestFlow(·)` generates the longest such execution flow, there cannot be more than L successive evictions of m_j. The nested loop (lines 5 to 16) is then used to upper bound how many times m_j will be evicted for every possible value of J (note that for sporadic tasks, at most $\left\lceil \frac{t}{T_j} \right\rceil$ jobs of τ_j may be released in any interval of length t, thus $J \leq \left\lceil \frac{t}{T_j} \right\rceil$). If the set of possible paths combinations returned by the function `FindPathsCombinations(·)` is empty, then m_j cannot be evicted for the disturbance value D and hence $\rho_{j,m_j}(D, J)$ is equal to 0 (line 6). Otherwise, if there exists some paths combinations for which m_j may be evicted with a disturbance D, i.e., $|PossibleFlows| > 0$, then two cases must be considered:

1) If $L \geq J - 1$, then we know from Lemma 2 of [1] that the CPRO suffered by J successive jobs of a task τ_j is upper bounded by $J - 1$ and thus $\rho_{j,m_j}(D, J) = J - 1$ (line 11).
2) If $L < J - 1$, then, by the definition of L, m_j may be evicted at most L times in every execution flow composed of $L + 1$ successive jobs of τ_j. Therefore, the maximum number of times m_j may be evicted for a succession of J jobs is bounded by $(J - 1) - \left\lfloor \frac{J}{L+1} \right\rfloor$ (line 13).

Example. Consider memory block m_1 in Fig. 5. Applying Algorithm 1 with $D^s_{j,k} = 1$ (i.e., $|ECB^s_k| = 1$), L=1 (i.e., the execution flow $p_1 \rightarrow p_2$) and J=$\{2, 3, 4\}$, we get $\rho_{j,m_1}(1, 2) = 1$, $\rho_{j,m_1}(1, 3) = 1$ and $\rho_{j,m_1}(1, 4) = 2$ which is consistent with the scenario depicted in Fig. 5b.

B. Bounding the CPRO

After creating the CPRO-table of every PCB of task τ_j using Algorithm 1, the total CPRO that task τ_j may suffer in a time interval of length t can be bounded using Algorithm 2. The inputs to Algorithm 2 are the CPRO-tables of every PCB m_j of task τ_j, the maximum disturbance task τ_j may suffer for every $s \in \mathbb{S}$ due to the execution of tasks in $\mathrm{hep}(i) \setminus \tau_j$, i.e., $D^s_{j,i}$ and the length of time interval t. The output of Algorithm 2 is the total CPRO of task τ_j in a time interval of length t denoted by

Algorithm 2 Computing the total CPRO of task τ_j in a time interval of length t

Input: Interval length t; CPRO-table of every PCB m_j of task τ_j; Disturbance $D^s_{j,i}$ for every $s \in \mathbb{S}$
Output: The total CPRO of task τ_j in a time interval of length t, i.e., $\hat{\rho}^{mul}_{i,j}(t)$.

1: $J := \left\lceil \frac{t}{T_j} \right\rceil$
2: $\hat{\rho}^{mul}_{i,j}(t) := 0$
3: **for** $\forall s \in \mathbb{S}$ **do**
4: $\quad \rho^{mul,s}_{i,j} := 0$
5: $\quad D := D^s_{j,i}$
6: $\quad$ **for** $\forall m_j \in PCB^s_j$ **do**
7: $\quad\quad \rho^{mul,s}_{i,j} := \rho^{mul,s}_{i,j} + \rho_{j,m_j}(D, J)$
8: $\quad$ **end for**
9: $\quad \hat{\rho}^{mul}_{i,j}(t) := \hat{\rho}^{mul}_{i,j}(t) + \rho^{mul,s}_{i,j}$
10: **end for**

$\hat{\rho}^{mul}_{i,j}(t)$. Given the length t of the time interval, $J := \left\lceil \frac{t}{T_j} \right\rceil$ upper bounds the number of jobs τ_j may execute in t (line 1). For every cache set $s \in \mathbb{S}$, the value of D is set using $D^s_{j,i}$ (line 5). Given the values of D and J the inner loop (lines 6 to 8) iteratively computes the CPRO of task τ_j in every cache set $s \in \mathbb{S}$, i.e., $\rho^{mul,s}_{i,j}$, using values from the CPRO-table of every PCB $m_j \in PCB^s_j$. The outer loop (lines 3 to 10) then sums up the values of $\rho^{mul,s}_{i,j}$ for all $s \in \mathbb{S}$ to bound $\hat{\rho}^{mul}_{i,j}(t)$.

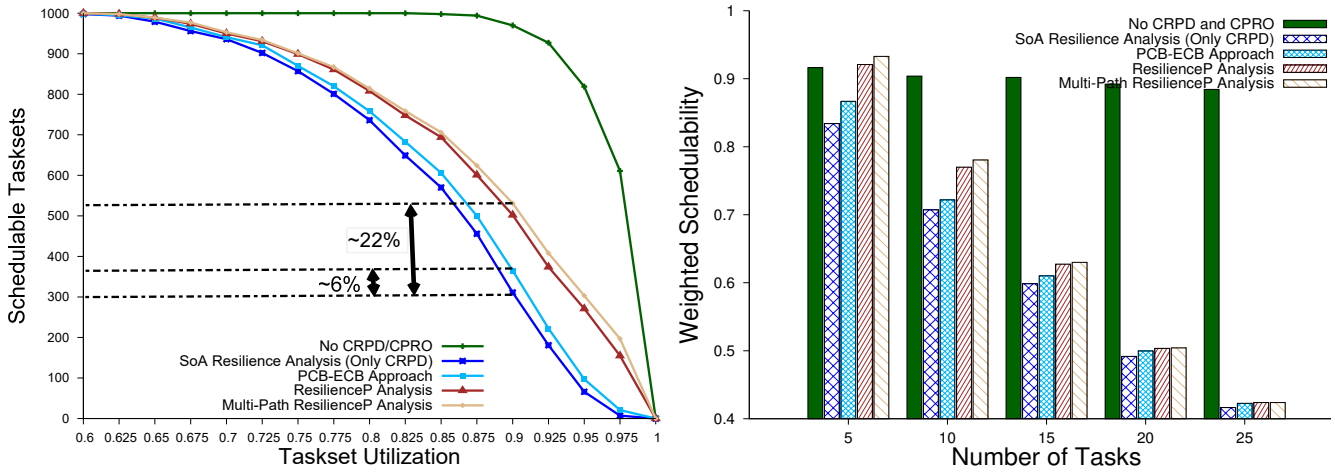
VI. WCRT Analysis

The WCRT analysis for FPPS that accounts for both CRPDs and CPRO considering direct-mapped caches was proposed in [1], [2]. It uses Eq. (13) to calculate the WCRT R_i of a task τ_i (see [1] for a formal proof of Eq. (13)).

$$R_i = C_i + \sum_{\forall j \in \mathrm{hp}(i)} \min \left\{ \left\lceil \frac{R_i}{T_j} \right\rceil C_j \; ; \left\lceil \frac{R_i}{T_j} \right\rceil PD_j + \hat{M}D_j(R_i) + \rho_{j,i}(R_i) \right\} + \sum_{\forall j \in \mathrm{hp}(i)} \gamma_{i,j}(R_i) \tag{13}$$

where C_i and C_j are the worst-case execution times of τ_i and τ_j, respectively. PD_j is the worst-case processing demand of τ_j and $\hat{M}D_j(R_i)$ is calculated using Eq. (4) and is an upper bound on the total memory access demand (in terms of time) of all jobs of τ_j that may execute during the WCRT R_i of τ_i. $\gamma_{i,j}(R_i)$ and $\rho_{j,i}(R_i)$ are upper bounds on the CRPD and CPRO considering the pair of tasks τ_i and τ_j, respectively.

Eq. (13) can also be used to calculate the WCRT R_i of a task τ_i when considering set-associative caches; (i) by calculating the CPRO $\rho_{j,i}(R_i)$ using any of the approaches presented in Sections IV and V, and (ii) by calculating the CRPD $\gamma_{i,j}(R_i)$ using the state-of-the-art resilience-analysis [3]. The WCRT R_i of task τ_i is then calculated by using simple fixed-point iteration on R_i, where R_i is initialized to C_i. In every iteration, the values of $\rho_{j,i}(R_i)$ are updated based on the chosen approach. For example, if multi-path ResilienceP analysis is used, Algorithm 2 is executed at every iteration to update the total CPRO suffered by the tasks based on the



(a) (b)

Fig. 6: Task sets schedulability by varying (a) total task set utilization and (b) the total number of tasks in a task set

CPRO-tables previously built using Algorithm 1. To reduce computation time, the CPRO-table of every PCB of each task can be built only once by setting $t = D_n$ in Algorithm 1, where D_n denote the deadline of the lowest priority task τ_n in the task set. The iteration stops as soon as R_i does not evolve anymore or $R_i > D_i$(i.e., the task is deemed unschedulable).

VII. Experimental Evaluation

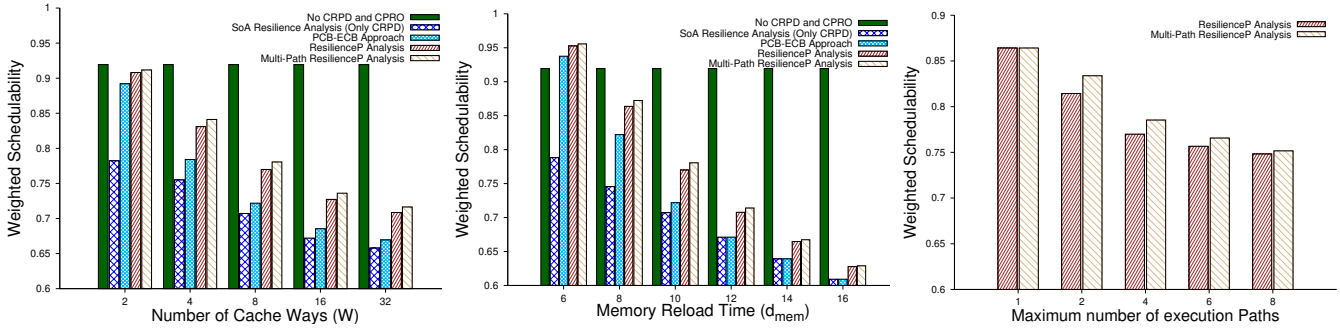
In this section, we evaluate how our proposed approaches that account for both cache persistence (i.e., CPROs) and CRPDs perform in terms of schedulability in comparison to the state-of-the-art resilience-analysis [3] that only considers CRPDs when analyzing set-associative caches. We performed experiments using synthetic task sets where tasks parameters C_i, PD_i, MD_i, UCB_i, ECB_i and PCB_i were taken from Table 1 of [2]. Each task in the task set was randomly assigned the values C_i, PD_i, MD_i, UCB_i, ECB_i and PCB_i of one of the benchmarks referred in that table. The system was setup to model a MIPS R2000/R3000 architecture assuming a 8-way set-associative cache with 64 sets, a line size of 32 Bytes (i.e., a total cache size of 16kB) and a memory reload time $d_{mem} = 10\mu$s.

The default number of tasks in a task set was 10 with task utilizations generated using UUnifast [9]. Task periods and deadlines were set such that D_i=T_i=C_i/U_i. Task priorities were assigned in a deadline monotonic order. Furthermore, to evaluate the performance of the multi-path ResilienceP analysis each task was randomly assigned between 1 to 4 execution paths.

We performed different experiments by varying the total core utilization, number of tasks, number of cache ways and memory reload time d_{mem}. A WCRT based schedulability analysis is performed using the same task sets for all the analyzed approaches.

1. Core Utilizations: In this experiment, we varied the total core utilization from 0.025 to 1 in steps of 0.025 and randomly generated 1000 tasksets at every value of the core utilization. Fig. 6a show the number of task sets that were deemed schedulable by all the analyzed approaches. The plot also show the number of task sets that were deemed schedulable without considering any CRPD and CPRO (i.e., green line). Note that we only show cropped version of the plot starting from a utilization of 0.6 as all approaches produce identical results below this utilization. Fig. 6a shows that our approaches that also account for cache persistence (i.e., CPROs) along with CRPDs dominate the state-of-the-art resilience-analysis that only consider CRPDs and does not account for cache persistence. Among the three proposed approaches, the PCB-ECB approach has the least number of task sets that were deemed schedulable. This is intuitive, since the PCB-ECB approach pessimistically assume that every PCB of a task in a cache set will be evicted if one or more ECBs of any other task are mapped to the same cache set. This pessimism is reduced by the ResilienceP analysis by considering the resilience of PCBs which results in accepting more task sets. Finally, the multi-path ResilienceP analysis was able to schedule even more task sets than the ResilienceP analysis by considering the variation in the resilience of PCBs over multiple job executions. Our proposed approaches improve task set schedulability by 6 to 22 percentage points over the state-of-the-art analysis. Note that on an Intel core i-7 processor (3.4GHz) the average computation time to generate the plot shown in Fig. 6a was 210 seconds.

2. Number of Tasks: In this experiment, we varied the total number of tasks in a task set between 5 to 25 in steps of 5 keeping default values of all other parameters. Since we varied both the number of tasks and core utilizations we have used the weighted schedulability measure defined in [10] to plot the results in Fig. 6b. The weighted schedulability measure reduces what would otherwise be a 3-dimensional plot to 2-dimensions by eliminating the axis of task set utilization. This performance metric gives more weight to task sets with higher utilization. We can see in Fig. 6b that by increasing the number of tasks the total number of task sets that were deemed schedulable by all the approaches decreases. Indeed, this is due to an increasing number of cache evictions and reloads



(a) Varying the number of cache ways (b) Varying memory reload time d_{mem} (c) Varying number of execution paths

Fig. 7: Weighted schedulability results

leading to higher CRPD and CPRO. However, we can still observe that our approaches always dominate the state-of-the-art analysis. Note that by increasing the number of tasks, all the three proposed approaches tend to produce similar results. This is mainly because by increasing the number of tasks, the number of ECBs of tasks sharing cache space with PCBs of other tasks also increases, i.e., the disturbance is increased. Therefore, even if PCBs of some tasks have a greater resilience they might still be evicted due to a higher disturbance.

3. Number of Cache Ways (W): The number of ways W defines how many memory blocks can be mapped to one cache set. We increased the number of cache ways from 2 to 32, keeping default values for all other parameters. The results are shown in Fig. 7a. We can see in Fig. 7a that by increasing the number of cache ways the total number of schedulable task sets decreases. This is mainly because we assume that the total cache size is constant hence by increasing the number of cache ways the number of cache sets decreases. This results in more tasks sharing the same cache sets which in turns leads to higher CRPD and CPRO. However, we can still see that all the three proposed approaches dominate the state-of-the-art resilience-analysis. In fact due to lower CPRO, for a cache associativity of 2 and 4, the performances of our analyses are considerably better. In all our experiments we assume a sequential layout of tasks in memory (and in cache), however, task layout optimization techniques, e.g., as proposed in [11], can also be used to improve task set schedulability.

4. Memory Reload Time d_{mem}: we varied the value of memory reload time d_{mem} from 6μs to 16μs in step of 2μs. The results are presented in Fig. 7b. We can see in Fig. 7b that for lower values of d_{mem} the difference between the weighted schedulability of our approaches and the state-of-the-art analysis is significantly higher. This is mainly because for lower values of d_{mem} the reduction in memory accesses demand due to cache persistence dominates the CPRO. We also note that for $d_{mem} \geq 12\mu$s the performances of the PCB-ECB approach and the state-of-the-art analysis are identical. This is due to the excessive pessimism in PCB-ECB approach which is removed by the ResilienceP (and multi-path ResilienceP) analysis.

5. Number of execution paths: It is obvious from the results that the multi-path ResilienceP analysis dominates all other approaches. However, the performance of the multi-path ResilienceP analysis depends on the number of execution paths of tasks and the resilience of PCBs along those paths. To evaluate this, we varied the maximum number of execution paths in each task between 1 to 8 and compared the performance of ResilienceP and multi-paths ResilienceP analyses. The results are presented in Fig. 7c. We can see in Fig. 7c that if tasks are only allowed to have a single execution path, both ResilienceP and multi-paths ResilienceP analyses produce identical results. Moreover, for number of execution paths between 2 to 6 the multi-path ResilienceP analysis tend to produce better results than the ResilienceP analyses. However, for a further increase in number of paths, the difference between the ResilienceP and multi-paths ResilienceP analyses tend to disappear. The is due to the fact that when the number of paths increase, it becomes more probable that there exist several execution flows for which the resilience of the PCBs is low. The function `FindPathsCombinations(·)` of Alg. 1 will then more easily return very long execution flows with low resilience. This eventually leads to account for the same number of evictions of PCBs as under the ResilienceP analysis.

VIII. Related Work

Many different approaches have been proposed in the state-of-the-art to bound cache overheads under preemptive scheduling (see [12] for a comprehensive survey).

Lee et al. [7] introduced the notion of UCBs and used the number of UCBs of the preempted tasks to bound CRPD considering both direct-mapped and set-associative caches. However, the resulting CRPD bounds were overly pessimistic since their analysis doesn't consider the preempting tasks. Tomiyama and Dutt [6] used only the number of ECBs of the preempting tasks to bound CRPD. However, their analysis resulted in underestimating the CRPD suffered by the tasks. Tan and Mooney [13] presented an approach to bound CRPDs of tasks using UCBs of the preempted tasks and ECBs of the preempting tasks. However, the analysis in [13] does not account for the cascading effect in set-associative caches and therefore results in unsound CRPD bounds. Burguiere et al. [14] also presented a CRPD analysis for set-associative caches that use the numbers of UCBs and ECBs of tasks. The analysis in [14] gives a bound on the CRPD that is sound but imprecise due to the assumption that every UCB of the preempted task in a cache set s is considered to be

evicted if only one ECB of the preempting task maps to the same cache set s. Altmeyer et al. [3] presented the resilience-analysis that dominates all other method in the state-of-the-art to compute CRPD. The resilience-analysis determines the set of UCBs of the preempted tasks that are guaranteed to remain in the cache even after the execution of the preempting tasks. This set of UCBs cannot contribute to the CRPD suffered by the preempted tasks leading to a tighter CRPD bound. However, since the resilience-analysis doesn't account for cache persistence it may lead to pessimistic WCRT bounds.

Cache persistence has been extensively studied by several works in literature, e.g., [8], [15]–[17], in the context of WCET analysis. However, most of these existing works focus on cache persistence within loops of a program which is distinct from the type of cache persistence studied in this work, i.e., between different jobs of the same task. In one of the earlier works that focus on cache persistence between jobs, Nemer et al. [18] presented an intra-task instruction cache analysis to accurately estimate the number of cache misses during the execution of tasks by considering task's entry and exit cache states. However, their approach is limited to non-preemptive task sets under static scheduling and do not apply to preemptive systems with commonly used priority based scheduling schemes. Under FPPS, few existing works that focus on cache persistence are [1], [2], [11], [19]. In [1] authors formally introduced the notion of cache persistence between different jobs of the same task and used it to reduce pessimism in the computation of interference from multiple jobs of a higher priority task in state-of-the-art WCRT analysis. Two analyses for CPRO were presented and integrated into an improved response time analysis. In [2] authors showed that the independent calculation of CRPD and CPRO may result in overestimating the total memory overhead suffered by tasks. An integrated approach to calculate CRPD and CPRO is presented in [2] that removes some of the pessimism in the CPRO analyses presented in [1]. In [11] authors evaluated the impact of memory layout of tasks on the CPRO and showed that an optimized layout of task in memory may result in an improved task set schedulability. In [19] the impact of cache persistence on memory bus contention in multicore system has been analyzed. However, all these existing works focus on the CPRO computation for direct-mapped caches.

IX. Conclusion

In this work, we proposed a solution to analyze the cache persistence for set-associative caches in the context of the WCRT analysis of FPPS. We showed how persistent cache blocks of tasks can be determined when considering set-associative caches. We then presented three different approaches to calculate the CPRO under set-associative caches. Our first analyses, i.e., the PCB-ECB approach, is very coarse-grain. To improve the analysis performance, we explained how the resilience of PCBs can be computed and factored in the analysis. Therefore, the resulting ResilienceP analysis performs considerably better. Lastly, the multi-path ResilienceP analysis uses the variation in the resilience of PCBs over different job execution flows to derive an even tighter CPRO bound. The experimental evaluation shows that our proposed approaches result in up to 22 percentage point higher task set schedulability than the state-of-the-art analyses.

As future work, we plan on extending our analysis to multi-level set-associative caches. We also aim to extent this analysis to multicore platforms having a shared last-level cache.

Acknowledgments. This work was partially supported by National Funds through FCT/MCTES (Portuguese Foundation for Science and Technology), within the CISTER Research Unit (UIDB/04234/2020); also by FCT and the ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH/BD/119150/2016.

References

[1] S. A. Rashid, G. Nelissen, D. Hardy, B. Akesson, I. Puaut, and E. Tovar, "Cache-persistence-aware response-time analysis for fixed-priority preemptive systems," in *ECRTS*, 2016, pp. 262–272.

[2] S. A. Rashid, G. Nelissen, S. Altmeyer, R. I. Davis, and E. Tovar, "Integrated analysis of cache related preemption delays and cache persistence reload overheads," in *RTSS*. IEEE, 2017, pp. 188–198.

[3] S. Altmeyer, C. Maiza, and J. Reineke, "Resilience analysis: Tightening the crpd bound for set-associative caches," in *LCTES*. ACM, 2010, pp. 153–162.

[4] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks: Past, present and future," in *OASIcs-OpenAccess Series in Informatics*, vol. 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[5] D. Hardy, B. Rouxel, and I. Puaut, "The heptane static worst-case execution time estimation tool," in *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[6] H. Tomiyama and N. D. Dutt, "Program path analysis to bound cache-related preemption delay in preemptive real-time systems," in *CODES*, 2000, pp. 67–71.

[7] C. G. Lee, J. Hahn, Y. M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *Computers, IEEE Transactions on*, vol. 47, no. 6, pp. 700–713, 1998.

[8] H. Theiling, C. Ferdinand, and R. Wilhelm, "Fast and precise wcet prediction by separated cache and path analyses," *Real-Time Systems*, vol. 18, no. 2-3, pp. 157–179, 2000.

[9] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, 2005.

[10] A. Bastoni, B. Brandenburg, and J. Anderson, "Cache-related preemption and migration delays: Empirical approximation and impact on schedulability," *Proceedings of OSPERT*, pp. 33–44, 2010.

[11] S. A. Rashid, G. Nelissen, and E. Tovar, "Trading between intra-and inter-task cache interference to improve schedulability," in *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, 2018, pp. 125–136.

[12] M. Lv, N. Guan, J. Reineke, R. Wilhelm, and W. Yi, "A survey on static cache analysis for real-time systems," *Leibniz Transactions on Embedded Systems*, vol. 3, no. 1, pp. 05–1, 2016.

[13] Y. Tan and V. Mooney, "Timing analysis for preemptive multitasking real-time systems with caches," *ACM TECS*, vol. 6, no. 1, p. 7, 2007.

[14] C. Burguière, J. Reineke, and S. Altmeyer, "Cache-related preemption delay computation for set-associative caches-pitfalls and solutions," in *OASIcs-OpenAccess Series in Informatics*, vol. 10. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[15] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems," *Real-Time Systems*, vol. 17, no. 2-3, pp. 131–181, 1999.

[16] F. Mueller, "Timing analysis for instruction caches," *Real-time systems*, vol. 18, no. 2-3, pp. 217–247, 2000.

[17] Z. Zhang and X. Koutsoukos, "Improving the precision of abstract interpretation based cache persistence analysis," *ACM SIGPLAN Notices*, vol. 50, no. 5, pp. 1–10, 2015.

[18] F. Nemer, H. Cassé, P. Sainrat, and J. P. Bahsoun, "Inter-task wcet computation for a-way instruction caches," in *2008 International Symposium on Industrial Embedded Systems*. IEEE, 2008, pp. 169–176.

[19] S. A. Rashid, G. Nelissen, and E. Tovar, "Cache persistence-aware memory bus contention analysis for multicore systems," in *DATE*, 2020.