Cache-Persistence-Aware Response-Time Analysis for Fixed-Priority Preemptive Systems

Syed Aftab Rashid
Geoffrey Nelissen
Damien Hardy
Benny Akesson
Isabelle Puaut
Eduardo Tovar

CISTER-TR-160503
Abstract

A task can be preempted by several jobs of higher priority tasks during its response time. Assuming the worst-case memory demand for each of these jobs leads to pessimistic worst-case response time (WCRT) estimations. Indeed, there is a big chance that a large portion of the instructions and data associated with the preempting task $\tau_j$ are still available in the cache when $\tau_j$ releases its next jobs. Accounting for this observation allows the pessimism of WCRT analysis to be significantly reduced, which is not considered by existing work.

The four main contributions of this paper are: 1) The concept of persistent cache blocks is introduced in the context of WCRT analysis, which allows re-use of cache blocks to be captured, 2) A cache-persistence-aware WCRT analysis for fixed-priority preemptive systems exploiting the PCBs to reduce the WCRT bound, 3) An multi-set extension of the analysis that further improves the WCRT bound, and 4) An evaluation showing that our cache-persistence-aware WCRT analysis results in up to 10% higher schedulability than state-of-the-art approaches.
Cache-Persistence-Aware Response-Time Analysis for Fixed-Priority Preemptive Systems

Syed Aftab Rashid*, Geoffrey Nelissen*, Damien Hardy†, Benny Akesson*, Isabelle Puaut‡, Eduardo Tovar*
* CISTER/INESC TEC, ISEP, Polytechnic Institute of Porto, Portugal
‡ University of Rennes 1/IRISA, France

Abstract—A task can be preempted by several jobs of higher priority tasks during its response time. Assuming the worst-case memory demand for each of these jobs leads to pessimistic worst-case response time (WCRT) estimations. Indeed, there is a big chance that a large portion of the instructions and data associated with the preempting task \( \tau_j \) are still available in the cache when \( \tau_j \) releases its next jobs. Accounting for this observation allows the pessimism of WCRT analysis to be significantly reduced, which is not considered by existing work.

The four main contributions of this paper are: 1) The concept of persistent cache blocks is introduced in the context of WCRT analysis, which allows re-use of cache blocks to be captured, 2) A cache-persistence-aware WCRT analysis for fixed-priority preemptive systems exploiting the PCBs to reduce the WCRT bound, 3) An multi-set extension of the analysis that further improves the WCRT bound, and 4) An evaluation showing that our cache-persistence-aware WCRT analysis results in up to 10% higher schedulability than state-of-the-art approaches.

I. INTRODUCTION

The existing gap between the processor and main memory operating speeds necessitates the use of intermediate cache memories to accelerate the average access time to instructions and data required by the processor. The introduction of cache memories in modern computing platforms causes big variations in the execution time of each instruction, depending on whether the instruction and data it requires are already loaded in the cache (cache hit) or not (cache miss).

Recent years have focused on analyzing the impact of preemptions on the worst-case execution time (WCET) and worst-case response time (WCRT) of tasks in preemptive systems. Indeed, the preempted tasks may suffer additional cache misses if its memory blocks are evicted from the cache during the execution of preempting tasks. These evictions cause extra accesses to main memory, which result in additional delays in the task execution. This extra cost is usually referred to as cache-related preemption delays (CRPDs).

Many different approaches have been proposed to counter the effect of preemptions. Some (e.g., [1], [2]) use non-preemptive or limited-preemption scheduling schemes to eliminate or reduce the number of preemptions. Others [3]–[9] use information about the tasks’ memory access patterns to bound and incorporate preemption costs into the WCRT analysis. These approaches differ in whether they consider the memory access patterns of the preempted task [4], the preempting tasks [3], [5], or both [5]–[9]. Regardless of this distinction, they all still result in pessimistic WCRT bounds due to the fact that they only consider the effect of preemptions on the memory demand of the preempted task, but not the variation in memory demand of the preempting tasks. Instead, they all assume that every job of a high priority task \( \tau_j \) preempting a low priority task \( \tau_i \) will ask for its maximum memory demand, i.e., its worst-case memory demand in isolation. Although this may be true for the first job released by the preempting task \( \tau_j \), subsequent jobs of \( \tau_j \) may re-use most of the data and instructions that were already loaded in the cache during the execution of its previous jobs. Analyses that exploit this observation have been proposed for both direct-mapped [10] and set-associative caches [11]. However, these are limited to non-preemptive task sets under static scheduling and do not apply to preemptive systems with commonly used priority-based scheduling schemes.

This work addresses this issue by proposing a novel analysis that captures the re-use of cache blocks between job executions, to reduce the negative impact of preemptions on the WCRT bound. The approach presented in this work is orthogonal to the state-of-the-art methods used for CRPD calculations and can be integrated with any of these methods. The four main contributions of the paper are as follows: 1) We introduce the concept of persistent cache blocks (PCBs) in the context of WCRT analysis. PCBs are cache blocks that, once loaded into the cache by a task \( \tau_i \) will never be evicted when \( \tau_i \) runs in isolation. This concept allows us to capture the re-use of cache blocks between executions of the same task and reduce the memory demand for subsequent jobs of a task, making its memory demand variable, 2) A cache-persistence-aware WCRT analysis for fixed-priority preemptive systems that exploits the variable memory demand of preempting tasks to reduce the WCRT bound, 3) An extension of the proposed WCRT analysis to a multi-set approach that further improves the WCRT bound by considering the total memory demand of the preempting tasks over a task response time rather than the worst-case memory demand of each independent job, and 4) An experimental evaluation showing that our cache-persistence-aware WCRT analysis results in up to 10% higher schedulability than state-of-the-art approaches.

The rest of the paper is organized as follows. Section II presents the system model, while Section III discusses the state-of-the-art in CRPD calculation. Section IV then motivates our approach and introduces the key concept of persistent cache blocks. The basic cache-persistence-aware WCRT analysis is presented in Section V and is then extended to a multi-set approach in Section VI. Section VII explains how the inputs for our approach is obtained using static analysis, before experimental results are presented in Section VIII. Lastly, conclusions are drawn in Section IX.
II. SYSTEM MODEL

This work targets single-core platforms with a single level (L1) data/instruction cache. The cache is assumed to be direct-mapped, which means that each memory block in the main memory can be mapped to only one block in the cache.

We consider sporadic tasks with constrained deadlines where each task has a fixed priority. Any priority assignment scheme (e.g., Rate Monotonic [12]) is acceptable. We also assume that the tasks are independent and do not suspend themselves during their execution. A task $\tau_i$ is defined by a triplet $(C_i, T_i, D_i)$, where $C_i$ is the WCET of $\tau_i$, $T_i$ is its minimum inter-arrival time and $D_i$ is the relative deadline of each instance (or job) of $\tau_i$. We assume that the tasks have constrained deadlines, i.e., $D_i \leq T_i$. Similarly to [13], we further decompose each task WCET into separate terms for processing demand and memory demand, respectively. Here, we use two terms, namely, the worst-case processing demand $P_i$ and the worst-case memory demand $MD_i$. $P_i$ denotes the worst-case execution time of $\tau_i$ considering that every memory access is a cache hit. Consequently, it only accounts for execution requirements of the task and does not include the time needed to fetch data and instructions from the main memory. $MD_i$ is the worst-case memory demand of any job of task $\tau_i$, that is, it is the maximum time during which any job of $\tau_i$ is performing memory operations. The values for $C_i$, $P_i$ and $MD_i$ are calculated assuming $\tau_i$ executes in isolation. It is also important to note that the worst-case processing demand and the worst-case memory demand may not necessarily be experienced on the same execution path of $\tau_i$, as a result it holds that $C_i \leq P_i + MD_i$.

The WCRT of task $\tau_i$ is denoted by $R_i$ and is defined as the longest time between the arrival and the completion of any of its jobs. A task $\tau_i$ is said to be schedulable if $R_i \leq D_i$. Similarly, a task set is schedulable if all of its tasks are schedulable.

In this work, we consider that preemption costs only refer to additional cache reloads due to those preemptions. Other overheads, e.g., due to context switches, scheduler invocations and pipeline flushes are assumed to be included in the WCET. The worst-case reload time of a cache block is denoted by $d_{mem}$.

We define the following task sets:

- $\text{hp}(i)$: the set of tasks with higher priority than $\tau_i$.
- $\text{hep}(i, j)$: the set of tasks with priorities higher than or equal to $\tau_i$.
- $\text{aff}(i, j)$: the set of tasks with priorities higher than or equal to the priority of $\tau_i$ (including $\tau_i$), but strictly lower than that of $\tau_j$. This set contains the intermediate priority tasks, which may affect the response time of $\tau_i$, but may also be preempted by $\tau_j$.

III. BACKGROUND

This section discusses state-of-the-art methods in more detail and establish the formal framework on which we later build our analysis. As previously mentioned, when a task $\tau_i$ is preempted by a higher priority task $\tau_j$, it is likely that $\tau_j$ will evict memory blocks of $\tau_i$ from the cache. On resumption, $\tau_i$ might consequently have to reload cache blocks from the main memory along with its normal memory requirements. This CRPD caused by $\tau_j$ on $\tau_i$ is denoted by $\gamma_{i,j}$. Several methods have been proposed in the literature to compute $\gamma_{i,j}$. In one of the earlier works, Lee et al. [4] introduced the concept of useful cache block (UCB), and defined it as, “a memory block $m$ is called a useful cache block (UCB) at program point $P$, if it is cached at $P$ and will be reused at program point $Q$ that may be reached from $P$ without eviction of $m$”. This definition was later improved by Altmeyer et al. [14], however in this work we only use the basic concept provided in [4]. Lee et al. [4] used the maximum number of UCBs among all the tasks in $\text{aff}(i, j)$ to upper bound the preemption cost $\gamma_{i,j}$. Busquets et al. [3] and Tomiyama et al. [5] rather used the notion of evicting cache block (ECB), i.e., any cache block accessed during the execution of the task and which can then evict the memory block cached by another task, to upper bound the preemption cost that can be caused by each preemption task. Other approaches by Tan and Mooney [7], Staschulat et al. [6] and Altmeyer et al. [8] used both the UCBs of the preempted tasks and ECBs of the preempting task in order to come up with more precise bounds on the preemption cost. Notably, the ECB and UCB-union and the multi-set approaches presented in [8] and [9] dominate all the existing approaches for CRPD calculation. We first detail the ECB-union approach and then the UCB-union multi-set. The formulations for the UCB-union and ECB-union multi-set can be found in [9].

The ECB-union approach [8] uses the ECBs of all tasks in $\text{hep}(j)$ maximized over the UCBs of tasks in $\text{aff}(i, j)$ to calculate the preemption cost $\gamma_{i,j}$. The resulting value for the preemption cost, denoted as $\gamma_{i,j}^{\text{ecb}}$, is given by

$$\gamma_{i,j}^{\text{ecb}} = d_{mem} \times \max_{k \in \text{aff}(i, j)} \left( \sum_{i,j} \left( UCB_k \cap \bigcup_{l \in \text{hep}(j)} ECB_l \right) \right)$$

(1)

where $d_{mem}$ is the time required to reload one memory block from the main memory to the cache, and $UCB_k$ and $ECB_j$ are the sets of UCBs and ECBs of task $\tau_k$ and $\tau_j$, respectively. The preemption cost can then be accounted for in the WCRT analysis using the following formulation:

$$R_i^{\text{ecb}} = C_i + \sum_{j \in \text{hep}(i)} \left( \frac{R_j^{\text{ecb}}}{T_j} \right) \times (C_j + \gamma_{i,j}^{\text{ecb}})$$

(2)

When combined, the ECB and UCB-union approaches provide a reasonably precise upper bound on the preemption cost. However, it can also lead to over-estimations in different situations, as shown in [9]. This is due to the fact that both ECB and UCB-union approaches do not take into account the actual number of preemptions of each low and intermediate priority task. For instance, with these approaches it is assumed that a high priority task $\tau_k$ can preempt any task $\tau_k \in \text{aff}(i, j)$ the same number of times it can preempt $\tau_i$. This can only be true if $\tau_k = \tau_i$, and will lead to over-estimation in all other cases where the cost of $\tau_j$ preempting $\tau_k$ is higher than the preemption cost of $\tau_j$ on $\tau_i$.

To reduce this pessimism associated to the ECB and UCB-union approaches, Altmeyer et al. [9] proposed two new so-
lutions, namely, the UCB-union multi-set and the ECB-union multi-set approaches. These multi-set versions of the UCB-union and ECB-union approaches additionally take into account the maximum number of jobs \( E_j(R_k) \) that each higher priority task \( \tau_j \) can release during the response time of \( \tau_i \) and the number of preemptions of each low and intermediate priority task by \( \tau_j \), i.e., \( E_j(R_k)E_k(R_i) \). Under this framework, the WCRT equation becomes:

\[
R_i^{mul} = C_i + \sum_{j \in \text{hp}(i)} \left[ \frac{R_i^{mul}}{T_j} \right] \times C_j + \sum_{j \in \text{hp}(i)} \gamma_{i,j}^{mul}
\]

(3)

where \( \gamma_{i,j}^{mul} \) accounts for the total preemption cost that can be caused by all jobs of \( \tau_j \) released during the response time of \( \tau_i \). We detail the UCB-union multi-set approach and refer the reader to [9] for the ECB-union multi-set formulation. Using the UCB-union multi-set approach \( \gamma_{i,j}^{mul} \) is upper bounded by \( \gamma_{i,j}^{uch-m} \) defined as follows:

\[
\gamma_{i,j}^{uch-m} = d_{mem} \times |M_{i,j}^{uch} \cap M_{i,j}^{ecb}|
\]

(4)

where \( M_{i,j}^{uch} \) and \( M_{i,j}^{ecb} \) are multi-sets defined as

\[
M_{i,j}^{uch} = \bigcup_{k \in \text{aff}(i,j)} \left( \bigcup_{E_j(R_k)E_k(R_i)} UCB_j \right)
\]

(5)

and

\[
M_{i,j}^{ecb} = \bigcup_{E_j(R_i)} ECB_j
\]

(6)

Here, \( M_{i,j}^{uch} \) is a multi-set comprising sets of UCBS of all low and intermediate priority tasks \( \tau_j \) added \( E_j(R_k)E_k(R_i) \) times, i.e., the maximum number of times \( \tau_j \) can preempt each \( \tau_k \) during the response time of \( \tau_i \). Similarly, \( M_{i,j}^{ecb} \) is a multi-set comprising the set of ECBS of all jobs of \( \tau_j \) executing within the response time of \( \tau_i \). The final value of the preemption cost \( \gamma_{i,j}^{uch-m} \) comes from the intersection of both these multi-sets.

The construction of the ECB-union multi-set approach is analogous to the UCB-union multi-set approach. Note that the ECB-union multi-set approach dominates the ECB-union approach [8], while the UCB-union multi-set approach dominates the UCB-union approach [7]. Yet, it is shown in [9] that the ECB-union and UCB-union multi-set approaches are incomparable. For a more detailed description of the formulation of Equations (2) to (6), the reader is referred to [9].

IV. PROBLEM DEFINITION

In this section, first we provide a basic example to affirm the motivation behind this work. Later, using this example as a base we provide some useful definitions that will be used in rest of the paper.

A. Motivational Example

As presented in the previous section, the impact of a high priority task \( \tau_1 \) on the WCRT of any lower priority task \( \tau_i \) can be estimated in a fairly accurate manner by analyzing the mapping of UCBS and ECBS in the cache. However, the impact of \( \tau_i \) on the memory demand of \( \tau_j \) is ignored during the WCRT analysis of \( \tau_i \). Yet, high priority tasks may often execute more than one job during the response time of a lower priority task. Therefore, to accurately estimate the WCRT of a low priority task \( \tau_i \), one must consider the impact of the preempted tasks on the memory demand of each job released by the preempting tasks. In the literature, this is dealt with by assuming that the memory demand for each job of a high priority task \( \tau_j \) executing within the response time of a low priority task \( \tau_i \) is always maximum, i.e., equal to the maximum memory demand \( MD_i \). Following that assumption, the total memory overhead \( MO_i \) that must be accounted by \( \tau_i \) during its worst-case response time is upper bounded by the following equation derived in [15].

\[
MO_i = MD_i + \sum_{j \in \text{hp}(i)} \left[ \frac{R_i}{T_j} \right] \times (MD_j + \gamma_{i,j})
\]

(7)

There is a significant level of pessimism involved in Equation (7), as we will demonstrate using the example below.

Example 1. Consider the two tasks \( \tau_1 \) and \( \tau_2 \) (where \( \tau_1 \) has a higher priority than \( \tau_2 \) ) presented in Figure 1. We assume that the time \( d_{mem} \) needed to access the main memory and load a memory block to the cache is equal to 10 time units and that the memory demand of \( \tau_1 \) and \( \tau_2 \) are \( MD_1 = 60 \) and \( MD_2 = 80 \). We also assume that memory block \( \{9\} \) accessed by \( \tau_1 \) contains data that must be updated at the beginning of the execution of each of its jobs. Figure 1 depicts a possible schedule together with the evolution of the cache contents over time. The memory blocks that must be loaded/reloaded from the main memory after each preemption or resumption are shown in bold with a bigger font size in Figure 1.

\[\text{Example 1. Consider the two tasks } \tau_1 \text{ and } \tau_2 \text{ (where } \tau_1 \text{ has a higher priority than } \tau_2 \text{) presented in Figure 1. We assume that the time } d_{mem} \text{ needed to access the main memory and load a memory block to the cache is equal to 10 time units and that the memory demand of } \tau_1 \text{ and } \tau_2 \text{ are } MD_1 = 60 \text{ and } MD_2 = 80 \text{. We also assume that memory block } \{9\} \text{ accessed by } \tau_1 \text{ contains data that must be updated at the beginning of the execution of each of its jobs. Figure 1 depicts a possible schedule together with the evolution of the cache contents over time. The memory blocks that must be loaded/reloaded from the main memory after each preemption or resumption are shown in bold with a bigger font size in Figure 1.}\]

\[\text{Example 1. Consider the two tasks } \tau_1 \text{ and } \tau_2 \text{ (where } \tau_1 \text{ has a higher priority than } \tau_2 \text{) presented in Figure 1. We assume that the time } d_{mem} \text{ needed to access the main memory and load a memory block to the cache is equal to 10 time units and that the memory demand of } \tau_1 \text{ and } \tau_2 \text{ are } MD_1 = 60 \text{ and } MD_2 = 80 \text{. We also assume that memory block } \{9\} \text{ accessed by } \tau_1 \text{ contains data that must be updated at the beginning of the execution of each of its jobs. Figure 1 depicts a possible schedule together with the evolution of the cache contents over time. The memory blocks that must be loaded/reloaded from the main memory after each preemption or resumption are shown in bold with a bigger font size in Figure 1.}\]
Initially, the cache is empty and \( \tau_2 \) loads all its ECBs from the main memory as soon as it starts to execute. When \( \tau_1 \) preempts \( \tau_2 \) for the first time, it also loads all its ECBs into the cache with a memory demand of \( MD_1 = 60 \). Since there is an overlap between the mapping of ECBs of \( \tau_1 \) and the mapping of UCBs of \( \tau_2 \) in the cache, \( \tau_1 \) evicts some of the useful cache blocks of \( \tau_2 \). In turn, when \( \tau_2 \) resumes its execution, it has to account for \( \gamma_{2,1} = 2 \times d_{mem} = 20 \), in order to load cache blocks \( \{5,6\} \) again from main memory. However, when the second job of \( \tau_1 \) preempts \( \tau_2 \), one can notice that it no longer needs to reload all of its ECBs. In fact, most of the memory blocks needed by \( \tau_1 \) are still in the cache. As a consequence, \( \tau_1 \) must only reload memory blocks \( \{5,6\} \), which have been evicted by \( \tau_2 \), as well as memory block \( \{9\} \) that must be reloaded for each new job execution of \( \tau_1 \). The same scenario happens for all jobs released by \( \tau_1 \), except the first one. The actual memory demand for the second and third job of \( \tau_1 \) is hence much less (i.e., 30) than \( MD_1 = 60 \), illustrating that it is not constant across all job executions.

In the presented example, memory blocks \( \{5, 6, 7, 8, 10\} \) are called persistent cache blocks (PCBs), as they are never evicted from the cache once loaded when \( \tau_1 \) executes in isolation. In contrast, cache block \( \{9\} \) is a non-persistent cache block (nPCB). nPCBs may be cache blocks that are shared by several memory blocks of the same task, or simply data (e.g., sensor readings, value on an input port, global shared data) that must be reloaded for each new access. One must note that PCBs and nPCBs are different from the notions of UCBs and ECBs in the sense that it does not matter if they are referenced more than once during a single execution of a task. However, a PCB must never be evicted from the cache by the task itself once it is fetched from main memory.

The state-of-the-art does not consider PCBs while calculating the memory overhead suffered by a task \( \tau_1 \) in case of preemptions. This results in pessimistic memory overhead evaluations and hence pessimistic WCRT computations. This can easily be shown using the example in Figure 1. If \( \tau_2 \)'s memory overhead is computed using Equation (7), one would get:

\[
MO_2 = MD_2 + 3 \times MD_1 + 3 \times \gamma_{2,1} = 80 + 3 \times 60 + 3 \times 20 = 320
\]

Equation (7) considers the worst-case memory demand, i.e., \( MD_1 \) for each job of \( \tau_1 \) that executes during the response time of \( \tau_2 \). As we have shown in Example 1, the actual memory demand of the second and third job of \( \tau_1 \) is in fact much less. Considering the PCBs of \( \tau_1 \) while calculating the memory overhead \( MO_2 \), the resulting value is given as:

\[
MO_2 = MD_2 + MD_1 + 2 \times (MD_1 - \left| PCB_1 \right| \times d_{mem}) + 3 \times \gamma_{2,1} = 80 + 60 + 2 \times (60 - 5 \times 10) + 3 \times 20 = 220
\]

This simple example highlights the necessity to consider PCBs when calculating the memory demand and hence the WCRT of a task.

### B. Problem Formalization

The previous example casually introduced the notions of PCB and nPCB. We now formally define those two types of cache blocks associated to the execution of a task \( \tau_i \).

**Definition 1** (Persistent cache block). A memory block of a task \( \tau_i \) is persistent if once loaded by \( \tau_i \), it will never be invalidated or evicted from the cache when \( \tau_i \) executes in isolation.

**Definition 2** (Non-persistent cache block). A non-persistent cache block (nPCB) of task \( \tau_i \) is an ECB that is not a PCB. That is, it is a memory block that may need to be reloaded at some point during the execution of \( \tau_i \) (in the same or different jobs), even when \( \tau_i \) executes in isolation.

The sets of PCBs and nPCBs associated to a task \( \tau_i \) are denoted by \( PCB_i \) and \( nPCB_i \), respectively. It follows from the two previous definitions that each cache block associated to a task \( \tau_i \) (ECB\(_i\)) is either a PCB or a nPCB, hence the following two relations:

\[
PCB_i \cup nPCB_i = ECB_i
\]

\[
PCB_i \cap nPCB_i = \emptyset
\]

By Definition 1, if \( \tau_i \) executes in isolation, a PCB is loaded only once from the main memory and hence contributes only once to the total memory demand of \( \tau_i \). Even though all the ECBs of \( \tau_i \) (i.e., PCBs and nPCBs) contribute to its worst-case memory demand in isolation (i.e., \( MD_i \)), only the nPCBs, a subset of \( ECB_i \), must be loaded by more than one job of \( \tau_i \). Considering the worst-case memory demand for each job released by higher priority tasks than \( \tau_i \) when computing the WCRT of \( \tau_i \), as is implicitly the case in Equations (2) and (3), is thus pessimistic. Therefore, we define the residual memory demand of a task \( \tau_i \) as the worst-case memory demand of \( \tau_i \) assuming that all the PCBs of \( \tau_i \) are already in the cache memory and therefore result in cache hits when being accessed.

**Definition 3** (Residual memory demand). The residual memory demand \( MD'_i \) of task \( \tau_i \) is the worst-case memory demand over all the jobs of \( \tau_i \) when all its PCBs are already loaded in the cache memory. Therefore, \( MD'_i \) only accounts for the accesses to the nPCBs of \( \tau_i \) and can occur during any job execution of \( \tau_i \).

An upper bound on the total memory demand \( MD_i(t) \) of a task \( \tau_i \) within a time window of length \( t \) when \( \tau_i \) executes in isolation is proven in the following lemma.

**Lemma 1.** If a task \( \tau_i \) executes in isolation, then its total memory demand \( MD_i(t) \) within a time window of length \( t \) is upper bounded by \( MD_i(t) \) where

\[
\hat{MD}_i(t) \overset{\text{def}}{=} \min \left\{ \left[ \frac{t}{T_i} \right] \cdot MD_i ; \left[ \frac{t}{T_i} \right] \cdot MD_i + \left| PCB_i \right| \times d_{mem} \right\}
\]

**Proof.** We prove that \( \left[ \frac{t}{T_i} \right] \cdot MD_i \) and \( \left[ \frac{t}{T_i} \right] \cdot MD_i + \left| PCB_i \right| \times d_{mem} \) are both upper bounds on the total memory demand.
MD_i(t) of τ_i. Thus, the minimum of those bounds is also an upper bound on MD_i(t).

1) τ_i can release at most \( \frac{\tau_i}{\ell_i} \) jobs in a time window of length \( \ell_i \). By definition of MD_i, each of these jobs has a worst-case memory demand MD_i. Therefore, \( \frac{\tau_i}{\ell_i} MD_i \) is an upper bound on the total memory demand of τ_i.

2) Recall from Equations (8) and (9) that PCB_i ∪ nPCB_i = ECB_i and PCB_i ∩ nPCB_i = ∅. Characterizing the worst-case contribution of the PCBs and nPCBs to the total memory demand is therefore sufficient to quantify the worst-case contribution of all the cache blocks of τ_i to MD_i(t). Since by Definition 1, the persistent cache blocks must be loaded only once, the maximum contribution of the cache blocks in PCB_i to MD_i(t) is \( |PCB_i| \times \Delta_{mem} \) (i.e., the total number of PCBs times the worst-case memory access time). By Definition 3, the worst-case contribution of nPCBs to the memory demand of each job released by τ_i is MD_i. Since a maximum of \( \frac{\tau_i}{\ell_i} \) jobs are released by τ_i in a time window of length \( \ell_i \), an upper bound on the total contribution of the nPCBs to MD_i(t) is given by \( \frac{\tau_i}{\ell_i} MD_i \). Adding the contributions of nPCBs and PCBs, we get that \( \frac{\tau_i}{\ell_i} MD_i + |PCB_i| \times \Delta_{mem} \) is an upper bound on the total memory demand of τ_i.

Although Equation (10) provides an upper bound on the total memory demand of τ_i in isolation, the total memory demand of τ_i when executing concurrently with other tasks can be much larger. Indeed, as can be observed in Example 1, the PCBs of a task τ_j can be evicted due to the execution of any task (i.e., tasks in hep(i) \ τ_j) between the execution of two successive jobs of τ_j. This requires the effect of all tasks in hep(i) \ τ_j on the memory demand of τ_j ∈ hep(i) during the WCRT of τ_j to be taken into account. We refer to this extra memory demand caused by the eviction of PCBs of τ_j by the tasks in hep(i) \ τ_j as cache-persistence reload overhead (CPRo) and denote it by ρ_j,i. CPRo is formally defined as:

**Definition 4** (Cache-persistence reload overhead). Cache-persistence reload overhead, denoted by ρ_j,i, is the maximum memory overhead of any task τ_j due to evictions of its PCBs resulting from the execution of all tasks in hep(i) \ τ_j, while τ_j is executing during the response time of τ_i.

V. CPRo UNION APPROACH

In this section, we present a simple approach similar to the ECB-union to calculate the CPRo (i.e., ρ_j,i). We further demonstrate how ρ_j,i can be incorporated in the WCRT analysis of a task τ_i. Later, in Section VI, we extend this simple union approach into a multi-set variant to remove some of the pessimism associated with this analysis.

A. Computation of Cache-Persistence Reload Overhead

As discussed in Section IV-B, ρ_j,i accounts for the extra memory demand of each job of τ_j ∈ hep(i) due to evictions of its persistent cache blocks by other tasks running concurrently on the processor. As one can see in Figure 1, the PCBs of a task τ_j ∈ hep(i) can be evicted by the ECBS of any other task running on the platform between two successive jobs of τ_j. The memory demand overhead ρ_j,i can thus be upper bounded by the intersection of the set PCB_j of all PCBs of τ_j with all cache blocks (i.e., ECBS) that can be loaded by any other task between two executions of τ_j. This observation leads to the following theorem.

**Theorem 1.** The cache-persistence reload overhead imposed by the eviction of PCBs of a job of task τ_j ∈ hep(i) on the worst-case response time of a task τ_i is upper bounded by

\[
\rho_{j,i} = d_{mem} \times |PCB_j \cap \bigcup_{\forall k \in hep(i) \setminus \tau_j} ECB_k| \tag{11}
\]

**Proof.** Since a fixed-priority scheduling algorithm is used, only tasks with priorities higher than or equal to the priority of τ_i (i.e., tasks in hep(i)) can execute during the response time of τ_i. Therefore, any task τ_k ∈ hep(i) \ τ_j can execute between two subsequent jobs of τ_j and hence evict some or all the PCBs of τ_j.

The worst-case memory interference of any task τ_k ∈ hep(i) \ τ_j on τ_j is when it reloads all its cache blocks (i.e., its ECBS) between two subsequent jobs of τ_j. Therefore, the largest set of memory blocks loaded by tasks in hep(i) \ τ_j between two jobs of τ_j is given by \( \bigcup_{\forall k \in hep(i) \setminus \tau_j} ECB_k \).

The set of persistent cache blocks that must be reloaded by τ_j during each job execution is thus upper bounded by the intersection between τ_j’s PCBs (i.e., PCB_j) and the set of all PCBs of τ_j that have been evicted by previous tasks (i.e., PCB_j \ ∪ \( \bigcup_{\forall k \in hep(i) \setminus \tau_j} ECB_k \)).

Since each cache block reload takes at most \( d_{mem} \) time units, the CPRo due to the eviction of PCBs of τ_j by tasks in hep(i) \ τ_j is upper bounded by

\[
d_{mem} \times |PCB_j \cap \bigcup_{\forall k \in hep(i) \setminus \tau_j} ECB_k| \tag{12}
\]

**Proof.** Having defined an expression to calculate ρ_j,i, we now define \( \hat{\rho}_{j,i}(t) \), i.e., the total cache-persistence reload overhead on τ_j in a time window of length \( \ell_i \) due to the eviction of its PCBs by tasks in hep(i) \ τ_j. \( \hat{\rho}_{j,i}(t) \) tells us by how much the memory demand of τ_j can vary in comparison to its memory demand in isolation (i.e., MD_j(t)) due to the interference generated by the other tasks executing concurrently with τ_j. Using Theorem 1, \( \hat{\rho}_{j,i}(t) \) can be easily computed as stated in Lemma 2 below.

**Lemma 2.** The total CPRo \( \hat{\rho}_{j,i}(t) \) on the execution time of τ_j due to the eviction of its PCBs by tasks in hep(i) \ τ_j in a time interval of length \( t \) is upper bounded by \( \hat{\rho}_{j,i}(t) \) where

\[
\hat{\rho}_{j,i}(t) \defeq \left( \frac{\ell_i}{T_j} \right) - 1 \times \rho_{j,i} \tag{12}
\]

**Proof.** It directly follows from the fact that τ_j releases at most \( \frac{\ell_i}{T_j} \) jobs in a time interval of length \( t \). As a result, at most
\[
\left(\left\lceil \frac{1}{T_j} \right\rceil - 1\right) \text{ evictions can happen between two subsequent jobs of } \tau_j. \text{ Since by Theorem 1, the CPRO suffered by a job of } \tau_j \text{ is upper bounded by } \rho_{i,j}, \text{ the total overhead } \rho_{j,i}(t) \text{ is upper bounded by } \left(\left\lceil \frac{1}{T_j} \right\rceil - 1\right) \times \rho_{j,i}.
\]

B. WCRT Analysis

After showing how the extra memory demand overhead \(\rho_{j,i}\) of a high priority task \(\tau_j\) can be computed, we now describe how it can be integrated into the WCRT analysis of any lower priority task \(\tau_i\). As mentioned in Section III, the WCRT analysis for fixed-priority preemptive systems was first presented in [16], [17] without considering memory overheads due to preemptions. It was then extended in several works (e.g., [3], [8], [9]) to account for the cache-related preemption delays. Some of the most prominent approaches resulted in Equations (2) and (3), previously presented in Section III.

Although these approaches are beneficial, their WCRT analysis still rely exclusively on the WCET \(C_j\) of high priority tasks when computing the worst-case response time of a low priority task \(\tau_i\). That is, it assumes that each job of a task \(\tau_j \in h\text{hp}(i)\) executing within the response time of \(\tau_i\) asks for its worst-case memory demand \(MD_j\). As discussed in Section IV, this assumption is pessimistic. In fact, due to the existence of persistent cache blocks, once \(\tau_j\) loads all its ECBs (i.e., PCBs and nPCBs), subsequent jobs of \(\tau_j\) will only need to reload nPCBs and some of the PCBs that may have been evicted due to the execution of tasks in \(h\text{hp}(i) \setminus \tau_j\). As a result, for subsequent jobs of \(\tau_j\), the memory demand will be significantly lower than \(MD_j\). To exploit this variable memory demand, we present a more elaborate formulation of the WCRT analysis.

We propose that for any task \(\tau_i\), the WCRT of task \(\tau_i\) is upper bounded by the smallest positive value \(R_i\) such that

\[
R_i = C_i + \sum_{\forall j \in h\text{hp}(i)} \left( P_j(R_i) + MD_j(R_i) + \rho_{j,i}(R_i) + \gamma_{i,j}(R_i) \right)
\]

Replacing each term with its given bound, we get that

\[
R_i \leq C_i + \sum_{\forall j \in h\text{hp}(i)} \left( \frac{R_j}{T_j} \right) P_j + \sum_{\forall j \in h\text{hp}(i)} MD_j(R_i) + \rho_{j,i}(R_i) + \gamma_{i,j}(R_i)
\]

In systems where the number of PCBs is high and the cache interference is low, the value provided by \(MD_j(R_i) + \rho_{j,i}(R_i)\) should always be smaller than \(\frac{R_j}{T_j} \cdot MD_i\), and therefore we should often have \(\frac{R_j}{T_j} P_j + MD_j(t) + \rho_{j,i}(R_i)\) smaller than \(\left[ \frac{R_j}{T_j} \right] C_j\). In this case, Equation (14) will result in a tighter WCRT bound than Equation (3). However, in some situations, since \(MD_j(t) + \rho_{j,i}(R_i)\) are upper bounds and not exact values, this formulation can result in an over-estimation of the interference generated by \(\tau_j\) on \(\tau_i\). In order to counter this effect, and knowing that Equation (3) is already an upper bound on the WCRT of \(\tau_i\), we further improve Equation (14) by always taking the minimum between \(\left[ \frac{R_j}{T_j} \right] C_j\) and \(\left[ \frac{R_j}{T_j} \right] P_j + MD_j(t) + \rho_{j,i}(R_i)\) as the total interference caused by \(\tau_j\) on \(\tau_i\) (see Equation (15) below).

Following this simple modification to Equation (14), Equation (15) will always return a value that is smaller than or equal to the solution to Equation (3). Our approach hence dominates the UCB union multi-set approach defined in [9].

\[
R_i^\text{un} = C_i + \min_{\forall j \in h\text{hp}(i)} \left\{ \left\lceil \frac{R_j}{T_j} \right\rceil C_j ; \left[ \frac{R_j}{T_j} \right] P_j + MD_j(R_i) + \rho_{j,i}(R_i) \right\} + \sum_{\forall j \in h\text{hp}(i)} \gamma_{i,j}^\text{mul}
\]

In this WCRT formulation, we separately account for the maximum processing demand \(P_j(R_i)\) and memory demand \(MD_j(R_i)\) (in isolation) that can be claimed by each higher priority task \(\tau_j\) within the response time \(R_i\) of \(\tau_i\). The terms \(\rho_{j,i}(R_i)\) and \(\gamma_{i,j}(R_i)\) denote the total cache-persistence reload overhead due to the eviction of PCBs of \(\tau_j\) by tasks in \(h\text{hp}(i) \setminus \tau_j\), and the total cache-related preemption delay due to the preemptions caused by \(\tau_j\) within the response time of \(\tau_i\), respectively. The terms \(P_j(R_i) + MD_j(R_i)\) assume values obtained in isolation, while the last two terms \(\left( \rho_{j,i}(R_i) + \gamma_{i,j}(R_i) \right)\) account for the overheads introduced by the eviction of cache blocks by other tasks sharing the cache.

As already discussed in Section III, \(\gamma_{i,j}(R_i)\) is upper bounded by \(\gamma_{i,j}^\text{mul}\). Furthermore, as proven in Lemmas 1 and 2, \(MD_j(R_i)\) and \(\rho_{j,i}(R_i)\) are upper bounded by Equations (10) and (12), respectively. Finally, because each task \(\tau_j\) releases at most \(\left[ \frac{R_i}{T_j} \right] \) jobs in a time window of length \(t\), \(P_j(R_i)\) is smaller than or equal to \(\left[ \frac{R_i}{T_j} \right] P_j\).

VI. CPRO Multi-Set Approach

The formulation in Equations (11) and (12) considers the ECBs of all tasks \(\tau_k \in h\text{hp}(i) \setminus \tau_j\) as interfering with every job of \(\tau_j\) released within the response time of \(\tau_i\). This is pessimistic. Indeed, considering two different tasks \(\tau_k\) and \(\tau_l\) pertaining to \(h\text{hp}(i) \setminus \tau_j\), the number of times \(\tau_i\) can execute between two successive jobs of \(\tau_j\) is not necessarily equal to the number of times \(\tau_k\) can execute between two successive jobs of \(\tau_j\). This situation is discussed in Example 2.

**Example 2.** Let \(\tau_1 = (1,4,4), \tau_2 = (4,30,30)\) and \(\tau_3 = (10,50,50)\), where \(\tau_1\) has the highest priority and \(\tau_3\) the lowest. Figure 2 presents a possible schedule that generates the worst-case response time of \(\tau_3\). As one can see, \(\tau_1\) releases 5 jobs during the response time of \(\tau_3\). As a result, Equation (15) upper bounds the total cache overheads on the PCBs of \(\tau_1\) with 4 times \(\rho_{1,3}\). That is, it assumes that both \(\tau_2\) and \(\tau_3\) execute and reload all their ECBs between every
two successive jobs of $\tau_1$. As can be seen in Figure 2, this is pessimistic. In fact, $\tau_2$ execute only twice between jobs of $\tau_1$! Its impact on the total CPRO of $\tau_1$ is therefore clearly overestimated.

In order to reduce the pessimism associated with the computation of $\rho_{j,i}$, we must consider the actual number of times each task $\tau_k \in \text{hep}(i) \setminus \tau_j$ can execute between two successive jobs of $\tau_j$. For this reason, this section presents a multi-set variant of Equation (12). The resulting quantity is an upper bound on $\bar{\rho}_{j,i}(t)$ denoted by $\rho_{j,i}^{\text{mul}}(t)$.

A. Computation of $\rho_{j,i}^{\text{mul}}(t)$
In this section, we first characterize the maximum number of times a task $\tau_k \in \text{hep}(j) \setminus \tau_j$ can execute between two successive jobs of $\tau_j$. To do so, we separately analyze the tasks in $\text{hep}(j) \setminus \tau_j$ (Lemma 3) and $\text{aff}(i,j)$ (Lemma 4). We then use this information to upper bound the total cache-persistence reload overhead $\rho_{j,i}(t)$ in Theorem 2.

**Lemma 3.** The maximum number of times a task $\tau_k \in \text{hep}(j) \setminus \tau_j$ can execute between two successive jobs of $\tau_j$ within the response time $R_i$ of $\tau_i$ is upper bounded by $E_k(R_i)$.

**Proof.** Remember that the maximum number of jobs that each higher priority task $\tau_k$ can release during the response time of a task $\tau_i$ is given by $E_k(R_i) = \left\lfloor \frac{R_i}{T_{\tau_k}} \right\rfloor$. Furthermore, because $\tau_k$ has a higher or equal priority than $\tau_j$, $\tau_j$ cannot preempt $\tau_k$. Hence, the maximum number of time $\tau_i$ can execute between two successive jobs of $\tau_j$ within a time window of length $R_i$ is upper bounded by its number of released jobs $E_k(R_i)$ (see Figure 3 for an example). □

**Lemma 4.** The maximum number of times a task $\tau_k \in \text{aff}(i,j)$ can execute between two successive jobs of $\tau_j$ within the response time $R_i$ of $\tau_i$ is upper bounded by

$$E_j(R_k) + 1 \times E_k(R_i)$$

**Proof.** $E_j(R_k) = \left\lfloor \frac{R_k}{T_{\tau_j}} \right\rfloor$ provides the maximum number of jobs that $\tau_j$ can release during the response time of a task $\tau_k$. Each of these released jobs may preempt the execution of $\tau_k$. Considering an arrival pattern such that $\tau_k$ started to execute just before the first arrival of $\tau_j$ preempting $\tau_k$ (see Figure 3), the maximum number of times a job of $\tau_k$ may execute between two successive jobs of $\tau_j$ is then given by $(E_j(R_k) + 1)$. Since $E_k(R_i)$ jobs of $\tau_k$ are released within the response time of $\tau_i$, the maximum number of times $\tau_k$ may execute between two successive jobs of $\tau_j$ within the response time of $\tau_i$ is upper bounded by $(E_j(R_k) + 1) \times E_k(R_i)$. □

Using Lemmas 3 and 4, one can derive an upper bound on $\rho_{j,i}(t)$. This upper bound is denoted by $\rho_{j,i}^{\text{mul}}(t)$ and is defined in the following theorem.

**Theorem 2.** The total cache-persistence reload overhead $\rho_{j,i}(R_k)$ on $\tau_j$ due to the eviction of its PCBs by tasks in $\text{hep}(i) \setminus \tau_j$ during the response time $R_i$ of $\tau_i$ is upper bounded by

$$\rho_{j,i}^{\text{mul}} = d_{\text{mem}} \times \left| M_j^{\text{ecb}} \cap M_j^{\text{pcb}} \right|$$

where $M_j^{\text{ecb}}$ and $M_j^{\text{pcb}}$ are multi-sets defined as

$$M_j^{\text{pcb}} = \bigcup E_j(R_i)-1 PCB_j$$

and

$$M_j^{\text{ecb}} = M_j^{\text{ecb-aff}} \cup M_j^{\text{ecb-hp}}$$

with

$$M_j^{\text{ecb-aff}} = \bigcup_{\tau_k \in \text{aff}(i,j)} \left( \bigcup_{E_j(R_k)+1} E_k(R_i) \right)$$

and

$$M_j^{\text{ecb-hp}} = \bigcup_{\tau_k \in \text{hep}(j) \setminus \tau_j} E_k(R_i)$$

**Proof.** The proof is based on the three following facts:

1. $\tau_j$ releases at most $\left\lfloor \frac{1}{T_{\tau_j}} \right\rfloor$ jobs in a time window of length $t$.
2. By Lemma 3, the maximum number of times a task $\tau_k \in \text{hep}(j) \setminus \tau_j$ can execute between two successive jobs of $\tau_j$ during the response time of $\tau_i$ is upper bounded by $E_{E_j(R_k)}$. Hence, the largest set of ECBs that can be loaded by $\tau_i$ is given by $\bigcup_{E_{E_j(R_k)}} ECB_i$.
3. By Lemma 4, the maximum number of times a task $\tau_k \in \text{hep}(i) \setminus \tau_j$ can execute between two successive jobs of $\tau_j$ during the response time of $\tau_i$ is upper bounded by $E_{E_j(R_k)}$. Hence, the largest set of ECBs that can be loaded by $\tau_i$ is given by $\bigcup_{E_{E_j(R_k)}} ECB_i$.
reloads all its ECBs whenever it resumes its execution). This
results in that the largest set of ECBs loaded by the tasks in
hep(j) \ τ_j between successive executions of τ_j is upper
bounded by $M_{j,i}^{ecb-hp} = \bigcup_{\forall k \in \text{hep}(j) \setminus \tau_j} \left( \bigcup_{E_k(R_j)} ECB_k \right)$.

3. By Lemma 4, the maximum number of times a task $\tau_k \in \text{aff}(i,j)$ can execute between two successive jobs of $\tau_j$ during the response time of $\tau_i$ is upper bounded by $(E_j(R_k) + 1) \times E_k(R_i)$. Hence, the largest set of ECBs that can be loaded by $\tau_k$ between successive jobs of $\tau_j$ during the response time of $\tau_i$ is given by $(E_j(R_k) + 1) E_k(R_i)$ (assuming that $\tau_k$ reloads all its ECBs whenever it resumes its execution). This results in that the largest set of ECBs loaded by the tasks in aff(i,j) between successive executions of $\tau_j$ is upper bounded by

$$M_{j,i}^{ecb-aff} = \bigcup_{\forall k \in \text{aff}(i,j)} \left( (E_j(R_k) + 1) E_k(R_i) \right) \bigcup ECB_k.$$ 

Therefore, by 2. and 3. the largest set of ECBs that can interfere with the PCBs of $\tau_j$ during the response time of $\tau_i$ is upper bounded by $M_{j,i}^{ecb} = M_{j,i}^{ecb-aff} \cup M_{j,i}^{ecb-hp}$.

Finally, the largest set of PCBs of $\tau_j$ that can be evicted by the tasks in hep(i) \ τ_j within the response time of $\tau_i$ is upper bounded by the intersection of $M_{j,i}^{ecb}$ with $M_{j,i}^{ecb}$. Since reloading a cache block takes at most $d_{mem}$ time units, the total cache-persistence reload overhead $\rho_{j,i}(R_i)$ is upper bounded by $d_{mem} \times \left| M_{j,i}^{ecb} \cap M_{j,i}^{ecb} \right|$.

B. Improving the Accuracy of $M_{j,i}^{ecb}$

Theorem 2 provides a good upper bound on the total cache-persistence reload overhead $\rho_{j,i}(R_i)$ during the response time of $\tau_i$. However, Equations (20) and (21) still consider that each job released by the tasks $\tau_k \in \text{hep}(i) \setminus \tau_j$ reload all their ECBs (i.e., PCBs and nPCBs) whenever they resume their execution. Even though this assumption may be valid for the tasks $\tau_j \in \text{hep}(i) \setminus \tau_j$, since each of their jobs contributes only once to $M_{j,i}^{ecb}$ (hence assuming that each job of $\tau_j$ accesses all its cache blocks during its execution), it is quite pessimistic for the tasks $\tau_k \in \text{aff}(i,j)$. Indeed, by Lemma 4 and Equation (20), each job of a task $\tau_k \in \text{aff}(i,j)$ is assumed to contribute $(E_j(R_k) + 1)$ times to $M_{j,i}^{ecb}$. However, a PCB of task $\tau_k$ will be accessed at most once during each job execution unless this PCB is also a UCB (in which case it may be used at several program points of the task). The nPCBs must always be considered to be loaded several times during each job execution though.

Indeed, since they are not persistent, it means that several memory blocks of $\tau_k$ are mapped to that same cache block, which can therefore be accessed more than once during each job execution.

It results from this discussion that $M_{j,i}^{ecb}$ can be more accurately modeled by the following equation:

$$M_{j,i}^{ecb} = M_{j,i}^{ecb-aff} \cup M_{j,i}^{ecb-hp}$$ (22)

with

$$M_{j,i}^{ecb-aff} = \bigcup_{\forall k \in \text{aff}(i,j)} \left[ \left( \bigcup_{E_k(R_j)} (PCB_k \setminus UCB_k) \right) \bigcup \left( nPCB_k \cup (PCB_k \cap UCB_k) \right) \right]$$

where $(PCB_k \cup UCB_k)$ is the set of PCBs of $\tau_k$ that are also UCBs, and $(nPCB_k \cup (PCB_k \cap UCB_k))$ is therefore the set of ECBs that may be loaded more than once by each job of $\tau_k$. All the other ECBs (those that are not in $(nPCB_k \cup (PCB_k \cap UCB_k))$ and are thus in $(PCB_k \setminus UCB_k)$ are loaded at most once per job of $\tau_k$ and are therefore accounted separately in the first term of Equation (23).

C. WCRT Analysis

Using the exact same argumentation as in Section V-B, the worst-case response time of task $\tau_j$ can be upper bounded by the smallest positive value $R_{j,mul}$ such that:

$$R_{j,mul} = C_j + \sum_{\forall j \in \text{hep}(i)} \min \left\{ \left\lfloor \frac{R_{j,mul}}{T_j} \right\rfloor C_j : \left\lfloor \frac{R_{j,mul}}{T_j} \right\rfloor P_j + MD_j(R_{j,mul}) + \rho_{j,i}(R_{j,mul}) \right\} + \sum_{\forall j \in \text{hep}(i)} \gamma_{j,i}(R_{j,mul})$$ (24)

It is important to note that, by construction, the WCRT formulation of Eq. (24) using the improved variant of the multi-set approach dominates the WCRT given by standard multi-set approach (Eq. (15)) which in turn dominates the simple union approach presented in Section V-A.

VII. STATIC ANALYSIS

Having presented our proposed cache-persistence-aware WCRT analysis, we proceed by explaining how the required input quantities, defined in Section IV-B, are obtained using standard static analysis techniques integrated in WCET estimation tools.

Static cache analysis techniques use abstract interpretation to determine the worst-case behavior with respect to caches for each memory reference. The outcome of such techniques is a classification of references (e.g., always-hit when the reference will always result in a cache hit, always-miss when the reference will always result in a cache miss, first-miss when all successive occurrences of a reference but the first one will result in hits). The classification of each reference allows to determine if a reference will never require a memory access (always-hit) or may require an access to memory. For the purpose of this paper, the method presented in [18] is used.

Most WCET estimation tools use IPET (Implicit Path Enumeration Technique) for WCET calculation. IPET is based on an Integer Linear Programming (ILP) formulation of the WCET calculation problem [19]. This formulation reflects the program structure and the possible execution flows using a set of linear constraints. The WCET estimate for a task is obtained by maximizing the following objective function:
\[ \sum_{b \in \text{BasicBlocks}} E_b \times f_b \] (25)

\( E_b \) (constant in the ILP problem) is the timing information of basic block \( b \). \( f_b \) (variables in the ILP system, to be instantiated by the ILP solver) correspond to the number of times basic block \( b \) is executed.

For a task \( \tau_i \), quantities \( P_i \) and \( MD_i \) are calculated using IPET by setting constant \( E_b \) accordingly for all basic blocks of \( \tau_i \). For the computation of \( P_i \) only the execution time of instructions is included in \( E_b \), ignoring memory accesses. Conversely, when computing \( MD_i \), only memory accesses (as detected by static cache analysis) are included in \( E_b \) and the execution times of instructions are ignored.

For the particular case of direct-mapped caches, determining \( PCB_i \) and \( ECB_i \) is straightforward. A memory block of task \( \tau_i \) belongs to \( PCB_i \) if it is the only one to map to a given cache block. \( ECB_i \) is simply the set of memory blocks of task \( \tau_i \). Determining \( UCB_i \) is achieved using the method presented in [4]. Finally, determining \( MD_i \) is very similar to \( MD_i \). IPET is applied with an execution cost of 0 and considering memory accesses, but in contrast to the computation of \( MD_i \), only memory accesses for cache blocks in \( nPCB_i \) are considered.

VIII. Experiments

In this section, we evaluate the effectiveness of our proposed approaches in comparison to state-of-the-art techniques. We conducted different experiments by varying the task utilizations, number of tasks and the size of cache.

The different inputs previously defined in Section VII were computed using the Heptane\(^2\) static WCET estimation tool. Heptane produces upper bounds on the execution times of hard-real-time applications. It computes WCETs using static analysis at the binary code level. In this paper, all experiments were conducted on C-code compiled with gcc 4.1 with no optimization for MIPS R2000/R3000. The default linker memory layout is used, i.e. functions are represented sequentially in memory, and unless explicitly stated, no alignment directive is used. Without loss of generality, all instructions are assumed to execute in \( I \) cycle (cache access included). Each memory access, regardless of its source, results in a penalty of \( d_{mem} = 100 \) cycles. By default a direct-mapped instruction cache of size \( 2 \) KB with a line size of 32 B is considered.

We have integrated the results obtained from Heptane using static analysis with the MRTA framework developed by Altmejer et al. [15] for multi-core response time analysis. The MRTA tool provides a compositional framework for timing verification in multi-core systems by explicitly modeling the interferences of the different components. We modified the MRTA tool to account for the new task parameters introduced in this paper. We have added a module in the MRTA framework that enables the calculation of the total CPRO \( \rho_{j,i}(R_i) \) using the multi-set approaches detailed in Section VI-B. Also, as we only consider a single-core system, the preemption overhead calculation and the WCRT analysis are altered accordingly. All the experiments were performed using the Mälardalen benchmark suite [20].

All the experiments are performed by randomly generating a large number of task sets and determining the schedulability of those tasksets using Equations (2) (denoted by ECB-Union in the plots), (3) (denoted by UCB-Union Multiset) and (24) (denoted by CPRO). Each task within the task set is randomly assigned parameters from the Mälardalen benchmarks. A subset of them is shown in Table I. Note that due to space limitations, it is not possible to show the details of all the benchmarks in Table I.

Also it should be clear from the numbers in Table I that the benchmark suite comprises tasks with both small and big memory footprint (that fill the entire cache), consequently removing any bias in the results.

With the exception of parameters defined in Table I, we used the following other parameters in our experiments:

- The default number of tasks in each task is 10.
- Task utilizations were generated using UUUnifast [21].
- Each task was randomly assigned one benchmark from the Mälardalen benchmark suite [20] with values of \( C_i \), \( P_i \), \( MD_i \), \( MD_i^e \) along with sets of \( UCB_i \), \( ECB_i \), \( PCB_i \) and \( nPCB_i \) obtained from the values given in Table I.
- Task periods are set according the WCET assigned to each task from the benchmarks and the randomly generated utilization, i.e., \( T_i = C_i/U_i \).
- Task deadlines are implicit with priorities assigned in deadline monotonic order.

1) Total Utilization: To evaluate how our proposed CPRO based WCRT analysis (i.e. Eq. (24)) performs in terms of schedulability in comparison to the ECB-union [8] and UCB-union multi-set approaches [9], we generated 100 task set at each utilization with task set utilizations varied from 0.1 to 1 in steps of 0.05. Each task set comprised 10 tasks, with benchmark parameters generated for a 2kB cache with 64 cache sets. The WCRT analysis is performed for all three approaches using the same task sets. A task set is deemed unschedulable if the calculated WCRT for any task within the task set is greater than its deadline.

Figure 4a shows an average number of task sets that were schedulable using the three analyzed approaches. It is important to note that we only show a cropped version of the plot starting from a utilization of 0.5 mainly because for task set utilizations less than 0.5 all approaches produced identical results. The ECB-union approach of Altmejer et al. [8] performs the worst. This is mainly due to the fact

\begin{table}[h]
\centering
\caption{Task parameters for a selection of benchmarks from the Mälardalen Benchmark Suite [20].}
\begin{tabular}{|l|l|l|l|l|l|l|l|l|}
\hline
Name & \( C_i \) & \( P_i \) & \( MD_i \) & \( MD_i^e \) & \( UCB_i \) & \( ECB_i \) & \( PCB_i \) & \( nPCB_i \) \\
\hline
icenum & 3440 & 984 & 2740 & 92 & 20 & 20 & 20 & 0 \\
ninsert & 7574 & 5974 & 2643 & 352 & 16 & 16 & 10 & 0 \\
bcs & 1399 & 203 & 1223 & 14 & 11 & 11 & 9 & 0 \\
bsdsm100 & 712289 & 710289 & 90893 & 88907 & 20 & 20 & 15 & 0 \\
lidcmp & 45153 & 27036 & 21511 & 11626 & 98 & 90 & 43 & 68 \\
ftct & 17330 & 6550 & 11525 & 11528 & 106 & 22 & 58 & 84 \\
adh & 28427 & 29267 & 10415 & 10417 & 75 & 83 & 41 & 22 \\
maschach & 516409 & 22009 & 294600 & 294400 & 1377 & 0 & 110 & 1377 \\
statecme & 190496 & 10586 & 180110 & 180110 & 275 & 0 & 81 & 275 \\
\hline
\end{tabular}
\end{table}

\(^2\)https://team.inria.fr/alf/software/heptane/
that they only use WCET (effectively the worst-case memory demand) of tasks during the WCRT analysis along with the CRPD cost defined by Equation (1), which is very pessimistic. As a result, a high number of tasks tend to be unschedulable especially at higher utilizations. However, the UCB-union multi-set approach [9] performs better in comparison to the ECB-union approach.

This is clearly due to the fact that the UCB-union multi-set approach also takes into account the actual number of preemptions of each task when calculating the CRPD. Yet, it can be seen from the presented results that our proposed WCRT analysis with CPRO (Eq. (24)) outperforms the other approaches. In fact, we have substantial gains in term of schedulability in comparison to UCB-union multi-set approach, for example at a utilization of 0.85, we gain around 10% of schedulability.

2) Number of Tasks: In preemptive systems, the number of tasks adversely affects the schedulability of the task set. Increasing the number of tasks will lead to more preemptions, resulting in increased memory overhead due to cache evictions. We varied the number of tasks from 5 to 25 increasing 5 tasks in each step. All the parameters other than the number of tasks have the same values as used in the previous section. Figure 4b shows the results of our experiment. We can see that the average schedulability (varying from 0.1 to 1 by step of 0.05) for all approaches decreases when the number of tasks increases. Indeed, this is due to an increasing number of cache evictions and reloads. On the other hand, we also observe that our CPRO-based WCRT analysis performs significantly better in comparison to the other two approaches. The average schedulability for our approach at each point in Figure 4b is up to 10% higher than the UCB-union multi-set and the ECB-union approach. Consequently, this proves the robustness of our approaches against the number of tasks.

3) Cache Size: The cache size is an important factor that can affect the schedulability of tasks. If the cache is large enough to accommodate all the tasks without any cache reuse no additional memory accesses are required. In fact, in this case all the ECBs of a task will be PCBs and will never be evicted from the cache. Another case is when the cache is very small and each task can fill the entire cache during its execution. Consequently, this will result in higher memory demand for each job of the task. To evaluate the impact of cache size on the performance of the analyses, we varied the number of cache sets from 32 to 512, keeping all other task parameters constant as in the case of the schedulability analysis described in Section VIII-1. Figure 4c shows the resulting average schedulability for each approach as a function of the number of cache sets. As the cache line size is kept constant (i.e. 32 B), increasing the number of cache sets effectively increase the cache size. We can see that our proposed CPRO-based WCRT dominate the other two approaches. In fact, by increasing cache size the overall schedulability also increased from 0.76 (with 32 cache sets) to 0.81 (with 512 cache sets) with our approach. This is due to the fact that with a bigger cache the number of PCBs for each task will also increase (hence reducing the residual memory demand). Whereas, for the other two approaches (consistently with [9]) the schedulability decreases due to an increase in the number of ECBs resulting in higher preemption overheads.

IX. CONCLUSION

This paper build upon the observation that a task can re-use cache contents between different jobs. A method is presented to capture these persistent cache blocks (PCBs) resulting in variable memory demand for different jobs from a task. The notion of cache-persistence reload overhead (CPRO) is introduced and different approaches are presented to calculate CPRO. These approaches are orthogonal to the state-of-the-art methods used for CRPD calculation and can be integrated with any of these methods. A WCRT analysis is then presented that exploits this variable memory demand to reduce the preemption cost of higher priority tasks under fixed-priority preemptive scheduling, thereby reducing the WCRT and improving schedulability.

We evaluated the performance of our approach against two prominent approaches from the state-of-the-art in terms of schedulability. Experiments were performed by varying different parameters with most of the values taken from the Mälardalen benchmarks. Experimental results show that our proposed WCRT analysis with CPRO dominates the UCB-union multi-set and the ECB-union approach with an average improvement of 10% in terms of schedulability.

In future work, we aim to extend this approach to multi-level set associative caches. We would like to evaluate our
approach against methods such as cache coloring and cache locking. We also plan to extend our analysis to multicore platforms.

Acknowledgments. This work was partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within project UID/CEC/04234/2013 (CISTER); also by FCT/MEC and the EU ARTEMIS JU within project(s) ARTEMIS/0003/2012 - JU grant nr. 333053 (CONCERTO) and ARTEMIS/0001/2013 - JU grant nr. 621429 (EMC2).

REFERENCES