Cache-aware Schedulability Analysis of PREM Compliant Tasks

Syed Aftab Rashid*§, Muhammad Ali Awan*†, Pedro F. Souto‡*, Konstantinos Bletsas*†, Eduardo Tovar*†
*CISTER Research Centre, Porto, Portugal, †ISEP, Polytechnic Institute of Porto, Portugal,
‡University of Porto, FEUP-Faculty of Engineering, Porto, Portugal, §VORTEX CoLab, Porto, Portugal,

***Abstract*—The Predictable Execution Model (PREM) is useful for mitigating inter-core interference due to shared resources such as the main memory. However, it is cache-agnostic, which makes schedulability analysis pessimistic, via overestimation of prefetches and write-backs. In response, we present cache-aware schedulability analysis for PREM tasks on fixed-task-priority partitioned multicores, that bounds the number of cache prefetches and write-backs. Our approach identifies memory blocks loaded in the execution of a previous scheduling interval of each task, that remain in the cache until its next scheduling interval. Doing so, greatly reduces the estimated prefetches and write backs. In experimental evaluations, our analysis improves the schedulability of PREM tasks by up to 55 percentage points.**

I. Introduction

In critical real-time systems, predictable timing behavior is crucial. PREM ("Predictable Execution Model") [1], [2] and cache-aware schedulability analysis techniques [3] both promote timing predictability (in terms of guarantees derivable offline) but have never been used in conjunction so far.

PREM tasks are sequences of *predictable* or *compatible* scheduling intervals. The former are non-preemptible and consist of separate regions for accessing memory (cache prefetching) and processor computation. This rids worst-case execution time (WCET) analysis of the pessimism resulting from uncertainty about cache state (and hits or misses), since the data is in the cache and cannot be evicted by preempting tasks.

For non-PREM tasks, cache analysis techniques can remove some of the uncertainty (and pessimism) related to cache misses. However, as we will show, even for PREM tasks, cache analysis can still help achieve tighter WCET estimates. Indeed, if analysis can show that some memory blocks need not be prefetched, because they are already in the cache, this comensurately lowers the estimates for prefetches (and resulting write-backs to main memory, of evicted cache blocks). Our present work leverages this principle and provides new cache-aware schedulability analysis for PREM tasks, offering significant schedulability improvement (up to 55 percentage points in our experiments).

This work was partially supported by National Funds through FCT/MCTES (Portuguese Foundation for Science and Technology), within the CISTER Research Unit (UIDP/UIDB/04234/2020); also by the Operational Competitiveness Programme and Internationalization (COMPETE 2020) under the PT2020 Partnership Agreement, through the European Regional Development Fund (ERDF), and by national funds through the FCT, within project PREFECT (POCI-01-0145-FEDER-029119); also by the European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 732505. Project "TEC4Growth - Pervasive Intelligence, Enhancers and Proofs of Concept with Industrial Impact/NORTE-01-0145-FEDER000020" financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement.

Related work: The PREM model, originally proposed for unicore systems [1], has since been extended to multicores [2], [4], [5] and heterogeneous systems [6], [7]. While PREM enables timing-predictable execution on non-predictable hardware, it has some restrictive assumptions on the behavior of the local memories, e.g., task-level partitioning of the outer cache [2], [4], [5], use of software-controlled scratchpad memories [6], [7] and the assumption of an unknown cache state at the start of every scheduling interval of a task [1]. These assumptions limit its applicability and cause overestimation of memory acceses.

Several cache analysis techniques [3], [8]–[10] can bound the main memory access demand of tasks executing on a multicore by accurately estimating the number of cache misses [8] and write-backs [9]. However, such techniques had not yet been applied to PREM tasks – which is what this work does.

II. System Model

Hardware Platform: We assume a multicore platform comprising K identical timing compositional cores [11]. The cores can have multiple cache levels, but we focus on the outer cache – assumed to be evenly partitioned among cores. The cache is direct-mapped[1], unified, i.e., it can store both data and instruction, and employ a write-back policy[2], i.e., it postpones writes to the main memory until the memory block holding the data in cache is evicted by another memory block. Furthermore, in case of a write-miss, we assume a write-allocate write-miss (WAWM) policy, i.e., the memory block being written to is first loaded in the cache and then a write is performed. The cache is assumed to be physically-indexed and physically-tagged and any set-based cache partitioning approach, e.g., provided by the Intel Xeon E5-2600 family [13], can be used to partition it.

PREM-based Execution and Task Model: As in [1], tasks are structured as a sequence of non-preemptive scheduling intervals. PREM has two kinds of scheduling intervals, *predictable* and *compatible*, but we only consider the former. A **predictable** interval has two phases. In the initial *memory phase*, the CPU accesses the main memory to prefetch cache lines and perform write-backs (evictions). Data and instructions needed for the subsequent *execution phase* are loaded during the memory phase, therefore it incurs no last-level cache misses.

Task set τ has n independent sporadic tasks, i.e., $\tau = \{\tau_0, \tau_1, \cdots, \tau_{n-1}\}$. Each task τ_i has a unique priority $\kappa_i \geq 0$

[1]Proposed analyses can be easily extended to consider set-associative Least-recently-used (LRU) caches by building on the work in [12].

[2]Many embedded processors (e.g. Freescale MPC740, Infineon Tricore TC1M, Renesas SH7750 and NEC V44181) support a write-back cache policy.

978-3-9819263-6-1/DATE22/©2022 EDAA

and all its instances, i.e., jobs, have a minimum inter-arrival time T_i and a relative deadline $D_i \leq T_i$.

Let N_i denote the number of scheduling intervals of a task τ_i and $E_i = \{E_{i,0}, E_{i,1}, \cdots, E_{i,N_i-1}\}$ the set of its scheduling intervals themselves. A scheduling interval is modeled by two parameters: the (worst-case) number of memory accesses performed in its memory phase, i.e., $\mu_{i,j}$, and the (worst-case) length of its execution phase, i.e., $C^e_{i,j}$. The memory accesses can be further divided into cache prefetches and write-backs. The former correspond to the data and instructions loaded into the cache for the subsequent execution interval. Recall that the WAWM policy first loads the data into the cache, from the main memory, in order to update its content in-place in the cache. Therefore, a write-miss also generates a read memory access. Under PREM, no memory accesses occur in the execution phase of a predictable interval. It is assumed that the compiler predicts such write memory locations in the static analysis and arranges for their contents be loaded in the memory phase. Hence, we treat them as prefetches. The write-backs write the data into memory upon eviction of a cache block. Note that a prefetch may evict the existing cache line, leading to a write-back memory access. We assume that the worst-case memory access latency for each memory request (prefetch or write-back), is upper bounded by d_{mem}. Then, each predictable scheduling interval $E_{i,j}$ can be modelled as $\{\mu^R_{i,j}, \mu^W_{i,j}, C^e_{i,j}\}$. Scalars $\mu^R_{i,j}$ and $\mu^W_{i,j}$ denote estimates of prefetches and write-backs, respectively. We have $\mu_{i,j} = \mu^R_{i,j} + \mu^W_{i,j}$ and the worst-case execution time of $E_{i,j}$, i.e., $C_{i,j}$, is given by $C_{i,j} = \mu_{i,j} \times d_{mem} + C^e_{i,j}$. Similarly, on the task level, we define total prefetches $\mu^R_i = \sum_{j=0}^{N_i-1} \mu^R_{i,j}$, total write-backs $\mu^W_i = \sum_{j=0}^{N_i-1} \mu^W_{i,j}$, total memory accesses $\mu_i = \mu^R_i + \mu^W_i$ and the worst-case execution time $C_i = \sum_{j=0}^{N_i-1} C_{i,j}$. Lastly, $hp(i)$, $hep(i)$ and $lp(i)$ denote the set of tasks with priorities higher, higher or equal and lower than that of τ_i, respectively.

III. Background and Problem Formulation

The schedulability analysis in [1], [2] does not rely on a cache analysis, therefore, for safety, it treats each memory prefetch in a predictable interval as a cache miss that causes a write-back. This may lead to an overestimation of memory accesses, and consequently, to conservative memory access overhead.

This work focuses on the analysis of the reuse of cache blocks in subsequent predictable scheduling intervals of a given task, so as to tightly bound the total number of memory accesses by it. Consider a memory block m used by two scheduling intervals $E_{i,j}$ and $E_{i,j+1}$. If m is prefetched during the execution of $E_{i,j}$ and is not evicted before the start of $E_{i,j+1}$, then, an access to m during the execution of $E_{i,j+1}$ will not incur a cache miss (and the corresponding write-back). Every block reused between $E_{i,j}$ and $E_{i,j+1}$ therefore eliminates two memory accesses. An example further illustrates the advantages of cache reuse:

Example: Task τ_i has 4 scheduling intervals, $E_{i,0}$ to $E_{i,3}$, that load $\mu^R_{i,0}$=$\mu^R_{i,1}$=3, $\mu^R_{i,2}$=4 and $\mu^R_{i,3}$=3 cache lines. The model in [1], [2] assumes unknown cache state at the start of each interval. Hence, in the worst case, the number of write-backs in each interval is: $\mu^W_{i,0}$=$\mu^W_{i,1}$=3, $\mu^W_{i,2}$=4, $\mu^W_{i,3}$=3; and overall: μ_i=μ^R_i+μ^W_i=(3+3+4+3)+(3+3+4+3)=26 (see Figure 1a).

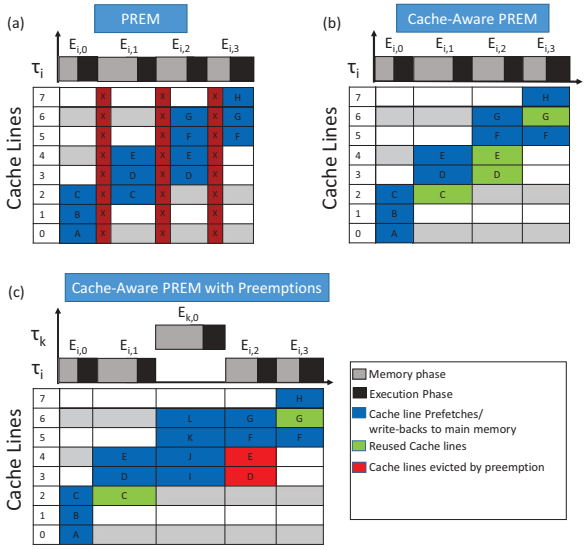


Fig. 1: Different execution scenarios for a PREM task τ_i

However, if some cache blocks are *re-usable* (highlighted in green in Figure 1b), the number of memory accesses may be much smaller than the worst-case computed above. For example, if there is no preemption during the execution of τ_i, $E_{i,1}$ will not need to load memory block C that was previously loaded from the main memory to the cache during the memory phase of $E_{i,0}$ Similarly, $E_{i,2}$ may re-use memory blocks D and E previously loaded during $E_{i,1}$; and $E_{i,3}$ may re-use memory block G, previously loaded during the execution of $E_{i,2}$. Considering the re-use of cache contents in Figure 1b, the total number of memory prefetches for each scheduling interval becomes $\mu^R_{i,0}$=3, $\mu^R_{i,1}$=$\mu^R_{i,2}$=$\mu^R_{i,3}$=2. Even if every such prefetch causes a write-back, with cache re-use, the worst-case number of memory accesses is μ_i=2×(3+2+2+2)=18. This shows that accounting for cache re-use among scheduling intervals of a task can significantly reduce the number of memory accesses compared to the re-use-oblivious worst case. In fact, as later shown in Section IV-B, further improvements may be had, by tightening the upper-bound on the number of write-backs.

A. Definition of Re-usable cache contents

We formally define the re-use of cache contents and explain how to compute the re-usable cache content between different scheduling/predictable intervals of a task τ_i.

A memory block m that is in the cache at the end of scheduling interval $E_{i,j}$ of task τ_i is re-usable, if it is still in the cache at the beginning of scheduling interval $E_{i,k}$, with $k > j$, and is accessed in the latter. This is similar to the notion of *useful cache blocks* (UCBs), defined in literature [14] as:

Useful Cache Block (UCB) [14]: A memory block m used by a task τ_i during its execution is considered a useful cache block w.r.t. a program point $\mathcal{P}$, if (i) m may be cached at $\mathcal{P}$ and (ii) m may be re-used at some program point reachable from $\mathcal{P}$ without being evicted along the corresponding path.

The number of UCBs of a task at a program point $\mathcal{P}$ over-estimates how many memory blocks may be re-used after $\mathcal{P}$.

However, for the safety of the analysis we later propose, we must only account for those ones that are guaranteed to be re-used. Therefore, we introduce a slightly different notion from UCBs, called *definitely reused cache blocks* (DRCBs).

Definitely Reused Cache Block (DRCB): A memory block m used by a task during its execution is a definitely reused cache block w.r.t. a program point $\mathcal{P}$, if (i) m is cached at $\mathcal{P}$ and (ii) in every path starting at $\mathcal{P}$, m is re-used at some program point $\mathcal{P}'$ without the eviction of m along the respective path.

The set of DRCBs of tasks can be computed similarly to UCBs, e.g., using the analysis in [14]. However, one should use the *Must* cache analysis [15] to compute DRCBs, instead of the *May* cache analysis used for UCBs. Moreover, since a PREM-compliant task τ_i can only be preempted at scheduling interval boundaries, the DRCB analysis need only be applied to program points at the end of every scheduling interval in τ_i.

Because a PREM task τ_i cannot be preempted when executing a predictable scheduling interval, and the analysis presented in the next section is done per scheduling interval, we restrict the concept of a DRCB to a scheduling interval:

DRCBs per scheduling interval: A memory block m is a DRCB w.r.t. a scheduling interval $E_{i,j}$ if m is (i) cached at the end point E of scheduling interval $E_{i,j-1}$ and (ii) re-used at some program point F in $E_{i,j}$ that is reachable from E without eviction of m along all possible execution paths from E to F.

At first sight, one might think that every memory block that may be accessed in $E_{i,j}$ that is cached at the end of $E_{i,j-1}$ is a DRCB w.r.t. $E_{i,j}$, but this may not be the case. For example, consider the DRCBs w.r.t. $E_{i,3}$ in Figure 1b, i.e., memory block G. Although, memory block F is also cached at the end of $E_{i,2}$, however, only memory block G is categorized as a DRCB. This can happen because there may be an execution path between $E_{i,2}$ and $E_{i,3}$, where memory block F is not accessed (re-used).

We use $DRCB_{i,j}$ to denote the set of DRCBs w.r.t. $E_{i,j}$. Because we assume a direct-mapped cache, to determine cache conflicts between memory blocks of different tasks, it suffices to track the indexes of the cache lines to which those memory blocks are mapped. Therefore, $DRCB_{i,j}$ only holds the indexes of the cache lines to which $E_{i,j}$'s DRCBs are mapped.

Handling Preemptions: So far, we only considered tasks executing in isolation, i.e., assuming τ_i is the only task executing on the core. However, τ_i can be preempted by any higher-priority task τ_k at the boundary of its scheduling intervals. The higher-priority task can then evict DRCBs of τ_i, leading to additional memory accesses during τ_i's execution. Figure 1c, illustrates such a scenario, where DRCBs of τ_i in cache lines 3 and 4 (memory blocks D and E) are evicted due to preempting task τ_k, which loads its memory blocks I and J in cache lines 3 and 4. The evicted DRCBs of τ_i, in the absence of preemptions, would be reused in $E_{i,2}$. Thus, when τ_i resumes it has to prefetch them again from the main memory. In the literature, the impact of preempting tasks on the cache content of a preempted task is bounded using the notion of *evicting cache blocks* (ECBs):

Evicting Cache Blocks (ECBs) [16]: All memory blocks accessed during the execution of a task.

Indeed, every memory block accessed by a preempting task τ_k can cause an eviction of a block of a preempted task and the evicted block may have to be written back. Furthermore, the preempted task may have to fetch the evicted memory block again upon resumption. Because our analysis in the next section is done per scheduling interval, we define the notion of ECBs w.r.t. each scheduling interval of a task τ_i. I.e., for any scheduling interval $E_{i,j}$ the set of cache lines holding the ECBs in $E_{i,j}$ is given by $ECB_{i,j}$. Furthermore, we define $ECB_i = \bigcup_{j=0}^{N_i-1} ECB_{i,j}$ and $\mu_{i,j} \leq 2 \times |ECB_{i,j}|$.

IV. Cache-Aware Schedulability Analyses of PREM Tasks

We present two cache-aware schedulability analyses of PREM tasks. The *DRCB-only* approach only considers the DRCBs of tasks, i.e., it only accounts for the cache re-use between scheduling intervals. The *FDCB-DRCB* approach (Section IV-B) improves on that by carefully analyzing cache write-backs.

A. DRCB-only Approach

The example in Section III, shows that subsequent scheduling intervals of the same task τ_i may reuse cache lines, i.e., DRCBs, loaded during a previous scheduling interval of τ_i. This re-use of cache lines results in reducing the number of main memory accesses of τ_i. However, we also know that under PREM, task τ_i can be preempted at the boundary of its scheduling intervals, e.g., between intervals $E_{i,j-1}$ and $E_{i,j}$, by higher priority tasks. Such preemptions can evict DRCBs of τ_i, that will impact the memory access demand of its scheduling intervals. Therefore, to bound the memory access demand of a scheduling interval $E_{i,j}$ of τ_i, we must first bound the number of DRCBs of τ_i that can be evicted due to preemptions. Assume that, after completing its interval $E_{i,j-1}$, task τ_i is preempted by a higher-priority task. Upon resumption of τ_i, the memory phase of $E_{i,j}$ will execute and τ_i will load from main memory all cache lines required during $E_{i,j}$. In the worst-case, the preemption by higher priority tasks can evict the following set of DRCBs of $E_{i,j}$:

$$DRCB^E_{i,j} = DRCB_{i,j} \bigcap \{ \bigcup_{\forall k \in hp(i)} ECB_k \} \tag{1}$$

The computation of $DRCB^E_{i,j}$ (i.e., DRCBs of $E_{i,j}$ evicted due to preemptions) in (1) considers all higher-priority tasks. On resumption of τ_i, the scheduling interval $E_{i,j}$ has to prefetch the following set of cache lines.

$$ECB^P_{i,j} = \{ ECB_{i,j} \setminus DRCB_{i,j} \} \bigcup DRCB^E_{i,j} \tag{2}$$

Since the policy is write-back, every line in $ECB^P_{i,j}$ may also incur a cache write-back if it holds a dirty cache line. Therefore, the worst-case memory access demand of scheduling interval $E_{i,j}$ of task τ_i is upper bounded by $2 \times |ECB^P_{i,j}| \times d_{mem}$.

Lemma 1. Under the DRCB-only approach, the worst-case memory access demand of a scheduling interval $E_{i,j}$ of task τ_i is upper bounded by $2 \times |ECB^P_{i,j}| \times d_{mem}$.

Proof. By definition of $ECB_{i,j}$ and $DRCB_{i,j}$, when a task τ_i executes in isolation, in scheduling interval $E_{i,j}$, in the worst case, it needs to load only the cache lines in $ECB_{i,j} \setminus DRCB_{i,j}$.

However, when τ_i executes with other tasks in the system, some of the DRCBs in $DRCB_{i,j}$ may also be evicted due to preemptions. In the worst case, all higher priority tasks may preempt τ_i between two scheduling intervals, thus $DRCB^E_{i,j}$ (1) upper bounds the set of cache blocks that may be evicted. Hence, in the worst case, in $E_{i,j}$, τ_i must load the cache lines in $ECB^P_{i,j} = (ECB_{i,j} \setminus DRCB_{i,j}) \cup DRCB^E_{i,j}$.

Furthermore, in the worst-case, at the beginning of $E_{i,j}$, each cache line in $ECB^P_{i,j}$ is dirty and must be written to main-memory, because the cache uses a write-back policy.

Therefore, the worst-case memory access demand of scheduling interval $E_{i,j}$ is upper bounded by $2 \times |ECB^P_{i,j}| \times d_{mem}$. □

Note that (1) assumes that all tasks in $\mathrm{hp}(i)$ will preempt τ_i at the boundary of all its scheduling intervals. This assumption is safe, but it can be refined by considering the actual arrivals of higher priority tasks during the execution of τ_i.

Having bounded the worst-case memory demand of a scheduling interval $E_{i,j}$ of task τ_i using Lemma 1, the WCET of scheduling interval $E_{i,j}$ can be computed using (3) and the total WCET of τ_i (a sum of its scheduling intervals) from (4):

$$C_{i,j} = (2 \times |ECB^P_{i,j}| \times d_{mem}) + C^e_{i,j} \tag{3}$$

$$C_i = \sum_{j=0}^{N_i-1} C_{i,j} \tag{4}$$

In addition to the WCET estimated in isolation, C_i also includes the total memory access latency and overhead of all memory reloads made due to preemptions from higher priority tasks. Having the total WCETs of all tasks, we compute the worst-case response time (WCRT) of each task via recurrence (5) [1], [17]

$$R_i^{k+1} = B_i + C_i + \sum_{\forall h \in hp(i)} \left\lceil \frac{R_i^k}{T_h} \right\rceil C_h \tag{5}$$

$$B_i = \max_{\forall l \in lp(i)} (\max_{j=0}^{N_l-1} C_{l,j}) \tag{6}$$

A task may be blocked due to non-preemptive execution of predictable interval of a lower-priority task. The worst-case blocking term for τ_i (Eq. 6) is the duration of the longest predictable interval of all tasks with lower priority than τ_i.

B. FDCB-DRCB Approach

The DRCB-only approach assumes that a task τ_i in each of its scheduling interval $E_{i,j}$ will perform $|ECB^P_{i,j}|$ write-backs. This assumption is safe but pessimistic because the number of write-backs task τ_i may have to perform depends on the memory writes of τ_i and of other tasks scheduled on the same core as τ_i. The FDCB-DRCB approach removes some pessimism from the DRCB-only approach. It relies on the concept of Final Dirty Cache Block (FDCB) from the literature [9]:

Final Dirty Cache Blocks [9] at Task Level: Memory blocks used by a task τ_i that may be written to during the execution of τ_i, and may still be cached after the completion of τ_i.

Note that since each scheduling interval $E_{i,j}$ of task τ_i is executed in a non-preemptive manner, we only need to consider cache blocks that are dirty at the completion of $E_{i,j}$. Therefore, we define FDCB at scheduling interval level.

Final Dirty Cache Blocks at Scheduling Interval Level: Memory block used by task τ_i that may be written to during the execution of scheduling interval $E_{i,j}$, and may still be available in cache after the completion of $E_{i,j}$.

Let $FDCB_{i,j}$ denote the set of cache lines where the FDCBs of scheduling interval $E_{i,j}$ are mapped and let $FDCB_i = \bigcup_{j=0}^{N_i-1} FDCB_{i,j}$. Then, in the worst case, any element of this set may still hold, at completion of τ_i, a FDCB of τ_i.

If a scheduling interval loads a memory block into a cache line currently holding a FDCB (from previously completed scheduling intervals of the same or other tasks), a write-back is required. We determine a tighter upper bound on the number of write-backs that each job of a task may have to make.

Let task τ_i be the task under analysis. The set of write-backs that τ_i may need to perform $WB^{tot}_{i,j}$, can be divided into two

1) WB^{lp}_i, i.e., the set of cache lines holding FDCBs of lower priority tasks that need to be written-back by task τ_i.
2) WB^{hep}_i, i.e., the set of cache lines holding FDCBs of either higher priority tasks or τ_i itself that need to be written-back by task τ_i.

First, we derive a safe upper bound on WB^{lp}_i. In the PREM model, no lower priority task is allowed to run after the arrival of τ_i, except the task running at the time τ_i arrived, if any. In that case, that task can run until the end of scheduling interval being executed. Independently of whether there is such a task, when τ_i starts executing, in the worst case, the set of FDCBs of lower priority tasks that may be in the cache is given by $\bigcup_{\forall l \in lp(i)} FDCB_l$. Thus, in the worst case τ_i will have to write-back memory blocks in the following set:

$$WB^{lp}_i = \left(\bigcup_{\forall l \in \mathrm{lp}(i)} FDCB_l \right) \bigcap ECB_i \tag{7}$$

We can further divide write-backs among τ_i's scheduling intervals. The set of cache lines that needs to be written-back in scheduling interval $E_{i,j}$ is given by:

$$WB^{lp}_{i,j} = \left(\bigcup_{\forall l \in \mathrm{lp}(i)} FDCB_l \setminus \bigcup_{\forall k < j} ECB_{i,k} \right) \bigcap ECB_{i,j} \tag{8}$$

The set subtraction in (8) ensures that an FDCB of a lower priority task that is accounted as written-back in a scheduling interval preceding $E_{i,j}$ is not accounted again in $WB^{lp}_{i,j}$.

We now derive a safe estimate of $WB^{hep}_{i,j}$, i.e. of the cache lines with FDCB left by either τ_i or higher priority tasks that may be written-back in scheduling interval $E_{i,j}$. Remember that $DRCB^E_{i,j} \subseteq DRCB_{i,j}$ (computed by (1)) gives the set of $E_{i,j}$'s DRCBs that maybe evicted by preempting higher priority tasks. In the worst-case, τ_i must write back each cache line in $DRCB^E_{i,j}$ in $E_{i,j}$, whereas the cache lines in $DRCB_{i,j} \setminus DRCB^E_{i,j}$ cause no write backs in $E_{i,j}$.

We now analyse the remaining cache lines that may be accessed in $ECB_{i,j}$, i.e. the cache lines in:

$$ECB^R_{i,j} = (ECB_{i,j} \setminus DRCB_{i,j}) \setminus WB^{lp}_{i,j} \tag{9}$$

Some cache lines in $ECB^R_{i,j}$ may have FDCBs of either τ_i or of higher priority tasks. Access to each of those lines may cause a write back. Thus, a safe estimate of $WB^{hep}_{i,j}$ is:

$$WB^{hep}_{i,j} = \left(\left(\bigcup_{\forall h \in hep(i)} FDCB_h \right) \bigcap ECB^R_{i,j} \right) \bigcup DRCB^E_{i,j} \tag{10}$$

Finally, by definition of $WB^{tot}_{i,j}$ we have:

$$WB^{tot}_{i,j} = WB^{lp}_{i,j} \bigcup WB^{hep}_{i,j} \tag{11}$$

Consequently, the worst-case memory access demand of a scheduling interval $E_{i,j}$ of task τ_i is upper bounded by $(|WB^{tot}_{i,j}| + |ECB^P_{i,j}|) \times d_{mem}$.

Lemma 2. Under the FDCB-DRCB approach, the worst-case memory access demand of a scheduling interval $E_{i,j}$ of task τ_i is upper bounded by $(|WB^{tot}_{i,j}| + |ECB^P_{i,j}|) \times d_{mem}$.

Proof. By Lemma 1, in the worst case, in $E_{i,j}$, τ_i must load the cache lines in $ECB^P_{i,j}$. Similarly, by construction $WB^{tot}_{i,j}$ contains all lines that may require write-backs during $E_{i,j}$.

Because the worst-case main memory access time is d_{mem}, $(|WB^{tot}_{i,j}| + |ECB^P_{i,j}|) \times d_{mem}$ upper bounds the worst-case memory access demand of scheduling interval $E_{i,j}$. □

Finally, the WCET of scheduling interval $E_{i,j}$ is given by:

$$C_{i,j} = (|WB^{tot}_{i,j}| + |ECB^P_{i,j}|) \times d_{mem} + C^e_{i,j} \tag{12}$$

The WCRT of task τ_i is then computed using Eq. (4)-(6).

V. Experimental Evaluation

Our experiments compare the performance of our proposed cache-aware schedulability analysis for PREM tasks against the state-of-the-art PREM analysis [1], [2] using synthetic task sets. Our simulator, available on request, models a quad-core with direct-mapped unified 64 KB outer cache (2048 cache sets, 32-byte blocks) evenly partitioned to the cores. The worst-case time for a cache line load from/write-back to the main memory is set to d_{mem}=100 μsec. The default task set size is 32, with 8 tasks per core. Task utilizations (U_i) are generated using UUnifast [18] assuming equally utilized cores. Task inter-arrival times are log-uniform-distributed in a 5 to 500 msec range. The number of scheduling intervals of a task is randomly chosen between 2 to 8, i.e., $N_i = rand(2, 8)$. The scheduling interval utilizations ($U_{i,j}$) are generated using UUnifast [18] for a task utilization U_i and N_i scheduling intervals. Scheduling interval WCETs are computed as $C_{i,j} = U_{i,j} \times T_i$. Memory access demands of scheduling intervals are set to $\mu_{i,j} = rand(0.1, 0.6) \times C_{i,j}$[3] The length of computation phase of a scheduling interval $E_{i,j}$ is then given by $C^e_{i,j} = C_{i,j} - \mu_{i,j} \times d_{mem}$. Obviously, $C_i = \sum_{j=0}^{N_i-1} C_{i,j}$.

The number of ECBs of a scheduling interval is generated as $|ECB_{i,j}| = \frac{\mu_{i,j}}{2}$. and the number of its DRCBs and FDCBs as $|DRCB_{i,j}| = rand(0.1, 0.3) \times |ECB_{i,j}|$ and $|FDCB_{i,j}| = rand(0.1, 0.6) \times |ECB_{i,j}|$. We assume that ECBs of tasks are sequentially arranged in cache using priority ordering. The ECB indexes of each task start from the last cache line used by the previous task +1 and may wrap-around the cache. DRCBs and FDCBs are then chosen randomly from ECBs. Task deadlines are implicit. Priorities are deadline-monotonic.

Task WCRTs are computed using (5). The number of cache loads and write-backs is computed considering (i) an unknown cache state at the start of every scheduling interval, i.e., the cache-agnostic PREM [1], (ii) the DRCB-only approach that only considers the improvement in cache misses due to cache re-use between scheduling intervals, and (iii) FDCB-DRCB, that additionally considers the improvement in cache write-backs due to FDCB analysis. Comparisons are either by number of schedulable task sets or by weighted schedulability[4]. A task set is deemed schedulable only if $R_i \leq D_i$ for all its tasks.

The first experiment varied the **core utilizations** from 0.05 to 1 in steps of 0.025, generating 1000 random task sets at each point. Figure 2a shows the number of task sets deemed schedulable by each approach. FDCB-DRCB performs best because it tightly bounds both cache line loads and write-backs. The DRCB-only analysis only slightly outperforms the cache-agnostic state-of-the-art, because it only considers cache re-use between intervals and overestimates the write-backs.

Next, we varied the **number of cores**, using default values for other parameters (Figure 2b). With more cores, fewer task sets are deemed schedulable, for all approaches. This is because the number of tasks increases comensurately, and the cache size per core decreases. This causes more cache conflicts between tasks, and, in turn, more cache line loads/write-backs. The cache-aware approaches again outperform the state-of-the-art.

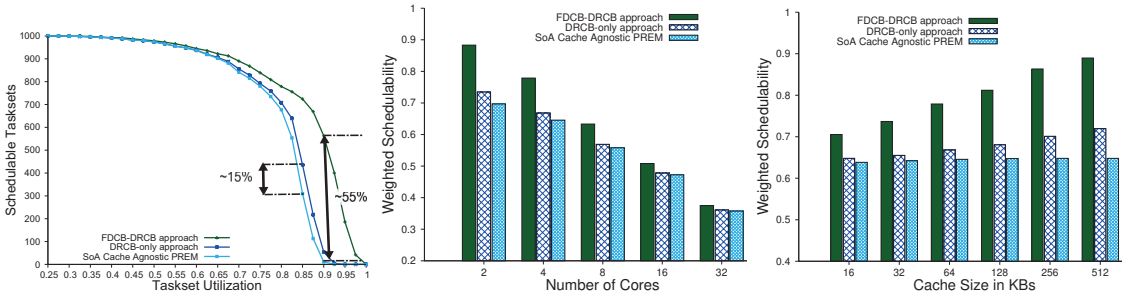
In Figure 2c, we varied the last-level **cache size** from 16 to 512 KB, using the defaults for other parameters. Small caches smother performance differences between the approaches. With larger caches, cache-awareness pays off, as the the cache re-use between scheduling intervals increases (i.e., DRCBs) and fewer cache conflicts (i.e., write-backs) occur between tasks.

To evaluate the impact of cache re-use on performance, we varied the **DRCB-ECB ratio** of tasks (i.e., fraction of a task's ECB that are also its DRCBs) from 10% to 80%, using the defaults for other parameters (Figure 2d). For a lower DRCB-ECB ratio, the performance difference between our proposed approaches and the state-of-the-art is small but it increases with higher ratios. Since the state-of-the-art is cache-agnostic, its performance is unaffected by this parameter. We similarly varied the **FDCB-ECB ratio** (Figure 2e). As only the FDCB-DRCB approach accounts for FDCBs, only its performance is impacted by higher FDCB-ECB ratios. Intuitively, as the number of FDCBs increases, the potential cache write-backs also increase, negatively impacting schedulability.

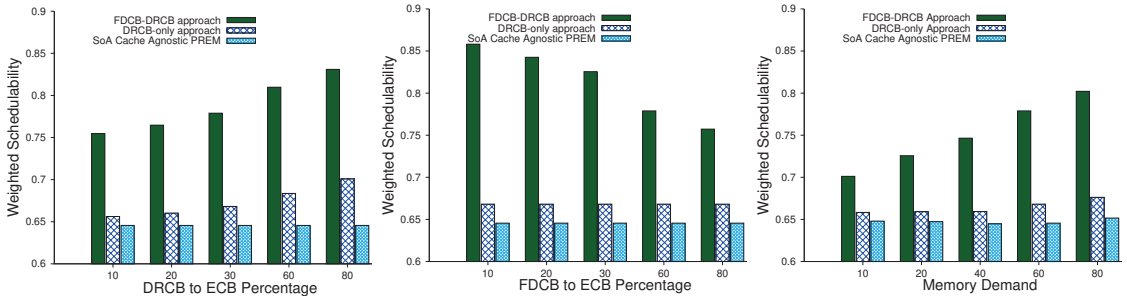
Finally, we varied the **relative memory demand**, i.e., the ratio of the memory access demand of a task to its WCET (C_i), from 10% to 80%, uniformly in its scheduling intervals (Figure 2f). When the memory phases are relatively small, the

[3]This is in-line with the memory access demand to WCET ratio of most benchmarks from the Mälardalen benchmark suite [19], [20].

[4]This metric [21] condenses three-dimensional plots to two-dimensional ones by eliminating the axis of task set utilization. Let $S_y(\tau, p)$ be the result of a schedulability test y for task set τ whose utilization is $U(\tau)$ with an input parameter p. Then the weighted schedulability is defined as $W_y(p) = \sum_{\forall \tau}(U(\tau) \times S_y(\tau, p)) / \sum_{\forall \tau} U(\tau)$.



(a) Varying the total Task set Utilization (b) Varying Number of Cores (c) Varying Cache Size



(d) Varying the DRCB-ECB percentage (e) Varying the FDCB-ECB percentage (f) Varying Memory Demand

Fig. 2: Weighted Schedulability ratio of proposed approaches by varying no. of cores, cache size, DRCBs, FDCBs and $\mu_{i,j}$.

difference between cache-aware approaches and the state-of-the-art is also small. Smaller values of $\mu_{i,j}$ also imply smaller ECB, DRCB and FDCB sets. For higher $\mu_{i,j}$ values, the sizes of those sets also increase, and the performance of FDCB-DRCB significantly improves. The trend is less pronounced for the DRCB-only approach, because, with increasing memory demand, the number of tasks' ECBs also increases alongside their DRCBs. Therefore, the conflicts between tasks' ECBs and DRCBs also increases. Naturally, the performance of the cache-agnostic state-of-the-art does not vary with this parameter.

VI. Conclusions

We ported established cache analysis techniques to the schedulability analysis of PREM tasks, and proposed two approaches that tightly estimate cache line loads and write-backs. Our DRCB-only approach exploits cache reuse among scheduling intervals to reduce the number of loads. Our FDCB-DRCB approach improves on that by carefully analyzing cache write-backs. Experiments performed by varying different parameters show that our approaches can significantly improve the schedulability success ratio by up to 55 percentage points.

References

[1] R. Pellizzoni *et al.*, "A predictable execution model for COTS-based embedded systems," in *RTAS*, 2011, pp. 269–279.

[2] G. Yao *et al.*, "Memory-centric scheduling for multicore hard real-time systems," *Real-Time Systems*, vol. 48, no. 6, pp. 681–715, 2012.

[3] M. Lv *et al.*, "A survey on static cache analysis for real-time systems," *Leibniz Transactions on Embedded Systems*, vol. 3, no. 1, pp. 05–1, 2016.

[4] S. Bak *et al.*, "Memory-aware scheduling of multicore task sets for real-time systems," in *Proc. 18th RTCSA*, 2012, pp. 300–309.

[5] G. Yao *et al.*, "Global real-time memory-centric scheduling for multicore systems," *IEEE Trans. on Computers*, vol. 65, no. 9, pp. 2739–2751, 2015.

[6] B. Forsberg *et al.*, "Gpuguard: Towards supporting a predictable execution model for heterogeneous soc," in *Proc. DATE*. IEEE, 2017, pp. 318–321.

[7] ——, "Heprem: Enabling predictable gpu execution on heterogeneous soc," in *Proc. DATE*. IEEE, 2018, pp. 539–544.

[8] S. A. Rashid *et al.*, "Cache persistence-aware memory bus contention analysis for multicore systems," in *2020 DATE*. IEEE, 2020, pp. 442–447.

[9] R. I. Davis *et al.*, "Response-time analysis for fixed-priority systems with a write-back cache," *Real-Time Systems*, vol. 54, no. 4, pp. 912–963, 2018.

[10] C. Maiza *et al.*, "A survey of timing verification techniques for multi-core real-time systems," *ACM Comput. Surv.*, vol. 52, no. 3, pp. 1–38, 2019.

[11] S. Hahn *et al.*, "Towards compositionality in execution time analysis: definition and challenges," *ACM SIGBED Review*, vol. 12, no. 1, pp. 28–36, 2015.

[12] S. Altmeyer *et al.*, "Resilience analysis: tightening the crpd bound for set-associative caches," *ACM Sigplan Not.*, vol. 45(4), pp. 153–162, 2010.

[13] A. Herdrich *et al.*, "Cache QoS: From concept to reality in the Intel Xeon processor E5-2600 v3 product family," in *IEEE HPCA*, 2016, pp. 657–668.

[14] C. G. Lee *et al.*, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *IEEE Trans. Comput.*, vol. 47, no. 6, pp. 700–713, 1998.

[15] C. Ferdinand *et al.*, "Efficient and precise cache behavior prediction for real-time systems," *Real-Time Systems*, vol. 17(2-3), pp. 131–181, 1999.

[16] H. Tomiyama *et al.*, "Program path analysis to bound cache-related preemption delay in preemptive real-time systems," in *CODES*, 2000, pp. 67–71.

[17] L. George *et al.*, "Preemptive and non-preemptive real-time uniprocessor scheduling," Ph.D. dissertation, Inria, 1996.

[18] E. Bini *et al.*, "Measuring the performance of schedulability tests," *Journal of Real–Time Systems*, vol. 30, no. 1-2, pp. 129–154, May 2005.

[19] S. A. Rashid *et al.*, "Integrated analysis of cache related preemption delays and cache persistence reload overheads," in *RTSS*, 2017, pp. 188–198.

[20] J. Gustafsson *et al.*, "The Mälardalen WCET benchmarks: Past, present and future," in *OASIcs*, vol. 15, 2010.

[21] A. Bastoni *et al.*, "Cache-related preemption and migration delays: Empirical approximation and impact on schedulability," OSPERT, 2010.