A

Cache Related Preemption Delay Analysis for Multi-level Non-inclusive Caches

SUDIPTA CHATTOPADHYAY, Linköping University
ABHIK ROYCHOUDHURY, National University of Singapore

With the rapid growth of complex hardware features, timing analysis has become an increasingly difficult problem. The key to solving this problem lies in the precise and scalable modeling of performance enhancing processor features (*e.g.* cache). Moreover, real-time systems are often multi-tasking and use preemptive scheduling, with fixed or dynamic priority assignment. For such systems, *cache related preemption delay* (CRPD) may increase the execution time of a task. Therefore, CRPD may affect the overall schedulability analysis. Existing works propose to bound the value of CRPD in a single-level cache. In this paper, we propose a CRPD analysis framework which can be used for a two-level, non-inclusive cache hierarchy. Besides, our proposed framework is also applicable in the presence of shared caches. We first show that CRPD analysis faces several new challenges in the presence of a multi-level, non-inclusive cache hierarchy. Our proposed framework overcomes all such challenges and we can formally prove the *correctness* of our framework. We have performed experiments with several subject programs, including an *unmanned aerial vehicle* (UAV) controller and an in-situ space debris monitoring instrument. Our experimental results suggest that we can provide *sound* and *precise* CRPD estimates using our framework.

1. INTRODUCTION

Hard real-time systems require absolute guarantees on their execution time to meet certain deadlines. Consequently, a significant research has been done in the schedulability analysis of real-time systems. Schedulability analysis requires worst case execution time (WCET) of each task as input. Additionally, for preemptive scheduling, task interferences need to be modeled. This is due to the preemption of a low priority task by a higher priority task. However, performance enhancing micro-architectural features in modern processors make such modeling a very challenging task.

Caches have a key role to play for enhancing performance of any running application on the underlying hardware platform. However, employing caches introduces additional complications to analyze the effect of *intra-task* and *inter-task* interferences on caches. Literature on static cache analysis handles the problem of *intra-task* interferences on caches. *Inter-task* interferences on caches are created due to preemption. Suppose a low priority task t is preempted by a higher priority task t'. The set of cache blocks used by t' is $\mathcal{C}_{t'}$ and the set of cache blocks used by t before the preemption took place is denoted by $\mathcal{C}_t$. If $\mathcal{C}_t \cap \mathcal{C}_{t'} \neq \phi$, t' may replace some of the cache blocks used by t and therefore, t may suffer additional cache misses when it resumes. This variety of inter-task interference on cache performance is called *cache related preemption delay* (CRPD).

Research on statically estimating CRPD has been done in the past few years [Lee et al. 1998; Negi et al. 2003; Tomiyama and Dutt 2000; Tan and Mooney 2004; Altmeyer and Burguiere 2009; Altmeyer et al. 2010]. All prior works on CRPD consider a single-level instruction or data cache. However, with the advent of complex hardware in real-time, embedded systems (*e.g.*, cache hierarchies), many processors (*e.g.*, ARM) employ a bigger second-level (L2) cache for improving performance. In this paper, we propose a CRPD analysis framework which can be applied to a two-level, non-inclusive cache hierarchy.

The key to estimate CRPD is based on the notion of *useful cache blocks* (UCB). UCBs denote cache blocks which might be used by a preempted task after the preemption. Therefore, the number of UCBs poses an upper bound on CRPD. Estimation can further be tightened by analyzing *evicting cache blocks* (ECB) in preempting tasks. ECBs denote cache blocks which might be used by a preempting task. In the presence of non-inclusive cache hierarchies, some memory blocks in a preempted task might be accessed from L2 cache *only* in preempted executions. Therefore, preemptions may increase the amount of *intra-task cache interferences* on L2 cache. Due to such variation in the *intra-task* L2 cache interferences after preemptions, CRPD computation might be affected in the presence of multi-level caches. As a consequence, a *sound* estimate of CRPD cannot be obtained solely based on the concept of UCBs and ECBs.

We show that in the presence of cache hierarchies, an instruction may suffer *one or more L2 cache misses due to a preemption*. We have given a theoretical bound on the number of such L2 cache misses. Such a theoretical bound depends on the organization of a cache hierarchy (in terms of the *number of cache sets* and *cache associativities*). We propose a CRPD analysis framework which uses these bounds and estimates the CRPD.

We have implemented our CRPD analysis framework using Chronos [Li et al. 2007] - a freely available, open-source, WCET analysis tool. To validate our analysis framework and measure the CRPD, we have extended the Simplescalar toolset [Austin et al. 2002]. We have evaluated our framework using several subject programs from [Gustafsson et al. 2010]. Besides, we have used different tasks from an *unmanned aerial vehicle* (UAV) control application [Nemer et al. 2006] and an in-situ, space debris monitoring instrument [Kuitunen et al. 2001]. Experiments show that our framework computes *precise* estimates for most of the subject programs.

2. RELATED WORK

Cache modeling for timing analysis has been an active topic of research for more than a decade. The key to the cache modeling lies in the precise estimation of different sources of cache conflicts. Cache conflict, in general, arises in three different forms: intra-task, inter-task and inter-core. In the following, we shall summarise existing works along the modeling of different cache conflicts.

Cache analysis of a single task

Existing works have primarily considered a single-level cache. Among others, abstract interpretation based cache analysis proposed in [Theiling et al. 2000] deserves mention. For analyzing the cache behaviour of a single task, *must*, *may* and *persistence* analyses have been proposed in [Theiling et al. 2000]. *Must*, *may* and *persistence* analyses categorize memory references as *all-hit* (AH), *all-miss* (AM) and *persistence* (PS), respectively. The memory block corresponding to an AH categorized memory reference is always in the cache when accessed. On the contrary, the memory block corresponding to an AM categorized memory reference is never in the cache when accessed. A memory reference is *persistent* (PS) if it suffers at most one cache miss. *Must* analysis can be used along with virtual inline and virtual unrolling (VIVU) to significantly improve the analysis precision [Theiling et al. 2000]. In VIVU approach, each loop is unrolled once to distinguish *cold cache misses* at first iteration of the loop. If a memory reference cannot be classified as AH, AM or PS, it is considered *unclassified* (NC). The analysis proposed in [Theiling et al. 2000] has later been extended to analyze multi-level, non-inclusive caches [Hardy and Puaut 2008].

Inter-task cache conflict analysis

Inter-task cache conflict analysis is required to find an upper bound on cache misses due to preemption. The bound on cache misses (or additional clock cycles) due to preemption is called cache related preemption delay (CRPD). In last decade, there has been an extensive amount of research to bound the cache related preemption delay (CRPD) [Lee et al. 1998; Negi et al. 2003; Tomiyama and Dutt 2000; Tan and Mooney 2004; Staschulat and Ernst 2004; Altmeyer and Burguiere 2009; Altmeyer et al. 2010]. There are three main approaches to statically bound the value of CRPD:

— Analyzing the preempted task ([Lee et al. 1998; Altmeyer and Burguiere 2009]),
— Analyzing the preempting task ([Tomiyama and Dutt 2000]), and
— Analyzing both the preempted and the preempting task ([Negi et al. 2003; Altmeyer et al. 2010]).

The analysis of the preempted task revolves around the concept of *useful cache block* (UCB) [Lee et al. 1998]. A UCB is a block that may be *cached* before preemption and may be *used* later, resulting in a *cache hit* in the absence of preemption. A *data flow analysis* is applied on the preempted task to statically predict the set of UCBs at each program point. The set of UCBs poses an upper bound on the *additional cache misses* for a single preemption. Recently, [Altmeyer and Burguiere 2009] has improved the state-of-the-art CRPD analysis [Lee et al. 1998] by reducing the number of UCBs to consider for CRPD computation. The key idea of [Altmeyer and Burguiere 2009] is

based on the observation that a CRPD-analysis is always used along with a WCET-analysis. Therefore, the technique proposed by [Altmeyer and Burguiere 2009] considers only those cache misses which were not predicted as cache misses by the WCET analysis. In this fashion, [Altmeyer and Burguiere 2009] may not be able to preserve the over-estimation of CRPD in isolation, however, it can guarantee the over-estimation of the sum of WCET and CRPD.

The analysis of a preempting task is based on the notion of *evicting cache blocks* (ECB). The set of cache blocks used by the preempting task is known as ECB. For example, if a cache set is entirely unused by the preempting task, it cannot evict any cache block used by the preempted task in the respective cache set. Therefore, researchers have proposed to use the set of ECBs for estimating CRPD [Tomiyama and Dutt 2000; Tan and Mooney 2004].

The work in [Negi et al. 2003] has proposed a precise CRPD analysis approach based on the combination of UCBs and ECBs. Therefore, [Negi et al. 2003] analyzes both the preempted task (for computing UCBs) and the preempting task (for computing ECBs). A UCB may lead to an additional cache miss after preemption only if it might be evicted by one or more ECBs. Such a CRPD analysis framework [Negi et al. 2003] is more precise than the CRPD analysis based on analyzing either the preempted or the preempting task in isolation. However, the analysis of [Negi et al. 2003] is based on *direct-mapped* caches. As shown in [Altmeyer et al. 2010], set-associative caches introduce additional complications in accurately estimating the set of UCBs that can be replaced by a set of ECBs. Subsequently, [Altmeyer et al. 2010] has proposed a CRPD analysis framework for set-associative caches with *least-recently-used* (LRU) replacement policy.

[Staschulat and Ernst 2004] shows that the precision of a CRPD-analysis does not only depend on the precision of UCBs and ECBs, but it also depends on the set of preemption points. The technique proposed by [Staschulat and Ernst 2004] is based on the following insight: if two different preemptions at p and p' may lead to a cache miss for the same memory reference in the preempted task, then for the set of preemption points $\{p, p'\}$, it is sufficient to consider only one additional cache miss. This could be possible, only if the analyzer has the knowledge of both the preemption points p and p'. However, if we compute the CRPD for p and p' in isolation, we shall consider duplicate cache misses for the same memory reference in the preempted task. Computing the CRPD for *all possible set of preemption points* will lead to an exponential slow-down. Therefore, [Staschulat and Ernst 2004] proposes efficient algorithms which account multiple preemption points to improve the precision of *state-of-the-art* CRPD analyses.

In summary, there has been an extensive set of works to estimate CRPD based on UCBs and ECBs. Several improvements over the state-of-the-art by combining UCBs and ECBs (*e.g.* [Negi et al. 2003; Altmeyer et al. 2010]) and by maintaining the knowledge of multiple preemptions (*e.g.* [Staschulat and Ernst 2004]) have also been proposed in previous years. However, existing works target only level-one caches. On the contrary, we leverage the concept of both UCB and ECB in the context of cache hierarchies. To the best of our knowledge, *ours is the first CRPD analysis framework that targets a cache hierarchy*.

Inter-core cache conflict analysis

Inter-core cache conflict analysis computes cache conflicts generated in shared caches. Conflicts in shared caches are generated by tasks running on different cores. Till now, only a few solutions have been proposed for analyzing the timing behaviour of shared caches [Li et al. 2009; Hardy et al. 2009; Yan and Zhang 2008]. Inter-core cache conflict analysis may change the categorization of a memory reference from all-hit (AH) to unclassified (NC). Assume a memory reference which accesses a memory block m. Shared cache conflict analysis phase first computes the number of unique and conflicting shared cache accesses from different cores. Then it is checked whether the number of conflicts from different cores can potentially replace m from the shared cache. In case the replacement of m is possible due to the computed set of inter-core conflicts, the categorization of the corresponding memory reference is changed from AH to NC.

Existing works on shared caches focus on estimating the uninterrupted worst case execution time (WCET) of a single task. To the best of our knowledge, there is no CRPD analysis framework in the

presence of shared caches. In this paper, we also show that our proposed CRPD analysis framework can be nicely integrated with the existing works on shared caches. Therefore, our CRPD analysis framework can also be adapted when the second-level cache is shared among multiple cores.

3. OVERVIEW OF OUR ANALYSIS

In this section, we shall first give an overview of our CRPD analysis framework for a two-level cache hierarchy. Subsequently, we shall discuss the *key challenges* in analyzing CRPD in the presence of level-two caches. We show through a few examples that major changes are required in the existing CRPD analysis frameworks based on the concept of UCBs and ECBs. We also show through an example that a *sound* CRPD estimation is *not possible* solely using the concept of UCBs and ECBs.

System Model

In this work, we only model the effect of instruction memory. Modeling the data memory requires the estimation of memory addresses accessed by each load/store instruction. Such an estimation of addresses is known in literature as address analysis [Balakrishnan and Reps 2004]. If the address analysis can accurately estimate the address accessed by each load/store instruction, our framework can be applied for data caches without any change. However, a load/store instruction may access different memory blocks in different executions (*e.g.* array accesses within a loop). A similar situation occurs in the presence of data aliasing, as the aliasing may vary at runtime. Therefore, the address analysis can only predict a range of addresses accessed by each load/store instruction. Such a range is an over-approximation of the set of addresses accessed by a load/store instruction in any execution. As a result, an accurate modeling of data caches, even in the absence of preemption, poses a challenge [Huynh et al. 2011]. The purpose of our work is to discuss challenges in CRPD analysis with non-inclusive cache hierarchies and their solutions. Therefore, overcoming the technical challenges due to data caches is a somewhat orthogonal field to the scope of this work. However, all the technical challenges described in this paper are applicable for any non-inclusive cache hierarchy (*i.e.* instruction, data and unified cache hierarchies). In the presence of data and unified caches, additional challenges to accurately bound the set of address references need to be investigated.

We assume a two-level and *non-inclusive* instruction cache hierarchy (*i.e.* L1 and L2 caches). A cache hierarchy is *non-inclusive* if it is neither *inclusive* (*i.e.* the content of L2 cache is a superset of the content in L1 cache) nor *exclusive* (*i.e.* the content of L1 cache and the content of L2 cache are mutually exclusive). For *inclusive* and *exclusive* cache hierarchies, additional operations are required to maintain the inclusion and exclusion property, respectively. Moreover, in the presence of multiple cores, inclusive cache hierarchies may limit the performance when the size of the largest cache is not significantly larger than the sum of the smaller caches. Therefore, processor architects sometimes resort to non-inclusive cache hierarchies [Zahran et al. 2007]. However, it is worthwhile to mention that inclusive cache hierarchies have their own advantages. The inclusion property greatly simplifies the design of cache coherence protocol in multi-core architectures. CRPD analysis for inclusive cache hierarchies is a subject of our future study.

For any memory reference, the memory subsystem (*i.e.* caches and main memory) is accessed as follows. A memory block is always accessed from L1 cache. If a memory block is not present in both L1 and L2 caches (*i.e.* an L2 miss), it is loaded from main memory to both the cache levels. However, if a memory block is not present in L1 cache but it is present in L2 cache (*i.e.* an L2 hit), the memory block is first loaded into L1 cache. A memory reference does not access L2 cache if the accessed memory block is present in L1 cache (*i.e.* an L1 cache hit). Finally, previous literature on CRPD analysis is primarily based on *least-recently-used* (LRU) cache replacement policy. Designing CRPD analysis frameworks for non-LRU replacement policies is still an open area of research. Therefore, in this work we only concentrate on LRU cache replacement policy.

Overall framework

Our CRPD analysis framework is shown in Figure 1. We first perform L1 and L2 cache analyses on the preempted task using [Theiling et al. 2000] and [Hardy and Puaut 2008], respectively. The

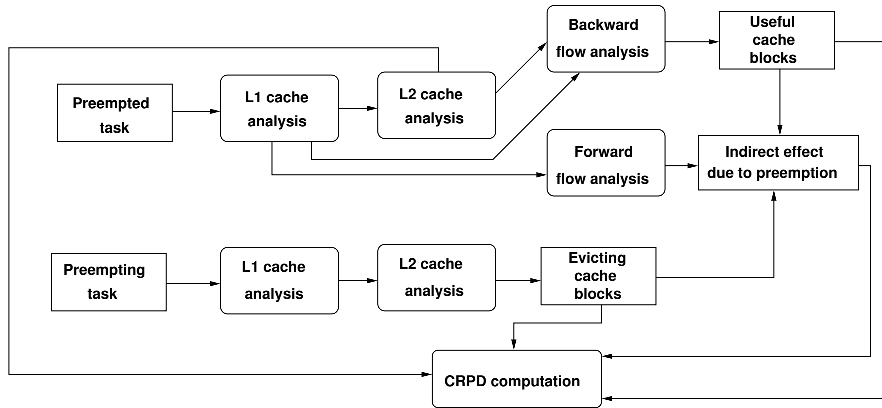

Fig. 1. CRPD analysis framework

cache analysis results are used by a *backward-flow analysis*, which in turn derives the set of useful cache blocks (UCB) in the context of a two-level cache hierarchy. A similar cache analysis on the preempting task derives the set of evicting cache blocks (ECB) in L1 and L2 caches. A separate *forward-flow analysis* is used to estimate the additional *intra-task L2 cache conflicts* generated due to preemptions. We call this additional intra-task L2 cache conflict *indirect effect of preemption* (as shown by the box labeled "indirect effect due to preemption" in Figure 1). Finally, backward and forward-flow analyses results are processed to compute the *cache related preemption delay* (CRPD) suffered by the underlying preempted task. Our CRPD analysis does not account for the cache misses computed by the intra-task L1 and L2 cache analyses (similar to [Altmeyer and Burguiere 2009]). Therefore, the CRPD computed by our framework is *safe* only when considered together with a WCET analysis.

Key challenges

The presence of *non-inclusive caches* makes the CRPD analysis complicated due to the *indirect effect of preemption*. The *indirect effect of preemption* is created when a particular memory reference was an *L1 cache hit* in the absence of preemption, but the same memory reference has to access L2 cache after preemption. This counter-intuitive scenario is explained via Figure 2.

Figure 2(a) captures the control flow graph (CFG) of a preempted task. Symbols inside each basic block capture the memory blocks accessed therein. Figure 2(b) demonstrates the indirect effect of preemption on a memory block m' which was contained *exclusively* in L2 cache before preemption. m' was not evicted by the preempting task. Consider a different memory block m that was contained exclusively in L1 cache before preemption. m was evicted by the preempting task. Consider the memory access sequence $m \rightsquigarrow m'$ after the preempted task resumes execution. m will be reloaded in both L1 and L2 caches. This will evict m' from L2 cache. Therefore, even though m' was not directly evicted by the preempting task, reference to m' will suffer an additional *L2 cache miss* after preemption.

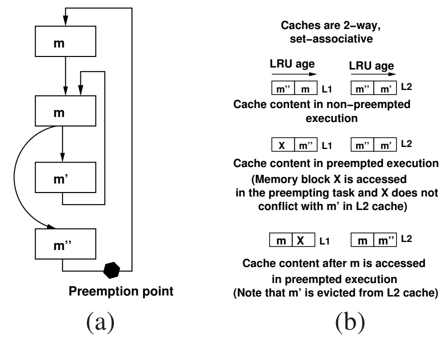

Fig. 2. (a) Control flow graph of the preempted task. Symbols inside each basic block capture the memory blocks accessed therein, (b) L1 and L2 cache set contents in non-preempted and preempted execution

Apart from accounting the cost of *indirect preemption effect* separately, the phenomenon shown in the preceding paragraph creates several other challenges during CRPD analysis. As a result,

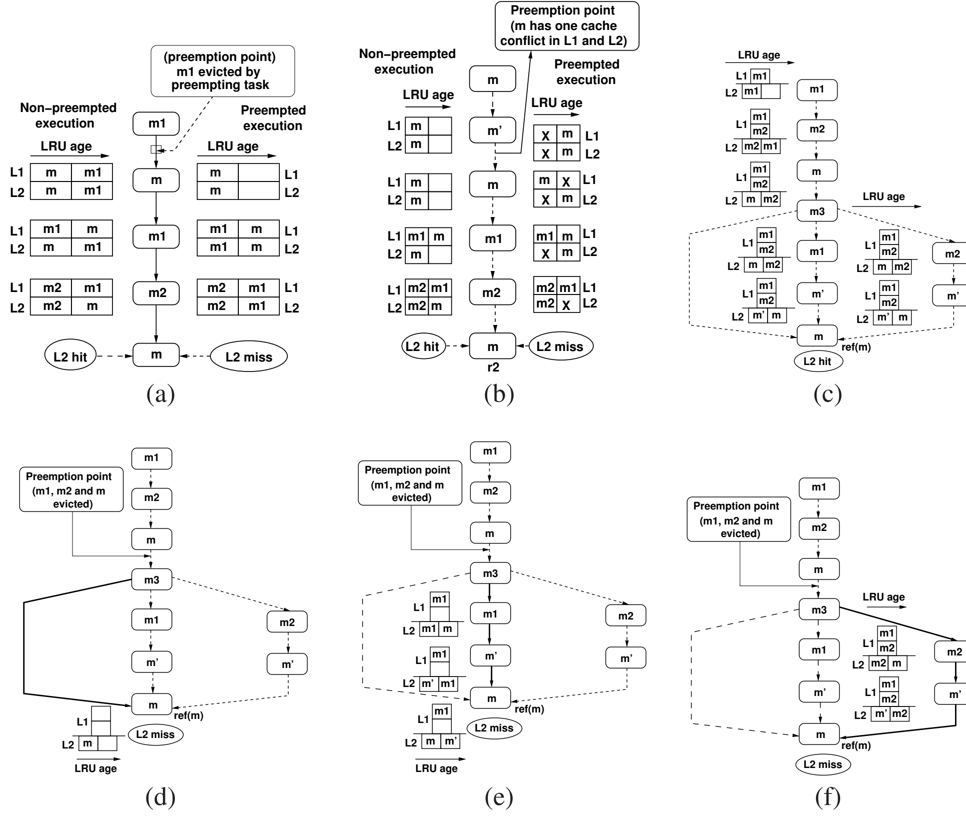

Fig. 3. For all figures, LRU age direction has been indicated. The direction of the arrow labelled "LRU age" points to the *older age* blocks. The relevant cache contents are shown. Each row of the cache content indicates a separate cache set. (a): Due to the *indirect effect* of preemption, preemption cost must go through all the memory references (not just all the memory blocks). The phenomenon is shown for memory block m in two-way, set-associative caches (b): An *L2 cache miss* occurs for the second access (but first access to L2 cache) of m after preemption. Caches are two-way, set-associative. (c)&(d)&(e)&(f): Demonstrating the indirect effect of preemption. L1 cache is direct-mapped and L2 cache is two-way, set-associative. (c): L1 and L2 cache contents in the absence of preemption, assuming $m3$ does not map to any of the cache sets shown in the figure, (d)&(e)&(f): Solid paths are the executed paths (in the order (d)→(e)→(f)) after preemption. L1 and L2 cache contents after preemption are shown when the solid path is executed.

some major changes are required in the CRPD analysis framework. Before going into the details of analysis, let us go through a few examples. The following examples will help in understanding the main difficulties in CRPD analysis in the presence of multi-level caches.

The first difficulty arises in deciding the *granularity* at which the preemption cost needs to be accounted for. In the presence of only L1 cache and LRU cache replacement policy, total preemption cost can be computed *soundly* by accumulating the preemption cost to reload *each L1 cache block*. The *soundness* of this approach can intuitively be explained as follows: once an L1 cache block is reloaded in the cache after preemption, it can only be evicted by the *intra-task cache conflicts*. Since L1 cache is always accessed, in the presence of LRU cache replacement policy, the amount of intra-task cache conflicts does not change due to preemption. Therefore, the CRPD computation in previous literature looks for the *next possible use* of a particular cache block after the preemption point [Lee et al. 1998; Altmeyer and Burguiere 2009]. If the *next use* of a cache block $\mathcal{C}$ is an *L1 cache hit* after preemption (or an *L1 cache miss* in the absence of preemption), no preemption cost is accounted for cache block $\mathcal{C}$.

Due to the indirect effect of preemption, the amount of *intra-task cache conflicts* generated in L2 cache *may increase* after preemption. Therefore the reasoning, as described in the preceding paragraph, may lead to underestimation in the CRPD computation. The situation can be explained via Figure 3(a). Figure 3(a) shows a sequence of memory references in the preempted task, the cache contents before each memory reference in the non-preempted execution (left side of memory references in Figure 3(a)) and the cache contents before each memory reference in the preempted execution (right side of memory references in Figure 3(a)). Assume two-way, set-associative caches and assume $m, m1, m2$ map to the same L1 and L2 cache sets as shown in Figure 3(a). Figure 3(a) demonstrates a sequence of memory references $m1 \rightsquigarrow m \rightsquigarrow m1 \rightsquigarrow m2 \rightsquigarrow m$. In the absence of preemption, L1 and L2 cache contents are shown at the left of each memory reference. The corresponding L1 and L2 cache contents are shown at the right of each memory reference in the preempted execution. Note that the last access to m is an *L2 cache hit* in the absence of preemption, but an *L2 cache miss* after preemption. Consider a CRPD analysis framework which is tailored to an LRU replacement policy based L1 cache. Such a CRPD analysis framework will only look till the first access of m, which is an *L1/L2 cache miss* even in the absence of preemption. Therefore, no preemption cost is added for the L1/L2 cache block corresponding to m. This leads to an underestimation in the CRPD computation as shown by our example.

The example in Figure 3(b) shows the necessity of considering memory references (instead of memory blocks) even in the absence of *indirect effect*. Analogous to Figure 3(a), Figure 3(b) also shows a sequence of memory references in the preempted task, the cache contents before each memory reference in the non-preempted execution (left side of memory references in Figure 3(b)) and the cache contents before each memory reference in the preempted execution (right side of memory references in Figure 3(b)). Assume two-way, set-associative L1 and L2 caches and assume that $m1$ and $m2$ both conflict with m in L1 cache, but only $m2$ conflicts with m in L2 cache. m' does not conflict with any of m, $m1$ or $m2$ in both the cache levels. The example shows that the second access of m after preemption ($r2$) suffers one *L2 cache miss*. This is because L2 cache is not always accessed. Therefore, the *inter-task L2 cache conflict* (denoted by "X" in Figure 3(b)) is only *realized* at $r2$ (*i.e.* when L2 cache was accessed to fetch m).

Our next example discusses the following question: *How many times a particular memory reference* $ref(m)$ *may suffer an L2 cache miss due to the indirect effect of preemption?* If $ref(m)$ is not accessed inside any loop, clearly, $ref(m)$ can suffer *at most one L2 cache miss* after preemption. Therefore, the more interesting scenario occurs when $ref(m)$ is accessed inside some loop.

Figure 3(c) shows a sequence of memory references in the absence of preemption. Assume a direct-mapped L1 cache and a two-way, set-associative L2 cache. Additionally, for the sake of illustration, we shall assume the following:

— m and m' map to the same L1 and L2 cache sets.
— $m1$ and $m2$ map to the same L2 cache set as m but m, $m1$ and $m2$ all map to different L1 cache sets.
— $m3$ is a loop header and has three different paths to $ref(m)$. $m3$ does not conflict with m, m', $m1$ or $m2$ in L1 and L2 caches.

Note that the above mapping is possible when L1 cache has higher number of *sets* than L2 cache. Figures 3(c)-(f) only demonstrate a portion of L1 and L2 caches which is relevant for this discussion. For example, we do not show the mapping of m or m' in L1 cache, as it is irrelevant for our current discussion. Figure 3(c) clearly shows that $ref(m)$ was an *L2 cache hit* and an *L1 cache miss* in the *absence of preemption*.

Figures 3(d)-(f) show executions of three different paths reaching $ref(m)$ after the preemption. The solid line captures the executed path. After executing the path shown in Figure 3(d), m is first loaded in L1 and L2 caches. Since $m1$ was evicted from L1 cache by the preempting task, it is loaded in both L1 and L2 caches as shown in Figure 3(e). This generates an additional L2 cache conflict to memory block m. As a result, $ref(m)$ suffers an L2 cache miss at the end. Since $m2$ was also evicted from L1 cache, $m2$ also generates an additional L2 cache conflict to memory block m,

as shown in Figure 3(f). Consequently, $ref(m)$ suffers a *second L2 cache miss* due to the indirect effect of preemption.

This example shows that $ref(m)$ suffers three L2 cache misses due to preemption: the first L2 cache miss (*i.e.* Figure 3(d)) is *directly* due to preemption, as m was evicted by the preempting task. However, the last two L2 cache misses suffered by $ref(m)$ resulted indirectly via two different memory blocks (*i.e.* $m1$ and $m2$).

It is, however, infeasible to enumerate all different paths to a particular memory reference as shown in Figures 3(d)-(f). Therefore, a reasonable question to ask is whether the number of L2 cache misses due to the indirect effect of preemption is *bounded*. Our work shows that this number is bounded and it depends on the organization of L1 and L2 caches. More precisely, we state the following properties: assume an L1 (L2) cache with number of cache sets S_1 (S_2) and associativity K_1 (K_2). For any memory reference $ref(m)$, assume that $IL2_{ind}$ denotes the number of *additional L2 cache misses* due to the indirect effect of preemption. We can prove the following bounds (for proofs, refer to Section 5):

— If $S_1 > S_2$ holds, then $IL2_{ind} \leq (\frac{S_1}{S_2})K_1 - 1$.
— If $S_1 \leq S_2$ and $K_1 > K_2$ hold, then $IL2_{ind} \leq K_1 - K_2$.
— If $S_1 \leq S_2$ and $K_1 \leq K_2$ hold, then $IL2_{ind} \leq 1$.

Of course, the third cache organization ($S_1 \leq S_2 \wedge K_1 \leq K_2$) is the most common and it is also used in most commercial architectures. Apart from bounding the number of L2 cache misses for a *realistic* cache architecture, the preceding properties also explain the disadvantage of using the other cache organizations to obtain real-time performance.

4. CRPD ANALYSIS

In this section, we shall describe the CRPD computation in detail. Throughout our discussion, we shall use the notations described in Table I.

Table I. Symbols used for modeling

Symbol	Description
$S1$	Number of cache sets in L1 cache
$S2$	Number of cache sets in L2 cache
K_1	Associativity of L1 cache
K_2	Associativity of L2 cache
LAT_1	L1 cache miss penalty
LAT_2	L2 cache miss penalty
$\mathbb{M}$	set of all memory blocks
$\mathbb{P}$	set of all program locations

4.1. Foundation

CRPD computation revolves around the concept of useful cache block (UCB). A UCB is a block that *must be cached* before preemption and *may be used* later [Altmeyer and Burguiere 2009]. As previous literature is based only on L1 cache, we first need to define the notion of UCB in a two-level cache hierarchy. Intuitively, a UCB might be used from L1 or L2 cache in the absence of preemption. Therefore, the *inter-task cache interferences* generated to a UCB may lead to additional cache reload latency in the preempted task.

DEFINITION 4.1. *(Useful cache block in a two-level cache hierarchy) With respect to a specific preemption point* p*, a memory block* m *is characterized by a tuple* $\langle age_1, age_2 \rangle$ *where* $age_1 \in [1, K_1] \cup \{\infty\}$ *and* $age_2 \in [1, K_2] \cup \{\infty\}$*. This characterization is defined as follows:*

— m *must be cached at* p *(either in L1 cache or in L2 cache or in both).*
— m *might be used at program point* q *that must be reached from* p *without* m *being evicted from both L1 and L2 caches, and*

— *At program point q, the LRU age of memory block m is age_1 (age_2) in L1 (L2) cache. If m is not cached in L1 (L2) at p or m is evicted from L1 (L2) cache before reaching q, age_1 (age_2) is ∞.*

EXAMPLE 4.2. *Consider the example code fragment and the respective preemption point shown in Figure 3(b). $m1$ and $m2$ are neither in L1 cache nor in L2 cache before the next use beyond preemption point. However, the memory block m is cached in both L1 and L2 before its next use beyond the preemption point. Therefore, according to the cache contents and the preemption point shown in Figure 3(b), we have the following useful cache blocks (UCBs): $m1 \mapsto \langle\infty,\infty\rangle$, $m2 \mapsto \langle\infty,\infty\rangle$, $m \mapsto \langle 1, 1\rangle$.*

In the following, we shall describe two different *flow analyses*. *Backward flow analysis* computes the useful cache blocks with respect to a program point p. *Forward flow analysis* computes the set of memory blocks which were *L1 cache hits* in the absence of preemption and might be potential causes for the indirect effect of preemption (cf. Figure 2) at a program point p. Note that an *L1 cache hit* may become an *L1 cache miss* after preemption and consequently, it may generate additional L2 cache conflict. Therefore, forward flow analysis is particularly important while computing the *indirect effect of preemption* (as demonstrated in Figures 3(e)-(f)).

Given the control flow graph (CFG) of each procedure, our analysis front-end uses *virtual inlining* [Theiling et al. 2000] to compute a *global control flow graph*. Such a *global CFG* captures the control flow graph of the entire program. Each procedure is *virtually inlined* at its calling locations. Such a *virtual inlining* is crucial for cache analysis, as the content of a cache may highly depend on procedure call contexts. Figure 4 shows an example of such *virtual inlining*. The control flow graph of procedure f is copied at two callsites. Therefore, *P1* and *P5* in the global CFG essentially access the same instruction, as shown in Figure 4.

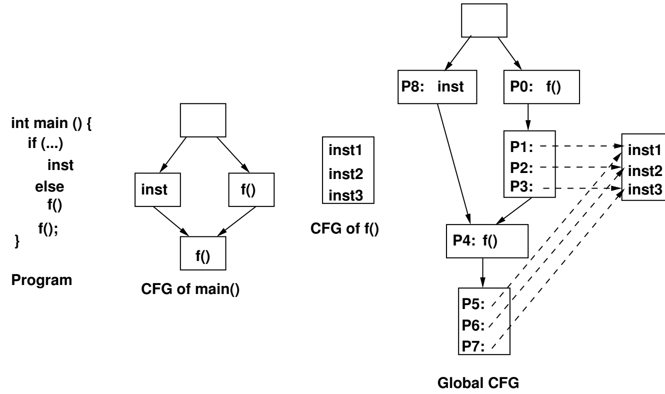

Fig. 4. Global CFG construction after *virtual inlining*

All of our analyses work on the global CFG computed after virtual inlining. In the rest of the discussion, we shall use the term *memory reference* to represent the different locations in this global CFG. As an example, $\{P0, \ldots, P8\}$ in Figure 4 capture different memory references. It is worthwhile to note that even though we only model the instruction memory, it is possible that different memory references in the program access the same memory block. As an example, memory references *P1* and *P5* access the same instruction and therefore, they access the same memory block.

4.2. Backward flow analysis

Backward flow analysis is used to compute the set of *useful cache blocks* (cf. Definition 4.1). Our analysis is based on abstract interpretation (AI). In the following, we shall describe the essential components of our AI-based analysis framework.

Abstract domain. Assume that $\mathbb{M}$ represents the set of all memory blocks and $\mathbb{D}_c$ captures the set of inclusion patterns of a memory block in a two-level cache hierarchy. The domain of the analysis ($\mathbb{D}$) is the set of all valid mappings from $\mathbb{M}$ to $\mathbb{D}_c$ as follows.

$$\mathbb{D} : \mathbb{M} \rightarrow (\mathbb{D}_c \cup \{\top\}) \quad (1)$$

with

$$\mathbb{D}_c : \{0, 1, \ldots, \mathcal{K}, \infty\} \times \{0, 1, \ldots, \mathcal{K}, \infty\} \quad (2)$$

$\top$ is an additional element in the abstract domain to capture the *uncertain* information during the analysis, $\mathcal{K} = \max(K_1, K_2)$ and ∞ captures numbers $\geq \mathcal{K} + 1$. Therefore, ∞ abstracts away all the scenarios where a specific memory block is not present in a certain cache level.

Transfer operation at each program point. We first perform the *must-cache analysis* on the preempted task. We use [Theiling et al. 2000] for analyzing L1 cache and we use [Hardy and Puaut 2008] for analyzing L2 cache. As an outcome of must-cache analysis, we obtain the abstract cache content at each program point. According to the *must-cache analysis*, let us assume that the tuple $MustAge_{m,p} = \langle age_1, age_2 \rangle$ captures *LRU ages* of memory block m (in both the cache levels) immediately before the program point p. If m is not present in L1 cache, $MustAge_{m,p}$ can be captured as a tuple $\langle \infty, age_2 \rangle$. Similarly, if m is not present in L2 cache, $MustAge_{m,p}$ can be captured as a tuple $\langle age_1, \infty \rangle$.

The transfer function τ modifies the abstract state (*i.e.* an element from the abstract domain $\mathbb{D}$) at each program point. Since the direction of the analysis is *backward*, such a transfer function takes the abstract state after a program point as *input* and computes the abstract state before the program point as *output*. Formally, the transfer function τ_b can be defined as follows.

$$\tau_b : \mathbb{D} \times \mathbb{P} \rightarrow \mathbb{D}$$

$$\tau_b(\mathcal{D}, p) = \mathcal{D}[m_p \mapsto MustAge_{m_p,p}] \quad (3)$$

$\mathbb{P}$ denotes the set of all program points and m_p captures the memory block accessed at program point p. $\mathcal{D}$ captures the abstract state after the program point p. $\mathcal{D}[m_p \mapsto MustAge_{m_p,p}]$ updates the mapping of memory block m_p (*i.e.* the memory block accessed at program point p) as $m_p \mapsto MustAge_{m_p,p}$. The mapping of all other memory blocks except m_p remain unchanged after applying the transfer function τ_b.

Join operation to merge multiple abstract states. Since programs usually contain branches and loops, an abstract *join* operation is used to combine multiple abstract states at the control flow merge points.

To define the *join* operation, we need to define a *partial order* $\preceq$ on $\mathbb{D}_c$. Recall that $\mathbb{D}_c$ captures the set of possible inclusion patterns of a memory block in the two-level cache hierarchy. In the following, we shall first define a couple of basic operations (Δ and $\boxplus$) on the domain $\mathbb{D}_c$. Such basic operations are required to establish a partial order $\preceq$ among the elements of $\mathbb{D}_c$.

Let us consider a tuple $cu = \langle cu_1, cu_2 \rangle \in \mathbb{D}_c$. For some memory block $m \in \mathbb{M}$, assume that the mapping $m \mapsto \langle cu_1, cu_2 \rangle$ belongs to some abstract state during the *backward-flow* analysis. We use a function Δ to compute the latency of fetching the memory block m. Δ is defined as follows.

$$\Delta : \mathbb{D}_c \rightarrow \mathbb{N}$$

$$\Delta(\langle cu_1, cu_2 \rangle) = \begin{cases} 0, & \text{if } cu_1 \neq \infty \wedge cu_1 \leq K_1; \\ LAT_1, & \text{else if } cu_2 \neq \infty \wedge cu_2 \leq K_2; \\ LAT_1 + LAT_2, & \text{otherwise.} \end{cases} \quad (4)$$

Recall that LAT_1 and LAT_2 denote the fixed L1 and L2 cache miss penalties, respectively.

In the following, the operation $\boxplus$ captures the element-wise addition operation in the domain $\mathbb{D}_c$. Assume $cu = \langle cu_1, cu_2 \rangle \in \mathbb{D}_c$ and $cv = \langle cv_1, cv_2 \rangle \in \mathbb{D}_c$. In the following, we define the operation $\boxplus$ between cu and cv. For the sake of simplicity, $\boxplus$ is defined using the *infix* notation.

$$\boxplus : \mathbb{D}_c \times \mathbb{D}_c \rightarrow \mathbb{D}_c$$

$$\langle cu_1, cu_2 \rangle \boxplus \langle cv_1, cv_2 \rangle = \langle cuv_1, cuv_2 \rangle \tag{5}$$

where

$$cuv_i = \begin{cases} \infty, & \text{if } cu_i = \infty \lor cv_i = \infty \lor cu_i + cv_i > K_i; \\ cu_i + cv_i, & \text{otherwise} \end{cases} \tag{6}$$

Note that saturation is used (using the element ∞) in the addition operation $\boxplus$ instead of overflow.

Given $cu_1, cu_2 \in \mathbb{D}_c$, an informal description of the partial order $\preceq$ can be introduced as follows: $cu_1 \preceq cu_2$ if and only if cu_2 leads to more cache reload latency compared to cu_1 in the presence of *any additional cache conflict* ce. Therefore, the partial order $\preceq$ can be captured by the following logical equivalence:

$$cu_1 \preceq cu_2 \Leftrightarrow \forall ce \in \mathbb{D}_c.\ \Delta(cu_1 \boxplus ce) - \Delta(cu_1) \leq \Delta(cu_2 \boxplus ce) - \Delta(cu_2) \tag{7}$$

However, it is possible that $cu_1 \not\preceq cu_2$ and $cu_2 \not\preceq cu_1$. Therefore, we introduce a *join semi-lattice* $\mathbb{D}_c \cup \{\top\}$ to define the *least upper bound* operator. $\top$ captures the *uncertain* information during the analysis and therefore, for any $cu \in \mathbb{D}_c$, $cu \preceq \top$. Now we can define the least upper bound operator $\bigsqcup$ on the set $\mathbb{D}_c \cup \{\top\}$ as follows.

$$\bigsqcup : (\mathbb{D}_c \cup \{\top\}) \times (\mathbb{D}_c \cup \{\top\}) \rightarrow \mathbb{D}_c \cup \{\top\}$$

$$\bigsqcup(cu_1, cu_2) = \begin{cases} \top, & \text{if } cu_1 = \top \ \lor \ cu_2 = \top; \\ cu_2, & \text{if } cu_1 \preceq cu_2; \\ cu_1, & \text{if } cu_2 \preceq cu_1; \\ \top, & \text{otherwise.} \end{cases} \tag{8}$$

The abstract join operation in our backward-flow analysis merges two abstract states from the abstract domain (*i.e.* $\mathbb{D}$). Assume that $\mathcal{D}_1, \mathcal{D}_2 \in \mathbb{D}$. For a memory block $m \in \mathbb{M}$, let us assume $\mathcal{D}_1(m) = cu_1$ and $\mathcal{D}_2(m) = cu_2$. After the join operation, the least upper bound of cu_1 and cu_2 (*i.e.* $\bigsqcup(cu_1, cu_2)$ as defined in Equation (8)) is mapped to memory block m. Therefore, the formal definition of the join operation $\widehat{J_{\mathbb{D}}}$ is as follows.

$$\widehat{J_{\mathbb{D}}} : \mathbb{D} \times \mathbb{D} \rightarrow \mathbb{D}$$

$$\widehat{J_{\mathbb{D}}}(\mathcal{D}_1, \mathcal{D}_2) = \bigcup_{m \in \mathbb{M}} \{m \mapsto \bigsqcup(cu_1, cu_2) |\ \mathcal{D}_1(m) = cu_1 \land \mathcal{D}_2(m) = cu_2\} \tag{9}$$

Initialization. We start our backward flow analysis with an abstract state $\{m \mapsto \langle \infty, \infty \rangle \mid m \in \mathbb{M}\}$. At each program point, we check the accessed memory block and apply our transfer function τ_b as described in Equation (3). Since our analysis is a backward-flow analysis, the abstract state at the exit of a basic block is computed by combining all the abstract states at the entry of its successors (via the join operation $\widehat{J_{\mathbb{D}}}$ in Equation (9)). The analysis terminates when a fixed-point is obtained at each program point.

4.3. Forward flow analysis

Forward flow analysis is primarily required to compute the indirect effect of preemption. Recall that the indirect effect of preemption is potentially caused by memory blocks which were *L1 cache hits* before the preemption, but they may access L2 cache after the preemption (cf. Figure 2).

With respect to a program point p, forward flow analysis computes a set of memory blocks $\mathcal{M}_p$ where each $m \in \mathcal{M}_p$ satisfies the following two conditions:

— m must be accessed along one of the paths starting from the entry point of the program and ending at p. We call such references of m *reachable references* to p.

— At least one of the reachable references of m (*w.r.t.* p) *must be an L1 cache hit* in the absence of preemption.

Therefore, the abstract domain of the analysis is all possible subsets of memory blocks accessed in the program (*i.e.* $2^{\mathbb{M}}$). The abstract transfer function τ_f is applied at each program point. Since the analysis direction is forward, τ_f takes the abstract state before a program point p (say $\mathcal{M} \in 2^{\mathbb{M}}$) as *input* and it computes the abstract state after the program point p as *output*. Additionally, τ_f uses the *must-cache analysis* results [Theiling et al. 2000] to detect L1 cache hits at a particular program point. According to the must-cache analysis, let us assume $MustACS_{p,1}$ captures the content of L1 cache immediately before the program point p. If m_p is the memory block accessed at program point p and m_p is contained in $MustACS_{p,1}$ (*i.e.* the memory access at p is a guaranteed L1 cache hit), τ_f augments the input abstract state $\mathcal{M}$ with m_p. The formal definition of τ_f is as follows.

$$\tau_f : 2^{\mathbb{M}} \times \mathbb{P} \rightarrow 2^{\mathbb{M}}$$

$$\tau_f(\mathcal{M}, p) = \begin{cases} \mathcal{M} \cup \{m_p\}, & \text{if } m_p \in MustACS_{p,1}; \\ \mathcal{M}, & \text{otherwise.} \end{cases} \tag{10}$$

The abstract join operation simply performs a *set union* of multiple abstract states at a control-flow merge point.

Our forward flow analysis starts with the *empty set* and at each program point, we apply the transfer function τ_f (as defined in Equation (10)). Since the direction of the analysis is *forward*, the abstract state at the entry of each basic block is computed by taking a simple set union of all the abstract states at the exit of its predecessors.

4.4. Analysis of the preempting task

For an accurate computation of CRPD, we need to compute the set of cache blocks possibly used by the preempting task. The set of used cache blocks by the preempting task is called *evicting cache blocks* (ECB) [Lee et al. 1998]. Since we consider set-associative LRU caches, for each cache set, we compute the maximum number of cache blocks used by the preempting task. ECBs can easily be computed by performing the *may-cache analysis* on the preempting task (using [Hardy and Puaut 2008]). The may-cache analysis computes an *over-approximation* of cache contents at each program point. Let us consider the *exit point* e of the preempting task. Therefore, the analyzed cache content at e must include all possibly accessed memory blocks (subject to the size of cache) in the preempting task. According to the may-cache analysis, let us assume that $MayACS_{e,1}(s)$ and $MayACS_{e,2}(s)$ denote the contents of L1 and L2 cache set s, respectively, at the *exit* point e of the preempting task. For each memory block m accessed in the preempted task, we define a tuple $\mathcal{CE}_m = \langle \mathcal{CE}_{m,1}, \mathcal{CE}_{m,2} \rangle$. Intuitively, $\mathcal{CE}_m$ captures the maximum number of ECBs mapping to the same cache sets as m. Let us assume that memory block m is mapped to cache set $\mathcal{S}_{m,1}$ ($\mathcal{S}_{m,2}$) in L1 (L2) cache. Therefore, we can define $\mathcal{CE}_m = \langle \mathcal{CE}_{m,1}, \mathcal{CE}_{m,2} \rangle$ as follows.

$$\mathcal{CE}_m = \langle \mathcal{CE}_{m,1}, \mathcal{CE}_{m,2} \rangle$$

where

$$\mathcal{CE}_{m,i} = |MayACS_{e,i}(\mathcal{S}_{m,i})| \tag{11}$$

4.5. Preemption delay computation

In this section, we show the CRPD computation using results of *backward-flow analysis* (Section 4.2), *forward-flow analysis* (Section 4.3) and *must-cache analyses* [Theiling et al. 2000; Hardy and Puaut 2008].

We start with a few definitions. The following definitions are based on the fixed-point computations by backward and forward flow analyses.

— $\mathcal{CU}_{m,p}$: After backward-flow analysis, let us assume that $\mathcal{D}_{b,p}$ captures the fixed-point on the abstract state at program point p. Therefore, $\mathcal{D}_{b,p} \in \mathbb{M} \rightarrow \mathbb{D}_c \cup \{\top\}$. We define $\mathcal{CU}_{m,p}$ as $\mathcal{D}_{b,p}(m)$. As a result, $\mathcal{CU}_{m,p} \in \mathbb{D}_c \cup \{\top\}$.
— $\mathcal{D}_{f,p}$: After forward-flow analysis, $\mathcal{D}_{f,p}$ captures the fixed-point on the abstract state at program point p. Therefore, $\mathcal{D}_{f,p} \in 2^{\mathbb{M}}$.

Additionally, we use $\mathcal{CE}_m = \langle \mathcal{CE}_{m,1}, \mathcal{CE}_{m,2} \rangle$ to capture evicting cache blocks (ECB) conflicting with memory block m (cf. Equation 11).

4.5.1. Indirect preemption factor. We had earlier illustrated that a CRPD analysis solely based on UCBs and ECBs may lead to an *unsafe* result (cf. Figures 3(a)-(b)). The root cause of such complications arises due to the presence of indirect effect (cf. Figure 2). Therefore, in the following, we first define a quantity which is crucial to take into account the indirect effect of preemption.

For the sake of clarity, we first show the CRPD computation with respect to a specific preemption point p. Subsequently, we show the computation of CRPD for an arbitrary preemption point. Given a preemption point p, we compute a quantity $\mathcal{ID}_{r,p}$ for a program point r. Let us assume m_r denotes the memory block accessed at r. Intuitively, $\mathcal{ID}_{r,p}$ captures an *over-approximation* on the set of memory blocks which may create indirect preemption effect to m_r. Such memory blocks must have been accessed from L1 cache in the absence of preemption, however, they might be accessed from L2 cache after preemption. Therefore, any memory block m in $\mathcal{ID}_{r,p}$ must satisfy all the conditions as stated below.

— m must be accessed along some path starting from the entry node and ending at r. Additionally, r must be reachable from at least one reference of m that must be an *L1 cache hit* in the absence of preemption. Therefore, $m \in \mathcal{D}_{f,r}$.
— m must be accessed after preemption point p and such an access must be an L1 cache hit in the absence of preemption. Therefore, $\mathcal{CU}_{m,p} \neq \langle \infty, \infty \rangle$. If $\mathcal{CU}_{m,p} = \langle \infty, \infty \rangle$, m is either not accessed after preemption point p or any immediate access of m beyond p might be an L1 cache miss in the absence of preemption (cf. Section 4.2).
— m might suffer an L1 cache miss due to preemption and m must map to the same L2 cache set as m_r. Let us assume $\mathcal{CU}_{m,p} = \langle \mathcal{CU}_{m,p,1}, \mathcal{CU}_{m,p,2} \rangle$. Therefore, $\mathcal{CU}_{m,p,1} + \mathcal{CE}_{m,1} > K_1$. If m is mapped to the L2 cache set $\mathcal{S}_{m,2}$, we additionally have $\mathcal{S}_{m,2} = \mathcal{S}_{m_r,2}$.

Aggregating the above notion of description, we can now formally define $\mathcal{ID}_{r,p}$ as follows.

$$\begin{aligned} \mathcal{ID}_{r,p} = \{m \quad | \quad & m \neq m_r \wedge m \in \mathcal{D}_{f,r} \wedge \mathcal{S}_{m,2} = \mathcal{S}_{m_r,2} \wedge \mathcal{CU}_{m,p} \neq \langle \infty, \infty \rangle \\ & \wedge \ (\mathcal{CU}_{m,p} = \top \vee (\mathcal{CU}_{m,p} \neq \top \wedge \mathcal{CU}_{m,p,1} + \mathcal{CE}_{m,1} > K_1))\} \end{aligned} \tag{12}$$

4.5.2. CRPD computation. For an arbitrary preemption point p, an overview of the preemption delay computation is shown in Figure 5. The computed preemption delay depends on cache hit-miss categorizations of memory references (*i.e.* L1 and L2 cache hit/miss) in the absence of preemption. For *L1 cache hits* in the absence of preemption, it is sufficient to check only the first access of the respective memory block after preemption [Lee et al. 1998]. This is because the respective memory block will be reloaded into L1 cache once it is first accessed after the preemption. For such memory blocks, $CRT_{p,1}$ captures the reloading delay from L2 cache and $CRT_{p,2}$ captures the reloading delay from main memory (cf. Figure 5).

Let us now consider the memory references which were L1 cache misses, but L2 cache hits, in the absence of preemption. As evidenced by our example in Figure 3(b), it is *insufficient* to consider only the first references of such memory blocks after preemption. Therefore, we need to go through all program locations which were L1 cache misses, but L2 cache hits in the absence of preemption. We distinguish between the first access and all other accesses to such a program location (say r) after preemption. The first access to r after preemption may suffer L2 cache miss penalty due to the combined effect of intra-task and inter-task L2 cache conflicts. Our examples in Figures 3(a)-(b)

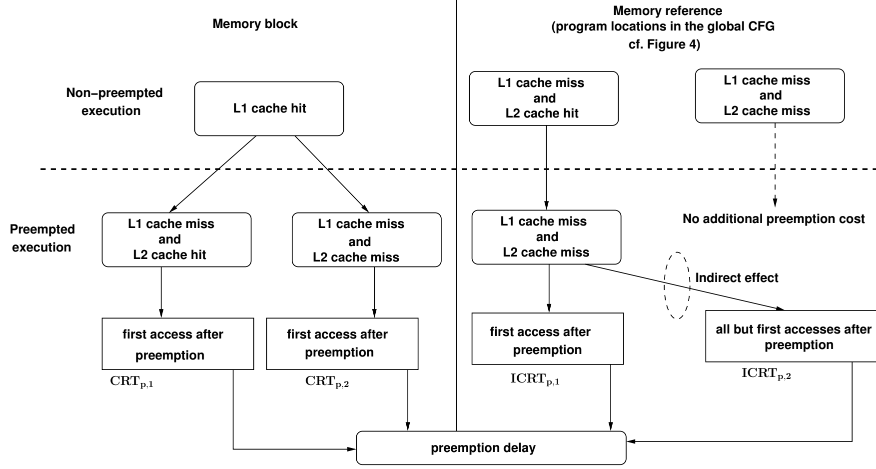

Fig. 5. Overview of preemption delay computation

illustrate such situations. In Figure 5, $ICRT_{p,1}$ captures this penalty. Any subsequent accesses to r after preemption may suffer L2 cache miss penalty only due to the indirect effect of preemption. Our examples in Figures 3(e)-(f) illustrate such situations. In Figure 5, $ICRT_{p,2}$ captures this penalty. Finally, in Theorem 5.3, we have proved an upper bound on the number of accesses that may suffer L2 cache misses solely due to the indirect effect of preemption.

Total preemption delay is computed by accumulating the preemption delay contributed by different components (*i.e.* $CRT_{p,1}$, $CRT_{p,2}$, $ICRT_{p,1}$ and $ICRT_{p,2}$). Finally, CRPD maximizes the preemption delay for any arbitrary preemption point. In the following, we shall describe the different components of the preemption delay computation as shown in Figure 5. It is worthwhile to note that the following computations are carried out with respect to an arbitrary preemption point p.

Assume that $\mathcal{CU}_{m,p} = \langle \mathcal{CU}_{m,p,1}, \mathcal{CU}_{m,p,2} \rangle$ and let m_r denote the memory block accessed at program point r. In the following, $\mathbb{M}_1$ captures the set of memory blocks which might be L1 cache hits in the absence of preemption, but may suffer L1 cache miss penalties after the preemption.

$$\begin{aligned}\mathbb{M}_1 = \{m \quad | \quad & \mathcal{CU}_{m,p} \neq \top \wedge \mathcal{CU}_{m,p,1} \neq \infty \wedge \mathcal{CU}_{m,p,2} \neq \infty \\ & \wedge\ \mathcal{CU}_{m,p,1} + \mathcal{CE}_{m,1} > K_1 \\ & \wedge\ \mathcal{CU}_{m,p,2} + \mathcal{CE}_{m,2} + \max_{r:m_r=m} |\mathcal{ID}_{r,p}| \leq K_2\}\end{aligned} \tag{13}$$

Note that the indirect preemption factor $\max_{r:m_r=m} |\mathcal{ID}_{r,p}|$ is essential for a sound estimation of L2 cache conflicts to m.

In a similar way, $\mathbb{M}_2$ captures the set of memory blocks that might be L1 cache hits in the absence of preemption, but may suffer the sum of L1 and L2 cache miss penalties after the preemption.

$$\begin{aligned}\mathbb{M}_2 = \{m \quad | \quad & (\mathcal{CU}_{m,p} = \top \wedge \mathcal{CE}_m \neq \langle 0, 0 \rangle) \vee (\mathcal{CU}_{m,p} \neq \top \wedge \mathcal{CU}_{m,p,1} \neq \infty \\ & \wedge\ \mathcal{CU}_{m,p,1} + \mathcal{CE}_{m,1} > K_1 \wedge m \notin \mathbb{M}_1)\}\end{aligned} \tag{14}$$

Finally, $CRT_{p,1}$ and $CRT_{p,2}$ (cf. Figure 5) can be computed from $\mathbb{M}_1$ and $\mathbb{M}_2$ as follows.

$$CRT_{p,1} = |\mathbb{M}_1| \times LAT_1 \tag{15}$$

$$CRT_{p,2} = |\mathbb{M}_2| \times (LAT_1 + LAT_2) \tag{16}$$

To compute $ICRT_{p,1}$ and $ICRT_{p,2}$, we need to check the set of program locations which had L1 cache misses, but L2 cache hits in the absence of preemption (cf. Figure 5). Let us assume $\mathbb{P}_2$

captures this set of program locations. To compute the cache hit/miss categorization in the absence of preemption, we use must-cache analysis [Theiling et al. 2000; Hardy and Puaut 2008]. Let us assume that the tuple $MustAge_{m,r} = \langle MustAge_{m,r,1}, MustAge_{m,r,2} \rangle$ captures *LRU ages* of memory block m (in both the cache levels) immediately before the program location $r \in \mathbb{P}_2$. If m is not present in L1 cache, $MustAge_{m,r}$ can be captured as a tuple $\langle \infty, MustAge_{m,p,2} \rangle$. Let us assume that m_r captures the memory block accessed at program location r. For any $r \in \mathbb{P}_2$, $ICRT_{p,1}$ computes the total cache miss penalty for the first visit to r after preemption.

$$ICRT_{p,1} = \sum_{r \in \mathbb{P}_2} \begin{cases} 0, \text{ if } MustAge_{m_r,r,2} + \mathcal{CE}_{m_r,2} + |\mathcal{ID}_{r,p}| \leq K_2; \\ LAT_2, \text{ otherwise.} \end{cases} \tag{17}$$

Similarly, for all program locations $r \in \mathbb{P}_2$, $ICRT_{p,2}$ (cf. Figure 5) computes the total cache miss penalty solely due to the indirect effect of preemption. $ICRT_{p,2}$ can be formally described as follows.

$$ICRT_{p,2} = IL2_{ind} \times \sum_{r \in \mathbb{P}_2} \begin{cases} 0, \text{ if } MustAge_{m_r,r,2} + |\mathcal{ID}_{r,p}| \leq K_2; \\ LAT_2, \text{ otherwise.} \end{cases} \tag{18}$$

In Equation (18), $IL2_{ind}$ denotes an upper bound on the number of L2 cache misses due to the indirect effect of preemption (for a proof of this bound, refer to Section 5, Theorem 5.3).

Final CRPD computation. Final CRPD computation maximizes the preemption delay for an arbitrary preemption point p. Therefore, the final CRPD computation can be defined by the following maximization function.

$$CRPD_{final} = \max_{p \in \mathbb{P}} (CRT_{p,1} + CRT_{p,2} + ICRT_{p,1} + ICRT_{p,2}) \tag{19}$$

4.6. Extension of the basic framework

4.6.1. Nested preemptions. In the preceding section, we have described the CRPD computation for a single preemption. Our framework can easily be extended with nested preemptions. Recall that our framework computes evicting cache blocks (ECB) using the *may-cache analysis* (cf. Equation (11)). To handle nested preemptions, we need to take into account all the ECBs from all higher priority tasks.

Assume that $T_1, T_2, \ldots, T_n$ are tasks in the decreasing order of priorities and we want to compute the CRPD for T_n. The *may-cache analysis* is performed for each task in the set $\{T_1, T_2, \ldots, T_{n-1}\}$. The set of ECBs computed for $T_1, T_2, \ldots, T_{n-1}$ are then merged (*i.e.* set union) together to produce a final estimation of ECBs. Let us assume that $MayACS_{T_i,e,1}(s)$ and $MayACS_{T_i,e,2}(s)$ denote the content of L1 and L2 cache set s, respectively, at the *exit* point e of the preempting task T_i. For each memory block m accessed in task T_n, Equation (11) is modified as follows.

$$\mathcal{CE}_m = \langle \mathcal{CE}_{m,1}, \mathcal{CE}_{m,2} \rangle$$

where

$$\mathcal{CE}_{m,i} = | \bigcup_{t \in \{T_1, \ldots, T_{n-1}\}} MayACS_{t,e,i}(\mathcal{S}_{m,i}) | \tag{20}$$

Recall that $\mathcal{S}_{m,1}$ ($\mathcal{S}_{m,2}$) captures L1 (L2) cache set in which memory block m is mapped.

Our rest of the framework remains unchanged except for the modified tuple $\mathcal{CE}_m$ (as computed in Equation (20)). It is worthwhile to note that we perform a set union of all the possible ECBs. As a result, the computed ECBs (*i.e.* $\mathcal{CE}_m$ in Equation (20)) clearly over-approximate the number of inter-task cache conflicts to memory block m.

4.6.2. Multiple preemptions. Our CRPD analysis framework for non-inclusive cache hierarchies can also be adapted in the presence of multiple preemptions. To adapt our framework in the presence of multiple preemptions, we rely on the existing literature [Altmeyer et al. 2010]. In the following, we first briefly describe the technical challenges in the presence of multiple preemptions and their solutions proposed in [Altmeyer et al. 2010].

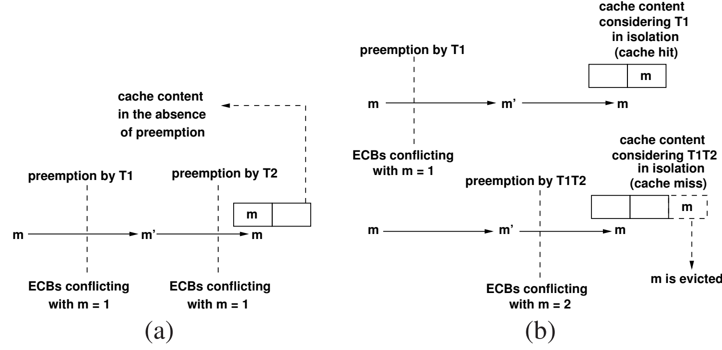

Fig. 6. Multiple preemptions by tasks $T1$ and $T2$ along the memory access sequence $m \rightarrow m' \rightarrow m$. Assume m' does not conflict with m in cache. (a) Cache contents in the non-preempted execution. Neither $T1$ nor $T2$ can evict memory block m from cache, but the interaction between tasks $T1$ and $T2$ will evict memory block m from cache, (b) sound CRPD computation by considering tasks $T1$ and the sequential composition $T1T2$ in isolation [Altmeyer et al. 2010]

Background. The key challenges in multiple preemptions appear in the presence of *set-associative* caches (as also shown in [Altmeyer et al. 2010]). For *direct-mapped* caches, multiple preemptions do not pose any significant challenges in computing the CRPD. For a direct-mapped cache, let us consider a useful cache block in the preempted task. If any memory block in any preempting task conflicts with such a useful cache block, the useful cache block might be replaced from cache. As a result, for direct-mapped caches, total CRPD in the presence of multiple preemptions is bounded by the sum of CRPDs each preempting task will cause in isolation. For set-associative caches, however, such a reasoning may lead to an unsafe CRPD estimation. We show an example to illustrate this point. Figure 6 shows a memory access sequence $m \rightarrow m' \rightarrow m$ in the preempted task. The preemption by task $T1$ takes place between the memory accesses m and m'. Another preemption by task $T2$ takes place between the memory accesses m' and m. Assume that both the preempting tasks have at most one evicting cache block that conflicts with memory block m. Figure 6(a) shows two different preemptions by task $T1$ and task $T2$ in the presence of a two-way, set-associative cache. We can observe that neither $T1$ nor $T2$ can cause the eviction of m in isolation. However, $T1$ and $T2$ together may evict the memory block m, leading to a cache miss after preemption. Therefore, for set-associative caches, it is crucial to consider the interactions among different preempting tasks (*e.g.* the interaction between tasks $T1$ and $T2$ in Figure 6(a)). To solve the challenges due to multiple preemptions, the work in [Altmeyer et al. 2010] considers preemption by the sequential compositions of different preempting tasks. Such a solution is primarily based on the following insight: if a memory block in the preempted task is evicted by the interaction between two different preempting tasks $T1$ and $T2$, the memory block will also be evicted by the sequential composition $T1T2$ of the preempting tasks $T1$ and $T2$. Figure 6(b) shows the solution, where the second access of m can now be predicted as a cache miss. Such a solution allows us to consider preemptions in isolation. At the same time, it allows to estimate a sound CRPD value. For a detailed description of this technique, readers are referred to [Altmeyer et al. 2010].

Multiple preemptions in the presence of non-inclusive cache hierarchies. Multiple preemptions do not pose any additional challenge for non-inclusive cache hierarchies. The key challenges in the presence of multiple preemptions appear due to the presence of interacting preempting tasks, as

shown in Figure 6(b). Considering each preemption in isolation may lead to an under-approximation of evicting cache blocks (ECBs), as also seen in Figure 6(b). Therefore, the solution proposed by [Altmeyer et al. 2010] uses a sequential composition of the interacting preempting tasks in order to compute a *sound* over-approximation of ECBs. Since we use the *may-cache analysis* to compute the set of ECBs, such an over-approximation of ECBs will be preserved even in the presence of cache hierarchies. Therefore, our CRPD analysis framework can be adapted into [Altmeyer et al. 2010] to handle multiple preemptions. It is worthwhile to mention that we do not claim any contribution in handling multiple preemptions. Our contribution is to address the technical challenges in the presence of non-inclusive cache hierarchies. This section discusses the methodology to adapt our basic framework into existing CRPD analysis frameworks that handle multiple preemptions (*e.g.* [Altmeyer et al. 2010]). There has been extensive research (*e.g.* [Staschulat and Ernst 2004]) on improving the estimation of CRPD in the presence of multiple preemptions. In future, we can study such solutions to see if they can be adapted in the presence of cache hierarchies.

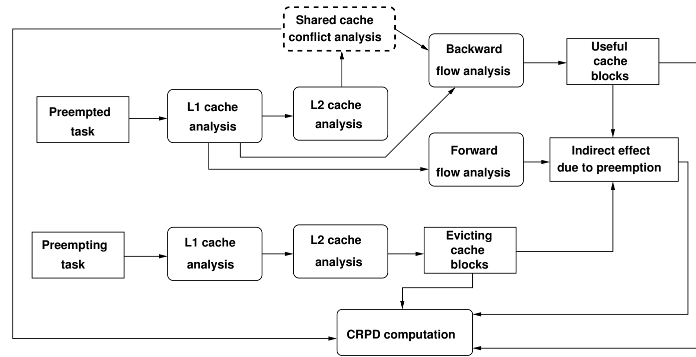

Fig. 7. CRPD analysis in the presence of shared caches

4.6.3. Shared caches. The presence of shared caches may generate additional L2 cache misses due to inter-core cache conflicts. Inter-core cache conflicts can easily be integrated into our framework. With such an integration, we can compare the different sources of overestimations from different cache conflict analyses (*i.e.* intra-task, inter-task and inter-core). According to the must-cache analysis [Theiling et al. 2000; Hardy and Puaut 2008] and without considering inter-core cache conflicts, let us assume $age(m)$ captures the LRU age of memory block m in L2 cache. Further assume that IC denotes the number of unique memory blocks accessed by programs running on different cores and that may potentially access the same L2 cache set as m. To consider the inter-core cache conflicts faced by memory block m, $age(m)$ is updated as $age(m) + IC$. As a result, the updated LRU age now captures both intra-task and inter-core cache conflicts. Such an integration is similar to the one used in [Li et al. 2009; Hardy et al. 2009]. However, it is worthwhile to note that the work proposed in [Li et al. 2009] discusses a worst-case response time (WCRT) analysis in multi-tasking systems. Integrating both inter-core cache conflicts and CRPD analysis into the WCRT analysis framework may pose additional challenges and it is an interesting topic to be studied in future. *In this work, we show the effect of three different cache conflicts – intra-task, inter-task and inter-core on the execution time of a single task*. Figure 7 shows the integration of inter-core cache conflicts via the dotted box. Once LRU ages are updated with inter-core cache conflicts, they are subsequently used during our backward-flow analysis.

5. SOUNDNESS OF THE ANALYSIS

In this section, we shall first briefly outline the *soundness* proof of our CRPD analysis framework. Subsequently, in Theorem 5.3, we shall prove theoretical bounds on the number of L2 cache misses due to the indirect effect of preemption.

Structure of the soundness proofs

Soundness of over-approximated ECB. Our analysis framework computes an over-approximation of evicting cache blocks (cf. Equation (11)). It is always *sound* to over-approximate the set of evicting cache blocks (ECB). Recall that the CRPD computation in our framework revolves around four quantities – $CRT_{p,1}$, $CRT_{p,2}$, $ICRT_{p,1}$ and $ICRT_{p,2}$ (Equations (13)-(18)). Equations (13)-(18) clearly show that an over-approximation of ECBs will only overestimate the value of $CRT_{p,1}$, $CRT_{p,2}$, $ICRT_{p,1}$ and $ICRT_{p,2}$. This will keep the overall CRPD analysis *sound*.

Soundness of WCET+CRPD. Since CRPD analysis is normally used with a WCET analysis [Altmeyer and Burguiere 2009], our approach guarantees a *sound* estimation of the sum of WCET and CRPD. Consider a memory reference that is predicted *cache miss* by the WCET analysis in both L1 and L2 caches. In our CRPD analysis, we do not consider any additional cache miss for such memory references. However, it is possible that a memory reference suffers different delays after preemption along different paths in the program. The least upper bound operator defined in Equation (8) ensures that we always account for the *maximum* among all possible cache reload delays. More precisely, the four key components of our CRPD computation (*i.e.* $CRT_{p,1}$, $CRT_{p,2}$, $ICRT_{p,1}$ and $ICRT_{p,2}$) are always overestimated using the partial order defined in Equation (7).

Number of cache misses due to the indirect effect of preemption. Recall that the presence of cache hierarchies may introduce multiple cache misses for the same memory reference after preemption. Such a scenario may arise due to the increased intra-task cache interferences after preemption. We call this effect of preemption *indirect effect*. In Section 3, we had introduced the bound on the number of cache misses due to the indirect effect (bound on $IL2_{ind}$). In Theorem 5.3, we formally prove this bound on $IL2_{ind}$.

5.1. Detailed proof

In this section, we shall prove theoretical bounds on the indirect effect of preemption. For the following discussion, we shall be using the terminologies defined in Table I.

PROPERTY 5.1. *Assume two memory blocks m_1 and m_2 that map to the same L2 cache set. If $S_1 \leq S_2$ holds, then m_1 and m_2 map to the same L1 cache set as well.*

PROPERTY 5.2. *If $S_1 > S_2$ holds, then at most $(\frac{S_1}{S_2})K_1$ cache blocks in L1 cache map to the same L2 cache set.*

THEOREM 5.3. *Consider any memory reference $ref(M)$ in the preempted task that accesses a memory block M. Assume $ref(M)$ was an L2 cache hit and an L1 cache miss in the absence of preemption. Further assume that $ref(M)$ suffers $IL2_{ind}$ number of L2 cache misses due to the indirect effect of preemption. $IL2_{ind}$ is bounded as follows:*

— *if $S_1 \leq S_2$ and $K_1 \leq K_2$ hold, then $IL2_{ind} \leq 1$,*
— *if $S_1 \leq S_2$ and $K_1 > K_2$ hold, then $IL2_{ind} \leq K_1 - K_2$, and*
— *if $S_1 > S_2$ holds, then $IL2_{ind} \leq (\frac{S_1}{S_2})K_1 - 1$.*

PROOF. If $ref(M)$ resides outside of all loops, then our claim is trivially satisfied, as $ref(M)$ can be executed at most once after the preemption. Therefore, in the following, we are concerned only about the case when $ref(M)$ is accessed within a loop.

Recall that the indirect effect of preemption may arise when some memory references access L2 cache *only after preemption*, but they do not access L2 cache in the absence of preemption (as demonstrated via Figure 2).

The basic idea of all the three proofs is as follows: assume that we want to impose a bound B on $IL2_{ind}$. For a memory reference $ref(M)$, we first construct B different program paths leading to $ref(M)$ after a preemption. Each path may result in the eviction of M. Therefore, B different program paths may generate a total of B different *L2 cache misses* for $ref(M)$ after the preemption.

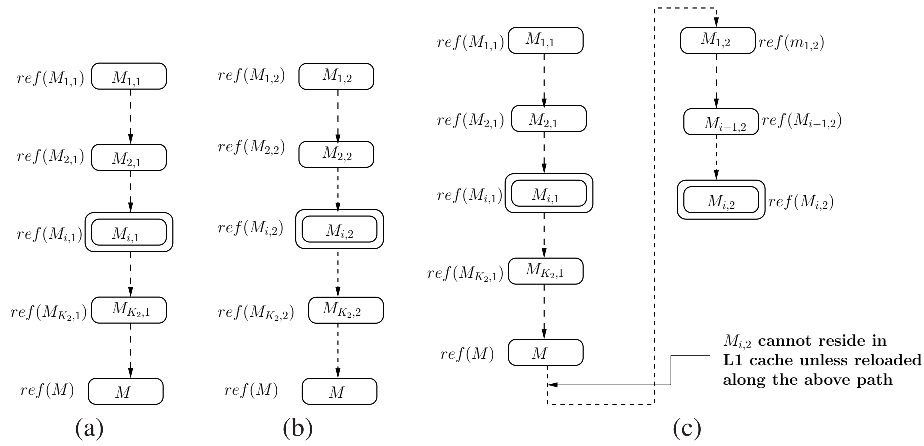

Fig. 8. Bounding the indirect effect of preemption when $S_1 \leq S_2$ and $K_1 \leq K_2$. $ref(M_{i,1})$ and $ref(M_{i,2})$ are *L1 cache hits* in the absence of preemption, but access L2 cache after preemption. (a)&(b): Indirect effect of preemption on memory reference $ref(M)$, (c): a potential scenario which shows that (a)&(b) cannot happen together.

If all of these L2 cache misses are generated due to the indirect effect of preemption, then each of the B constructed path must contain at least one memory reference which accesses L2 cache *only after preemption* (and not in the absence of preemption). Subsequently, we show the impossibility of constructing a $B+1$-th path (say $\mathcal{P}_{B+1}$) in a similar fashion. We show that any such path $\mathcal{P}_{B+1}$ will contain only memory references that are either *L1 cache hits after preemption* or *L1 cache misses even in the absence of preemption*. As a result, $\mathcal{P}_{B+1}$ cannot lead to an L2 cache miss for $ref(M)$ due to the indirect effect of preemption.

$\boxed{S_1 \leq S_2 \wedge K_1 \leq K_2}$. If M is evicted from L2 cache, then M must have faced K_2 unique conflicts since it is *last accessed* from L2 cache. Consider the scenario in Figure 8(a). Let us assume one program path $\mathcal{P}_1 := ref(M_{1,1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i,1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{K_2,1}) \rightsquigarrow ref(M)$ which accesses K_2 unique memory blocks $\{M_{1,1}, \ldots, M_{i,1}, \ldots, M_{K_2,1}\}$, all mapping to the same L2 cache set as M. If all the references in $\{ref(M_{1,1}), \ldots, ref(M_{K_2,1}\}$ access L2 cache in the absence of preemption, then M would be evicted from L2 cache (after accessing $ref(M_{K_2,1})$) even in the absence of preemption. This leads to a contradiction that $ref(M)$ is an *L2 cache hit* in the absence of preemption. Therefore, to consider the *indirect effect*, there must be one memory reference, say $ref(M_{i,1}) \in \{ref(M_{1,1}), \ldots, ref(M_{i,1}), \ldots, ref(M_{K_2,1})\}$, which does not generate L2 cache conflict in the absence of preemption (being an *L1 cache hit*), but $ref(M_{i,1})$ generates L2 cache conflict after preemption (as $M_{i,1}$ might be evicted from L1 cache due to preemptions).

Now consider the scenario in Figure 8(b) which illustrates a program path $\mathcal{P}_2 := ref(M_{1,2}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i,2}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{K_2,2}) \rightsquigarrow ref(M)$. Assume that the execution of $\mathcal{P}_2$ can create an indirect effect of preemption on memory reference $ref(M)$ after the execution of $\mathcal{P}_1$. Therefore, we must have a memory reference $ref(M_{i,2}) \in \{ref(M_{1,2}), \ldots, ref(M_{i,2}), \ldots, ref(M_{K_2,2})\}$, that was an *L1 cache hit* in the absence of preemption, but it may access L2 cache after preemption. Note that L1 cache is always accessed. Therefore, in the absence of preemption, $ref(M_{i,2})$ can be an *L1 cache hit* only if the following condition holds:

— $M_{i,2}$ is accessed in the path $\mathcal{P} := ref(M_{1,1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i-1,1}) \rightsquigarrow ref(M_{i,1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{K_2,1}) \rightsquigarrow ref(M) \rightsquigarrow \ldots \rightsquigarrow ref(M_{1,2}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i-1,2})$. This scenario is illustrated in Figure 8(c). If $M_{i,2}$ is not accessed in path $\mathcal{P}$, $M_{i,2}$ cannot exist in L1 cache after $\mathcal{P}_1$ is executed. This is because $K_2 \geq K_1$ and K_2 unique memory blocks $M_{1,1}, \ldots, M_{K_2,1}$ are also mapped to the same L1 cache set as M (since $S_1 \leq S_2$). Therefore, M will be evicted from L1 cache by the set of memory blocks $M_{1,1}, \ldots, M_{K_2,1}$ after $\mathcal{P}_1$ is executed.

If the preceding condition holds, then $M_{i,2}$ is already reloaded after preemption and before the memory reference $ref(M_{i,2})$. As a result, $ref(M_{i,2})$ is an *L1 cache hit* even after preemption. If $M_{i,2}$ is not reloaded before the memory reference $ref(M_{i,2})$, then $ref(M_{i,2})$ will be an *L1 cache miss* even in the absence of preemption. Therefore, we reach a contradiction with our assumptions.

$\boxed{S_1 \leq S_2 \wedge K_1 > K_2}$. In the preceding construction of $\mathcal{P}$, $M_{i,2}$ may not be evicted from L1 cache if $K_1 > K_2$ holds. Therefore, we first construct $K_1 - K_2$ program paths $\mathcal{P}_1, \ldots, \mathcal{P}_{K_1-K_2}$ – all of which lead to memory reference $ref(M)$ as follows.

— $\mathcal{P}_1 := ref(M_{1,1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i-1,1}) \rightsquigarrow ref(M_{i,1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{K_2,1}) \rightsquigarrow ref(M)$
— $\mathcal{P}_2 := ref(M_{1,2}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i-1,2}) \rightsquigarrow ref(M_{i,2}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{K_2,2}) \rightsquigarrow ref(M)$
— ...
— $\mathcal{P}_{K_1-K_2} := ref(M_{1,K_1-K_2}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i-1,K_1-K_2}) \rightsquigarrow ref(M_{i,K_1-K_2}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{K_2,K_1-K_2}) \rightsquigarrow ref(M)$

where $\{ref(M_{i,1}), \ldots, ref(M_{i,K_1-K_2})\}$ is the set of memory references that were *L1 cache hits* in the absence of preemption but may access L2 cache after preemption. Let us construct another path, say, $\mathcal{P}_{K_1-K_2+1} := ref(M_{1,K_1-K_2+1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i,K_1-K_2+1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{K_2,K_1-K_2+1}) \rightsquigarrow ref(M)$, where $ref(M_{i,K_1-K_2+1})$ accesses L2 cache *only after preemption*, but not in the absence of preemption. After the execution of $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_{K_1-K_2}$, at least K_1 unique memory-block accesses map to the same L1 cache set as M. Therefore, M_{i,K_1-K_2+1} must have been accessed along the path $\mathcal{P}' := \mathcal{P}_1 \rightsquigarrow \mathcal{P}_2 \rightsquigarrow \ldots \rightsquigarrow \mathcal{P}_{K_1-K_2} \rightsquigarrow \ldots \rightsquigarrow ref(M_{i-1,K_1-K_2+1})$. In this case, $ref(M_{i,K_1-K_2+1})$ must be an *L1 cache hit* after preemption. If M_{i,K_1-K_2+1} is not accessed along the path $\mathcal{P}'$, memory block M_{i,K_1-K_2+1} is not cached in L1 while executing the reference $ref(M_{i,K_1-K_2+1})$. Therefore, $ref(M_{i,K_1-K_2+1})$ will be an *L1 cache miss* even in the absence of preemption. In both the cases, we reach contradictions.

$\boxed{S_1 > S_2}$. Assume that $B = (\frac{S_1}{S_2})K_1 - 1$. We construct $B + 1$ program paths all leading to the memory reference $ref(M)$ as follows:

— $\mathcal{P}_1 := ref(M_{1,1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i-1,1}) \rightsquigarrow ref(M_{i,1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{K_2,1}) \rightsquigarrow ref(M)$
— $\mathcal{P}_2 := ref(M_{1,2}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i-1,2}) \rightsquigarrow ref(M_{i,2}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{K_2,2}) \rightsquigarrow ref(M)$
— ...
— $\mathcal{P}_B := ref(M_{1,B}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i-1,B}) \rightsquigarrow ref(M_{i,B}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{K_2,B}) \rightsquigarrow ref(M)$

In the preceding paragraph, $\{ref(M_{i,1}), ref(M_{i,2}), \ldots, ref(M_{i,B})\}$ is the set of memory references that were *L1 cache hits* in the absence of preemption but may access L2 cache after preemption. Suppose we want to construct another program path $\mathcal{P}_{B+1} := ref(M_{1,B+1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{i,B+1}) \rightsquigarrow \ldots \rightsquigarrow ref(M_{K_2,B+1}) \rightsquigarrow ref(M)$, where $ref(M_{i,B+1})$ accesses L2 cache *only after preemption*, but not in the absence of preemption. After $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_B$ are executed, there are at least $(\frac{S_1}{S_2})K_1$ unique memory-block accesses in L1 cache ($\{M_{i,1}, \ldots, M_{i,B}\} \cup \{M\}$) mapping to the same L2 cache set as memory block M. According to Property 5.2, there could be at most $(\frac{S_1}{S_2})K_1$ blocks in L1 cache that may map to the same L2 cache set as M. Therefore, $M_{i,B+1}$ must have been accessed along the path $\mathcal{P}_1 \rightsquigarrow \mathcal{P}_2 \rightsquigarrow \ldots \rightsquigarrow \mathcal{P}_B \rightsquigarrow \ldots \rightsquigarrow ref(M_{i-1,B+1})$. In this case, $ref(M_{i,B+1})$ will be an L1 cache hit even after preemption. If $M_{i,B+1}$ is not accessed along the path $\mathcal{P}_1 \rightsquigarrow \mathcal{P}_2 \rightsquigarrow \ldots \rightsquigarrow \mathcal{P}_B \rightsquigarrow \ldots \rightsquigarrow ref(M_{i-1,B+1})$, then $M_{i,B+1}$ must have been evicted from L1 cache (by the set of memory blocks $\{M_{i,1}, \ldots, M_{i,B}\} \cup \{M\}$) before executing $ref(M_{i,B+1})$. As a result, $ref(M_{i,B+1})$ was an L1 cache miss even in the absence of preemption. Therefore, we reach a contradiction with our assumptions. □

6. EXPERIMENTAL EVALUATION

Experimental setup. We have chosen medium to large size benchmarks from [Gustafsson et al. 2010], which are generally used to validate timing analysis. The code size of different benchmarks

ranges from 2779 bytes (`bsort100`) to 118351 bytes (`nsichneu`), with an average code size of 18500 bytes. Throughout our evaluation, we shall assume that each task has been statically mapped to a particular core and all the tasks have fixed static priorities. We compile each benchmark into Simplescalar PISA (Portable Instruction Set Architecture) [Austin et al. 2002] — a MIPS like instruction set architecture. The control flow graph (CFG) of each benchmark is extracted from its PISA compliant binary and is used for all the analysis results reported here.

We choose `cnt` and `compress` from [Gustafsson et al. 2010] to generate different amounts of *inter-task* cache conflicts. `cnt` (which is a small program having a code size of 2880 bytes) is used to generate low inter-task cache conflict, whereas, `compress` (which is a relatively large program having a code size of 13411 bytes) is used to generate relatively high inter-task cache conflict. We also conduct experiments for private as well as shared L2 caches. The default micro-architectural setup is captured by Figure 9(a) when each core has a private L2 cache and by Figure 9(b) when L2 cache is shared among multiple cores. For the experiments featuring a *shared L2 cache*, we use `qurt` (code size 4898 bytes) and `statemate` (code size 52618 bytes) from [Gustafsson et al. 2010] to generate low and high inter-core cache interferences, respectively.

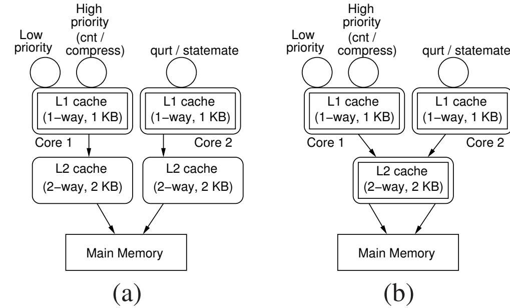

Fig. 9. We use either `cnt` or `compress` [Gustafsson et al. 2010] to generate inter-task cache conflicts. (a) Default architecture used for the results reported as "preemption + no L2 cache sharing". (b) Default architecture used for the results using shared caches. Either `qurt` or `statemate` [Gustafsson et al. 2010] is used to generate inter-core cache conflicts.

To validate our analysis method, the Simplescalar toolset [Austin et al. 2002] was extended to support the simulation of shared L2 caches. The original Simplescalar toolset supports *cycle accurate* simulation in the presence of L1 and L2 caches. However, the Simplescalar toolkit does not support the simulation of shared caches in the presence of multiple cores. Such a Simplescalar extension was developed in our prior work [Chattopadhyay et al. 2010; Chattopadhyay et al. 2012]. The extended Simplescalar framework is also *cycle accurate*. The key to such an extension is to modify the main simulation loop for multiple cores. Each iteration of the main simulation loop updates the execution state on each core – capturing the change in execution states for each cycle on each core. As a result, the state of the shared cache is also updated appropriately for each cycle. Currently, the simulation infrastructure is limited to the simulation of homogeneous processor cores, meaning that each processor core runs at the same frequency.

As part of our work in this paper, we have extended the multi-core Simplescalar developed in our prior work [Chattopadhyay et al. 2010; Chattopadhyay et al. 2012] to capture the effect of preemptions. Inside the simulator, we have implemented features using which a task can be preempted by a higher priority task and after the higher priority task finishes execution, the preempted task will resume. Before the preemption takes place, the pipeline state of the preempted task is flushed. This is acceptable, as we just want to measure the number of additional cache misses due to preemptions. Such a measurement from simulation will help to evaluate the *precision* of our CRPD analysis framework. However, since the preemption point is unknown, the search space for measuring the worst-case preemption delay is huge. Therefore, the observed CRPD in our experiments may highly underestimate the actual worst-case CRPD.

Our default system configuration uses a *direct-mapped*, 1 KB L1 cache and a 2-way associative, 2 KB L2 cache, both having 32 bytes cache block size. L1 cache miss penalty is 6 cycles and L2 cache miss penalty is 30 cycles.

We report the analysis overestimation ratio for the following evaluations. Overestimation ratio compares the analysis result (using our CRPD analysis framework) with the results observed from real execution (using our modified simulation infrastructure). To compare the overestimation solely

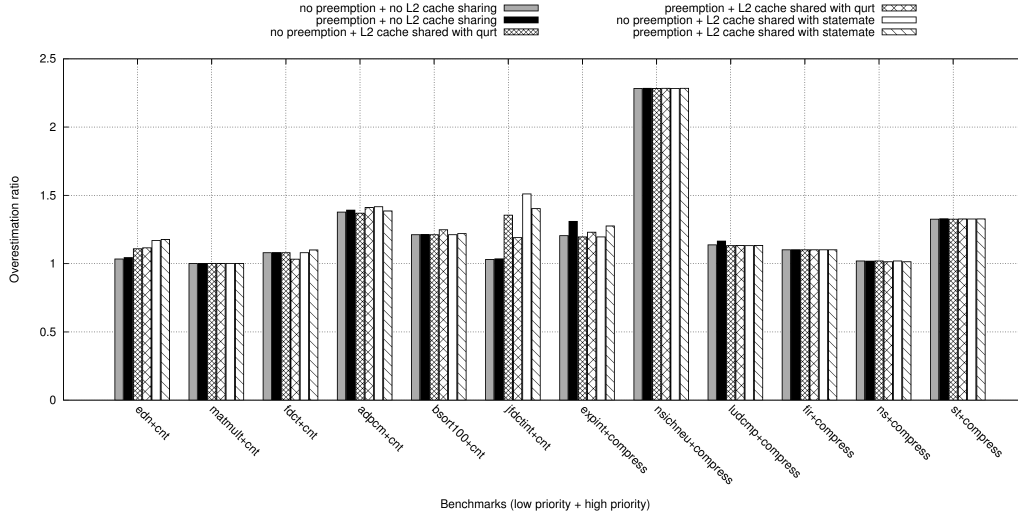

Fig. 10. $WCET + \#p \cdot CRPD$ overestimation for the set of tasks used from [Gustafsson et al. 2010]. Along the x-axis, $A + B$ denotes the scenario when task A is preempted by task B (where applicable)

due to the CRPD analysis, we record both the WCET overestimation and the overestimation of the quantity $WCET + \#p \cdot CRPD$, where $\#p$ captures the number of preemptions. $\#p$ is chosen in a fashion so that the value of $WCET$ and the value of $\#p \cdot CRPD$ are comparable. In the absence of preemption, a low priority task will not be interrupted by any higher priority task. Therefore, we plot the WCET overestimation ratio in the absence of preemption. If preemption is enabled, a low priority task can be preempted by a high priority task. Therefore, we record the overestimation of $WCET + \#p \cdot CRPD$ for the low priority task. The estimation is taken using our CRPD analysis framework and Chronos WCET analysis tool [Li et al. 2007]. To measure the quantity $WCET + \#p \cdot CRPD$ for a task T, we run T for a few inputs. We also allow a high priority task to preempt T exactly $\#p$ times. We observe the maximum execution time of T over different inputs and we record the observed value as a measurement of the quantity $WCET + \#p \cdot CRPD$.

Accuracy of our analysis. The *soundness* of our CRPD analysis is guaranteed only when used in conjunction with the WCET analysis. Therefore, only the sum of WCET and CRPD can be compared with the measurement. Figure 10 shows the combined WCET and CRPD overestimation ratio in the presence of different benchmarks from [Gustafsson et al. 2010]. Note that the rightmost four bars in Figure 10 uses the dual-core setup, as shown in Figure 9(b). Similarly, the leftmost two bars in Figure 10 uses the setup shown in Figure 9(a). Figure 10 clearly shows that our analysis computes *precise* estimates in most of the cases. Benchmark `nsichneu` is an exception. `nsichneu` is a benchmark with over two hundred branch instructions and many *infeasible paths*. Therefore, the overestimation largely arises from the *path analysis* during the WCET computation. This can be illustrated by results labeled “no preemption” in Figure 10.

Analysis result sensitivity w.r.t L1 and L2 caches. Figure 11(a) shows our analysis result sensitivity with respect to different L1 cache sizes and configurations. Similarly, Figure 11(b) shows the analysis result sensitivity with respect to different L2 cache sizes and configurations. Increasing the size of L1 cache usually increases the number of useful cache blocks, as a bigger L1 cache can hold more cache blocks to be reused later. Consequently, CRPD may increase with bigger L1 caches, as more cache blocks can be replaced by a preempting task. However, increasing the associativity usually decreases the CRPD. This is expected, as high cache associativity could decrease cache conflict misses. In a similar fashion, increasing the size of L2 cache usually increases the CRPD due

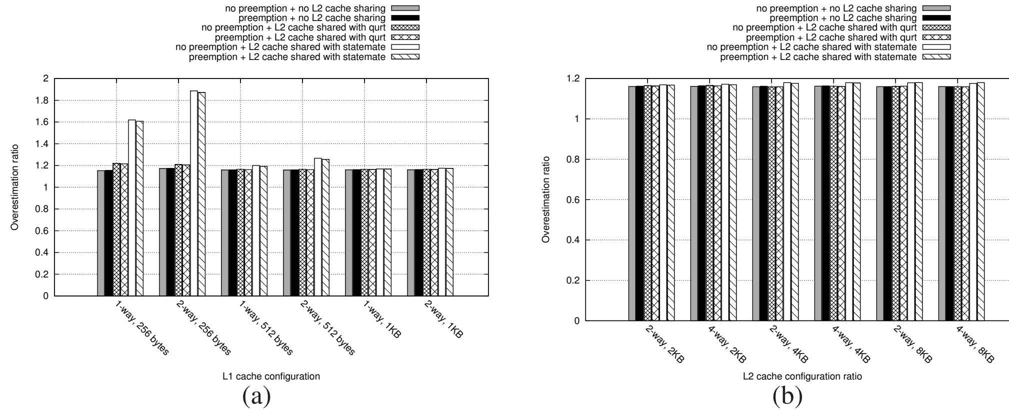

Fig. 11. CRPD and WCET analysis sensitivity with respect to (a) L1 cache configuration and (b) L2 cache configuration

to the replacement of more useful cache blocks. However, beyond a certain size of L2 cache, many useful cache blocks are not replaced due to the reduced cache interference. As a result, CRPD also decreases. Figures 11(a)-(b) show that our analysis is precise except for very small L1 caches in multi-cores (*e.g.* 256 bytes). This is primarily due to the difficulty in analyzing the inter-core cache conflicts. Note that the overestimation is high even in the absence of preemption (cf. Figure 11(a)).

Effect of cache sharing. It is worthwhile to mention that the *measured CRPD* (using our simulation infrastructure) can be *negative* in the presence of shared caches. Since the lifetime of a task is shifted due to preemptions, inter-core interferences may reduce after preemptions. As a result, preempting a low priority task may reduce the number of shared L2 cache misses in the preempted task. This leads to a *negative* CRPD value. In our measurements, we indeed found such scenarios. However, to consider such scenarios in our analysis, we may need to model an unbounded number of thread interleaving patterns. Since our analysis conservatively models all possible thread interleavings, the CRPD computed by our analysis is always *positive*.

Indirect effect of preemption. We have separately measured the cache reload latency due to the indirect effect of preemption (as computed by Equation (18)). Moreover, we have analyzed this effect for all the three different cases reported in Theorem 5.3 (*i.e.* for $S_1 \leq S_2 \wedge K_1 \leq K_2$, $S_1 \leq S_2 \wedge K_1 > K_2$ and $S_1 > S_2$). In general, due to the structure of programs, the additional cache reload latency resulting from the indirect effect is *minimal*. In the worst case (over all the used benchmarks from [Gustafsson et al. 2010]), the indirect effect of preemption is around 8% of the total CRPD cost computed by our analysis.

Case studies - `papabench` and `Debie`. We have also evaluated our framework on two freely available embedded software - `papabench` [Nemer et al. 2006], a derivation from the unmanned aerial vehicle (UAV) controller `Paparazzi` and `DEBIE-I` DPU [Kuitunen et al. 2001], an in-situ space debris monitoring instrument developed by Space Systems Finland Ltd. The controller of `papabench` contains two modules, `fly_by_wire` and `autopilot`. `fly_by_wire` module is responsible for managing radio-command orders, whereas the *autopilot* module runs the navigation and stabilization tasks of the aircraft. Table II(a) and Table II(b) describe the set of tasks used from `papabench` and `DEBIE-I` DPU, respectively. We evaluate our framework for different preemption scenarios that may appear in some executions of `papabench` and `DEBIE-I` DPU software. For the following experiments, we choose a 4-way, 16 KB L1 cache and an 8-way, 128 KB L2 cache. The configuration of caches are typical for embedded processors, such as the Freescale i.MX31 processor equipped with an ARM1136 CPU [Freescale 2008]. Figure 12(a) shows the combined

Table II. (a) `Papabench` task set used for evaluation, (b) `Debie` task set used for evaluation

Task	Description	code size (bytes)
T1	navigation task	20240
T2	stabilisation task	2744
T3	SPI serial link control 1	1840
T4	GPS control	4048
T5	fly_by_wire servo control	1696
T6	radio control	5520
T7	SPI serial link control 2	992

(a)

Task	Description	code size (bytes)
T1	telecommand	23288
T2	health monitoring 1	448
T3	Sensor unit (SU) interface task 1	6512
T4	Sensor unit (SU) interface task 2	4392
T5	Sensor unit (SU) interface task 3	1320
T6	health monitoring 2	16992

(b)

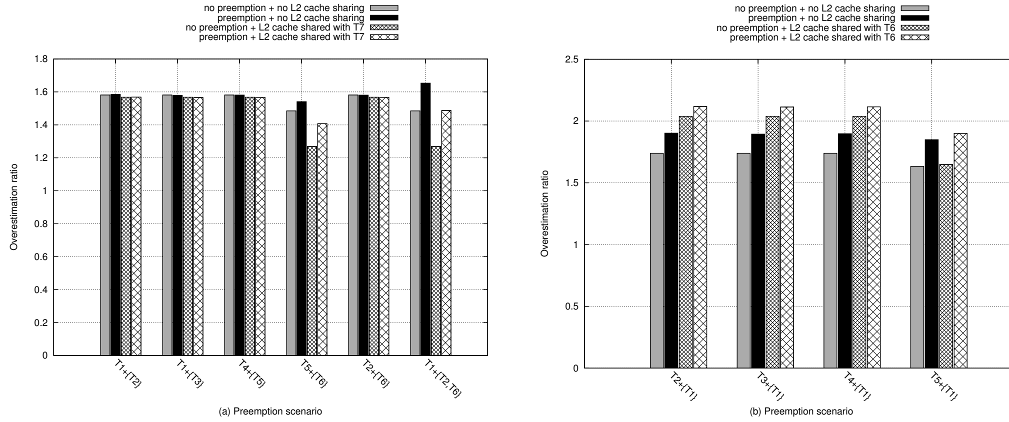

Fig. 12. $WCET + \#p \cdot CRPD$ overestimation for (a) `papabench` and (b) `DEBIE-I` DPU. Along the x-axis, $A + \{B\}$ denotes the scenario when task A is preempted by the set of tasks in B (where applicable)

WCET and CRPD overestimation for different preemption scenarios in `papabench`. On average, our framework generates around 55% overestimation. For nested preemptions (*e.g.* preemption of $T1$ using $T2$ and $T6$ as shown in Figure 12), evicting cache blocks (ECB) are merged from all the high priority tasks (*e.g.* evicting cache blocks from $T2$ and $T6$). We observe that the overestimation solely due to our CRPD analysis (*i.e.* comparing results labeled with "preemption" and "no preemption") is bounded by 10% on average. Figure 12(b) shows the results obtained for `DEBIE-I` DPU software. Unfortunately, concrete inputs for `DEBIE-I` DPU software is not available in public domain. Since simulation requires concrete input values, we cannot run simulation for the set of tasks in `DEBIE-I` DPU software (*i.e.* the set of tasks shown in Table II(b)). Therefore, in Figure 12(b), we present the combined WCET and CRPD overestimation solely due to the presence of L2 caches. The *baseline* in Figure 12(b) captures WCET and CRPD values when all L1 cache misses are treated as *L2 cache hits*. A fraction of such L1 cache misses can also be L2 cache misses and such L2 cache misses are computed by enabling the analysis of L2 caches. We primarily observe that the timing is not affected more than 20% due to L2 cache misses induced by preemptions.

Analysis time. We have performed all experiments on an 8-core, Intel Xeon machine with a 4 GB of RAM and running Fedora core 4 operating system. Our analysis finishes within a few seconds for most of the experiments. The maximum time taken by our framework is 5 minutes. This was observed when we analyzed tasks from `DEBIE-I` DPU, a real-life embedded software.

7. CONCLUSION

In this paper, we have proposed a CRPD analysis framework in the presence of multi-level, non-inclusive caches. We have shown that the presence of *non-inclusive* caches poses several new challenges in estimating the CRPD. These new challenges arise due to the *indirect effect of preemption*.

We have proved theoretical bounds on the indirect effect of preemption. Based on these theoretical bounds, we have proposed a CRPD analysis framework in the presence of multi-level, non-inclusive caches. We have performed experiments to evaluate our CRPD analysis framework on standard WCET benchmarks as well as an unmanned aerial vehicle (UAV) controller and a space debris monitoring software. Our observations suggest that we can provide *precise* estimates for most of the cases. In future, we plan to extend our CRPD analysis framework for data caches and multi-level, inclusive caches.

Acknowledgements

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

REFERENCES

ALTMEYER, S. AND BURGUIERE, C. 2009. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *ECRTS*.

ALTMEYER, S., MAIZA, C., AND REINEKE, J. 2010. Resilience analysis: tightening the CRPD bound for set-associative caches. In *LCTES*.

AUSTIN, T., LARSON, E., AND ERNST, D. 2002. Simplescalar: An infrastructure for computer system modeling. *Computer 35,* 2.

BALAKRISHNAN, G. AND REPS, T. W. 2004. Analyzing memory accesses in x86 executables. In *CC*.

CHATTOPADHYAY, S., CHONG, L. K., ROYCHOUDHURY, A., KELTER, T., MARWEDEL, P., AND FALK, H. 2012. A unified wcet analysis framework for multi-core platforms. In *RTAS*.

CHATTOPADHYAY, S., ROYCHOUDHURY, A., AND MITRA, T. 2010. Modeling shared cache and bus in multi-cores for timing analysis. In *SCOPES*.

FREESCALE. 2008. i.MX31 Applications Processor. `http://www.freescale.com/files/32bit/doc/data_sheet/MCIMX31.pdf`.

GUSTAFSSON, J., BETTS, A., ERMEDAHL, A., AND LISPER, B. 2010. The Mälardalen WCET benchmarks – past, present and future. In *WCET*.

HARDY, D., PIQUET, T., AND PUAUT, I. 2009. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *RTSS*.

HARDY, D. AND PUAUT, I. 2008. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS*.

HUYNH, B. K., JU, L., AND ROYCHOUDHURY, A. 2011. Scope-aware data cache analysis for WCET estimation. In *RTAS*.

KUITUNEN, J., DROLSHAGEN, G., MCDONNELL, J., SVEDHEM, H., LEESE, M., MANNERMAA, H., KAIPIAINEN, M., AND SIPINEN, V. 2001. DEBIE-first standard in-situ debris monitoring instrument. *EUROPEAN SPACE AGENCY-PUBLICATIONS-ESA SP 473*.

LEE, C.-G., HAHN, J., SEO, Y.-M., MIN, S. L., HA, R., HONG, S., PARK, C. Y., LEE, M., AND KIM, C. S. 1998. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput. 47,* 6.

LI, X., LIANG, Y., MITRA, T., AND ROYCHOUDHURY, A. 2007. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*. `http://www.comp.nus.edu.sg/~rpembed/chronos`.

LI, Y., SUHENDRA, V., LIANG, Y., MITRA, T., AND ROYCHOUDHURY, A. 2009. Timing analysis of concurrent programs running on shared cache multi-cores. In *RTSS*.

NEGI, H. S., MITRA, T., AND ROYCHOUDHURY, A. 2003. Accurate estimation of cache-related preemption delay. In *CODES+ISSS*.

NEMER, F., CASSÉ, H., SAINRAT, P., BAHSOUN, J., AND DE MICHIEL, M. 2006. Papabench: a free real-time benchmark. In *WCET Workshop*.

STASCHULAT, J. AND ERNST, R. 2004. Multiple process execution in cache related preemption delay analysis. In *EMSOFT*.

TAN, Y. AND MOONEY, V. 2004. Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems. In *SCOPES*.

THEILING, H., FERDINAND, C., AND WILHELM, R. 2000. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems 18,* 3.

TOMIYAMA, H. AND DUTT, N. D. 2000. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *CODES*.

YAN, J. AND ZHANG, W. 2008. WCET analysis for multi-core processors with shared l2 instruction caches. In *RTAS*.

ZAHRAN, M. M., ALBAYRAKTAROGLU, K., AND FRANKLIN, M. 2007. Non-inclusion property in multi-level caches revisited. *I. J. Comput. Appl. 14,* 2.