Cache-Related Preemption Delay Analysis for Multi-Level Inclusive Caches

Zhenkai Zhang
Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN, USA
zhenkai.zhang@vanderbilt.edu

Xenofon Koutsoukos
Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN, USA
xenofon.koutsoukos@vanderbilt.edu

ABSTRACT

Cache-related preemption delay (CRPD) analysis is crucial when designing embedded control systems that employ preemptive scheduling. CRPD analysis for single-level caches has been studied extensively based on useful cache blocks (UCBs). As high-performance embedded processors are increasingly used, which are often equipped with multi-level caches, CRPD analysis for cache hierarchies also needs to be investigated. Recently, an approach has been proposed to estimate CRPD for multi-level non-inclusive caches. Since multi-level inclusive caches are also commonly used, especially in some multi-core processors, it becomes important to study how to analyze CRPD for inclusive cache hierarchies. However, as shown in this paper, new challenges appear due to the strict inclusion enforcement in the multi-level inclusive caches, which make the traditional UCB concept hard to use. In this paper, we propose a new concept of useful positive references (UPRs) to replace the UCB concept. Based on UPRs, we propose an approach to bound the additional cache misses due to a preemption in a two-level inclusive cache hierarchy. We present theoretical analysis to show the approach is safe, and we evaluate the proposed approach on a set of benchmarks to demonstrate its effectiveness. To the best of our knowledge, this is the first attempt to analyze CRPD for multi-level inclusive caches.

CCS Concepts

•Computer systems organization → Embedded systems; Real-time systems;

Keywords

CRPD Analysis; Timing Analysis; Cache Analysis

1. INTRODUCTION

Worst-case execution time (WCET) estimation is required by schedulability analysis when designing embedded control systems. Usually, WCET estimation is performed under the assumption that tasks being analyzed are non-preemptive. However, many embedded control systems use preemptive scheduling strategies, in which, the execution of a task can be frequently interrupted by other tasks with higher priorities. When such an interruption occurs, the states of many underlying micro-architectural components (such as caches, pipelines, and branch predictors) are often changed. Therefore, after the preempted task resumes, there will be some additional overhead on its execution time due to the "polluted" hardware states. Since caches usually have the most significant impact on the variation of execution time, most of the overhead is related to cache state changes, which is often referred to as cache-related preemption delay (CRPD).

The importance of CRPD in schedulability analysis has been shown in [2, 11, 17]. Over the past two decades, CRPD analysis for single-level caches has been studied extensively [14, 18, 3]. However, as mentioned in [7], CRPD analysis for multi-level caches becomes much harder since the amount of intra-task interference can vary at lower cache levels after a preemption. Consequently, the existing analysis methods for single-level caches cannot be directly used for multi-level caches. As embedded processors are increasingly equipped with cache hierarchies, CRPD analysis for multi-level caches becomes a significant problem.

There are three cache hierarchy types: inclusive, exclusive, and non-inclusive. Multi-level inclusive caches require that the contents at upper cache levels must be a subset of the contents at lower levels. Multi-level exclusive caches require that the contents at a cache level should not be duplicated at any other cache level. Multi-level non-inclusive caches allow duplicated contents existing at any cache level, but they do not strictly enforce the inclusion property.

In [7], CRPD analysis for multi-level non-inclusive caches has been investigated. Since cache hierarchies of different types may have different behaviors even for the same memory reference sequence, CRPD analysis for multi-level non-inclusive caches may be not fit for multi-level inclusive caches. Therefore, in this paper, we study the distinctions between non-inclusive and inclusive cache hierarchies in the context of CRPD analysis, and we propose an approach which can safely analyze CRPD for multi-level inclusive caches. To the best of our knowledge, this is the first time to analyze CRPD in terms of multi-level **inclusive** caches.

The main contributions of this paper are: (1) We identify the challenges of analyzing CRPD for multi-level inclusive caches, especially those that do not appear in multi-level non-inclusive caches; (2) We propose a new concept of useful positive references, and based on this new concept we pro-

EMSOFT'16, October 01-07, 2016, Pittsburgh, PA, USA

DOI: http://dx.doi.org/10.1145/2968478.2968481

pose an approach to analyze CRPD for a two-level inclusive cache; (3) We prove the proposed approach is safe, namely it can conservatively bound the additional cache misses due to a preemption; (4) We evaluate the proposed approach on a set of benchmarks showing the effectiveness of the method.

The rest of this paper is organized as: Section 2 describes the system model under consideration; Section 3 briefly sets the background; Section 4 states why CRPD analysis for multi-level inclusive caches is a hard problem; Section 5 presents the proposed approach to CRPD analysis for multi-level inclusive caches; Section 6 evaluates the proposed approach; Section 7 gives the related work; and Section 8 concludes this paper and states some future work.

2. SYSTEM MODEL & ASSUMPTIONS

Similar to [7], in this paper, we also focus on a two-level cache hierarchy (i.e. L1 and L2) and consider only instruction references. Different from [7], the second cache level of our model maintains the strict inclusion property. Although data references are not considered, this work can serve as a basis for any future work on CRPD analysis for multi-level inclusive data/unified caches.

In our system model, we assume L1 and L2 caches are both set associative, and they use LRU (Least Recently Used) replacement policy. The size of a L1 cache block can be smaller than or equal to the size of a L2 cache block.

Since inclusive cache hierarchies are often used in multi-core architectures [4], the inclusive L2 cache is usually shared by multiple processor cores. In this work, we only focus on a single processor core without considering inter-core interferences. We also assume the time to access L2 is bounded and predictable, which can be achieved by using some deterministic interconnect to connect the caches, like TDMA buses [12]. For example, Fig. 1 shows a model of interest focusing on the first core and the two cache levels that can be affected by this core.

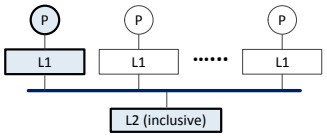

Figure 1: An example of the system model

We assume the estimated CRPD is used together with the estimated WCET for schedulability analysis, which is almost every use case of CRPD in practice. Thus, we only need to guarantee that the combination of the estimated WCET and CRPD over-approximates the overall task execution time under preemptions [1]. In addition, we assume there are no timing anomalies [16]. Thus, an upper bound on additional L1 and L2 cache misses (which are those that do not appear in WCET estimation) can be used to estimate CRPD for a preemption.

3. BACKGROUND

Cache analysis for WCET estimation usually only takes into account intra-task interference to assign a cache hit/miss classification (CHMC) to each reference according to the abstract cache states (ACSs) derived by three different analyses [23]. At a program point, a *must* analysis determines a set of memory blocks that are *definitely* in the cache, so a reference to a block being in the set is classified as *always hit* (AH); a *may* analysis determines a set of memory blocks that are *possibly* in the cache, so a reference to a block not being in the set is classified as *always miss* (AM); a *persistence* analysis determines a set of memory blocks that stay in the cache once they are loaded, and a reference to such a block is classified as *persistent* (PS); and, if a reference cannot be classified as AH, AM, or PS, it is *non-classified* (NC). The analyses are usually performed on the control-flow graph (CFG) reconstructed from the low-level code of the program. In each of the analyses, an *update* function is defined to account for the effect of a memory reference on the ACS, and a *join* function is defined to safely combine ACSs at a merge point.

While L1 cache in a cache hierarchy is always accessed by each memory reference, the caches at lower levels may only be intermittently accessed. Accordingly, for each reference at a cache level, a cache access classification (CAC) is used to represent whether the cache at this level will be accessed: *always* (`A`) denotes the access will always occur (so the ACS at this level will always be updated by this reference); *never* (`N`) denotes the access will never happen (so the ACS at this level will never be changed by this reference); and *uncertain* (`U`) denotes the access may occur (so the ACS becomes the join of two possible cases corresponding to access occurring and not occurring) [9]. Note that if a reference always/never accesses a cache level in reality but its CAC at the level is `U` in an analysis, the analysis based on this CAC is still safe but may not be precise.

As described in [25], other than cache access filtering behavior, multi-level inclusive caches also have invalidation behavior. When a memory block is evicted from a lower inclusive cache level, all the contents belonging to this memory block need to be invalidated from its upper cache levels. The combination of these two behaviors makes multi-level inclusive cache analysis very challenging even only considering intra-task interference. Fortunately, there have been several approaches proposed to analyze inclusive cache hierarchies for WCET estimation [10, 26, 25]. In this paper, the approach proposed in [25] is employed, which can derive CAC and CHMC for each reference at each cache level more precisely compared to others.

WCET estimation usually depends on the upper bound on cache misses according to each reference's CHMC on a path in the absence of preemptions. However, when preemptions are possible, there will also be inter-task interference which can cause the derived CHMC not to hold so that the upper bound on cache misses may become unsafe. Therefore, in the presence of preemptions, CRPD analysis is needed to bound the number of additional cache misses caused by inter-task interference.

CRPD analysis for single-level caches relies on the concepts of useful cache blocks (UCBs) and evicting cache blocks (ECBs) [14, 24]. In the following, let us only focus on set associate caches with LRU replacement policy. At a program point of the preempted task, a memory block is a UCB if it *may* be cached at this point and it *may* be reused later at some reachable program point without being evicted along some path to its reuse. The set of ECBs consists of the memory blocks that *may* be accessed by the preempting tasks. Combing UCBs and ECBs can result in a tight CRPD analysis for single-level set-associative caches [3]. Note that if

the estimated CRPD will be used with the WCET estimate together, the UCB concept can be refined by requiring the memory block *must* be cached at the program point and along the path to its reuse [1].

At a program point, a UCB is only related to the *first* reachable references to the memory block after that program point. The number of UCBs at a program point can serve as an upper bound on the additional cache misses when a preemption happens at this program point. This upper bound is safe because: (1) a single-level cache is always accessed, so the amount of intra-task interference to a memory block will not be changed by the preemption; (2) after any *first* reachable reference to the memory block of a UCB beyond the preemption point, the memory block will become the youngest and its LRU age can only be affected by intra-task interference which is not changed as stated above. Therefore, in terms of single-level caches, the first reachable references to the memory block of a UCB beyond the preemption point can be treated as "firewalls" which prevent the inter-task interference from affecting the LRU age of the memory block afterwards. As a result, after the first reference to a memory block beyond the preemption point on a path, any further reference to this memory block would have the same CHMC as that in the absence of the preemption.

However, as stated in [7], only considering the effect of inter-task interference on the *first* reachable references to a memory block may not be safe in terms of multi-level caches (specifically, a two-level non-inclusive cache is studied in [7]). While L1 is always accessed, the other lower cache levels are usually not accessed by every memory reference. Compared to the case where there is no preemption, some references may need to access the lower cache levels in the presence of a preemption. Therefore, the amount of intra-task interference to a memory block at a lower level may be increased due to the preemption. Because of the possibility that the amount of intra-task interference to a memory block can change at a lower level, we cannot treat the *first* references to the memory block as "firewalls" to stop the effect of preemption on the memory block's LRU age afterwards at that level. This phenomenon is referred to as *the indirect effect of preemption* in [7].

4. PROBLEM FORMULATION

Similar to multi-level non-inclusive caches, multi-level inclusive caches can also suffer indirect preemption effects. To be specific, with respect to our model, the amount of intra-task interference at L2 may be increased after a preemption. However, this is not the only indirect preemption effect that makes CRPD analysis hard for multi-level inclusive caches. There can be two new indirect preemption effects induced by the invalidation behavior as described below.

4.1 Broken L1 "Firewalls"

As studied in [7], when analyzing CRPD for a two-level non-inclusive cache, for any reachable reference to a specific memory block after a preemption point, it is the *first* one on a path whose L1 cache behavior needs to be checked, and the other references to that block afterwards on the path will have the same L1 CHMC as what they were classified in the absence of the preemption. That is to say, we only expect L2 cache suffers the indirect preemption effect but not the L1 cache. In this case, L1 cache behaves like a single-level cache, so the *first* reachable reference to a memory block on a path after a preemption point acts like a L1 "firewall".

However, when analyzing CRPD for multi-level inclusive caches, due to the possible invalidation behavior, it may be unsafe to consider the other references after the *first* one to the same memory block will have the same L1 CHMC as what they were classified in the absence of the preemption. For example, a sequence of memory references is executed on

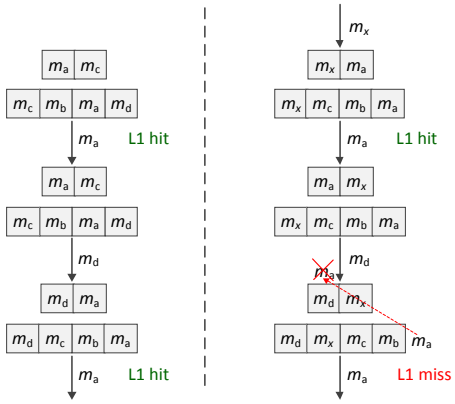

Figure 2: L1 "firewalls" are broken by invalidation behavior. Invalidations are shown as dotted lines with cross marks.

a two-level inclusive cache as shown in Fig. 2. For simplicity, we assume the L1 and L2 cache block sizes are the same, and the memory blocks m_a, m_b, m_c, m_d, and m_x are mapped to the same cache set at each of these two levels. The left part of the figure shows the states of the cache hierarchy when there is no preemption, and the right part shows the states in the presence of a preemption. From the left part of the figure, we can observe that the *second* reference to m_a is a L1 cache hit. However, when a preemption happens before the first reference to m_a and introduces another memory block m_x into the cache hierarchy, the *second* reference to m_a cannot be L1 cache hit anymore. This is because when m_d is referenced in the presence of the preemption, it evicts m_a from L2 cache, which invalidates m_a in L1 cache.

4.2 Reduced Intra-Task Interference

In the case of multi-level non-inclusive caches, we can deduce the following invariant: For any reference whose L1/L2 CHMC is AM, its L1/L2 CHMC is always AM no matter whether there is any preemption. The reason for this is because the amount of interference will never be reduced at L1/L2. Due to this invariant, a L1 AM reference in a loop remains L1 AM during all the iterations. Based on this, how much a L2 AH reference in a loop may contribute to the CRPD can be bounded in [7][1]. However, we find this bound may not be suitable for multi-level inclusive caches, since the invariant may not hold due to a possibly reduced amount of intra-task interference to a memory block at L1/L2 after a preemption.

It may seem impossible that the amount of intra-task interference can be reduced due to a preemption, but it actually can happen because of the invalidation behavior. If a memory block is invalidated at L1 due to the eviction of its

[1]Although not stated explicitly, we can find one implication in [7] is that the L1 and L2 cache block sizes should be the same. However, in this work, we do not impose this restriction.

super-block at L2 caused by a preemption, a "hole" will be left in the cache; until this "hole" is filled by some memory block, any access to the corresponding cache set will not increase the LRU ages of the memory blocks that are behind this "hole"; therefore, compared to that in the absence of the preemption, some memory blocks may live longer at L1 such that the subsequent references to them may not need to access L2. For example, consider the situation shown in Fig.

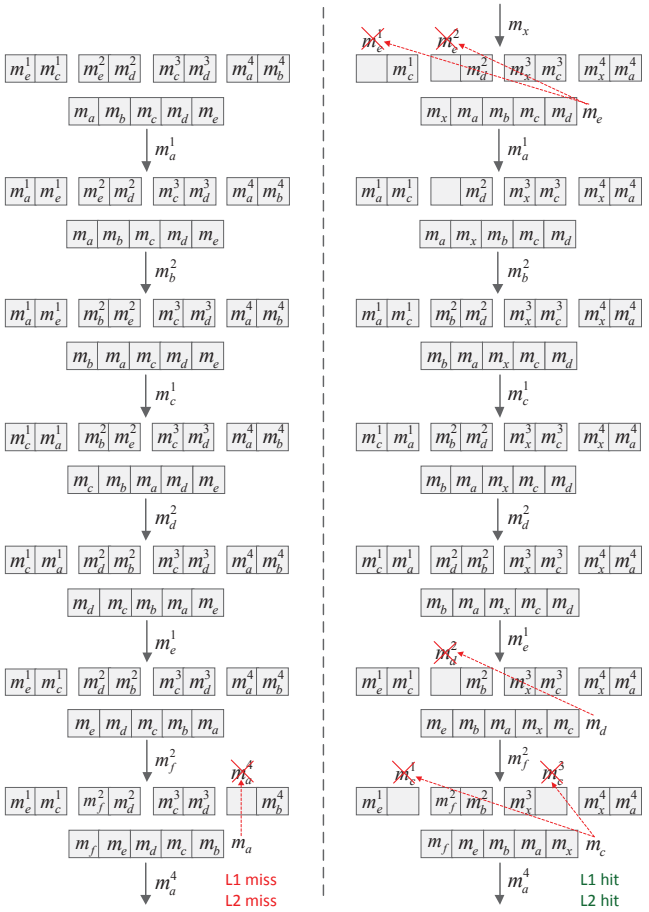

Figure 3: The amount of intra-task interference may be reduced due to a preemption. Invalidations are shown as dotted lines with cross marks.

3. In this example, we suppose L2 cache is fully associative and L1 cache is 2-way set associative with 4 cache sets. The cache block size at L1 is a quarter of the cache block size at L2. Since the cache block size at L1 is smaller than that at L2, given a memory block m in terms of L2 cache, we use m^i to denote the i^{th} sub-block of m in terms of L1 cache. Therefore, each sub-block of a L2 memory block is mapped to the corresponding L1 cache set (e.g., m_a^1 is mapped to the first L1 cache set and so forth). Let us also assume after a preemption one extra memory block m_x is introduced into the cache hierarchy. As shown in the first state on the right, the introduction of m_x evicts m_e from L2 cache, which invalidates both m_e^1 and m_e^2 in L1 cache. Thus, two "holes" are left in the first and second cache sets of L1 cache. Compared with the counterparts on the left, we can observe that references to m_a^1 and then m_b^2 do not evict m_c^1 and m_d^2 because of the "holes"; and, the following references to m_c^1 and then m_d^2 do not need to access L2, since the needed sub-blocks can be found in L1 cache. As a consequence, after references to m_e^1 and m_f^2, m_a is not evicted from L2 cache in the presence of the preemption, but m_a is evicted from L2 cache in the absence of the preemption. In other words, the introduced m_x causes less amount of intra-task interference to m_a in L2 cache. The result of this phenomenon is: Even if the last reference to m_a^4 is classified as L1 AM and L2 AM in the absence of the preemption, the reference can surprisingly hit both caches in the presence of the preemption.

Therefore, CRPD analysis for multi-level inclusive caches becomes a very challenging problem due to all these possible indirect preemption effects. In the next section, we propose an approach that can derive a safe CRPD estimate with respect to inclusive cache hierarchies.

5. CRPD ANALYSIS

As observed in Section 4, when analyzing CRPD for multi-level inclusive caches, it becomes really difficult to rely on the traditional UCB concept to capture the potential CRPD contributors. Taking that in consideration, we devise a new concept of useful positive references (UPRs) to replace UCBs as the basis of our work. The new concept intends to capture which references beyond a program point may be influenced by a preemption directly or indirectly such that they may contribute more to the execution time. As mentioned above, we only focus on how to make the combination of WCET and CRPD sound, so the references of interest should be "positively" classified by the WCET analysis, and these references can be categorized into 7 types as shown in Tab. 1. In the following, we call the references of these types as *positive references* (i.e. their referenced memory blocks are definitely/persistently in L2 cache when the references occur). Note that due to the inclusion property, there are no such CHMC pairs like "L1 is AH but L2 is not AH" and "L1 is PS but L2 is not AH nor PS".

Table 1: Seven types of positively classified references

	Type 1	Type 2	Type 3	Type 4	Type 5	Type 6	Type 7
L1	AH	AM	NC	PS	PS	AM	NC
L2	AH	AH	AH	AH	PS	PS	PS

Note: PS classification implies the reference is located in a loop.

At a program point, if a reachable positive reference can lead to additional cache misses after a preemption, we call this reference a useful positive reference (UPR) at that program point. Specifically, there are two UPR types: if a UPR can contribute only one additional L1 cache miss, we call it an L1-UPR; and if a UPR can contribute either one additional L1 cache miss with one additional L2 cache miss or just one additional L2 cache miss, we call it an L2-UPR. Note that if a positive reference is inside a loop, it may occur several times; even it is considered as a UPR at some point, it may not always contribute additional cache misses for all iterations. Thus, when we say a positive reference is a UPR, we actually mean this positive reference can act as a UPR at least once.

Clearly, at a program point, if we can conservatively derive a set of reachable positive references that can be considered as UPRs and bound the number of times each of them may act as a UPR, we can bound the CRPD for multi-level inclusive caches safely.

5.1 UPR Derivation

First, the approach proposed in [25] is used to analyze the inclusive cache hierarchy for WCET estimation and also we can obtain which references are positive references. Then, we propose a backward-flow analysis to conservatively derive a set of UPRs at each program point. In order to facilitate the presentation, given a reference r, we assume r has the following attributes, as shown in Tab. 2, among which $r.c_{L1}$, $r.c_{L2}$, and $r.a_{L2}$ are all set by the cache hierarchy analysis in the first step.

Table 2: Attributes of a reference r

Attribute	Description
$r.m_{L1}$	memory block accessed by r w.r.t. L1 cache block size
$r.m_{L2}$	memory block accessed by r w.r.t. L2 cache block size
$r.c_{L1}$	r's L1 cache hit/miss classification
$r.c_{L2}$	r's L2 cache hit/miss classification
$r.a_{L2}$	r's L2 cache access classification

Note: Since r's L1 CAC is always A, we do not show $r.a_{L1}$.

Before presenting the detailed analysis, let us discuss the following question: Suppose there is a positive reference $\dot{r}$ that is reachable from a program point p; how can we safely decide whether or not to consider $\dot{r}$ as a UPR at p?

Although $\dot{r}$ may be of either type shown in Tab. 1, when $\dot{r}$ is inside a loop with L1/L2 *PS*, we can treat *PS* as *AH* to focus only on the first three types, namely if $\dot{r}$ is of type 4 or 5, then $\dot{r}$ is treated as if it is of type 1; if $\dot{r}$ is of type 6, then $\dot{r}$ is treated as if it is of type 2; and if $\dot{r}$ is of type 7, then $\dot{r}$ is treated as if it is of type 3. This is because "*AH* implies *PS*", namely "not *PS* implies not *AH*", which means $\dot{r}$ is more likely considered UPR with L1/L2 *AH* than *PS*. Thus, we can always conservatively derive the set of UPRs at a program point.

In the following, let us use a notation $\pi_{x\rightsquigarrow y}$ to represent a control flow path from a program point x to a program point y. Since a reference is associated with a program point, we also allow x and/or y to represent references as long as there is no ambiguity.

Clearly, if there is a path $\pi_{p\rightsquigarrow\dot{r}}$ without any reference to $\dot{r}.m_{L2}$ (except for $\dot{r}$), $\dot{r}.m_{L2}$ must be always in L2 cache at p; otherwise, $\dot{r}$ cannot be classified as L2 *AH* by the cache hierarchy analysis in the first place. In this case, a preemption occurring at p can possibly evict $\dot{r}.m_{L2}$ out of L2 cache to make $\dot{r}$'s L2 *AH* not hold, so it means we should consider $\dot{r}$ as a UPR at p (to be specific, it is considered as an L2-UPR). Thus, the interesting scenario is that on every path $\pi_{p\rightsquigarrow\dot{r}}$, in addition to $\dot{r}$, there is at least one reference to $\dot{r}.m_{L2}$: Given such a path $\pi_{p\rightsquigarrow\dot{r}}$, let the sequence of references to $\dot{r}.m_{L2}$ be represented as $\langle\hat{r}_1,\cdots,\hat{r}_k,\dot{r}\rangle$ where $k\geq 1$.

In order to facilitate the discussion, let us define a few notations. Let u be a reference and v be a reference reachable from u. Given a path $\pi_{u\rightsquigarrow v}$, we use $\Theta(\pi_{u\rightsquigarrow v})$ to represent the set of references *in between* u and v on this path:

$$\Theta(\pi_{u\rightsquigarrow v})=\{w|w\in\pi_{u\rightsquigarrow v}\wedge w\neq u\wedge w\neq v\}$$

We also use $\Delta(\pi_{u\rightsquigarrow v})$ to represent the set of L2 memory blocks accessed by the references *in between* u and v on the path *that are mapped to the same L2 cache set as* $v.m_{L2}$:

$$\Delta(\pi_{u\rightsquigarrow v})=\bigcup_{w\in\Theta(\pi_{u\rightsquigarrow v})}\{w.m_{L2}|\sigma(w.m_{L2})=\sigma(v.m_{L2})\}$$

where $\sigma(m)$ gives the L2 cache set that the memory block m is mapped to. Based on these notations, let us define a predicate $\varphi(\pi_{u\rightsquigarrow v})$ on the path $\pi_{u\rightsquigarrow v}$ as:

$$\varphi(\pi_{u\rightsquigarrow v}):\forall m\in\Delta(\pi_{u\rightsquigarrow v}),\exists w\in\Theta(\pi_{u\rightsquigarrow v}),$$
$$w.m_{L2}=m\wedge(w.a_{L2}=\texttt{A}\vee w.a_{L2}=\texttt{U})$$

namely if $\varphi(\pi_{u\rightsquigarrow v})$ is true, for any memory block in $\Delta(\pi_{u\rightsquigarrow v})$ there is at least one reference to this block on the path whose L2 CAC is A or U.

The following lemma states when we can safely treat $\dot{r}$ as no additional L2 cache miss contribution.

LEMMA 1. *Given any path $\pi_{p\rightsquigarrow\dot{r}}$, if there are multiple references to $\dot{r}.m_{L2}$, i.e. $\langle\hat{r}_1,\cdots,\hat{r}_k,\dot{r}\rangle$ where $k\geq 1$, and the condition $\varphi(\pi_{\hat{r}_1\rightsquigarrow\hat{r}_2})\wedge\varphi(\pi_{\hat{r}_2\rightsquigarrow\hat{r}_3})\wedge\cdots\wedge\varphi(\pi_{\hat{r}_k\rightsquigarrow\dot{r}})$ also holds, $\dot{r}$ does not need to be considered as an L2-UPR at p.*

PROOF. For a path $\pi_{p\rightsquigarrow\dot{r}}$, we know $\dot{r}$ is a positive reference, but with respect to $\hat{r}_1,\cdots,\hat{r}_k$ there are two cases: (1) all of them are positive references; or (2) at least one of them is not a positive reference. If it is **the first case**, a preemption occurring at p may make any of them suffer an L2 cache miss (including $\dot{r}$). (I) Suppose the first one suffering an L2 cache miss after a preemption is $\hat{r}_i\in\langle\hat{r}_1,\cdots,\hat{r}_k\rangle$ where $1\leq i\leq k$. When there is no preemption, after $\hat{r}_i$ occurs, $\dot{r}.m_{L2}$'s LRU age is upper bounded by α. Let $\mathscr{A}_{L2}$ denote the associativity of L2 cache, so we have $1\leq\alpha\leq\mathscr{A}_{L2}$. Since $\varphi(\pi_{\hat{r}_i\rightsquigarrow\hat{r}_{i+1}})$ holds, each memory block in $\Delta(\pi_{\hat{r}_i\rightsquigarrow\hat{r}_{i+1}})$ has been considered as an interference to $\dot{r}.m_{L2}$ during the cache analysis (as described in Section 3 in terms of A and U CACs). Since $\hat{r}_{i+1}$ is also a positive reference when there is no preemption, we have $\beta=\alpha+|\Delta(\pi_{\hat{r}_i\rightsquigarrow\hat{r}_{i+1}})|\leq\mathscr{A}_{L2}$ where β is the upper bound on $\dot{r}.m_{L2}$'s LRU age when $\hat{r}_{i+1}$ occurs. Since $\hat{r}_i$ suffers an L2 cache miss in the presence of the preemption, $\dot{r}.m_{L2}$ becomes the youngest in the corresponding L2 cache set, i.e. its LRU age $\alpha'=1\leq\alpha$. Note that the amount of intra-task interference to $\dot{r}.m_{L2}$, denoted as ξ, on the path $\pi_{\hat{r}_i\rightsquigarrow\hat{r}_{i+1}}$ is at most $|\Delta(\pi_{\hat{r}_i\rightsquigarrow\hat{r}_{i+1}})|$ no matter whether there is a preemption. Therefore, when $\hat{r}_{i+1}$ occurs, the LRU age of $\dot{r}.m_{L2}$ is $\beta'=\alpha'+\xi\leq\alpha+|\Delta(\pi_{\hat{r}_i\rightsquigarrow\hat{r}_{i+1}})|=\beta$, namely $\dot{r}.m_{L2}$ is still in L2 cache. By mathematical induction, we can prove $\dot{r}$ still has an L2 hit, namely $\dot{r}$ does not contribute additional L2 cache misses. (II) It is also possible $\dot{r}$ is the first one in the sequence suffering an L2 cache miss after a preemption. However, since there is no reference to $\dot{r}.m_{L2}$ on $\pi_{p\rightsquigarrow\hat{r}_1}$, we know $\hat{r}_1$ is always considered as an L2-UPR at p. Therefore, it is still safe not to consider $\dot{r}$ as an L2-UPR at p, since $\hat{r}_1$'s compensation makes the amount of additional L2 cache misses not underestimated. On the other hand, if it is **the second case**, then we have a subsequence $\langle\hat{r}_j,\cdots,\hat{r}_k,\dot{r}\rangle$ where $1\leq j\leq k$ and $\hat{r}_j$ is the closest one to $\dot{r}$ that is not a positive reference (i.e. the rest of them are all positive). If $\hat{r}_j$ misses L2, $\dot{r}.m_{L2}$ will become the youngest in the cache set. Based on the discussion above, we know $\dot{r}$ will not suffer an L2 cache miss. However, due to the indirect preemption effect shown in Section 4.2, even $\hat{r}_j.c_{L2}$ was *AM*, it is possible that $\hat{r}_j$ does not suffer an L2 cache miss after a preemption but $\dot{r}.m_{L2}$ may have a relatively high LRU age. Under this scenario, any one of the rest of the references $\hat{r}_{j+1},\cdots,\hat{r}_k,\dot{r}$ may suffer an L2 cache miss if $\dot{r}.m_{L2}$ is evicted in the process. However, similar to the discussion for the first case, we can prove there is at most one L2 cache miss, which can be compensated by $\hat{r}_j$ if we do not count it: $\hat{r}_j$ hits L2 cache but it is treated as an L2 cache miss by the WCET estimation. Thus, no matter which case it is, $\dot{r}$ does not need to be considered as an L2-UPR at p. □

Note that "$\dot{r}$ does not need to be considered as an L2-UPR" does not mean "$\dot{r}$ may not suffer an L2 cache miss after a preemption". It is just that: Either $\dot{r}$ does not suffer an L2 cache miss, or when it does there is always some reference that has an L2 cache hit but has been accounted for as an L2 cache miss such that the compensation makes the number of L2 cache misses not underestimated. Thus, even

$\dot{r}$ does not need to be considered as an L2-UPR, it may still need to be considered as an L1-UPR in some cases (due to the indirect preemption effect shown in Section 4.1). The following theorem answers the question we have asked.

THEOREM 1. *Given a positive reference $\dot{r}$ reachable from a program point p, we can decide whether or not to consider $\dot{r}$ as a UPR at p according to three cases in sequential order.*

Case A: *If the condition of Lemma 1 for $\dot{r}$ does not hold, $\dot{r}$ is considered as an L2-UPR at p.*

Case B: *Otherwise, if $\dot{r}.c_{L1}$ is AH and on some path $\pi_{p\rightsquigarrow\dot{r}}$ there is no other reference to $\dot{r}.m_{L1}$, $\dot{r}$ is considered as an L1-UPR at p.*

Case C: *Otherwise, $\dot{r}$ is not considered as a UPR at p.*

PROOF. Considering $\dot{r}$ as an L2-UPR is the most conservative decision, so Case A holds. If $\dot{r}$ does not need to be considered as an L2-UPR and $\dot{r}.c_{L1}$ is AH, as discussed above, $\dot{r}$ may still suffer an L1 cache miss, it is also conservative to consider $\dot{r}$ as an L1-UPR at p. Therefore, Case B holds. However, when on any path $\pi_{p\rightsquigarrow\dot{r}}$ there is at least one reference to $\dot{r}.m_{L1}$ in $\langle\hat{r}_1,\cdots,\hat{r}_k\rangle$, even $\dot{r}.c_{L1}$ is AH, we can prove that $\dot{r}$ does not need to be considered as a (L1-)UPR at p (it is straightforward to see if $\dot{r}.c_{L1}$ is not AH and $\dot{r}$ does not need to be an L2-UPR, $\dot{r}$ does not need to be considered as a UPR at all). Let $\hat{r}_i \in \langle\hat{r}_1,\cdots,\hat{r}_k\rangle$ where $1 \le i \le k$ be the first reference to $\dot{r}.m_{L1}$ on a given path $\pi_{p\rightsquigarrow\dot{r}}$. (I) If $\hat{r}_i.c_{L1}$ is AH, according to Case B, $\hat{r}_i$ will be considered as at least an L1-UPR (if $i=1$, $\hat{r}_i$ will also be considered as an L2-UPR). If $\hat{r}_i$ suffers an L1 cache miss, after it occurs, $\dot{r}.m_{L1}$ and $\dot{r}.m_{L2}$ will become the youngest in their cache sets. According to the proof of Lemma 1, $\dot{r}.m_{L2}$ will be in L2 cache when $\dot{r}$ occurs, so $\dot{r}.m_{L1}$ will not be invalidated (it will not be evicted neither, since the amount of intra-task interference on L1 will never be increased, although it may be reduced). Therefore, $\dot{r}$ will not suffer an L1 cache miss. If $\hat{r}_i$ does not suffer an L1 cache miss, any one of the following references to $\dot{r}.m_{L1}$ (including $\dot{r}$) may suffer an L1 cache miss. As just argued, once such an L1 cache miss occurs, the rest of them whose L1 CHMC is AH will always hit L1 cache. Therefore, even there is such an additional L1 cache miss that we do not count, the total number will be compensated, since we always treat $\hat{r}_i$ as an additional L1 cache miss contributor. (II) If $\hat{r}_i.c_{L1}$ is not AH, it may still hit L1 cache, even $\hat{r}_i.c_{L1}$ is AM (Note that if $\hat{r}_i.c_{L1}$ suffers an L1 cache miss when it occurs, $\dot{r}$ will not suffer an L1 cache miss as discussed above). When $\hat{r}_i.c_{L1}$ hits L1 cache, any one of the following references to $\dot{r}.m_{L1}$ (including $\dot{r}$) may suffer an L1 cache miss. As we have argued, there will be at most one such additional L1 cache miss, which we do not need to count due to the compensation made by $\hat{r}_i.c_{L1}$ (it is always treated as an L1 cache miss). Thus, Case C also holds. □

We can observe that if $\dot{r}$ do not need to be considered as a UPR or only needs to be considered as an L1-UPR at a set of program points $\{p_1,\cdots,p_i\}$ where $i \ge 1$ according to Theorem 1, $\dot{r}$ can also be considered as such at another point q, as long as $\dot{r}$ is always reached from q through some point in $\{p_1,\cdots,p_i\}$. This observation is not difficult to see since Theorem 1 holds no matter what disturbance is caused by a preemption, namely $\dot{r}$ can be considered as such under any state at a point in $\{p_1,\cdots,p_i\}$. Thus, given a program point p, if at all p's immediate following points we do not need to consider $\dot{r}$ as a UPR or just need to consider $\dot{r}$ as an L1-UPR, it is safe to consider $\dot{r}$ as the same at p. Accordingly, we formulate a backward-flow analysis to derive a set of UPRs at each program point.

Let R denote the set of all the references, $\dot{R} \subseteq R$ denote the set of all the positive references, and M denote the set of all the L2 memory blocks (namely the memory blocks w.r.t. the L2 cache block size). The domain D of the analysis is defined as:

$$D = (\dot{R} \to (\mathcal{P}(M) \times \mathcal{P}(M))^{\top}_{\bot}) \times \mathcal{P}(\dot{R})$$

where $\mathcal{P}$ constructs the power set of a given set and we use $(\mathcal{P}(M) \times \mathcal{P}(M))^{\top}_{\bot}$ to mean the product domain is lifted by adding two elements $\bot$ and $\top$. Thus, a value in D will be a tuple $\langle\varrho,\tau\rangle$, where the first component $\varrho \in \dot{R} \to (\mathcal{P}(M) \times \mathcal{P}(M))^{\top}_{\bot}$ is a mapping, and the second component $\tau \in \mathcal{P}(\dot{R})$ is a set of positive references.

Starting from the exit point of the program with a value $\langle\varrho_{\bot},\emptyset\rangle \in D$ where $\forall\dot{r} \in \dot{R}, \varrho_{\bot}(\dot{r}) = \bot$, the analysis goes backwards against the control flow to derive a value in D at each program point. Such a value is computed by reaching a fixed point through iterations. Given a derived value $\langle\varrho,\tau\rangle \in D$ at a program point p and a positive reference $\dot{r} \in \dot{R}$, if we have $\varrho(\dot{r}) = \bot$, then $\dot{r}$ does not need to be considered as a UPR at p; otherwise, $\dot{r}$ should be considered as a UPR at p. If we have $\varrho(\dot{r}) = \top$, it means there is some path $\pi_{p\rightsquigarrow\dot{r}}$ on which $\varphi(\pi_{\hat{r}_k\rightsquigarrow\dot{r}})$ does not hold where $\hat{r}_k$ is the last reference to $\dot{r}.m_{L2}$ before $\dot{r}$; otherwise, $\varrho(\dot{r}) = \langle\eta,\gamma\rangle$, it means there is some path $\pi_{p\rightsquigarrow\dot{r}}$ without any other reference to $\dot{r}.m_{L2}$, where η keeps track of $\dot{r}.m_{L2}$'s conflicting L2 memory blocks to which all the references on some path $\pi_{p\rightsquigarrow\dot{r}}$ have L2 CAC as `N`, and γ keeps track of $\dot{r}.m_{L2}$'s conflicting L2 memory blocks to which at least one reference on each path $\pi_{p\rightsquigarrow\dot{r}}$ has L2 CAC as `A` or `U`. Therefore, we can compute the set of UPRs at p as $\{\dot{r}|\dot{r} \in R^{+} \wedge \varrho(\dot{r}) \neq \bot\}$. For a UPR $\dot{r}$, if $\dot{r} \in \tau$, it is treated as an L1-UPR at p; otherwise, it is treated as an L2-UPR at p.

Algorithm 1: Definition of the *update* function $\mathscr{U}$

```
   Input: ⟨ϱ,τ⟩ ∈ D, r ∈ R
   Output: ⟨ϱ,τ⟩ ∈ D
1  foreach positive reference ṙ ∈ Ṙ that satisfies
   ϱ(ṙ) ≠ ⊤ ∧ ϱ(ṙ) ≠ ⊥ ∧ σ(r.m_L2) = σ(ṙ.m_L2) do
2      ⟨η,γ⟩ ← ϱ(ṙ)
3      if r.m_L2 ≠ ṙ.m_L2 then
4          if r.a_L2 = A ∨ r.a_L2 = U then
5              ϱ[ṙ ↦ ⟨η\{r.m_L2}, γ ∪ {r.m_L2}⟩]
6          else if r.m_L2 ∉ γ then
7              ϱ[ṙ ↦ ⟨η ∪ {r.m_L2}, γ⟩]
8      else
9          if η ≠ ∅ then
10             ϱ[ṙ ↦ ⊤]
11         else if ṙ.c_L1 ≠ AH ∧ ṙ.c_L1 ≠ PS then
12             ϱ[ṙ ↦ ⊥]
13         else if r.m_L1 = ṙ.m_L1 then
14             ϱ[ṙ ↦ ⊥]
15             τ ← τ\{ṙ}
16         else
17             τ ← τ ∪ {ṙ}
18 if r.c_L2 = AH ∨ r.c_L2 = PS then
19     ϱ[r ↦ ⟨∅,∅⟩]
```

The *update* function $\mathscr{U} : D \times R \to D$ is employed to take into account the effect of a reference on a value in D. This function is applied sequentially from the last instruction to the first one of a basic block. Given a value $\langle\varrho,\tau\rangle \in D$ and a reference $r \in R$, the *update* function is defined by Alg. 1. For a positive reference $\dot{r} \in \dot{R}$, if we have $\varrho(\dot{r}) = \langle\eta,\gamma\rangle$ and $\dot{r}.m_{L2}$ is mapped to the same cache set as $r.m_{L2}$, namely $\dot{r}$ satisfies the condition given in line 1, we update the value $\langle\varrho,\tau\rangle$ depending on whether r accesses the same L2 memory block

as $\dot{r}$: If we have $r.m_{L2} \neq \dot{r}.m_{L2}$, namely $r.m_{L2}$ conflicts with $\dot{r}.m_{L2}$ in the same L2 cache set, we keep track of $r.m_{L2}$ according to r's L2 CAC and whether $r.m_{L2}$ has been in γ (lines 3–7); otherwise, we consider $\dot{r}$ according to Theorem 1 (lines 8–17). If we have $\eta \neq \emptyset$, it means on some path $\pi_{r \rightsquigarrow \dot{r}}$ the condition $\varphi(\pi_{r \rightsquigarrow \dot{r}})$ does not hold, so we set $\varrho(\dot{r})$ as $\top$ that implies $\dot{r}$ will always be considered as a UPR at a program point that can reach r (lines 9–10 correspond to Case A of the theorem); otherwise, lines 11–15 correspond to Case C where line 15 ensures $\dot{r}$ will not be considered as an L1-UPR in case $\dot{r}$ has been considered before; and lines 16–17 correspond to Case B (but we still keep tracking $\langle \eta, \gamma \rangle$ in this case until Case A or Case C is met). If r is also a positive reference, we need to record it for analyzing the following program points (lines 18–19).

Algorithm 2: Definition of the *join* function $\mathscr{J}$

```
   Input: ⟨ϱ1, τ1⟩ ∈ D, ⟨ϱ2, τ2⟩ ∈ D
   Output: ⟨ϱ, τ⟩ ∈ D
1  foreach positive reference ṙ ∈ Ṙ do
2      if ϱ1(ṙ) = ⊤ ∨ ϱ2(ṙ) = ⊤ then
3          ϱ[ṙ ↦ ⊤]
4      else if ϱ1(ṙ) = ⟨η1, γ1⟩ ∧ ϱ2(ṙ) = ⊥ then
5          ϱ[ṙ ↦ ⟨η1, γ1⟩]
6      else if ϱ1(ṙ) = ⊥ ∧ ϱ2(ṙ) = ⟨η2, γ2⟩ then
7          ϱ[ṙ ↦ ⟨η2, γ2⟩]
8      else if ϱ1(ṙ) = ⟨η1, γ1⟩ ∧ ϱ2(ṙ) = ⟨η2, γ2⟩ then
9          ϱ[ṙ ↦ ⟨η1 ∪ η2, γ1 ∩ γ2⟩]
10     else
11         ϱ[ṙ ↦ ⊥]
12 τ ← τ1 ∩ τ2
```

The *join* function $\mathscr{J} : D \times D \rightarrow D$ is used to merge values in D at a join point. In the backward-flow analysis, a join point corresponds to a branching point in the CFG. Given two values $\langle \varrho_1, \tau_1 \rangle \in D$ and $\langle \varrho_2, \tau_2 \rangle \in D$, the *join* function is defined by Alg. 2. Since we need to overestimate the set of UPRs at each program point, for a positive reference $\dot{r} \in \dot{R}$, it is not considered as a UPR at the join point only if it is not considered as a UPR in both given values (lines 10–11); otherwise, $\dot{r}$ should be considered as a UPR at the join point, even if it is not considered as a UPR in one given value (lines 4–7). Likewise, if $\dot{r}$ is mapped to $\top$ in either value, it should also be mapped to $\top$ in the joined value (lines 2–3); and if $\dot{r}$ is mapped to $\langle \eta_1, \gamma_1 \rangle$ and $\langle \eta_2, \gamma_2 \rangle$ respectively in these two values, in the joined value $\dot{r}$ should be mapped to $\langle \eta, \gamma \rangle$ where η overestimates the set of L2 memory blocks that may lead to $\top$ (i.e. $\eta = \eta_1 \cup \eta_2$) and γ underestimates the set of L2 memory blocks that may lead to $\bot$ (i.e. $\gamma = \gamma_1 \cap \gamma_2$) (lines 8–9). Moreover, we take the intersection of τ_1 and τ_2 to safely obtain L1-UPRs (line 12).

5.2 Bound on Number of Times Being A UPR

In order to calculate CRPD from the derived set of references that are considered as UPRs at a program point, we need to bound the number of times a reference may act as a UPR. If a reference is not in a loop, it is straightforward to see it may act as a UPR at most once. The problem emerges when a reference considered as a UPR is inside a loop, in which case the reference can execute many times.

First, let us discuss the necessary conditions on a loop when a reference inside the loop is positively classified. In the following, we use $\mathscr{A}_{L2}$ to denote the associativity of L2 cache.

LEMMA 2. *If a reference $\dot{r}$ in a loop is a positive reference, then one of the following two cases holds.*

Case A: *There are fewer than $\mathscr{A}_{L2}$ conflicting L2 memory blocks of $\dot{r}.m_{L2}$ accessed on any cyclic path $\pi_{\dot{r} \rightsquigarrow \dot{r}}$.*

Case B: *If there is a cyclic path $\pi_{\dot{r} \rightsquigarrow \dot{r}}$ on which there are at least $\mathscr{A}_{L2}$ conflicting L2 memory blocks of $\dot{r}.m_{L2}$ accessed, there is at least one reference to $\dot{r}.m_{L2}$ classified as L1* AM *in between $\dot{r}$ on the cyclic path.*

PROOF. We prove this lemma by contraposition, namely we are to prove

$$\neg\text{Case A} \wedge \neg\text{Case B} \implies r \text{ is not a positive reference}$$

We have ¬Case A as "there are at least $\mathscr{A}_{L2}$ conflicting L2 memory blocks of $\dot{r}.m_{L2}$ accessed on some cyclic path $\pi_{\dot{r} \rightsquigarrow \dot{r}}$". Also, we have ¬Case B as "given any cyclic path $\pi_{\dot{r} \rightsquigarrow \dot{r}}$ on which there are at least $\mathscr{A}_{L2}$ conflicting L2 memory blocks of $\dot{r}.m_{L2}$ accessed, there is no reference to $\dot{r}.m_{L2}$ classified as L1 *AM* in between $\dot{r}$ on the path". Combining ¬Case A and ¬Case B, on such a cyclic path $\pi_{\dot{r} \rightsquigarrow \dot{r}}$, we can deduce that at least $\mathscr{A}_{L2}+1$ distinct L2 memory blocks need to appear in the corresponding L2 cache set due to the inclusion property. As a result, at least one L2 memory block, say m, among these distinct L2 memory blocks is not definitely/persistently in L2 cache at the end of this path (i.e. $\dot{r}$ is met again since the path is cyclic). Therefore, we have: (1) If m is $\dot{r}.m_{L2}$, it means $\dot{r}.m_{L2}$ is not definitely/persistently in L2 cache when $\dot{r}$ occurs, so $\dot{r}$ is not a positive reference. (2) If m is not $\dot{r}.m_{L2}$, since m is not in L2 cache at the beginning of this cyclic path, after m is first referenced on this path, m will become the youngest in the cache set, namely it is possible that m is younger than $\dot{r}.m_{L2}$ at that point. Since the *must* and *persistence* analyses are safe, in the ACSs of these two analyses, $\dot{r}.m_{L2}$ should not be younger than m at that point (see Section 3). Since there is no reference to $r.m_{L2}$ classified as L1 *AM* in between $\dot{r}$ on the cyclic path, $\dot{r}.m_{L2}$ cannot be put into the first position in the ACSs. Therefore, at the end of this cyclic path, $\dot{r}.m_{L2}$ is still treated as possibly older than m, which means $\dot{r}.m_{L2}$ should be classified as "not definitely/persistently in L2 cache when $\dot{r}$ occurs". Thus, $\dot{r}$ is not a positive reference. □

As shown in Section 4.2, in the presence of a preemption, we can no longer guarantee that L1/L2 *AM* remains valid, which implies Case B in Lemma 2 may not hold when there exists inter-task interference. On the contrary, if Case A in Lemma 2 holds, it will always hold whether or not there is a preemption. Therefore, we have the following theorem:

THEOREM 2. *Given a positive reference $\dot{r} \in \dot{R}$ in a loop, if Case A in Lemma 2 holds, it can suffer L2 cache misses at most once after a preemption.*

PROOF. We prove this theorem by contradiction. Let us assume $\dot{r}$ can suffer at least two L2 cache misses in the presence of a preemption. After $\dot{r}$'s first L2 cache miss, $\dot{r}.m_{L2}$ becomes the youngest memory block in the corresponding cache set. Since $\dot{r}$ can suffer at least another L2 cache miss, it means $\dot{r}.m_{L2}$ will be evicted from L2 cache. Therefore, there are at least $\mathscr{A}_{L2}$ conflicting L2 memory blocks of $r.m_{L2}$ accessed on some cyclic path $\pi_{\dot{r} \rightsquigarrow \dot{r}}$, which means Case A does not hold. Therefore, we reach a contradiction. □

Note that if $\dot{r}$ is also classified as L1 *AH*/*PS*, it may also suffer L1 cache misses after a preemption; but it is straightforward to see if Case A holds, it will also suffer at most one L1 cache miss since, when it does, it will bring both $\dot{r}.m_{L1}$ and $\dot{r}.m_{L2}$ to their youngest positions and no invalidation and eviction would happen to them later on. Therefore, the number of times that a positive reference $\dot{r}$ in a loop $\mathcal{L}$ may

act as a UPR can be bounded by $\kappa(\mathcal{L}, \dot{r})$, which is defined as:

$$\kappa(\mathcal{L}, \dot{r}) = \begin{cases} 1 & \text{if } \mu(\mathcal{L}, \dot{r}) < \mathscr{A}_{L2} \\ lb(\mathcal{L}) & \text{otherwise} \end{cases}$$

where $\mu(\mathcal{L}, \dot{r})$ over-approximates the number of conflicting L2 memory blocks of $\dot{r}.m_{L2}$ accessed on any cyclic path $\pi_{\dot{r} \rightsquigarrow \dot{r}}$ in the loop $\mathcal{L}$, and $lb(\mathcal{L})$ gives the loop bound of $\mathcal{L}$. While $lb(\mathcal{L})$ can be manually input or derived by other techniques (which is not in the scope of this paper), $\mu(\mathcal{L}, \dot{r})$ is defined as:

$$\mu(\mathcal{L}, \dot{r}) = \\ |\{m | \exists r \in \mathcal{L}, m = r.m_{L2} \wedge m \neq \dot{r}.m_{L2} \wedge \sigma(m) = \sigma(\dot{r}.m_{L2})\}|$$

which calculates in the loop how many memory blocks other than $\dot{r}.m_{L2}$ are mapped to the same L2 cache set as $\dot{r}.m_{L2}$. Since only a subset of these memory blocks is accessed on any path in the loop, $\mu(\mathcal{L}, \dot{r})$ over-approximates the number of distinct L2 conflicting memory blocks of $\dot{r}.m_{L2}$ accessed on any cyclic path $\pi_{\dot{r} \rightsquigarrow \dot{r}}$.

5.3 CRPD Calculation

After the backward-flow analysis, we derive a set of UPRs at each program point. If a preemption occurs at some program point, we can safely estimate the corresponding CRPD using the derived set of UPRs at that point. Since the execution can be at an arbitrary point when a preemption occurs, the maximum CRPD among all the possible ones is regarded as the task's CRPD due to a single preemption.

Algorithm 3: CRPD calculation at a program point

```
   Input: ⟨ϱ, τ⟩ ∈ D, p ∈ P
   Output: crpd
1  crpd ← 0
2  foreach positive reference ṙ ∈ Ṙ that satisfies ϱ(ṙ) ≠ ⊥ do
3      if ṙ is in a loop ℒ then
4          n ← κ(ℒ, ṙ)
5          if ṙ.c_L1 = AH then
6              crpd ← crpd + T_1 × n
7          else if ṙ.c_L1 = PS then
8              if p is not in loop ℒ then
9                  crpd ← crpd + T_1 × (n − 1)
10             else
11                 crpd ← crpd + T_1 × max(1, n − 1)
12         if ṙ ∉ τ then
13             if ṙ.c_L2 = AH then
14                 crpd ← crpd + T_2 × n
15             else if ṙ.c_L2 = PS then
16                 if p is not in loop ℒ then
17                     crpd ← crpd + T_2 × (n − 1)
18                 else
19                     crpd ← crpd + T_2 × max(1, n − 1)
20     else
21         if ṙ.c_L1 = AH then
22             crpd ← crpd + T_1
23         if ṙ ∉ τ then
24             crpd ← crpd + T_2
```

Let P denote the set of all the program points, T_1 denote L1 cache reload time, and T_2 denote L2 cache reload time. Given the derived value $\langle \varrho, \tau \rangle \in D$ at a program point $p \in P$, we overestimate CRPD at p by applying Alg. 3. For each positive reference $\dot{r}$ that can act as a UPR at p, we consider whether $\dot{r}$ is inside a loop. If $\dot{r}$ is not in a loop, it can only be of type 1, 2, or 3 shown in Tab. 1, and it can act as a UPR at most once. In this case, no matter whether $\dot{r}$ acts as an L1-UPR or an L2-UPR, as long as it is of type 1, a T_1 is added to the CRPD (lines 21–22). If $\dot{r}$ is an L2-UPR, we also need to add a T_2 to the CRPD (lines 23–24). On the other hand, if $\dot{r}$ is inside a loop $\mathcal{L}$, $\dot{r}$ can be of either type shown in Tab. 1. The number of times n that $\dot{r}$ can act as an UPR is obtained by applying $\kappa(\mathcal{L}, \dot{r})$ (line 4). If $\dot{r}$ is of type 1, 2, or 3, accounting for $\dot{r}$ is similar to that when $\dot{r}$ is outside of loops, but we need to consider it n times instead of just one time (lines 5–6 and lines 13–14). If $\dot{r}$ has L1/L2 *PS* CHMC, we also need to consider whether p is in the same loop as $\dot{r}$. If they are not in the same loop, we only need to take into account $n-1$ times that $\dot{r}$ acts as a UPR, since the WCET estimation already considers the first time as a cache miss for *PS* CHMC (lines 8–9 and lines 16–17). Thus, if we have $n = 1$, $\dot{r}$ will contribute nothing to additional cache misses. If $\dot{r}$ and p are in the same loop, we should consider $\dot{r}$ as a UPR at least once even we have $n = 1$ (lines 10–11 and lines 18–19). This is because when a preemption occurs at p, the loop may have already passed its first iteration.

Note that in the current work we do not consider preempting tasks, namely any cache hierarchy state is regarded as possible after a preemption. In addition, we do not consider whether the derived UPRs at a program point can all occur after a preemption, e.g., two UPRs derived at a point may be not reachable from each other when the execution passes this program point. Although the precision may be reduced, it is a safe CRPD analysis for multi-level inclusive caches. How to improve the precision is our future work.

6. EXPERIMENTAL RESULTS

In this section, we evaluate the proposed approach on a set of benchmarks. The benchmarks are shown in Tab. 3 that are maintained by the Mälardalen WCET research group [8]. For each benchmark, we estimate its CRPD for only one preemption (if there are multiple preemptions, we can simply adopt the method used in [14] to construct a table whose i^{th} entry corresponds to the i^{th} biggest possible CRPD, which is always safe although may be not precise). Each benchmark is compiled for MIPS R3000 using gcc-3.4.4.

Table 3: Cache capacity configs for each benchmark

Benchmark	Code Size	Config. 1		Config. 2	
		L1	L2	L1	L2
fibcall	220B	256B	1KB	32B	128B
bs	320B	512B	2KB	64B	256B
janne	324B	512B	2KB	64B	256B
insertsort	440B	512B	2KB	64B	256B
bsort100	564B	1KB	4KB	128B	512B
ns	588B	1KB	4KB	128B	512B
fir	600B	1KB	4KB	128B	512B
expint	888B	1KB	4KB	128B	512B
matmult	932B	1KB	4KB	128B	512B
cnt	944B	1KB	4KB	128B	512B
qurt	1328B	2KB	8KB	256B	1KB
select	1580B	2KB	8KB	256B	1KB
ludcmp	2276B	4KB	16KB	512B	2KB
jfdctint	2580B	4KB	16KB	512B	2KB
lms	2588B	4KB	16KB	512B	2KB
minver	3052B	4KB	16KB	512B	2KB
compress	3564B	4KB	16KB	512B	2KB
edn	3576B	4KB	16KB	512B	2KB
statemate	10296B	16KB	64KB	2KB	8KB
nsichneu	40036B	64KB	256KB	8KB	32KB

As described in Section 5, the proposed CRPD analysis for two-level inclusive caches is based on the multi-level inclusive cache analysis proposed in [25]. We implement both analyses in our research tool which first constructs a context-sensitive call graph for the program and CFGs for all the procedures from the compiled binary. Thus, we can have multiple different references to the same memory block, which correspond to an instruction belonging to a procedure reached by different contexts.

In our experiments, we only consider a two-level inclusive cache but do not consider other micro-architectural features. As stated in our system model, both L1 and L2 caches are set associative, and LRU replacement policy is used in them. Some basic parameters for this two-level cache hierarchy are not changed in different experiments: L1 cache is 2-way set associative, its cache block size is 8-byte, and its access latency is 1-cycle. L2 cache is 4-way set associative, its cache block size is 16-byte, and its access latency is 5-cycle. For every needed information, we assume it can be found in the main memory with a 20-cycle latency.

We perform two experiments on each benchmark by changing L1 and L2 cache capacities. In the first configuration, we require that L1 cache size is bigger than (but not bigger than twice of) the code size, and L2 cache size is bigger than four times that of L1 cache size. In the second configuration, we require that L2 cache size is smaller than (but not smaller than half) the code size, and L1 cache size is smaller than four times that of L2 cache size. The two configurations for each benchmark with its code size are shown in Tab. 3.

Table 4: Experimental results – WCET and CRPD

Benchmark	Config. 1			Config. 2		
	WCET	CRPD	Ratio	WCET	CRPD	Ratio
fibcall	1225	205	16.7%	2860	125	4.4%
bs	1052	485	46.1%	1152	265	23.0%
janne	1448	395	27.3%	4327	185	4.3%
insertsort	4329	385	8.9%	21374	125	0.6%
bsort100	269254	460	0.2%	1466629	265	<0.1%
ns	26895	390	1.5%	92425	225	0.2%
fir	8193	630	7.7%	42096	265	0.6%
expint	9957	520	5.2%	40427	305	0.8%
matmult	508737	505	0.1%	4214252	245	<0.1%
cnt	12856	560	4.4%	79948	150	0.2%
qurt	12004	1560	13.0%	52835	140	0.3%
select	8767	2395	27.3%	43199	400	0.9%
ludcmp	15835	760	4.8%	22410	720	3.2%
jfdctint	14965	2230	14.9%	22265	1465	6.6%
lms	450970	1330	0.3%	3067340	1245	<0.1%
minver	15240	885	5.8%	24299	505	2.1%
compress	40151	1505	3.7%	353129	470	0.1%
edn	154948	1985	1.3%	287438	520	0.2%
statemate	23638	5480	23.2%	33130	290	0.9%
nsichneu	147848	76525	51.8%	264308	4970	1.9%

The experimental results are given in Tab. 4. Under each capacity configuration, both estimated WCET and CRPD in terms of clock cycles are reported. We can observe that for each benchmark the estimated WCET under the first configuration is always lower that under the second one, whereas, the estimated CRPD under the first configuration is always higher than that under the second one. The reason for this is: Under the first configuration, most likely, the code segment of the program can all be loaded into L1 cache without eviction[2]. Thus, many references can be positively classified by the cache hierarchy analysis, and most of them are of type 1, 4, and 5, which implies not even too many L1 cache misses are taken into account in the WCET estimation; but many of these positive references may be considered as UPRs at some program points, especially the points in some big loops (e.g. in the cases of statemate and nsichneu benchmarks). On the contrary, under the second configuration, either the references are not positively classified, or the positive references are of type 2, 3, 6, and 7. Therefore, most of the cache misses are taken into account in the WCET estimation; and most of the positive references can be classified according to Case C of Theorem 1, so they do not need to be considered as UPRs. We can also observe that the sum of WCET and CRPD under the first configuration is much smaller than that under the second configuration in most cases.

In addition, the ratio of CRPD to WCET (i.e. $\frac{\text{CRPD}}{\text{WCET}}$) is also computed. From the results, we can see the estimated CRPD can be in large proportion to the estimated WCET in some cases. On one hand, this justifies the purpose of CRPD analysis when designing embedded control systems. On the other hand, this also implies sometimes the approach may be too conservative to analyze CRPD for multi-level inclusive caches.

The experiments are all carried out on a machine with a 3.4GHz quad-core processor and 16GB memory. Most of the analysis time is spent on the cache hierarchy analysis, which can up to a few thousand seconds in some cases. The proposed CRPD analysis runs fast, which is always in the range of a few seconds.

7. RELATED WORK

CRPD analysis for single-level caches has been studied extensively in the past two decades: In [14], the important concept of UCB is proposed to bound the CRPD conservatively at a program point of the preempted task. Later in [1], a new notion of UCB is defined, which takes into account only the definitely-cached UCBs (DC-UCB). Using this concept of DC-UCB may under-approximate the CRPD itself, but over-approximation is ensured when the estimated CRPD is used together with the estimated WCET. Since the CRPD of the preempted task also depends on how much "damage" a preempting task can cause, the concept of ECB is used to bound the CRPD in [24].

Compared to the approaches that rely on only either UCBs or ECBs in isolation, the methods that combine them together can result in more precise analyses, e.g. such methods are proposed in [15, 18] to analyze the CRPD for direct-mapped caches. However, as stated in [6], a simple strategy to combine UCBs and ECBs may result in an unsafe CRPD analysis for set-associative caches, such as the approach proposed in [22] which simply counts on the minimum number of UCBs, ECBs, and associativity in each set to bound the impact of a preemption. For set-associative caches, there is often deferred impact on UCBs after a preemption, and a method to compute how resilient an UCB can be is proposed in [3], which gives rise to a safe and precise CRPD analysis for LRU set-associative caches [13]. For set-associative caches using other replacement policies, some preliminary work has also been done [6, 5].

[2]For some benchmarks, e.g. bsort100, some unused procedures may be in between the used procedures in the code segment of the binary. This makes instruction addresses non-consecutive, which may cause some cache set having much more conflicting memory blocks mapped than others.

Under preemptive scheduling, a task can be preempted multiple times. A method proposed in [20] tries to reduce CRPD overestimation when considering multiple preemptions, and later this method is used in [21] which takes into account the estimated CRPD to analyze the response time of a task in a much efficient way under fixed-priority preemptive scheduling. Schedulability analysis accounting for CRPD is further studied in [2] for fixed-priority preemptive scheduling. For dynamic priority preemptive scheduling, several approaches have also been proposed to test the schedulability with the awareness of CRPD [11, 17].

The first approach analyzing CRPD in term of multi-level caches is proposed in [7]. That work focuses on CRPD analysis for multi-level *non-inclusive* caches, which relies on the cache behavior analysis for *non-inclusive* cache hierarchies that is proposed in [9]. In this paper, our approach is for multi-level *inclusive* caches, which is based on the cache behavior analysis for *inclusive* cache hierarchies that is proposed in [25]. The approaches proposed in [7] and in this paper only focus on multi-level instruction caches. While there have been some approaches proposed to analyze CRPD for single-level data caches (e.g. a method based on data cache access patterns rather than UCBs is proposed in [19]), CRPD analysis for multi-level data/unified caches remains an open research problem, especially for multi-level *inclusive* caches, the existing methods only focus on instruction cache behavior analysis [10, 26, 25].

8. CONCLUSION & FUTURE WORK

In this paper, we investigate how to analyze CRPD for multi-level inclusive caches, which has not been studied before. First, we identify some indirect preemption effects due to the strict inclusion enforcement in inclusive cache hierarchies. These indirect preemption effects make the traditional UCB concept difficult to use for CRPD analysis in terms of multi-level inclusive caches. We propose a new concept of UPRs, and based on this new concept we propose an approach to conservatively analyze CRPD for a two-level inclusive cache hierarchy.

In order to measure CRPD empirically to compare with the analysis results, we plan to use and integrate some cycle-accurate instruction set simulator like gem5 in our research tool. The simulation results can serve as a baseline to investigate the analysis precision.

There are multiple known sources of pessimism in the current approach, which we need to address in the future. For example, we do not analyze preempting tasks, so we assume the cache hierarchy can be in any state after a preemption. By integrating the traditional concept of ECBs into our approach, the analysis precision should be improved. Another instance is that we do not consider if two UPRs can occur in the same execution. Implicit path enumeration technique (IPET) should be helpful, but it may incur too much computational overhead if it is used at every program point. At last, we also need to study how to improve the precision in the case of multiple preemptions.

Acknowledgments: This work is supported in part by the National Science Foundation (CNS-1035655).

9. REFERENCES

[1] S. Altmeyer and C. Burguiere. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *ECRTS '09*, pages 109–118, 2009.

[2] S. Altmeyer, R. I. Davis, and C. Maiza. Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. In *RTSS '11*, pages 261–271, 2011.

[3] S. Altmeyer, C. Maiza, and J. Reineke. Resilience analysis: Tightening the crpd bound for set-associative caches. In *LCTES '10*, pages 153–162, 2010.

[4] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *ISCA '88*, pages 73–80, 1988.

[5] C. Ballabriga, L. K. Chong, and A. Roychoudhury. Cache-related preemption delay analysis for fifo caches. In *LCTES '14*, pages 33–42, 2014.

[6] C. Burguière, J. Reineke, and S. Altmeyer. Cache-related preemption delay computation for set-associative caches - pitfalls and solutions. In *WCET'09*, volume 10, pages 1–11, 2009.

[7] S. Chattopadhyay and A. Roychoudhury. Cache-related preemption delay analysis for multilevel noninclusive caches. *ACM Trans. Embed. Comput. Syst.*, 13(5s):147:1–147:29, July 2014.

[8] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks - past, present and future. In *WCET '10*, 2010.

[9] D. Hardy and I. Puaut. Wcet analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS '08*, pages 456–466, 2008.

[10] D. Hardy and I. Puaut. Wcet analysis of instruction cache hierarchies. *J. Syst. Archit.*, 57(7):677–694, Aug. 2011.

[11] L. Ju, S. Chakraborty, and A. Roychoudhury. Accounting for cache-related preemption delay in dynamic priority schedulability analysis. In *DATE '07*, pages 1623–1628, 2007.

[12] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Bus-aware multicore wcet analysis through tdma offset bounds. In *ECRTS '11*, pages 3–12, 2011.

[13] J. C. Kleinsorge, H. Falk, and P. Marwedel. A synergetic approach to accurate analysis of cache-related preemption delay. In *EMSOFT '11*, pages 329–338, 2011.

[14] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.*, 47(6):700–713, June 1998.

[15] C.-G. Lee, K. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Bounding cache-related preemption delay for real-time systems. *IEEE Trans. Softw. Eng.*, 27(9):805–826, Sept. 2001.

[16] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '99*, pages 12–21, 1999.

[17] W. Lunniss, S. Altmeyer, C. Maiza, and R. I. Davis. Integrating cache related pre-emption delay analysis into edf scheduling. In *RTAS '13*, pages 75–84, 2013.

[18] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS '03*, pages 201–206, 2003.

[19] H. Ramaprasad and F. Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *RTAS '06*, pages 71–80, 2006.

[20] J. Staschulat and R. Ernst. Multiple process execution in cache related preemption delay analysis. In *EMSOFT '04*, pages 278–286, 2004.

[21] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *ECRTS '05*, pages 41–48, 2005.

[22] Y. Tan and V. Mooney. Timing analysis for preemptive multitasking real-time systems with caches. *ACM Trans. Embed. Comput. Syst.*, 6(1), Feb. 2007.

[23] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise wcet prediction by separated cache andpath analyses. *Real-Time Syst.*, 18(2/3):157–179, May 2000.

[24] H. Tomiyama and N. D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *CODES '00*, pages 67–71, 2000.

[25] Z. Zhang and X. Koutsoukos. Precise multi-level inclusive cache analysis for wcet estimation. In *RTSS '15*, pages 350–360, 2015.

[26] Z. Zhang and X. Koutsoukos. Top-down and bottom-up multi-level cache analysis for wcet estimation. In *RTAS '15*, pages 24–35, 2015.