Top-Down and Bottom-Up Multi-Level Cache Analysis for WCET Estimation

Zhenkai Zhang Xenofon Koutsoukos
Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN, USA
Email: {zhenkai.zhang, xenofon.koutsoukos}@vanderbilt.edu

***Abstract*—In many multi-core architectures, inclusive shared caches are used to reduce cache coherence complexity. However, the enforcement of the inclusion property can cause invalidation of memory blocks at higher cache levels. In order to ensure safety, analysis of cache hierarchies with inclusive caches for worst-case execution time (WCET) estimation is typically based on conservative decisions. Thus, the estimation may not be tight. In order to tighten the estimation, this paper proposes an approach that can more precisely analyze the behavior of a cache hierarchy maintaining the inclusion property. We illustrate the approach in the context of multi-level instruction caches. The approach first analyzes all the inclusive caches in the hierarchy in a *bottom-up* direction, and then analyzes the remaining non-inclusive caches in a *top-down* direction. In order to capture the inclusion victims and their effects, we also propose a concept of aging barrier and integrate it with the traditional *must* and *persistence* analyses to safely slow down their aging process so as to derive more precise analyses. We evaluate the proposed approach on a set of benchmarks and the evaluation reveals that the estimations are tightened.**

I. Introduction

Hard real-time system design requires WCET estimation for each task. Since the exact WCET of a task is impossible to derive in general, an overestimation is necessary to ensure safety. Yet, in order to maximize resource utilization, the estimation should be as tight as possible. However, due to the complex behavior of many performance enhancing features in modern processors, it is very challenging to safely and tightly estimate the WCET.

Caches are very common in processors in order to bridge the increasing gap between the processor clock cycle time and main memory access time. Although the presence of caches improves the average performance, it poses great challenges on the tightness of WCET estimation. Over the past two decades, the analysis of the effects of single-level cache behavior on WCET estimation has been studied extensively [17, 19].

Recently, multi-level cache analysis has drawn much attention in real-time systems [8, 12, 4, 18, 9], since there is a rising need of exploiting the high-performance processors, which are often equipped with multi-level caches. However, compared to single-level cache analysis, multi-level cache analysis is much more challenging. Besides the sequence of memory references, there is a need to take into account the effects of the behavior of one cache level on the behavior of other cache levels (e.g. filtering memory accesses and invalidating memory blocks), which can be different depending on the type of the cache hierarchy.

Typically, there are three cache hierarchy types, which are 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 cache levels. On the contrary, multi-level exclusive caches require that the contents at a cache level should not be duplicated at any other cache levels. Multi-level non-inclusive caches allow duplicated contents existing at any cache level, but they do not strictly enforce the inclusion. Moreover, there are some hybrid cache hierarchies, which have some inclusive and/or exclusive cache levels and other levels being non-inclusive. In this paper, we call a cache hierarchy a multi-level ***inclusive*** cache as long as it maintains the inclusion property at some cache level(s).

Compared to an exclusive/non-inclusive cache hierarchy, a cache hierarchy enforcing inclusion has less effective cache capacity, but the inclusion property can significantly simplify the maintenance of cache coherence [1]. Therefore, multi-level inclusive caches are widely used in many multi-core architectures. A multi-level cache analysis framework that can precisely analyze cache hierarchies that enforce inclusion becomes necessary for WCET estimation.

Most of the current approaches target multi-level non-inclusive cache analysis, and it is not straightforward to extend these approaches to tightly analyze inclusive caches, since the invalidation behavior introduced by maintaining the inclusion property requires making conservative decisions in order to ensure safety [9]. The main idea in this paper is that this pessimism can actually be reduced by analyzing the multi-level inclusive caches in a *bottom-up* direction, which is counter-intuitive in contrast with the natural *top-down* cache hierarchy access direction that is used in existing methods for multi-level cache analysis. In this paper, the *top-down* direction is referring to the direction from the uppermost cache level (i.e. L1) downto the lowest cache level, and the *bottom-up* direction is referring to the opposite.

The main technical contributions of this paper are: (1) We propose an approach which analyzes all the inclusive caches in the *bottom-up* direction first, and then analyzes the rest non-inclusive caches in the *top-down* direction. Due to the *bottom-up* analysis, the invalidation behavior becomes visible at the time of analyzing upper levels; (2) We propose a concept of aging barrier to capture the effects of the invalidations caused by inclusive caches, and by using the aging barriers, we can safely slow down the increase of memory block ages in a cache that is above an inclusive cache level, so more precise *must* and *persistence* analyses can be achieved; (3) We evaluate the

proposed approach using a set of benchmarks, and we find the proposed approach can tighten the WCET estimation by 12.2% on average, compared to the approach proposed in [9]. In this paper, we only consider multi-level inclusive instruction caches for a single processor. Although the effects of data references and inter-core interferences are not considered, this approach can serve as a basis for such extensions.

The rest of the paper is organized as: Section II shows why a multi-level inclusive cache is hard to analyze for WCET estimation; Section III gives the system model considered in this paper; Section IV presents our multi-level inclusive cache analysis; Section V evaluates the proposed approach; Section VI describes the related work, and Section VII concludes this paper.

II. Problem Statement

In the case of single-level cache analysis, only the effects of the memory reference sequences need to be taken into account. In order to make the analysis scalable, most of the approaches are based on abstract interpretation. An abstract interpretation based approach aims to assign a cache hit/miss classification (CHMC) to each memory reference according to the abstract cache states (ACSs) derived by three different analyses [19, 5]. The analyses are usually performed on the control-flow graph (CFG) reconstructed from the low-level code of the program. At a given program point, a *must* analysis is used to determine the set of memory blocks that are *definitely* in the cache, so a memory reference to a block being in the set can be classified as *always hit* (*AH*); a *may* analysis is used to determine the set of memory blocks that are *possibly* in the cache, so a memory reference to a block not being in the set can be classified as *always miss* (*AM*); a *persistence* analysis is used to determine the set of memory blocks that stay in the cache once they are loaded, and a memory reference to such a block is classified as *persistent* (*PS*) or *first miss* (*FM*); and, if a memory reference cannot be classified as *AH*, *AM*, or *PS*, it is classified as *not classified* (*NC*).

When analyzing multi-level caches, it is also important to consider the effects of other cache levels, like cache access filtering and memory block invalidation. For example, if we treat every possible access at a level as always happening, the analysis may become unsafe, since doing so may underestimate the set reuse distances[1] of memory blocks [8].

For a reference at a cache level, a cache access classification (CAC) can be used to represent whether the cache access at this level will occur: *always* (*A*) denotes the access will always occur, *never* (*N*) denotes the access will never happen, and *uncertain* (*U*) denotes the access may occur [8]. In order to ensure safety, the updates of the abstract cache states due to *U* accesses need to take into account the two possible cases (access occurring and not occurring).

In the case of multi-level non-inclusive cache analysis, the CAC for a reference r at a cache level l can be derived from the CHMC and CAC for r at $l-1$ (as described in [8]), and the behavior of l will not be affected by any lower cache level. However, in the case of analyzing cache hierarchies containing inclusive caches, the CAC for r at l cannot be safely derived from CHMC and CAC for r at $l-1$. The reason is the behavior of l depends not only on the behavior of $l-1$, but also on the invalidation behavior induced by some lower inclusive cache level(s): When a memory block is evicted from a lower inclusive cache level, all the contents that belong to this memory block need to be invalidated from its upper cache levels (the invalidated memory blocks are called *inclusion victims*).

Example: Fig. 1 shows a 3-level inclusive cache, where L1 is 2-way set associative, L2 is 4-way set associative, and L3 is 8-way set associative (at each level, only one set is shown). We assume L1 has the smallest cache block size and L3 has the biggest, so a block in L1 is a sub-block of some block in L2 and that block in L2 is a sub-block of some block in L3. For a memory block m in L3, let $\dot{m}$ denote a m's sub-block in L2, and let $\ddot{m}$ denote a $\dot{m}$'s sub-block in L1. For example, we have $\ddot{m}_a \subset \dot{m}_a \subset m_a$. If the next reference needs the information that is in m_x (m_x is also mapped to the shown set of L3), the oldest m_a in that set needs to be evicted. The eviction of m_a will also invalidate $\ddot{m}_a$ in L1 and $\dot{m}_a$ in L2 to maintain the inclusion property. Due to the invalidation, $\ddot{m}_h$ in L1 can live longer, and depending on which sub-block of m_x is needed by the reference, there may be some "holes" left in L1 and L2.

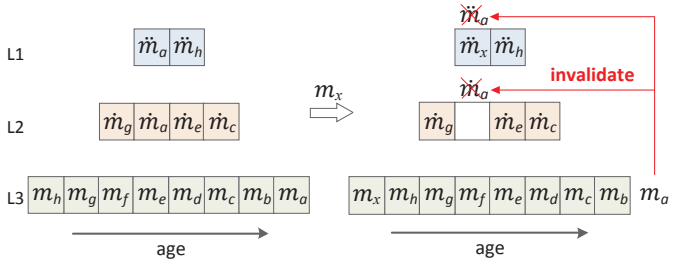

Fig. 1. Invalidation due to the maintenance of the inclusion property of L3

In [9], multi-level non-inclusive cache analysis is adapted to multi-level inclusive cache analysis. To achieve this, several conservative decisions are made on the CAC and CHMC for a reference at a cache level due to any possible invalidation to ensure safety: (1) Except for L1 which is always accessed, the CAC at any other level should be classified as *U*; (2) If a reference is classified as *AH* or *PS* at a level, this CHMC may be changed into *NC* depending on the analysis of lower inclusive levels; (3) Even if a memory reference is classified as *AM* at a level, this CHMC has to be changed into *NC*. In this way, although safety is ensured, the tightness of the estimation may suffer a lot. Therefore, we need a method that can more precisely analyze the effects of multi-level inclusive caches on WCET estimation.

III. System Model

We focus on a general multi-level inclusive cache model. The model has p cache levels, where $p \geq 2$, among which q levels are inclusive, where $p > q \geq 1$, and the other $p-q$ levels are non-inclusive[2]. We also assume the time for a processing

[1] In [8], the set reuse distance between two memory references to the same block at a cache level is defined as the relative age of the memory block when the second reference occurs.

[2] It has no meaning for L1 cache to be inclusive/non-inclusive. Later, we treat L1 as non-inclusive to facilitate the presentation. Thus, we assume $p > q$ not $p \geq q$.

element to access a cache level is bounded and predictable, which can be achieved by using deterministic interconnects to connect the caches, like TDMA buses [11].

Let $L = \{l_x | 1 \leq x \leq p\}$ be the set of all the cache levels, in which l_x denotes the x^{th} cache level. Let I be the set of all the inclusive cache levels, and let N be the set of all the non-inclusive cache levels. Thus, we have $L = I \cup N \bigwedge I \cap N = \emptyset \bigwedge |I| = q$. Since it does not matter whether l_1 is inclusive or non-inclusive, we can simply assume $l_1 \in N$, so neither I nor N is an empty set. Fig. 2 gives two examples of the models focusing on single cores of two multi-core architectures.

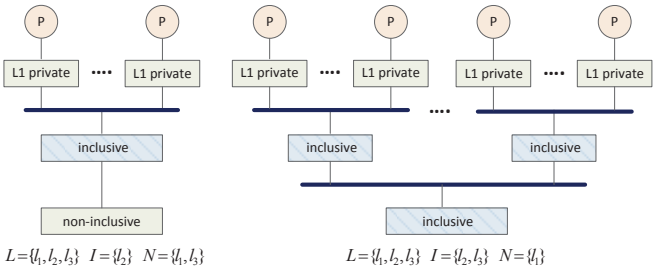

Fig. 2. Two examples of the models with respect to a single core

We assume at each level the cache is set associative, and least recently used (LRU) replacement policy is used. The size of a cache block can be different at different cache levels, and it is common to assume the block size does not increase as the level goes up. It is also common to assume the capacity decreases as the level goes up. Let C_{l_x} denote the cache at the cache level l_x, let A_{l_x} denote the associativity of C_{l_x}, and let s_{l_x} denote the number of cache sets of C_{l_x}. Sometimes we use "cache level" to actually mean the cache located at that level if there is no ambiguity.

Although we do not consider exclusive caches in the model, we can easily add them into our analysis by using the approach proposed in [9]. Basically, the exclusive cache levels can be collapsed by concatenating them to the end of the upper level to form a single level for the analysis, as long as they all have the same number of cache sets and the same cache block size. In this paper, we focus on how to analyze multi-level caches in the presence of invalidations caused by inclusion enforcement, so we simply consider multi-level instruction caches in terms of a single processor. This work can serve as a basis for analysis of multi-level data or unified caches, that may also suffer from invalidations, in terms of a multi-core processor.

In order to facilitate the presentation, we introduce the following notations. As described in [19], an abstract cache state is a mapping from a cache set number to an abstract set state, where an abstract set state is a mapping from a position to a set of memory blocks. For the cache C_{l_x}, let $\alpha_{l_x}^{must}$, $\alpha_{l_x}^{may}$, and $\alpha_{l_x}^{pers}$ denote the abstract cache states of C_{l_x} with respect to the cache *must*, *may*, and *persistence* analysis respectively; and let ACS^{must}, ACS^{may}, and ACS^{pers} denote the sets of all of the abstract cache states of these three analyses. For an abstract cache state α_{l_x} (that is either $\alpha_{l_x}^{must}$, $\alpha_{l_x}^{may}$, or $\alpha_{l_x}^{pers}$), let $\alpha_{l_x}(i)$ give the i^{th} abstract set state of α_{l_x}, and let $\alpha_{l_x}(i)(h)$ give the set of memory blocks corresponding to the h^{th} position in $\alpha_{l_x}(i)$.

Let $\mathcal{U}^{must}$ and $\mathcal{J}^{must}$ represent the update and join functions for single-level cache *must* analysis. Similarly, let $\mathcal{U}^{may}$ and $\mathcal{J}^{may}$ represent the update and join functions for single-level cache *may* analysis. These two sets of functions are well-known and defined in [19]. Furthermore, let $\mathcal{U}^{pers}$ and $\mathcal{J}^{pers}$ represent the update and join functions for single-level cache *persistence* analysis. Since the original *persistence* analysis has been known unsafe, we can use the corresponding functions of the safe *persistence* analyses defined in [10] or [5].

For a memory reference r at a cache level l_x, let $m_{l_x}^r$ denote the memory block that contains the information r needs with respect to the cache block size and the number of cache sets in C_{l_x}. We use $m_{l_x}^r \in C_{l_x}$ to denote the needed memory block is in the corresponding concrete set state of C_{l_x}, and use $m_{l_x}^r \in \alpha_{l_x}^t$ to denote the block is in the corresponding abstract set state of t-analysis at this level, where t is either must, may, or persistence.

IV. Multi-Level Inclusive Cache Analysis: Going Top-Down or Bottom-Up?

To our knowledge, existing work analyzes the cache hierarchies in a *top-down* direction, since it is the natural direction of accessing a multi-level cache. As long as there are no invalidations at any cache level, a *top-down* analysis can be safe and precise. However, when there are inclusive caches in the cache hierarchy, a *top-down* analysis cannot capture the possible invalidation behavior precisely, since the invalidations appearing at a cache level are actually caused by the inclusive caches located below this level. Thus, as discussed in [9], conservative decisions have to be made to ensure safety which makes the analysis pessimistic.

In order to make the analysis of multi-level inclusive caches more precise, we propose a safe approach which analyzes the cache hierarchy in a rather counter-intuitive way: We first analyze all the inclusive cache levels in the *bottom-up* direction so as to make the possible invalidation behavior visible at a cache level, and then we analyze all the non-inclusive levels in the traditional *top-down* direction taking into account the revealed invalidations. The analysis process is shown in Fig. 3.

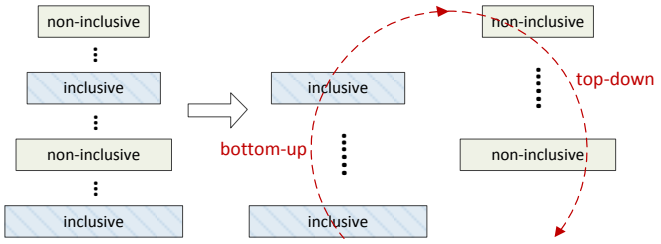

Fig. 3. Multi-level inclusive cache analysis: going *bottom-up* and *top-down*

Our *bottom-up* analysis of inclusive caches is based on the following observation, that is related to the amount of information that can be derived for the access to an inclusive cache level l_y from the state of C_{l_y}.

Lemma 1. *When a memory reference r occurs,*

1) *l_y will be definitely accessed, if $m_{l_y}^r \notin C_{l_y}$.*
2) *l_y will be possibly accessed, if $m_{l_y}^r \in C_{l_y}$.*

Proof: If $m_{l_y}^r$ is not in C_{l_y}, it means all the contents of $m_{l_y}^r$ are not in any C_{l_x} neither, where $l_1 \leq l_x < l_y$, due to

the enforced inclusion property of C_{l_y}; so l_y will be definitely accessed. However, if $m^r_{l_y}$ is already in C_{l_y}, we cannot determine whether there are some sub-blocks of $m^r_{l_y}$ that have the needed contents at above levels only from the state of C_{l_y}, so l_y will be possibly accessed. ■

Based on this lemma, we show that we can first analyze each inclusive level in the *bottom-up* direction safely, and use the results of one inclusive level's analyses to guide its upper levels' analyses to derive more precise CHMC. Note that for a memory reference r which may access the cache level l_x, we can always use $\mathcal{J}^t(\mathcal{U}^t(\alpha^t_{l_x}, m^r_{l_x}), \alpha^t_{l_x})$ to handle the access uncertainty so as to carry out a safe t-analysis at this level, where t is either *must*, *may*, or *persistence* [8]. However, the more uncertainty we can resolve, the more precise the analysis can become.

A. Last Inclusive Cache Analysis

The proposed multi-level inclusive cache analysis begins with the last inclusive cache. There can be other non-inclusive caches located between the last inclusive cache and the main memory. Let us assume the last inclusive cache level corresponds to $l_{\text{LIC}} \in I$, so we have $\forall l_x \in L : x > \text{LIC} \implies l_x \in N$.

1) Last Inclusive Cache May Analysis: At a program point, if a memory block is not in the abstract cache state of a safe *may* analysis of the cache, it is definitely not in any concrete state of the cache. Therefore, if we can safely perform a *may* analysis of the last inclusive cache, we can use the $\alpha^{may}_{l_{\text{LIC}}}$ to safely classify some memory references as *AM* at a cache level l_x where $1 \le x \le \text{LIC}$ based on the inclusion property.

For the *may* analysis of the last inclusive cache, we define the join function $\mathcal{J}^{may}_{\text{LIC}}$ and update function $\mathcal{U}^{may}_{\text{LIC}}$ as follows:

$$\mathcal{J}^{may}_{\text{LIC}} = \mathcal{J}^{may}$$

$$\mathcal{U}^{may}_{\text{LIC}}(\alpha^{may}_{l_{\text{LIC}}}, m^r_{l_{\text{LIC}}}) = \begin{cases} \mathcal{J}^{may}(\mathcal{U}^{may}(\alpha^{may}_{l_{\text{LIC}}}, m^r_{l_{\text{LIC}}}), \alpha^{may}_{l_{\text{LIC}}}) & \text{if } m^r_{l_{\text{LIC}}} \in \alpha^{may}_{l_{\text{LIC}}} \\ \mathcal{U}^{may}(\alpha^{may}_{l_{\text{LIC}}}, m^r_{l_{\text{LIC}}}) & \text{otherwise} \end{cases}$$

where $\mathcal{J}^{may}_{\text{LIC}}$ is the join function of the single-level cache *may* analysis, and the update function $\mathcal{U}^{may}_{\text{LIC}}$ is defined with respect to the two cases in Lemma 1 for a memory reference r: If $m^r_{l_{\text{LIC}}} \notin \alpha^{may}_{l_{\text{LIC}}}$, we can deduce $m^r_{l_{\text{LIC}}} \notin C_{l_{\text{LIC}}}$ (this is formally proven in Lemma 2 in the appendix), so it is certain that l_{LIC} will be accessed, and using $\mathcal{U}^{may}(\alpha^{may}_{l_{\text{LIC}}}, m^r_{l_{\text{LIC}}})$ is certainly safe; if $m^r_{l_{\text{LIC}}} \in \alpha^{may}_{l_{\text{LIC}}}$, $m^r_{l_{\text{LIC}}}$ may be in $C_{l_{\text{LIC}}}$ and l_{LIC} may be accessed, so we use $\mathcal{J}^{may}(\mathcal{U}^{may}(\alpha^{may}_{l_{\text{LIC}}}, m^r_{l_{\text{LIC}}}), \alpha^{may}_{l_{\text{LIC}}})$ to safely update the $\alpha^{may}_{l_{\text{LIC}}}$ by taking into account both the access occurring and not occurring.

Therefore, at a program point, $\alpha^{may}_{l_{\text{LIC}}}$ contains all the memory blocks that are possibly in $C_{l_{\text{LIC}}}$ when the execution reaches this point. If a memory reference r is classified as *AM* by the last inclusive cache *may* analysis (i.e. $m^r_{l_{\text{LIC}}} \notin \alpha^{may}_{l_{\text{LIC}}}$), we can safely categorize r as *AM* at any cache level l_x where $1 \le x \le \text{LIC}$, since, according to the inclusion property, if a memory block is absent from the underlying inclusive cache, it is also absent from all of the included upper-level caches. Therefore, compared to the *top-down* approach proposed in [9], which needs to conservatively change any reference classified as *AM* to *NC* at any cache level, the approach is more precise.

2) Last Inclusive Cache Must and Persistence Analysis: At a program point, the proposed *must* and *persistence* analyses of the last inclusive cache depend on the $\alpha^{may}_{l_{\text{LIC}}}$ of that point. This is because only the information deduced from $\alpha^{may}_{l_{\text{LIC}}}$ can be used to determined whether the l_{LIC} will be definitely accessed according to Lemma 1.

For the last inclusive cache *must* (resp. *persistence*) analysis, we define the join function $\mathcal{J}^{must}_{\text{LIC}}$ (resp. $\mathcal{J}^{pers}_{\text{LIC}}$) and update function $\mathcal{U}^{must}_{\text{LIC}}$ (resp. $\mathcal{U}^{pers}_{\text{LIC}}$) as follows:

$$\mathcal{J}^{must}_{\text{LIC}} = \mathcal{J}^{must}$$

$$\mathcal{U}^{must}_{\text{LIC}}(\alpha^{must}_{l_{\text{LIC}}}, m^r_{l_{\text{LIC}}}) = \begin{cases} \mathcal{J}^{must}(\mathcal{U}^{must}(\alpha^{must}_{l_{\text{LIC}}}, m^r_{l_{\text{LIC}}}), \alpha^{must}_{l_{\text{LIC}}}) & \text{if } m^r_{l_{\text{LIC}}} \in \alpha^{may}_{l_{\text{LIC}}} \\ \mathcal{U}^{must}(\alpha^{must}_{l_{\text{LIC}}}, m^r_{l_{\text{LIC}}}) & \text{otherwise} \end{cases}$$

$$\mathcal{J}^{pers}_{\text{LIC}} = \mathcal{J}^{pers}$$

$$\mathcal{U}^{pers}_{\text{LIC}}(\alpha^{pers}_{l_{\text{LIC}}}, m^r_{l_{\text{LIC}}}) = \begin{cases} \mathcal{J}^{pers}(\mathcal{U}^{pers}(\alpha^{pers}_{l_{\text{LIC}}}, m^r_{l_{\text{LIC}}}), \alpha^{pers}_{l_{\text{LIC}}}) & \text{if } m^r_{l_{\text{LIC}}} \in \alpha^{may}_{l_{\text{LIC}}} \\ \mathcal{U}^{pers}(\alpha^{pers}_{l_{\text{LIC}}}, m^r_{l_{\text{LIC}}}) & \text{otherwise} \end{cases}$$

where $\mathcal{J}^{must}_{\text{LIC}}$ (resp. $\mathcal{J}^{pers}_{\text{LIC}}$) is just the join function of the single-level cache *must* (resp. *persistence*) analysis, and similar to $\mathcal{U}^{may}_{\text{LIC}}$, for a memory reference r, the update function $\mathcal{U}^{must}_{\text{LIC}}$ (resp. $\mathcal{U}^{pers}_{\text{LIC}}$) is defined to safely update the $\alpha^{must}_{l_{\text{LIC}}}$ (resp. $\alpha^{pers}_{l_{\text{LIC}}}$) by using the join function to merge the two abstract cache states (i.e. one state corresponds to the access occurring and the other corresponds to the access not occurring), if $m^r_{l_{\text{LIC}}} \in \alpha^{may}_{l_{\text{LIC}}}$ (i.e. $m^r_{l_{\text{LIC}}}$ is possibly in $C_{l_{\text{LIC}}}$); otherwise, $m^r_{l_{\text{LIC}}}$ is definitely not in $C_{l_{\text{LIC}}}$, so it can more precisely update the abstract cache state by knowing the access definitely occurs.

Thus, at any program point, the memory blocks contained in $\alpha^{must}_{l_{\text{LIC}}}$ are definitely in $C_{l_{\text{LIC}}}$, and the memory blocks not contained in the $\top$ age positions of $\alpha^{pers}_{l_{\text{LIC}}}$ are persistent when the execution reaches this point. If a memory reference r is classified as *AH* by the last inclusive cache *must* analysis (i.e. $m^r_{l_{\text{LIC}}} \in \alpha^{must}_{l_{\text{LIC}}}$), this reference will cause no cache misses at l_{LIC}, but may result in misses at a cache level l_x where $1 \le x < \text{LIC}$. In other words, this classification for this memory reference is only locally safe. If r is classified as *AH* by the last inclusive cache *must* analysis, no memory blocks need to be evicted from C_{l_x} because of this reference, so no invalidations are enforced by $C_{l_{\text{LIC}}}$. Similarly, if a memory reference r is classified as *PS* by the last inclusive cache *persistence* analysis (i.e. $m^r_{l_{\text{LIC}}}$ is not in $\top$ of the corresponding set of $\alpha^{pers}_{l_{\text{LIC}}}$), r will result in at most one cache miss at l_{LIC}, but may cause more than one misses at a cache level l_x where $1 \le x < \text{LIC}$. Finally, if r is classified as *PS* by the last inclusive cache *persistence* analysis, at most one memory block will be evicted from $C_{l_{\text{LIC}}}$ so that at most one invalidation enforcement can be caused because of r.

B. Aging Barriers

In order to analyze a cache located above an inclusive cache level more precisely, the effects of the invalidations need to be captured. Since the invalidations are caused by lower inclusive caches, compared to the *top-down* approach, one advantage of

the *bottom-up* approach is the invalidation behavior becomes visible when analyzing an upper level.

At a cache level, if a memory block is invalidated due to the maintenance of the inclusion property, 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 ages of the memory blocks that are behind this "hole". Yet, it does not mean the age of a memory block behind the "hole" will not be decreased, since a reference to such a block will decrease its age to 1 and fill the "hole", in which case another "hole" will be created behind the filled "hole". A "hole" will be filled and no new one will be created when the referenced memory block is not in the cache.

We propose a concept of aging barrier to capture this "hole" behavior so as to perform more precise *must* and *persistence* analyses of a cache that may suffer from invalidations. Without loss of generality, we present the concept in terms of an A-way set associative cache C which has s cache sets.

Definition 1 (Aging Barrier)**.** *A valid aging barrier (i, j) satisfies $1 \leq i \leq s \bigwedge 1 \leq j \leq A$, and represents an unused position within the range $[1, j]$ in the i^{th} cache set, which prevents the age of any memory block in the i^{th} abstract set state of α^{must} or α^{pers} from increasing if the age is already greater than or equal to j for an access.*

We treat an aging barrier (i, j) as an abstract *must* "hole": if there is a valid aging barrier (i, j) at a program point, in any concrete state of C, there must be a corresponding real "hole" appearing in the i^{th} cache set of C within the position range $[1, j]$. Thus, j serves as the position upper bound of the real "hole". For example, the aging barrier $(1, 2)$ represents either the 1^{st} or the 2^{nd} young memory block in the 1^{st} cache set is invalidated and the position it occupied becomes available.

It is possible to have multiple valid aging barriers with respect to the i^{th} cache set, which are listed as $(i, j_1), \cdots, (i, j_k)$ where $k \geq 1$. In that case, there are certainly at least k real "holes" in the i^{th} cache set, whose positions are bounded by $j_1, \cdots, j_k$ respectively. Note that it is valid to have multiple identical j's with respect to the i^{th} cache set, as long as the multiset [3] formed by these upper bounds satisfies the condition: Given any position pos in the cache set, the total number of j's with $j \leq pos$ is at most pos. Let Ξ denote the set of all of the valid multisets formed by "hole" position upper bounds of a cache set. Formally, we have:

$$\xi \in \Xi \iff \max(\xi) \leq A \bigwedge \forall pos \in \{1, \cdots, A\} : \sum_{j=1}^{pos} \nu(\xi, j) \leq pos$$

where $\max(\xi)$ gives the maximum member and $\nu(\xi, j)$ gives the multiplicity of j in the multiset ξ.

Definition 2 (Aging Barrier State)**.** *An aging barrier state $\beta : \{1, \cdots, s\} \to \Xi$ is a mapping from a cache set number to a multiset of "hole" position upper bounds.*

Given an aging barrier state β, the set of all the valid aging barriers is $\{(i, j)^{\nu(\beta(i), j)} | i \in \{1, \cdots, s\} \bigwedge j \in \beta(i)\}$, which is a multiset and uses $\nu(\beta(i), j)$ as the multiplicity of (i, j). Let ABS denote the set of all the aging barrier states of C. We define three functions to operate on the aging barrier states.

Let $\top = A + 1$ be the invalid aging barrier indicator. The function $\mathscr{A} : ABS \times \{1, \cdots, s\} \times \{1, \cdots, A, \top\} \to ABS$ is used to add an aging barrier into the state and is defined as: [4]

$$\mathscr{A}(\beta, i, j) = \beta\big[i \mapsto \beta(i) \uplus_c \{j\}\big]$$

$$\text{where } \beta(i) \uplus_c \{j\} = \begin{cases} \beta(i) \uplus \{j\} & \text{if } \beta(i) \uplus \{j\} \in \Xi \\ \beta(i) & \text{otherwise} \end{cases}$$

The function adds the aging barrier (i, j) into the state β only if the result of $\beta(i) \uplus \{j\}$ ($\uplus$ is the multiset sum operation) is a member of Ξ; otherwise, it keeps β unchanged. For example, given a 4-way set associative cache (i.e. A is 4), when we want to add an aging barrier $(1, 3)$ into the state β, the function $\mathscr{A}$ needs to check if $\beta(1) \uplus \{3\}$ is a member of Ξ. Assume we have $\beta(1) = \{2, 2\}$; then $\beta(1) \uplus \{3\} = \{2, 2, 3\}$ is a member of Ξ according to the condition given above – the maximum member in $\{2, 2, 3\}$ is 3 that is less than 4 and no matter what pos is, the total number of the members that are less than or equal to pos is at most pos. Therefore, after applying $\mathscr{A}(\beta, 1, 3)$, we will have $\beta(1) = \{2, 2, 3\}$.

The function $\mathscr{U} : ABS \times \{1, \cdots, s\} \to ABS \times \{1, \cdots, A, \top\}$ is used to acquire an aging barrier from the state and is defined as:

$$\mathscr{U}(\beta, i) = \big\langle \beta\big[i \mapsto \beta(i) \backslash \{\min_c(\beta(i))\}\big], \min_c(\beta(i)) \big\rangle$$

$$\text{where } \min_c(\beta(i)) = \begin{cases} \min(\beta(i)) & \text{if } \beta(i) \neq \emptyset \\ \top & \text{otherwise} \end{cases}$$

Given a cache set number i, the resultant aging barrier depends on whether the mapped multiset $\beta(i)$ is empty: If $\beta(i)$ is not empty, $\min_c(\beta(i))$ equals $\min(\beta(i))$ that is the minimum member in $\beta(i)$, and the composite $(i, \min(\beta(i)))$ will be a valid aging barrier; otherwise, $\min_c(\beta(i))$ equals $\top$ and there is no valid aging barrier for the i^{th} cache set. Since a valid aging barrier may be acquired in which case this aging barrier should no longer be in the state, the function changes the state by mapping i to $\beta(i) \backslash \{\min_c(\beta(i))\}$ ($\backslash$ is the multiset asymmetric difference operation). For example, let us continue with the last example in which we have $\beta(1) = \{2, 2, 3\}$. Since the minimum member in $\{2, 2, 3\}$ is 2, after applying $\mathscr{U}(\beta, 1)$, we have a valid aging barrier $(1, 2)$ and $\beta(1)$ becomes $\{2, 3\}$.

The function $\mathscr{J} : ABS \times ABS \to ABS$ is used to join two aging barrier states and is defined as:

$$\mathscr{J}(\beta_1, \beta_2) = \big[i \mapsto \beta_1(i) \sqcap_c \beta_2(i) | i = 1, \cdots, s\big]$$

$$\text{where } \beta_1(i) \sqcap_c \beta_2(i) = \begin{cases} \emptyset & \text{if } \beta_1(i) = \emptyset \bigvee \beta_2(i) = \emptyset \\ \{j_1, \cdots, j_k\} & \text{otherwise} \end{cases}$$

$$\text{where } k = \min(|\beta_1(i)|, |\beta_2(i)|) \bigwedge$$
$$j_1 = \max(\min_c(\beta_1(i)), \min_c(\beta_2(i)) \bigwedge$$
$$j_2 = \max(\min_c^2(\beta_1(i)), \min_c^2(\beta_2(i)) \bigwedge$$
$$\cdots$$
$$j_k = \max(\min_c^k(\beta_1(i)), \min_c^k(\beta_2(i)))$$

[3] A multiset is a set in which members are allowed to appear more than once.

[4] For a function $f : X \to Y$, $f\big[i \mapsto k\big]$ means $f(i) = k \bigwedge \forall x \in X \wedge x \neq i : f(x) = f(x)$.

where $\min_c^k(\beta(i))$ is similar to $\min_c(\beta(i))$ except it gives the k^{th} minimum member of $\beta(i)$ if $\beta(i)$ has at least k members (of course, if $\beta(i)$ does not have that many members, it gives $\top$). When joining two aging barrier states, for the i^{th} cache set, the cardinality of $\beta_1(i) \sqcap_c \beta_2(i)$ (i.e. k) is the smaller one of the cardinalities of $\beta_1(i)$ and $\beta_2(i)$, which implies the number of aging barriers that can be derived from $\mathscr{J}(\beta_1, \beta_2)$ will never exceed that derived from either β_1 or β_2. In the case of $k \geq 1$, j_1 is the bigger one between the two minimum members of $\beta_1(i)$ and $\beta_2(i)$, which safely captures an aging barrier since there must be a "hole" within position range $[1, j_1]$ along either path; and j_2 is the bigger one between the 2^{nd} minimum members of $\beta_1(i)$ and $\beta_2(i)$. We repeat this process until we have j_k which is the bigger one between the k^{th} minimum members of $\beta_1(i)$ and $\beta_2(i)$. For example, assume we have $\beta_1(1) = \{2, 2\}$ and $\beta_2(1) = \{1, 3, 4\}$. After applying $\beta = \mathscr{J}(\beta_1, \beta_2)$, we will have $\beta(1) = \{2, 3\}$, since, for the 1^{st} cache set, we have $k = 2$, $j_1 = 2$, and $j_2 = 3$ when performing $\{2, 2\} \sqcap_c \{1, 3, 4\}$.

Definition 3 (Partial Ordering)**.** *Let β_1 and β_2 be two aging barrier states. We define $\beta_1 \sqsubseteq \beta_2$ if and only if* $\forall i \in \{1, \cdots, s\}$: $|\beta_2| \leq |\beta_1| \bigwedge \min_c(\beta_1(i)) \leq \min_c(\beta_2(i)) \bigwedge \min_c^2(\beta_1(i)) \leq \min_c^2(\beta_2(i)) \bigwedge \cdots \bigwedge \min_c^{|\beta_2|}(\beta_1(i)) \leq \min_c^{|\beta_2|}(\beta_2(i))$.

Therefore, we have $\beta_1 \sqsubseteq \beta_2$, if and only if, for any cache set i, the mapped multisets $\beta_1(i)$ and $\beta_2(i)$ satisfy: the number of members of $\beta_2(i)$ is not greater than that of $\beta_1(i)$, and when we iterate the two multisets in the ascending order in parallel, the iterated number from $\beta_2(i)$ is not smaller than that from $\beta_1(i)$. According to Definition 3, we can deduce that $\beta_1 \sqsubseteq \mathscr{J}(\beta_1, \beta_2)$ and $\beta_2 \sqsubseteq \mathscr{J}(\beta_1, \beta_2)$. Let $\beta_\perp = \big[i \mapsto \{1, \cdots, A\} | i = 1, \cdots, s\big]$ and $\beta_\top = \big[i \mapsto \emptyset | i = 1, \cdots, s\big]$; thus, according to Definition 3, we can deduce that $\forall \beta \in ABS : \beta_\perp \sqsubseteq \beta \sqsubseteq \beta_\top$.

C. Integrating Aging Barriers into Update Functions

In order to realize more precise *must* and *persistence* analyses of the caches which suffer from invalidations, we need to integrate the aging barriers into the update functions of these analyses. Let M denote the set of all the memory blocks. Given a reference to a memory block $m \in M$ that is mapped to the i^{th} set of C and an aging barrier (i, j), where $j \in \{1, \cdots, A, \top\}$ (recall that we use $\top$ as the invalid aging barrier indicator), we redefine the update function $\overline{\mathcal{U}}^{must} : ACS^{must} \times M \times \{1, \cdots, A, \top\} \to ACS^{must}$ for the *must* analysis as:

$$\overline{\mathcal{U}}^{must}(\alpha^{must}, m, j) = \begin{cases} \mathcal{U}^{must}(\alpha^{must}, m) & \text{if } k \leq j \\ \alpha^{must}\big[i \mapsto \epsilon_i\big] & \text{otherwise} \end{cases}$$

$$\text{where } k = \begin{cases} h & \text{if } m \in \alpha^{must}(i)(h) \\ \top & \text{otherwise} \end{cases} \bigwedge$$

$$\epsilon_i = \begin{array}{l} \big[\iota_1 \mapsto \{m\}, \\ \quad \iota_n \mapsto \alpha^{must}(i)(\iota_{n-1}) | n = 2, \cdots, j-1, \\ \quad \iota_j \mapsto \alpha^{must}(i)(\iota_j) \cup \alpha^{must}(i)(\iota_{j-1}), \\ \quad \iota_n \mapsto \alpha^{must}(i)(\iota_n) \backslash \{m\} | n = j+1, \cdots, A\big] \end{array}$$

The rationale of the redefined update function is: If there is no valid aging barrier available (i.e. $j = \top$), or if the current valid aging barrier (i, j) is not needed (i.e. $m \in \alpha^{must}(i)(h) \bigwedge j \leq A \bigwedge h \leq j$, in which case this update never attempts to affect the ages of the memory blocks "protected" behind this aging barrier), then we can simply use the $\mathcal{U}^{must}$ to update the α^{must}; otherwise, the current aging barrier can prevent the memory blocks that are behind it in the corresponding abstract set state from aging, since it means there is a "hole" before j (including j) that needs to be filled, and we can only increase the ages of the memory blocks until j, and keep the ages of other blocks not increased (excluding m which will be moved to the first age position if it is in the current state). Fig. 4 shows an example of using an aging barrier to update α^{must} more precisely – if m_c in α^{must} is invalidated, since it is definitely in the cache before the invalidation with an overestimated maximal age 3, a "hole" will definitely appear within the range $[1, 3]$, namely we have an aging barrier with $j = 3$; when m_d is referenced, even if it is not in the cache, there is a "hole" to fill, the maximal ages of m_b and m_a should not be increased. Therefore, using the redefined function $\overline{\mathcal{U}}^{must}$ leads to more precise analysis.

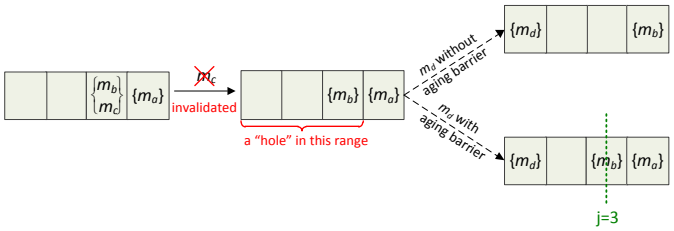

Fig. 4. Must analysis with aging barriers

Similarly, when updating α^{pers}, given an aging barrier (i, j) and the k which is the affected position range upper bound when applying the normal $\mathcal{U}^{pers}$, if we have $k \leq j$, we simply perform the normal $\mathcal{U}^{pers}$; otherwise, we know in any concrete state of C there will be a "hole" in the i^{th} cache set within the position range $[1, j]$, so we can take advantage of this information to carry out a more precise update. We redefine the update function $\overline{\mathcal{U}}^{pers} : ACS^{pers} \times M \times \{1, \cdots, A, \top\} \to ACS^{pers}$ for the *persistence* analysis. When performing $\overline{\mathcal{U}}^{pers}(\alpha^{pers}, m, j)$, if we have $j < k$, for the memory blocks whose maximal ages are already greater than or equal to j in the i^{th} abstract set state, their ages will not be increased (but one of them may be decreased to 1 if that block is the referenced one).

We maintain an aging barrier state for each cache which is located above at least one inclusive cache level so as to achieve more precise analysis (described in the next subsection).

D. Cache Analysis above One Inclusive Cache Level

When the last inclusive cache analysis is finished, we move up to the second last inclusive level if there is any; otherwise, we start from the first cache level l_1 and move down to analyze the non-inclusive caches. No matter which level l_x (where $1 \leq x < \text{LIC}$) we are going to analyze, this level is located above at least one inclusive cache level (i.e. the last inclusive cache level l_{LIC}), so the cache at this level may suffer from invalidations caused by the underlying inclusive cache(s).

When we analyze C_{l_x} with respect to the CFG of the program, at a join point, given the abstract cache states $\alpha^t_{l_x,1}, \alpha^t_{l_x,2}$ of the exit points of two predecessors, where t is either *may*, *must*, or *persistence*, we can simply perform $\mathcal{J}^t(\alpha^t_{l_x,1}, \alpha^t_{l_x,2})$ to safely join the abstract cache states. However, at a program point in a basic block where r is the reference that is going to occur, we need to take into account the invalidation behavior

to safely update the abstract cache state of the corresponding analysis.

In order to facilitate the presentation, in the following, let us assume l_y is the uppermost inclusive level that includes l_x, and all the abstract states (i.e. $\alpha_{l_x}^{may}$ $\alpha_{l_x}^{must}$, $\alpha_{l_x}^{pers}$, and β_{l_x}) and all the arguments (e.g. the number of cache sets s_{l_x} and the associativity A_{l_x}) at the cache level l_x are the attributes of C_{l_x}.

Since we first analyze the inclusive caches in the *bottom-up* direction, the analyses of C_{l_y} are already completed at the time of analyzing C_{l_x}, and these analyses of C_{l_y} have captured the possible invalidations caused by the inclusive levels lower than l_y if there are any. Thus, from $\alpha_{l_y}^{may}$, we can deduce whether the contents of a memory block are definitely absent from C_{l_y}, and from $\alpha_{l_y}^{pers}$, we can deduce whether the contents of a memory block are possibly absent from C_{l_y}. Thus, we only need to check l_x against l_y and not any other lower inclusive cache levels.

1) May Analysis: As described in [9], it is unsafe to update the abstract cache state $\alpha_{l_x}^{may}$ without considering the possible invalidations caused by its underlying inclusive levels, since there possibly exist some "holes" so that some memory blocks at l_x may live longer. Fortunately, since we first analyze all the inclusive caches in the *bottom-up* direction, when we analyze C_{l_x}, the invalidation behavior induced by its underlying inclusive levels has already become visible.

First, let us redefine the update function $\overline{U}^{may} : ACS^{may} \times M \times \{1, \cdots, A, \top\} \rightarrow ACS^{may}$ for the *may* analysis of C_{l_x} that is located above the inclusive level l_y. Similar to the $\overline{\mathcal{U}}^{must}$ and $\overline{\mathcal{U}}^{pers}$ described in IV-C, given a memory reference r, in the $\overline{\mathcal{U}}^{may}(\alpha_{l_x}^{may}, m_{l_x}^r, j)$, j controls the upper bound on the aging process. However, different from the $\overline{\mathcal{U}}^{must}$ and $\overline{\mathcal{U}}^{pers}$, where j is given by an aging barrier, here j is decided by finding the *youngest* position in which there is a possible inclusion victim (i.e. there is *possibly* a "hole" within the range $[j, A_{l_x}]$ if such a j can be found). Thus, if we have $j = \top$, we just perform the normal $\mathcal{U}^{may}$; otherwise, for the memory blocks whose ages are already greater than or equal to j, their ages will not be increased (but may be decreased to 1 by the reference). The steps to update $\alpha_{l_x}^{may}$ are given in Algorithm 1.

Algorithm 1: Update $\alpha_{l_x}^{may}$ above an inclusive level l_y

Input: r, l_x, l_y
Result: updated $\alpha_{l_x}^{may}$
1 $i \leftarrow m_{l_x}^r$ mapped set number;
2 $j \leftarrow \top$;
3 $k \leftarrow 1$;
4 **for** $j = \top \wedge k \leq A_{l_x}$ **do**
5 **if** *the contents of a memory block* $m_{l_x} \in \alpha_{l_x}^{may}(i)(k)$ *are possibly evicted according to* $\alpha_{l_y}^{pers}$ *after* r **then** $j \leftarrow k$;
6 **else** $k \leftarrow k + 1$;
7 **if** l_x *is inclusive* **then**
8 **if** $m_{l_x}^r \notin \alpha_{l_x}^{may}(i)$ **then** $\alpha_{l_x}^{may} \leftarrow \overline{\mathcal{U}}^{may}(\alpha_{l_x}^{may}, m_{l_x}^r, j)$;
9 **else** $\alpha_{l_x}^{may} \leftarrow \mathcal{J}^{may}(\overline{\mathcal{U}}^{may}(\alpha_{l_x}^{may}, m_{l_x}^r, j), \alpha_{l_x}^{may})$;
10 **else**
11 get CAC for r at l_x from CHMC and CAC at l_{x-1};
12 **if** CAC *is always* **then** $\alpha_{l_x}^{may} \leftarrow \overline{\mathcal{U}}^{may}(\alpha_{l_x}^{may}, m_{l_x}^r, j)$;
13 **else if** CAC *is never* **then** $\alpha_{l_x}^{may} \leftarrow \alpha_{l_x}^{may}$;
14 **else** $\alpha_{l_x}^{may} \leftarrow \mathcal{J}^{may}(\overline{\mathcal{U}}^{may}(\alpha_{l_x}^{may}, m_{l_x}^r, j), \alpha_{l_x}^{may})$;

The first loop (line 4-6) checks whether there is a memory block m_{l_x} whose contents are in a block located in a $\top$ position of $\alpha_{l_y}^{pers}$ after the reference r (i.e. $\alpha_{l_y}^{pers}$ has taken into account the effect of the reference), namely it checks if m_{l_x} is a sub-block of a possibly evicted memory block due to the reference at l_y. If there is such a block found in a position $k \leq A_{l_x}$, increasing the ages of the memory blocks which are not less than k may make the *may* analysis unsafe (since there may be a "hole" within the range $[k, A_{l_x}]$), so we set j as the *youngest* k; otherwise, j is $\top$.

If l_x is an inclusive level (line 7-9), we are still moving up in the cache hierarchy, so it is not possible to decide the access occurrence by using the traditional CAC method. Therefore, like in the last inclusive cache analyses, the algorithm checks against itself (i.e. $\alpha_{l_x}^{may}$) to find out if the memory block $m_{l_x}^r$ referenced by r is possibly in the cache. If not, this inclusive level will be definitely accessed, so we update $\alpha_{l_x}^{may}$ directly; otherwise, we have to safely update $\alpha_{l_x}^{may}$ by taking into account the two cases (i.e. access occurring and not occurring). If l_x is a non-inclusive level (line 10-14), we have already analyzed all the inclusive levels and are moving down in the cache hierarchy. Therefore, no matter which type l_{x-1} is, where $x > 1$ (when l_x is l_1, it is always accessed), the analyses of $C_{l_{x-1}}$ have been completed, so it is possible to derive the CAC for r at l_x from the CHMC and CAC for r at l_{x-1}, and then to update the $\alpha_{l_x}^{may}$ according to the derived CAC.

The last step is to update $\alpha_{l_x}^{may}$ by removing all the memory blocks whose contents are definitely not in $\alpha_{l_x}^{may}$. We perform this step by referring to the contents of $\alpha_{l_y}^{may}$ at the same point, after the *may* analysis of C_{l_x} is completed (i.e. at each program point, its $\alpha_{l_x}^{may}$ has reached a fixed-point).

2) Must Analysis: In the *must* analysis of C_{l_x}, we maintain both the abstract cache state $\alpha_{l_x}^{must}$ and the aging barrier state β_{l_x}. As we discussed above, at a join point, we simply perform $\mathcal{J}^{must}(\alpha_{l_x,1}^{must}, \alpha_{l_x,2}^{must})$ to safely join two abstract cache states. Similarly, given two aging barrier states $\beta_{l_x,1}, \beta_{l_x,2}$, we simply perform $\mathscr{J}(\beta_{l_x,1}, \beta_{l_x,2})$ to join these two aging barrier states. At a program point in a basic block, we update the $\alpha_{l_x}^{must}$ and β_{l_x} following the steps described in Algorithm 2.

The loop (line 1-7) first checks whether a memory block in $\alpha_{l_x}^{must}$ is definitely an inclusion victim (i.e. the contents of the block are not in $\alpha_{l_y}^{may}$ after the reference r). If there is such a block, there will be a "hole" created by removing this block from $\alpha_{l_x}^{must}$, since it was definitely in the cache C_{l_x} before the reference r. Thus, we add an aging barrier corresponding to this certainly invalidated block into β_{l_x} (line 3-6). In order to guarantee safety of the *must* analysis, the algorithm also (line 7) takes into account all the possibly evicted memory blocks by removing them from the $\alpha_{l_x}^{must}$.

In the next steps, we first acquire an aging barrier (i, j) by applying $\langle \beta'_{l_x}, j \rangle = \mathscr{U}(\beta_{l_x}, i)$ (line 9). Since l_x can be either inclusive or non-inclusive, line 11-18 take into account the two possibilities, which is similar to the corresponding steps in the *may* analysis. A valid aging barrier (i, j) (i.e. we have $j \neq \top$) means there must be a "hole" in the i^{th} cache set within the position range $[1, j]$, different from that in Algorithm 1 where j is chosen to be the position lower bound of a possible "hole". After updating $\alpha_{l_x}^{must}$, we update the aging barrier state by performing $\mathscr{A}(\beta'_{l_x}, i, \max(j, k))$ to add an aging barrier back to the

Algorithm 2: Update $\alpha_{l_x}^{must}$ and β_{l_x} above an inclusive level l_y

Input: r, l_x, l_y
Result: updated $\alpha_{l_x}^{must}$, updated β_{l_x}
1 **foreach** *memory block* $m_{l_x} \in \alpha_{l_x}^{must}$ **do**
2 **if** *the contents of* m_{l_x} *are definitely evicted according to* $\alpha_{l_y}^{may}$ *after* r **then**
3 $i \leftarrow m_{l_x}$ mapped set number;
4 $j \leftarrow$ the position where m_{l_x} is in $\alpha_{l_x}^{must}(i)$;
5 $\beta_{l_x} \leftarrow \mathscr{A}(\beta_{l_x}, i, j)$;
6 remove m_{l_x} from $\alpha_{l_x}^{must}$;
7 **else if** *the contents of* m_{l_x} *are possibly evicted according to* $\alpha_{l_y}^{pers}$ *after* r **then** remove m_{l_x} from $\alpha_{l_x}^{must}$;
8 $i \leftarrow m_{l_x}^r$ mapped set number;
9 $\langle \beta'_{l_x}, j \rangle \leftarrow \mathscr{U}(\beta_{l_x}, i)$;
10 $k \leftarrow$ the position where $m_{l_x}^r$ is in $\alpha_{l_x}^{must}(i)$ ($\top$, if not found);
11 **if** l_x *is inclusive* **then**
12 **if** $m_{l_x}^r \notin \alpha_{l_x}^{may}$ **then** $\alpha_{l_x}^{must} \leftarrow \overline{\mathcal{U}}^{must}(\alpha_{l_x}^{must}, m_{l_x}^r, j)$;
13 **else** $\alpha_{l_x}^{must} \leftarrow \mathcal{J}^{must}(\overline{\mathcal{U}}^{must}(\alpha_{l_x}^{must}, m_{l_x}^r, j), \alpha_{l_x}^{must})$;
14 **else**
15 get CAC for r at l_x from CHMC and CAC at l_{x-1};
16 **if** CAC *is always* **then** $\alpha_{l_x}^{must} \leftarrow \overline{\mathcal{U}}^{must}(\alpha_{l_x}^{must}, m_{l_x}^r, j)$;
17 **else if** CAC *is never* **then** $\alpha_{l_x}^{must} \leftarrow \alpha_{l_x}^{must}$;
18 **else** $\alpha_{l_x}^{must} \leftarrow \mathcal{J}^{must}(\overline{\mathcal{U}}^{must}(\alpha_{l_x}^{must}, m_{l_x}^r, j), \alpha_{l_x}^{must})$;
19 $\beta_{l_x} \leftarrow \mathscr{A}(\beta'_{l_x}, i, \max(j, k))$;

state (line 19): (1) If we have $k \leq j$, we perform the normal update function $\mathcal{U}^{must}$, and line 19 will add the acquired aging barrier back to the aging barrier state (since we have $k \leq j$, $\max(j,k)$ is always j, and no matter whether j is $\top$ or not, after line 19 the β_{l_x} will be the same as the input β_{l_x}) – in the case of $j \neq \top$, the acquired aging barrier is valid, since we have $k \leq j$, the "hole" represented by the aging barrier has not been filled yet, so after line 19, β_{l_x} becomes the same as the input β_{l_x}; in the case of $j = \top$, no valid aging barrier has been acquired from the input β_{l_x} at line 9, so β'_{l_x} was still the same as the input β_{l_x}, and after line 19, β_{l_x} is the same as β'_{l_x} as well as the input β_{l_x}. (2) If we have $j < k = \top$, it means the referenced memory block $m_{l_x}^r$ is not in the i^{th} set state of α^{must} (since $k = \top$), and the acquired aging barrier is valid (i.e. $j \neq \top$); so $m_{l_x}^r$ intends to fill the "hole" represented by this valid aging barrier; since we have $\max(j,k) = k = \top$, $\mathscr{A}(\beta'_{l_x}, i, \top)$ will not change the state β'_{l_x} which represents the valid aging barrier has already been used. (3) If we have $j < k < \top$, it means $m_{l_x}^r$ is definitely present in any concrete state, so no other memory blocks will be loaded due to this reference, and we can safely guarantee there will be a "hole" in the range $[1, k]$, even if the "hole" that was in the range $[1, j]$ has been filled; we have $\max(j,k) = k < \top$, and (i, k) is an valid aging barrier; so $\mathscr{A}(\beta'_{l_x}, i, k)$ will add the new valid aging barrier into the state β'_{l_x}.

3) Persistence Analysis: For the *persistence* analysis, the steps to update $\alpha_{l_x}^{pers}$ are similar to the steps in Algorithm 2. The differences are: (i) We set j according to the aging barrier state β_{l_x} maintained by the *must* analysis of C_{l_x}, but we do not change β_{l_x} in the steps, namely we only use the fact that if there is a valid aging barrier available before executing the reference, there is a "hole" within the position range $[1, j]$; (ii) We do not remove memory blocks from $\alpha_{l_x}^{pers}$, but for any memory block in $\alpha_{l_x}^{pers}$ which is not in the $\top$ position yet, if its contents are not in $\alpha_{l_y}^{may}$ after the reference r or its contents are in a $\top$ position of $\alpha_{l_y}^{pers}$ after the reference r, move it to the corresponding set's $\top$ position.

There can also be some non-inclusive caches located below the last inclusive cache level, but they do not suffer from any invalidation. When moving down in the cache hierarchy, the analysis of any of them is the same as the traditional multi-level non-inclusive cache analysis. Theoretical analysis of the approach's **safety** and **termination** is provided in the appendix.

V. Evaluation

The objective of this paper is to tighten the WCET estimation in the presence of inclusive caches. We evaluate the proposed approach and compare with the approach proposed in [9]. In order to analyze the effects of multi-level inclusive caches, we developed a research prototype tool, which reconstructs the CFG from the binary executable of the program and recursively derives the fixed-points of the abstract cache states at each level. Currently, our tool does not distinguish calling contexts, so overestimations are possible. However, in terms of precision, handling contexts is orthogonal to the problem considered in this paper.

In order to calculate the WCET bound, we apply the widely used IPET (Implicit Path Enumeration Technique) [14]. IPET uses a set of integer linear constraints to combine the flow information and the timing effects of the multi-level caches [9, 12]. In terms of the flow information, the structural constraints are generated directly, but currently the loop bounds need to be determined and input manually in our tool. The CPLEX solver is used to solve the generated ILP (Integer Linear Programming) problems.

Due to the limitations of our current tool, we only take into account the timing effects of multi-level caches on the WCET estimation and do not consider the effects of other micro-architectural components like pipelines and branch predictors, so we assume there are no timing anomalies. Therefore, a reference that is classified as *NC* can be safely treated as a *AM* when used to estimate the WCET. However, if the timing anomalies are considered, we will gain more precision using the proposed approach, since it can safely classify some references as *AM* compared to the approach in [9]. We leave this as future work.

Our experiments are carried out on the set of benchmarks maintained by the Mälardalen WCET research group [6], and they are compiled for MIPS R3000 processor using gcc-3.4.4. Since the approach proposed in [9] only considers strict multi-level inclusive caches (i.e. it does not consider mixed inclusive and non-inclusive cache levels), we carry out the experiments on a three-level cache hierarchy and configure L2 and L3 to be inclusive. The parameters of the cache at each level are shown in Tab. I. Moreover, we assume every needed information can be found in the main memory with a 200-cycle latency.

TABLE I. 3-level inclusive cache parameters

Level	Cache Capacity	Block Size	Associativity	Latency
L1	2KB	8B	4-way	1-cycle
L2	8KB	32B	8-way	10-cycle
L3	16KB	64B	8-way	80-cycle

The experimental results are shown in Tab. II. For a benchmark, $\text{WCET}_{\text{top-dw}}$ is derived by using the method proposed in [9], and $\text{WCET}_{\text{bot-up}}$ is derived by using the method proposed

in this paper. The WCET estimation is reported in clock cycles. The precision improvement is calculated by $\frac{\text{WCET}_{\text{top-dw}}}{\text{WCET}_{\text{bot-up}}} - 1$. We also report the computation time overhead in seconds, along with the reported WCET. The experiments are performed on a Linux machine with a $1.2GHz$ quad-core processor and $12GB$ memory.

We sort Tab. II in descending order of the precision improvement. From the results, we can see that the bound can be tightened about 12.2% on average. In some cases, the improvement is more than 20%, e.g. up to 57.3% is gained in the case of *fibcall* and up to 44.4% is gained in the case of *insertsort*. For some benchmarks, the improvement rate is not that substantial (less than 3%), e.g. only 2.7% is gained in the case of *ludcmp* and only 2.4% is gained in the case of *adpcm*. We find most of these benchmarks contain nested loops and/or are context-sensitive. The advantage of the proposed method may become larger if the *persistence* analysis is multi-leveled to handle the nested loops [2] and contexts are taken into account in the inter-procedural analysis. Furthermore, as mentioned above, our prototype tool does not analyze other micro-architectural features than multi-level caches for the present. Since the proposed approach can classify some references as *AM* while the method in [9] cannot, we would expect more precision gains if timing anomalies are considered. Although these techniques are not integrated in our tool yet, the improvement is still significant. Even in some cases the improvement rate is less than 3%, thousands of overestimated cycles are reduced (e.g. up to 12400 clock cycles are reduced in the case of *adpcm*). However, it should be noted that the proposed approach is standalone and can be integrated with other techniques without any changes.

TABLE II. EXPERIMENT RESULTS OF ESTIMATED WCET AND COMPUTATION TIME OVERHEAD

Benchmark	Method in [9]		Method in This Paper		Precis. Improv.
	Ovhd.	$\text{WCET}_{\text{top-dw}}$	Ovhd.	$\text{WCET}_{\text{bot-up}}$	
fibcall	0	7250	0	4610	57.3%
insertsort	0	18349	0	12709	44.4%
recursion	0	6942	0	4982	39.3%
bs	0	10979	0	8579	28.0%
fir	0	28046	1	22406	25.2%
sqrt	0	19662	0	15742	24.9%
janne_cmplx.	0	11367	0	9247	22.9%
cnt	0	43138	1	35778	20.6%
ns	0	45731	0	38131	19.9%
duff	0	23169	1	19609	18.2%
prime	0	31690	0	27890	13.6%
edn	3	303483	4	272123	11.5%
expint	0	35855	0	32775	9.4%
qurt	0	41122	1	37922	8.4%
statemate	17	404050	31	377550	7.0%
lcdnum	0	18939	0	17819	6.3%
fdct	1	92089	1	88329	4.3%
minver	5	111053	5	106533	4.2%
select	3	63744	3	61344	3.9%
compress	13	299514	14	288514	3.8%
cover	9	187579	10	182259	2.9%
ludcmp	3	87526	3	85206	2.7%
qsort_exam	3	69903	5	68063	2.7%
adpcm	41	522619	42	510219	2.4%
ndes	10	737997	11	728637	1.3%
bsort100	0	287104	1	281904	1.8%
st	6	380532	5	374572	1.6%
jfdctint	1	99865	1	98465	1.4%
matmult	0	513672	0	508032	1.1%
crc	1	95794	0	95274	0.5%
lms	5	1226776	6	1221496	0.4%
nsichneu	383	2985648	476	2985088	0.02%
average					12.2%

From the results, we can see the computation time overhead differences between the two methods are within a few seconds in most cases. The biggest difference is about 93 seconds in the case of *nsichneu*. Since this difference is just a small portion of the overheads, which are 6.4 and 7.9 minutes respectively, we believe the computation time overhead is acceptable.

VI. RELATED WORK

Abstract interpretation based single-level cache analysis has been widely used in WCET analysis [19]. However, it has been found its original persistence analysis is not safe, and the safe persistence analysis is proposed in [5, 10]. The first multi-level cache analysis is proposed in [16], which is an extension to another well-established single-level cache analysis method called static cache simulation [17]. Later, in [8], it is pointed out that this method is actually unsafe for analyzing multi-level set associative caches, and it is proposed to use CAC to filter the references at each level and defines an update strategy to take into account the uncertain accesses.

Based on the work in [8] which does not take into account data caches, a method for analyzing multi-level non-inclusive data caches is proposed in [12], and a method for analyzing non-inclusive cache hierarchies with unified caches is proposed in [4]. In [18], an abstract domain called live caches is used to model the relationships between cache levels and the analysis based on this domain can handle unified caches using write-back policy.

Cache hierarchies are natural in multi-core processors, for which the analysis needs to take into account the inter-core interferences. In [20], a dual-core processor with a shared L2 cache model is considered. In [13], task lifetime information is computed and utilized to refine possible interferences. In [7], a method for identifying and bypassing the static single usage memory blocks so as to reduce the number of interferences is proposed. In [15], abstract interpretation based cache analysis is combined with model checking based bus analysis to achieve more precise interference analysis. In [3], a WCET analysis framework that covers different micro-architectural components in a multi-core processor is presented. All these works assume multi-level non-inclusive caches are used.

In [9], the methods to analyze cache hierarchies of different types (non-inlucisve, inclusive, and exclusive) are presented. It shows the difficulties in deriving a tight WCET estimation for systems using multi-level inclusive caches and non-LRU replacement policies. It considers different multi-level instruction cache types separately without taking into account hybrid types like a combination of non-inclusive and inclusive caches.

VII. CONCLUSION AND FUTURE WORK

In this paper, we propose an approach that can safely and more precisely analyze multi-level inclusive caches for WCET estimation. The approach first analyzes all the inclusive levels in the *bottom-up* direction and then analyzes the rest non-inclusive levels in the *top-down* direction. Although *bottom-up* sounds counter-intuitive considering the cache levels are accessed in the *top-down* direction, we show that it is actually very suitable for analyzing inclusive caches. In order to capture the effects of the invalidations caused by an inclusive level, we propose a concept of aging barrier. Aging barriers can safely

slow down the increase of memory blocks' ages, and we show how to integrate them into the *must* and *persistence* analyses to gain more precision. From the experiment results, we can observe the proposed approach can tighten the bound by 12.2% on average. In the future, we want to extend the approach to take into account the effects of data references and inter-core interferences, and we also want to enhance our tool to consider the interactions between multi-level caches and other micro-architectural features.

ACKNOWLEDGMENT

This work is supported in part by the NSF (CNS-1035655). The authors would like to thank the anonymous shepherd and reviewers for their comments and suggestions which greatly help us improve the quality of the paper.

REFERENCES

[1] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *ISCA '88*, pages 73–80, 1988.
[2] C. Ballabriga and H. Casse. Improving the first-miss computation in set-associative instruction caches. In *ECRTS '08*, pages 341–350, 2008.
[3] S. Chattopadhyay, C. L. Kee, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk. A unified wcet analysis framework for multi-core platforms. In *RTAS '12*, pages 99–108, 2012.
[4] S. Chattopadhyay and A. Roychoudhury. Unified cache modeling for wcet analysis and layout optimizations. In *RTSS '09*, pages 47–56, 2009.
[5] C. Cullmann. Cache persistence analysis: Theory and practice. *ACM Trans. Embed. Comput. Syst.*, 12(1s):40:1–40:25, Mar. 2013.
[6] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks - past, present and future. In *WCET '10*, pages 137–147, jul 2010.
[7] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches. In *RTSS '09*, pages 68–77, 2009.
[8] D. Hardy and I. Puaut. Wcet analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS '08*, pages 456–466, 2008.
[9] D. Hardy and I. Puaut. Wcet analysis of instruction cache hierarchies. *J. Syst. Archit.*, 57(7):677–694, Aug. 2011.
[10] B. K. Huynh, L. Ju, and A. Roychoudhury. Scope-aware data cache analysis for wcet estimation. In *RTAS '11*, pages 203–212, 2011.
[11] 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.
[12] B. Lesage, D. Hardy, and I. Puaut. WCET Analysis of Multi-Level Set-Associative Data Caches. In *WCET '09*, pages 1–12, 2009.
[13] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *RTSS '09*, pages 57–67, 2009.
[14] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC '95*, pages 456–461, 1995.
[15] M. Lv, W. Yi, N. Guan, and G. Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *RTSS '10*, pages 339–349, 2010.
[16] F. Mueller. Timing predictions for multi-level caches. In *In ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 29–36, 1997.
[17] F. Mueller. Timing analysis for instruction caches. *Real-Time Syst.*, 18(2/3):217–247, May 2000.
[18] T. Sondag and H. Rajan. A more precise abstract domain for multi-level caches for tighter wcet analysis. In *RTSS '10*, pages 395–404, 2010.
[19] 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.
[20] J. Yan and W. Zhang. Wcet analysis for multi-core processors with shared l2 instruction caches. In *RTAS '08*, pages 80–89, 2008.

APPENDIX

In order to prove the proposed multi-level (inclusive) cache analysis is safe, we need to prove the *may*, *must*, and *persistence* analyses of the last inclusive cache are safe, and we also need to prove the analyses of the cache located above at least one inclusive cache are safe.

When analyzing a cache level, we can safely use the well-defined join function of the single-level cache *may*, *must*, or *persistence* analysis at a join point for the corresponding analysis [8], so we can focus more on proving the defined update functions are safe.

A. Safe Analyses of the Last Inclusive Cache

Given the last inclusive cache level l_{LIC}, we first prove the proposed *may*, *must*, and *persistence* analyses are safe.

Lemma 2. *The last inclusive cache may analysis is safe. In other words, at a program point p, $\alpha^{may}_{l_{\text{LIC}}}$ contains all of the memory blocks that are possibly in $C_{l_{\text{LIC}}}$ when the execution reaches p.*

Proof: Since $\mathcal{J}^{may}_{\text{LIC}}$ is $\mathcal{J}^{may}$ which is safe, we only need to prove $\mathcal{U}^{may}_{\text{LIC}}$ is safe, which we do by mathematical induction.
Base case: At the beginning of any execution, $C_{l_{\text{LIC}}}$ does not have any valid blocks (cold start), and the $\alpha^{may}_{l_{\text{LIC}}}$ is also empty showing no memory block is possibly in $C_{l_{\text{LIC}}}$.
Inductive hypothesis: Before a reference r which accesses the memory block $m^r_{l_{\text{LIC}}}$, $\alpha^{may}_{l_{\text{LIC}}}$ contains all the memory blocks that are possibly in $C_{l_{\text{LIC}}}$.
Inductive step: When executing the reference r, we have two possibilities. (1) If $m^r_{l_{\text{LIC}}} \notin \alpha^{may}_{l_{\text{LIC}}}$, $m^r_{l_{\text{LIC}}}$ is definitely not in $C_{l_{\text{LIC}}}$ (deduced from the inductive hypothesis). Based on Lemma 1, we know that $C_{l_{\text{LIC}}}$ will be definitely accessed. Therefore, $\mathcal{U}^{may}(\alpha^{may}_l, m^r_{l_{\text{LIC}}})$ gives the safe result. (2) If $m^r_{l_{\text{LIC}}} \in \alpha^{may}_{l_{\text{LIC}}}$, $m^r_{l_{\text{LIC}}}$ is possibly (may or may not be) in $C_{l_{\text{LIC}}}$ (given by the inductive hypothesis), so it is uncertain whether $C_{l_{\text{LIC}}}$ will be accessed. Thus, $\mathcal{J}^{may}(\mathcal{U}^{may}(\alpha^{may}_{l_{\text{LIC}}}, m^r_{l_{\text{LIC}}}), \alpha^{may}_{l_{\text{LIC}}})$ captures this uncertainty and gives the safe result. Combing (1) and (2), we conclude Lemma 2 holds. ■

Lemma 3. *The last inclusive cache must analysis is safe. In other words, at a program point p, the memory blocks that are contained in $\alpha^{must}_{l_{\text{LIC}}}$ are definitely in $C_{l_{\text{LIC}}}$ when the execution reaches p.*

Proof: Since $\mathcal{J}^{must}_{\text{LIC}}$ is $\mathcal{J}^{must}$ which is safe, we only need to prove $\mathcal{U}^{must}_{\text{LIC}}$ is safe. As shown in the definition of $\mathcal{U}^{must}_{\text{LIC}}$, for a memory reference r, only when $m^r_{l_{\text{LIC}}} \notin \alpha^{may}_{l_{\text{LIC}}}$, we directly use $\mathcal{U}^{must}(\alpha^{must}_{l_{\text{LIC}}}, m^r_{l_{\text{LIC}}})$; otherwise, we conservatively join the two states coming from two possibilities (the access occurring and not occurring). Thus, as long as when $m^r_{l_{\text{LIC}}} \notin \alpha^{may}_{l_{\text{LIC}}}$, $C_{l_{\text{LIC}}}$ will be definitely accessed, the update function $\mathcal{U}^{must}_{\text{LIC}}$ is safe. From Lemma 2, it is straightforward to see that this is true. ■

Lemma 4. *The last inclusive cache persistence analysis is safe. In other words, at a program point p, any memory block that has been loaded into $C_{l_{\text{LIC}}}$ is in an age position of $\alpha^{pers}_{l_{\text{LIC}}}$ which is greater than or equal to its possible maximal age when the execution reaches p (which implies if it is possibly absent from $C_{l_{\text{LIC}}}$, it is in a $\top$ position of $\alpha^{pers}_{l_{\text{LIC}}}$).*

Proof: This proof will be the same as the proof of Lemma 3, except we prove the defined $\mathcal{U}^{pers}_{\text{LIC}}$ is safe. ■

B. Safe Analyses of Inclusive Caches Located above One Inclusive Cache

Since we analyze all the inclusive caches in the *bottom-up* direction at first, we prove the analyses of the inclusive caches

that are located above the last inclusive cache are safe. Let l_v be the second last inclusive cache level.

Lemma 5. *The may analysis of C_{l_v} is safe. In other words, at a program point p, $\alpha_{l_v}^{may}$ contains all the memory blocks that are possibly in C_{l_v} when the execution reaches p.*

Proof: As $\alpha_{l_v}^{may}$ is updated according to Algorithm 1, we need to prove the steps in the algorithm will not overestimate the age of a memory block. In the algorithm, j is calculated and used to control the upper bound on the aging process of updating $\alpha_{l_v}^{may}$. Note that if we have $j \leq j'$, where j' represents the smallest position where has a "hole", line 7-9 will be always safe (some blocks' ages will be underestimated but will not be overestimated). Based on Lemma 4, we know the last inclusive cache *persistence* analysis captures all the possibly evicted memory blocks in the $\top$ positions of $\alpha_{l_{\text{LIC}}}^{pers}$. Thus, line 4-6 will give a j such that $j \leq j'$ holds.

When $\alpha_{l_v}^{may}$ of each point reaches the fixed-point, we also remove the memory blocks from $\alpha_{l_v}^{may}$ whose contents are not in the $\alpha_{l_{\text{LIC}}}^{may}$ of that point. Based on Lemma 2, we know if a memory block is not in $\alpha_{l_{\text{LIC}}}^{may}$, it is definitely not in the last inclusive cache, so its contents are also invalidated at l_v. Thus, $\alpha_{l_v}^{may}$ is safely derived at each point, and Lemma 5 holds. ■

Lemma 6. *The must analysis of C_{l_v} is safe. In other words, at a program point p, any aging barrier (i, j) derived from β_{l_v} corresponds to a "hole" in the i^{th} set within the position range of $[1, j]$, and the memory blocks contained in $\alpha_{l_v}^{must}$ are definitely in C_{l_v} when the execution reaches p.*

Proof: As discussed in IV-B concerning the definition of $\mathscr{J}$ function, we know the $\mathscr{J}$ function ensures only the "holes" that definitely exist along either path are kept and the function overestimates the position upper bounds of these "holes". Since the join function $\mathcal{J}^{must}$ does not underestimate the age of a memory block, we only need to prove updating β_{l_v} and $\alpha_{l_v}^{must}$ are safe. We prove this by mathematical induction.

Base case: At the beginning, $\beta_{l_v} = \beta_{\perp}$, which means all the positions in all sets are "holes", and $\alpha_{l_v}^{must}$ corresponds to an empty state. We have a cold start, there is no memory blocks loaded. Therefore, the lemma holds in the base case.

Inductive hypothesis: Before a reference r which accesses the memory block $m_{l_v}^r$ that is mapped to the i^{th} cache set, any aging barrier (i, j) derived from β_{l_v} corresponds to a "hole" in the i^{th} set within the position range of $[1, j]$, and the memory blocks contained in $\alpha_{l_v}^{must}$ are definitely in C_{l_v}.

Inductive step: Based on the inductive hypothesis and Lemma 2, if a memory block is in the current $\alpha_{l_v}^{must}$, but its contents are not in $\alpha_{l_{\text{LIC}}}^{may}$ after the reference, this memory block needs to be invalidated, so a "hole" will be created. Since the *must* analysis captures the maximal ages of memory blocks, adding the created "hole" into β_{l_v} will not violate the lemma. Based on Lemma 4, the memory blocks in the $\top$ positions of $\alpha_{l_{\text{LIC}}}^{pers}$ after the reference are possibly evicted; thus, after line 7 the lemma still holds with respect to the updated β_{l_y} and $\alpha_{l_y}^{must}$. When updating the states according to the rest of Algorithm 2, after line 9, j has a position and if we have $j \neq \top$, there is a "hole" within the range $[1, j]$, and any of the rest aging barriers derived from the used β_{l_v}, namely β'_{l_v}, still corresponds to a "hole" (deduced from the inductive hypothesis). There are two possibilities when updating $\alpha_{l_v}^{must}$. (1) If $m_{l_v}^r \notin \alpha_{l_v}^{may}$, based on Lemma 5, we have $k = \top$ and we are sure that $m_{l_v}^r$ is not in C_{l_v}. Based on Lemma 1, C_{l_v} will be definitely accessed due to the reference r. Therefore, line 16 (i.e. applying $\overline{\mathcal{U}}^{must}$ which takes into account the effects of the existence of a "hole") can safely update $\alpha_{l_v}^{must}$, and that "hole" is possibly filled. In this case, $\max(j, k) = \top$ no matter what j is, so $\mathscr{A}$ will not change the β'_{l_v} at line 19. (2) If $m_{l_v}^r \in \alpha_{l_v}^{may}$, we do not know if C_{l_v} will be accessed or not, so line 17 can safely update $\alpha_{l_v}^{must}$ by taking into account the access occurring and not occurring. We have $j = \top$ or $j \neq \top$, and $k = \top$ or $k \neq \top$. If $j = \top$ or $k = \top$, $\max(j, k) = \top$, so $\mathscr{A}$ will not change the β'_{l_v} at line 19. The only case in which $\mathscr{A}$ will change β'_{l_v} is when $j \neq \top \bigwedge k \neq \top$. In this case, although the "hole" with the range $[1, j]$ may be possibly filled, there is still a "hole" within the range $[1, \max(j, k)]$ – this is because, based on the hypothesis, $m_{l_v}^r$ is definitely in C_{l_v} if $k \neq \top$, and in either case of $k < j$ or $j < k$, the reference does not load a new memory block into C_{l_v}; so after applying $\mathscr{A}$ on β'_{l_v}, the resultant β_{l_v} does not violate the lemma. Thus, after line 19, this lemma still holds with respect to the updated β_{l_v} and $\alpha_{l_v}^{must}$. ■

Lemma 7. *The persistence analysis of C_{l_v} is safe. In other words, at a program point p, any memory block that has been loaded into C_{l_v} is in an age position of $\alpha_{l_v}^{pers}$ which is greater than or equal to its possible maximal age.*

Proof: Since $\mathcal{J}^{pers}$ does not underestimate the age of a memory block, we only need to prove this lemma holds in terms of updating, which we do by mathematical induction.

Base case: At the beginning, no memory block is loaded, and all the positions of $\alpha_{l_v}^{pers}$ are empty. The lemma holds.

Inductive hypothesis: Before a reference r which accesses the memory block $m_{l_v}^r$, any memory block that has been loaded into C_{l_v} is in an age that is greater than or equal to its possible maximal age.

Inductive step: When updating $\alpha_{l_v}^{pers}$, we first move the blocks which are possibly or definitely invalidated to $\top$ positions according to $\alpha_{l_{\text{LIC}}}^{pers}$ and $\alpha_{l_{\text{LIC}}}^{may}$ after the reference. Since doing so does not decrease any block's age, the lemma still holds. Then, we get j from the aging barrier state β_{l_v} maintained by the *must* analysis (but we do not change β_{l_v}). There are two possibilities to continue updating. (1) If $m_{l_v}^r \notin \alpha_{l_v}^{may}$, based on Lemma 5, we know $m_{l_v}^r$ is not in C_{l_v}; and based on Lemma 1, C_{l_v} will be accessed due to the reference r. According to the definition of $\overline{\mathcal{U}}^{pers}$, when $j = \top$, it is $\mathcal{U}^{pers}$ and it will not underestimate the possible maximal ages of the blocks; when $j \neq \top$, no matter what k is, it will never increase the ages of the blocks that are already greater than or equal to j, so we need to prove in this case the possible maximal ages of these memory blocks are actually not greater than these unchanged ages: since $j \neq \top$, there is definitely a "hole" within the position range $[1, j]$, so even C_{l_v} is accessed and $m_{l_v}^r$ is not in C_{l_v}, the ages of the blocks that are behind this "hole" will not be increase, which means the possible maximal ages of the memory blocks which are already greater than or equal to j will not be increase; from the inductive hypothesis, we know that before this reference, for a memory block, the position where it is in $\alpha_{l_v}^{pers}$ is the upper bound of its possible maximal age position; thus, even though the ages of the memory blocks that are already greater than or equal to j are unchanged after applying $\overline{\mathcal{U}}^{pers}$, they are still not less than the possible maximal ages of these memory blocks, based on the arguments above. (2) If $m_{l_v}^r \in \alpha_{l_v}^{may}$, we do not know if C_{l_v} is accessed or not, so we safely join the two states corresponding to the access occurring and not occurring. Thus, the lemma holds with respect to the updated $\alpha_{l_v}^{pers}$. ■

Theorem 1. *The proposed may, must, and persistence analyses of the inclusive caches in the bottom-up direction are safe.*

Proof: Since we have proven the analyses of the last inclusive cache are safe (Lemma 2, Lemma 3, and Lemma 4), we only need to prove the analyses of the rest inclusive caches in the *bottom-up* direction are safe by mathematical induction.
Base case: The analyses of C_{l_v} are safe, where l_v is the second last inclusive cache level.
Inductive hypothesis: The analyses of all the inclusive caches that are located beneath C_{l_y} are safe, where l_y is an inclusive level above the last inclusive level l_{LIC}.
Inductive step: Let us assume the next inclusive level located beneath l_y in the *top-down* direction is l_y^i. Following the proofs of Lemma 5, Lemma 6, and Lemma 7, we can prove the *may*, *must*, and *persistence* analyses of C_{l_y} are safe, as long as the analyses of $C_{l_y^i}$ are safe. Since the inductive hypothesis gives the analyses of $C_{l_y^i}$ are safe, the analyses of C_{l_y} are safe. ■

C. Safe Analyses of Non-Inclusive Caches

After the analyses of the inclusive caches are completed, we start from l_1 and analyze all the non-inclusive caches in the *top-down* direction. Let us assume, for a non-inclusive cache level l_z, l_z^p is the previous cache level in the *top-down* direction in the cache hierarchy when $z > 1$ (l_z^p can be either inclusive or non-inclusive), and l_z^i is the first inclusive cache level that is beneath l_z if there is one (i.e. $C_{l_z^i}$ directly includes C_{l_z}).

Lemma 8. *The may analysis of C_{l_1} is safe. In other words, at a program point p, $\alpha_{l_1}^{may}$ contains all the memory blocks that are possibly in C_{l_1} when the execution reaches p.*

Proof: Since C_{l_1} is always accessed for a reference, we just need to prove line 12 can safely update $\alpha_{l_1}^{may}$ each time, which implies we prove j should always satisfy $j \leq j'$, where j' represents the smallest position where has a "hole" (in which case, some blocks' ages will be underestimated but will never be overestimated). Following the proof of Lemma 5, we can see j set by the loop (line 4-6) satisfies $j \leq j'$, since the *persistence* analysis of $C_{l_1^i}$ has already been safely performed (given by Theorem 1). Thus, Lemma 8 holds. ■

Lemma 9. *The must analysis of C_{l_1} is safe. In other words, at a program point p, any aging barrier (i, j) derived from β_{l_1} corresponds to a "hole" in the i^{th} set within the position range of $[1, j]$, and the memory blocks contained in $\alpha_{l_1}^{must}$ are definitely in C_{l_1} when the execution reaches p.*

Proof: Following the proof of Lemma 6, we can prove the lemma holds by using mathematical induction. The difference is: since C_{l_1} is always accessed for a reference, we only use line 16 to update $\alpha_{l_1}^{must}$ each time. Following that proof, we can prove it safely updates β_{l_1} and $\alpha_{l_1}^{must}$, as long as the *may* and *persistence* analyses of $C_{l_1^i}$ are safe, which is true based on Theorem 1. Thus, Lemma 9 holds. ■

Lemma 10. *The persistence analysis of C_{l_1} is safe. In other words, at a program point p, any memory block that has been loaded into C_{l_1} is in an age position of $\alpha_{l_1}^{pers}$ which is greater than or equal to its possible maximal age.*

Proof: Similarly, following the proof of Lemma 7, we can prove this lemma holds. ■

Theorem 2. *The proposed may, must, and persistence analyses of the non-inclusive caches in the top-down direction are safe.*

Proof: We prove this theorem by mathematical induction.
Base case: The analyses of C_{l_1} are safe.
Inductive hypothesis: The analyses of all of the non-inclusive caches located above C_{l_z} are safe, where l_z is a non-inclusive cache level and $z > 1$.
Inductive step: When updating the abstract cache states of C_{l_z}, we need to derive the CAC at l_z for a reference. As described in [8], for a reference, the CAC at l_z can be safely derived if the CHMC and CAC at l_z^p are known. Based on Theorem 1 and the inductive hypothesis, we know, from l_1 to l_z^p, no matter whether a level is inclusive or non-inclusive, it is safely analyzed. By taking into account the effects of filtering accesses, the CHMC and CAC at l_z^p can be safely derived, so that the CAC at l_z can be safely derived. If l_z is located beneath the last inclusive cache level l_{LIC}, C_{l_z} does not suffer from any invalidation, so the unmodified analyses of C_{l_z} will be safe. On the contrary, if l_z is located above l_{LIC}, we use the methods described in IV-D to analyze C_{l_z}; following the previous proofs, we can also prove the analyses are safe. Thus, combing these two cases, we can conclude this theorem holds. ■

Theorem 3. *The proposed approach to analysis of multi-level inclusive caches is safe.*

Proof: We can directly conclude this Theorem holds based on Theorem 1 and Theorem 2. ■

D. Termination of the Analysis

In order to prove the proposed multi-level inclusive cache analysis will terminate, we need to prove the aging barrier state domain ABS is a partially ordered set with a finite height; and we need to prove the joining and updating of the aging barrier states are monotonic at a program point during the iterations at one level. Since the number of sets and the associativity of a cache are both finite, based on the Definition 2 and Definition 3, it is trivial to see ABS is finite and partially ordered. Also, we have seen the join function $\mathscr{J}$ is monotone with respect to the partial ordering defined in Definition 3. Thus, we need to prove the aging barrier state updating is also monotone.

Lemma 11. *Given an aging barrier state β' which is updated by a reference from β, $\beta \sqsubseteq \beta'$ always holds.*

Proof: When updating β, first we have $\langle \beta'', j \rangle = \mathscr{U}(\beta, i)$, where i is fixed for a reference in a cache. Therefore, we have $\beta \sqsubseteq \beta''$ according to the definition of $\mathscr{U}$. Then, we have $\beta' = \mathscr{A}(\beta'', i, \max(j, h))$. According to the definition of $\mathscr{A}$: if $\max(j, h) = \top$, we have $\beta' = \beta''$, so $\beta \sqsubseteq \beta'' = \beta'$ holds; if $\max(j, h) = j \neq \top$, we have $\beta' = \beta$, since the partial ordering $\sqsubseteq$ is reflexive, so $\beta \sqsubseteq \beta'$ holds; if $\max(j, h) = h \neq \top$, we have $|\beta'(i)| = |\beta(i)|$ and since the only difference between $\beta(i)$ and $\beta'(i)$ is in $\beta'(i)$ we have $j = \min_c(\beta(i))$ replaced by h which is $\max(j, h)$, so $\beta \sqsubseteq \beta'$ holds. ■

Theorem 4. *The proposed multi-level inclusive analysis approach terminates in finite iterations at each level.*

Proof: Since the analyses of the last inclusive cache are not affected by other factors, they will terminate. The analyses of a cache located above at least one inclusive level are affected by its aging barrier state and the abstract cache states of some safely analyzed caches. Although aging barriers can slow down the age increasing, the abstract cache states at this level are still updated along an ascending chain. Based on Lemma 11, we know the aging barrier states are also updated along an ascending chain. Since all the domains are finite partially ordered sets, the proposed analysis will terminate. ■