2015 IEEE Real-Time Systems Symposium

Precise Multi-Level Inclusive 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*—Multi-level inclusive caches are often used in multi-core processors to simplify the design of cache coherence protocol. However, the use of such cache hierarchies poses great challenges to tight worst-case execution time (WCET) estimation due to the possible invalidation behavior. Traditionally, multi-level inclusive caches are analyzed in a level-by-level manner, and at each level three analyses (i.e. *must*, *may*, and *persistence*) are performed separately. At a particular level, conservative decisions need to be made when the behaviors of other levels are not available, which hurts analysis precision. In this paper, we propose an approach which analyzes a multi-level inclusive cache by integrating the three analyses for all levels together. The approach is based on the abstract interpretation of a concrete operational semantics defined for multi-level inclusive caches. We evaluate the proposed approach and also compare it with two state-of-the-art approaches. From the experimental results, we can observe the proposed approach can significantly improve the analysis precision under relatively small cache size configurations.**

I. Introduction

When designing real-time systems, especially hard real-time systems, worst-case execution time (WCET) estimation is an essential task required by schedulability analysis. There are two criteria in such a task – the WCET estimate must be safe, namely it may not be lower than the actual WCET; and the estimate should be as tight as possible to maximize system resource utilization. To this end, we need to take into account the underlying micro-architectural features, such as caches, pipelines, and branch predictors, since their behavior has a huge impact on the variation of execution time.

Over the past two decades, WCET estimation in the context of single-level caches has been studied extensively [6], [19]. Recently, multi-level cache analysis has drawn much attention [22], [9], [14], [3], [20], [10], since there is a rising need to use high-performance processors in real-time systems, which are often equipped with multi-level caches. However, compared to single-level caches, the behavior of multi-level caches is much more challenging to analyze. In addition to memory reference sequences, there is also a need to take into account the effect of one cache level's behavior on other cache levels' behavior (e.g. filtering memory accesses and invalidating memory blocks), which can be different depending on the type of the cache hierarchy.

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. Moreover, there are some hybrid cache hierarchies, which combine heterogeneous cache level policies. 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).

While some methods have been proposed to analyze multi-level non-inclusive caches precisely for WCET estimation [9], [14], [3], [20], there is only little progress in terms of how to precisely analyze the other two types of cache hierarchies, especially multi-level inclusive caches [10]. However, inclusive cache hierarchies appear commonly in many multi-core architectures, since the inclusion property can significantly simplify the maintenance of cache coherence [1]. Many commercial processors are equipped with multi-level inclusive caches, such as the IBM PowerPC and Intel Xeon. Thus, it is necessary to have a multi-level cache analysis framework that can precisely analyze cache hierarchies that enforce inclusion for WCET estimation.

Most of the current approaches attempt to separately analyze each cache level of a cache hierarchy, usually starting from the topmost level and moving downward. For multi-level non-inclusive caches, this analysis style does not have a large impact on the precision, since the interactions between different cache levels in such a hierarchy are only related to the memory access filtering behavior which appears in a *top-down* direction. However, for multi-level inclusive caches, such an analysis style may have a big influence on the precision, because some blocks may be invalidated by lower inclusive levels (i.e., the invalidation behavior appears in a *bottom-up* direction). Due to the unknown underlying invalidation behavior, when analyzing an upper level, conservative decisions have to be made in order to ensure safety [10]. In order to reduce the number of conservative decisions, in our previous work, we propose an approach to separately analyze all the inclusive levels in a *bottom-up* direction [22]. However, as shown in this paper, this *bottom-up* approach may not perform well when the cache size is relatively small compared to the program size.

In this paper, we propose an approach that can precisely

1052-8725/15 $31.00 © 2015 IEEE
DOI 10.1109/RTSS.2015.40

IEEE computer society

analyze multi-level inclusive caches, even in the case when the ratio of cache size to program size is low. The main idea is to analyze a multi-level inclusive cache as a whole following its concrete behavioral semantics, instead of analyzing it in a level-by-level manner.

The main contributions of this paper are: (1) We define a concrete operational semantics which formally describes how a multi-level inclusive cache changes its state when a memory reference occurs. (2) Based on the abstract interpretation of this concrete semantics, we propose an approach that analyzes a multi-level inclusive cache as a whole by integrating three analyses (i.e., *must*, *may*, and *persistence*). (3) We evaluate the proposed approach on a set of benchmarks, and the evaluation results show significant precision improvements when the ratio of cache size to program size is low, compared to the state-of-the-art approaches.

The rest of the paper is organized as: Section II describes the background on cache analysis; Section III gives the system model; Section IV states why multi-level inclusive cache analysis is challenging; Section V presents the proposed approach to multi-level inclusive cache analysis; Section VI evaluates the proposed approach; Section VII describes the related work, and Section VIII concludes this paper.

II. Background

Cache analysis for WCET estimation is usually based on abstract interpretation in order to make the analysis scalable. Such approaches aim to assign a *cache hit/miss classification* (CHMC) to each memory reference according to the abstract cache states (ACSs) derived by three different analyses [6], [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 a 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 a 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 a 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*).

Given a reference r to a memory block m, the effect of this reference on the ACS θ^t, where t is either *must*, *may*, or *persistence*, is defined by an *update* function $\mathcal{U}^t : \Theta^t \times M \rightarrow \Theta^t$, where Θ^t is the set of all the ACSs of the cache (i.e. $\theta^t \in \Theta^t$), and M is the set of all the memory blocks *w.r.t.* the cache block size (i.e. $m \in M$). In order to safely combine information at a join point during the analysis on the CFG, a *join* function $\mathcal{J}^t : \Theta^t \times \Theta^t \rightarrow \Theta^t$ is also defined. The definitions of the *update* and *join* functions can be found in [6], [5].

Single-level caches are always accessed by each memory reference, so we only need to consider the effects of memory reference sequences in the analysis. However, in the case of multi-level caches, it is also important to consider the effects of other cache levels, in particular, cache access filtering and memory block invalidation. For example, if we treat a 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 [9].

For a reference r, a *cache access classification* (CAC) at a cache level l is used to represent the possibility that l will be accessed [9]. Let $\theta^{t,i}_{l,r}$ denote the ACS at this level immediately before r, and let $\theta^{t,o}_{l,r}$ denote the ACS at this level immediately after r. Let m^r_l denote the memory block *w.r.t.* the cache block size at l containing the information needed by r. If the CAC is *always* (*A*), the access will always occur, so r will always affect the ACS:

$$\theta^{t,o}_{l,r} = \mathcal{U}^t(\theta^{t,i}_{l,r}, m^r_l)$$

On the other hand, if the CAC is *never* (*N*), the access will never happen, so the ACS at l is not affected by r:

$$\theta^{t,o}_{l,r} = \theta^{t,i}_{l,r}$$

If the CAC cannot be either *A* or *N*, it is *uncertain* (*U*), which means the access may or may not happen. In order to ensure safety, the updates of the ACS due to *U* accesses need to take into account the two possible cases (access occurring and not occurring) by joining them:

$$\theta^{t,o}_{l,r} = \mathcal{J}^t(\underbrace{\mathcal{U}^t(\theta^{t,i}_{l,r}, m^r_l)}_{\text{access occurring}}, \overbrace{\theta^{t,i}_{l,r}}^{\text{access not occurring}})$$

As described in [9], for a reference r that is possible to access a cache level (i.e. its CAC is not *N* at this level), if r can be safely classified as *AH* at this level, r will never need to access all the lower levels, namely its CAC is *N* at any lower level; if r can be safely classified as *AM* at this level, r is also possible to access the next lower level, namely its CAC at the next lower level is the same as the CAC at this level (i.e. *A* or *U*). Note that if a reference always/never accesses a cache level in reality, but its CAC at the level is *U* in the analysis, the analysis is still safe but may not give a tight result.

III. System Model

In this paper, we focus on a generalized cache hierarchy model, in which the inclusion property is enforced at some cache level(s). The model has n cache levels, represented by $L = \{l_1, \cdots, l_n\}$. Each cache level is either inclusive or non-inclusive[2]. Let $inc : L \rightarrow \{\text{true}, \text{false}\}$ be an auxiliary function that returns true if the level is inclusive and false otherwise. The inclusive cache hierarchy model C is a two-tuple $\langle L, inc \rangle$.

Although we do not consider exclusive caches in the model, we can easily add them into our analysis by using the approach proposed in [10]. The exclusive cache levels can be collapsed

[1]In [9], 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]Note that it has no meaning for L1 cache to be inclusive/non-inclusive, i.e., it can be either one.

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.

For a cache level l_x (where $1 \leq x \leq n$), we assume that the cache is set associative, and LRU (Least Recently Used) replacement policy is used. The size of a cache block can be different at different cache levels, and it is assumed the block size does not increase as the level goes up. It is also assumed the capacity decreases as the level goes up.

We also assume the time to access a cache level is bounded and predictable, which can be achieved by using deterministic interconnects to connect the caches, like TDMA buses [12]. Fig. 1 gives an example of the model focusing on a single core and all the cache levels that can be affected by this core in a multi-core architecture.

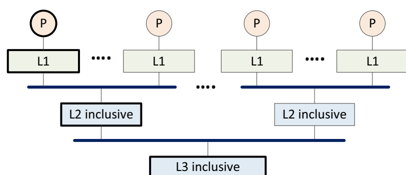


Fig. 1. An example of the system model: only the cache levels that can be affected by the first core are considered, i.e., $L = \{l_1, l_2, l_3\}$ and $inc(l_1) =$ false, $inc(l_2) =$ true, and $inc(l_3) =$ true.

In this paper, we focus only on how to analyze multi-level caches in the presence of invalidations caused by the inclusion enforcement, so we simply consider instruction references in terms of a single processor (i.e. no data references and inter-core interferences). This work can serve as a basis for analysis of multi-level data or unified caches, that enforce the inclusion property, in terms of a multi-core processor.

IV. Problem Statement

In the case of multi-level non-inclusive cache analysis, the CAC for a reference r at a cache level l_x can be derived from the CHMC and CAC for r at l_{x-1} ($x > 1$ is assumed, since l_1 is always accessed, i.e. the CAC is *A* at l_1), and the cache behavior at any level will not be affected by any lower cache level. Thus, the cache hierarchy can be analyzed level-by-level in a *top-down* direction.

However, in the case of analyzing cache hierarchies containing inclusive caches, the CAC for r at l_x cannot be safely derived from CHMC and CAC for r at l_{x-1}. The reason is the behavior of l_x depends not only on the behavior of l_{x-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 (these invalidated memory blocks are called *inclusion victims*).

Example: Fig. 2 shows a 3-level inclusive cache, where L1 is 2-way set associative, L2 is 4-way set associative, and L3 is 4-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, $\ddot{m}_b$ 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.

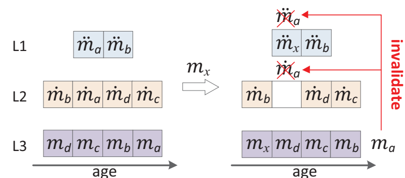


Fig. 2. Invalidation due to the maintenance of the inclusion property of L3

Due to the possible invalidation behavior and the induced consequences (i.e., some blocks may live in the cache longer, but some blocks may live in the cache shorter – being invalidated instead of being evicted), the traditional CAC derivation method for multi-level non-inclusive cache analysis becomes unsuitable for multi-level inclusive cache analysis. At a cache level, it is challenging to safely classify a reference as *AH* or *AM* without knowing the behavior of lower inclusive cache level(s).

Since safely and tightly determining the CAC (i.e. trying to derive *A* or *N* instead of *U*) for a reference at a cache level relies on the safe CHMC at the upper levels, the approach proposed in [10] decides to classify the CAC for every reference at any level (except for the L1 which is always accessed) as *U*. In this way, although the safety is ensured, the tightness of the analysis may suffer considerably.[3]

A naive method may involve refining the ACSs of upper levels when finishing a lower inclusive level analysis, which also refines the CHMC and CAC for a reference at these levels. However, such a method may not ensure the monotonicity so that it may not guarantee termination.

The approach proposed in [22] can have the CACs for some references at some levels classified as *A* instead of *U*. However, it cannot have the CAC for any reference at the inclusive levels classified as *N*, since it analyzes all of the inclusive levels in a *bottom-up* direction (i.e. when analyzing an inclusive level, the ACSs of its upper levels remain unknown). Moreover, as the ratio of the cache size at an inclusive level to the program size decreases, its ability to safely classify the CAC as *A* instead of *U* at this level also decreases.

Thus, how to precisely analyze multi-level inclusive caches is still a very challenging problem. Specifically, we need to

[3]The approach proposed in [10] also changes every reference's *AM* CHMC into *NC* at any level, and may also change some references' *AH* or *PS* CHMC into *NC* at a level depending on the analysis of lower inclusive levels.

find ways to safely determine the *A* or *N* instead of *U* CACs for as many references as possible at a cache level.

V. Precise Multi-Level Inclusive Cache Analysis Based on Abstract Interpretation

The approach proposed in this paper is based on abstract interpretation which is a framework for deriving sound analyses (i.e., the results are sound approximations) [4]. In order to make use of abstract interpretation, we first need to define the concrete semantics. The concrete semantics for single-level caches and multi-level non-inclusive caches have been given in [6] and [20], respectively. In this section, we use an operational semantics to describe how multi-level inclusive caches change their states when a reference occurs. Based on the abstraction of this concrete semantics, we propose an approach that can more precisely analyze multi-level inclusive caches due to its ability to determine *A* or *N* CACs for memory references at a cache level. Since a cache level in an inclusive cache hierarchy can be affected by two behaviors coming in two directions, i.e. the memory access filtering behavior appearing in a *top-down* direction and the invalidation behavior appearing in a *bottom-up* direction, the intuition behind the approach is to analyze a multi-level inclusive cache as a whole instead of level-by-level in isolation so as to make both behaviors available at any cache level[4].

A. Concrete Semantics

Since we focus only on instruction references, we just give a concrete operational semantics to the multi-level inclusive instruction caches. The semantics related to data references and dirty blocks can be added (we leave this as our future work).

The state $\bar{s}$ of a multi-level inclusive cache $\langle L, inc\rangle$ is a n-tuple (recall that $L = \{l_1, \cdots, l_n\}$). Each tuple component is a cache state s_{l_x} of the cache level l_x and they are ordered by the level numbers: $\bar{s} = \langle s_{l_1}, \cdots, s_{l_n}\rangle$.

Given a cache level l_x, let c_{l_x} denote its capacity, let b_{l_x} denote its cache block size, and let k_{l_x} denote its associativity. Therefore, the cache at this level has $d_{l_x} = \frac{c_{l_x}}{b_{l_x} \times k_{l_x}}$ cache sets. The cache state s_{l_x} of this cache level is a mapping:

$$s_{l_x} : \{1, \cdots, d_{l_x}\} \rightarrow \Big(\{1, \cdots, k_{l_x}\} \rightarrow M_{l_x}\Big)$$

where M_{l_x} is the set of all of the memory blocks *w.r.t.* b_{l_x}, and we also assume there is an element $I \in M_{l_x}$ which denotes the memory block is invalid. The cache state maps a cache set number to a cache set state and the cache set state is also a mapping that maps a logical position (ordered by LRU ages) to a memory block. For a reference r, let $m^r_{l_x} \in M_{l_x}$ denote the memory block *w.r.t.* b_{l_x} containing the information needed by r. We have an auxiliary function $set : M_{l_x} \rightarrow \{1, \cdots, d_{l_x}\}$ which gives us the cache set number to which $m^r_{l_x}$ is mapped.

When a reference r occurs, the multi-level inclusive cache carries out a sequence of actions that may change the state of a cache level intermittently. For a multi-level inclusive cache $\langle L, inc\rangle$, let $\bar{s} = \langle s_{l_1}, \cdots, s_{l_n}\rangle$ denote the cache hierarchy state before r, and let $\bar{s}' = \langle s'_{l_1}, \cdots, s'_{l_n}\rangle$ denote the state after r. The operational semantics is described as follows:

(1) [Search Cache Levels] Starting from $l_x = l_1$, check whether the needed information is at l_x:

$$\exists p \in \{1, \cdots, k_{l_x}\} : s_{l_x}(set(m^r_{l_x}))(p) = m^r_{l_x}$$

If it is true (i.e. cache hit at l_x), stop searching and go to step (2). Otherwise (i.e. cache miss at l_x), if $x < n$, go to the next cache level (i.e. increment x) to continue searching; else (i.e. x is n which means the needed information is absent from the whole cache hierarchy), increment x (i.e. x becomes $n+1$ which denotes the main memory) and go to step (3).

(2) [Update LRU Ages] Change the logical position (i.e. the LRU age) of $m^r_{l_x}$ from p to 1 in the cache set it is mapped to, and increment the positions of other blocks which were smaller than p. The resultant cache state s'_{l_x} is:

$$\forall i \in \{1, \cdots, d_{l_x}\} : s'_{l_x}(i) =$$
$$\begin{cases} s_{l_x}(i) & \text{if } i \neq set(m^r_{l_x}) \\ [1 \mapsto m^r_{l_x}, & \\ q \mapsto s_{l_x}(i)(q-1) | q = 2 \cdots p, & \text{otherwise} \\ q \mapsto s_{l_x}(i)(q) | q = p+1 \cdots k_{l_x}] & \end{cases}$$

When the updating is completed, go to step (3).

(3) [Move Upwards or Terminate] If $2 \leq x \leq n+1$, decrement x and go to step (4). Otherwise, terminate (and send the needed information to the processor).

(4) [Find New Position] Find the smallest logical position p where $s_{l_x}(set(m^r_{l_x}))(p) = I$, if there is any. Otherwise, p is k_{l_x}. Go to step (5).

(5) [Load Memory Block] Load the memory block $m^r_{l_x}$ into $s_{l_x}(set(m^r_{l_x}))(p)$ to replace the previous memory block $\hat{m}_{l_x}$ in that position. Go to step (6).

(6) [Invalidate Memory Blocks] If $inc(l_x) = $ false or $\hat{m}_{l_x} = I$, go back to step (2). Otherwise, for all cache levels located above l_x, check if any block in their states is a sub-block of $\hat{m}_{l_x}$, and invalidate it if so:

$$\forall y \in \{1, \cdots, x-1\}, \exists i \in \{1, \cdots, d_{l_y}\}, \exists q \in \{1, \cdots, k_{l_y}\} :$$
$$s_{l_y}(i)(q) \subseteq \hat{m}_{l_x} \Rightarrow s_{l_y}(i)(q) = I$$

When the invalidation is finished, go back to step (2).

Let $\bar{S} = S_{l_1} \times \cdots \times S_{l_n}$ denote the set of all the states of a multi-level inclusive cache, where S_{l_x} $(1 \leq x \leq n)$ denotes the set of all the states of the cache level l_x. Let R denote the set of all the references the program can generate. We define a function $f : R \times \bar{S} \rightarrow 2^L \times \bar{S}$ as:

$$f(r, \bar{s}) = \langle \{l_1, \cdots, l_z\}, \bar{s}'\rangle$$

where $\bar{s}'$ is semantically updated from $\bar{s}$ due to the reference r and l_z is the last cache level being updated by step (2) (i.e. $1 \leq z \leq n$) during the process. We can also lift the f function

[4]Level-by-level approaches can only make one of these two behaviors available when analyzing a cache level, so conservative decisions have to be made concerning the other unknown behavior.

to deal with a sequence of references $\pi = (r_1, \cdots, r_h)$, i.e. we sequentially apply f to each reference in π with its prior state, and the result consists of the levels updated by r_h and the state at r_h. The collecting semantics $cs : R \to 2^L \times 2^{\bar{S}}$ is defined as:

$$cs(r) = \bigcup_{\pi \in \Pi_r} \bigcup_{\bar{s} \in \bar{S}_0} f(\pi, \bar{s})$$

where $\bar{S}_0$ is the set of all the initial states, and Π_r is the set of all the possible program reference sequences that reaches r. Note that here we use a loose notation to avoid cluttering: We treat the second component of $f(\pi, \bar{s})$ (i.e. a state) as a singleton, and we also treat $\bigcup$ can realize the set union of the first and second components respectively.

B. Abstract Semantics-Based Approach

Based on the concrete semantics, we propose an approach which attempts to analyze the cache levels together. Given a multi-level inclusive cache $\langle L, inc \rangle$, we define its abstract cache hierarchy state domain Ω as:

$$\Omega = \Phi_{l_1} \times \cdots \times \Phi_{l_n}$$

where Φ_{l_x} $(1 \leq x \leq n)$ is the abstract state domain for the cache level l_x. Before giving the definition of Φ_{l_x} and the semantic functions on Φ_{l_x}, we want to present the approach first in order to give a general idea. To this end, we assume the abstract state domain Φ_{l_x} and the operations on Φ_{l_x} meet the conditions:

- From an abstract state $\phi_{l_x} \in \Phi_{l_x}$, we can safely derive a set of memory blocks that are definitely in the cache.
- From an abstract state $\phi_{l_x} \in \Phi_{l_x}$, we can safely derive a set of memory blocks that are possibly in the cache.
- From an abstract state $\phi_{l_x} \in \Phi_{l_x}$, we can safely derive a set of memory blocks that may be out of the cache after being loaded into the cache.
- An *update* function $\mathcal{U} : \Phi_{l_x} \times M_{l_x} \to \Phi_{l_x}$ can safely update the abstract states in the presence of possible "holes" caused by invalidations.
- A *join* function $\mathcal{J} : \Phi_{l_x} \times \Phi_{l_x} \to \Phi_{l_x}$ can safely join the abstract states in the presence of possible "holes" caused by invalidations.
- An *invalidate* function $\mathcal{I} : \Phi_{l_x} \times 2^{M_{l_x}} \to \Phi_{l_x}$ can safely perform invalidations on the abstract states.

Note that the first three conditions mean that we can safely derive the CHMC for a reference at a cache level l_x from its ϕ_{l_x}. Let us use an auxiliary function $chmc : \Phi_{l_x} \times M_{l_x} \to \{AH, AM, PS, NC\}$ to do this. In addition, we have another auxiliary functions $pout : \Phi_{l_x} \to 2^{M_{l_x}}$ to acquire an over-approximated set of memory blocks that may be out of the cache after being loaded into the cache.

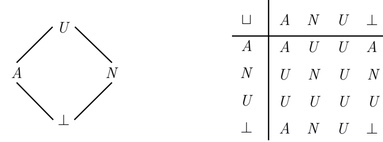
We also define a function domain Ψ that captures all the mappings from references to CACs at all the cache levels:

$$\Psi = R \to \Big(L \to \{U, A, N\}_\perp\Big)$$

In order to ensure the analysis *update* function monotone, we establish a lattice on the lifted set of CACs $\{U, A, N\}_\perp = \{U, A, N\} \cup \{\perp\}$ where $\perp$ means there has not been any CAC yet, and we also define a least upper bound operator $\sqcup$, as shown in Fig. 3. This ensures that if the CAC for a reference at a cache level becomes U, it will never be changed. A semantic function $\mathcal{F} : \Psi \times R \times L \times \{U, A, N\} \to \Psi$ is defined as:

$$\mathcal{F}(\psi, r, l, \text{CAC}) = \psi\Big[r \mapsto \psi(r)[l \mapsto \text{CAC} \sqcup \psi(r)(l)]\Big]$$

which derives the least upper bound of all possible CACs for a reference. Instead of having an abstract element of Ψ at each program point, we have a global abstract object $\psi \in \Psi$ for the whole program which is initialized as $\forall r \in R, l \in L : \psi(r)(l) = \perp$.



$\sqcup$	A	N	U	$\perp$
A	A	U	U	A
N	U	N	U	N
U	U	U	U	U
$\perp$	A	N	U	$\perp$

Fig. 3. CAC lattice and least upper bound operator

At a join point of CFG, we use a *join* function $\mathscr{J} : \Omega \times \Omega \to \Omega$ to combine the abstract cache hierarchy states. Given two abstract states $\omega_1 = \langle \phi_{1,1}, \cdots, \phi_{n,1} \rangle \in \Omega$ and $\omega_2 = \langle \phi_{1,2}, \cdots, \phi_{n,2} \rangle \in \Omega$, the *join* function $\mathscr{J} : \Omega \times \Omega \to \Omega$ is defined as:

$$\mathscr{J}(\omega_1, \omega_2) = \langle \mathcal{J}(\phi_{1,1}, \phi_{1,2}), \cdots, \mathcal{J}(\phi_{n,1}, \phi_{n,2}) \rangle$$

namely it independently combines each level's abstract states.

An *update* function $\mathscr{U} : \Omega \times \Psi \times R \to \Omega \times \Psi$ is used to take into account the effect of a reference r on the abstract hierarchy state and the ψ abstract object. The *update* function is defined based on the concrete operational semantics regarding how a multi-level inclusive cache changes its states when a reference occurs, as described in Section V-A. The steps to perform this *update* function are given in Algorithm 1.

The first loop (lines 3 – 15) abstracts the first step of the concrete operational semantics. It starts from the cache level l_1 and moves downwards through L in sequence to check if a cache level l_x will be updated. Line 4 safely approximates the first component of the collecting semantics cs for a reference r (given $cs(r) = \langle L_r, \bar{S}_r \rangle$ where L_r is the set of all the cache levels that can be updated due to r, if we have $l_x \in L_r$, then safety means we should have $\psi(r)(l_x) \neq N$ when the analysis reaches its fixed-point). Lines 5 – 13 performs the same CAC derivation as described in [9]. Different from the traditional multi-level cache analysis method, we do not use the resultant CAC to update this cache level's state directly but we use it to derive the least upper bound for the reference's CACs at this level.

The second loop (lines 18 – 31) begins from the last cache level l_n and moves upwards through L in sequence to carry out updating. It first acquires the least upper bound of the CACs at a cache level l_x (line 19) and updates the level's abstract state ϕ_{l_x} according to the acquired CAC[5]. The abstract cache

[5]If the acquired CAC is A or N, it means the CAC for this reference at this level is always the same (A or N) during the analysis iterations.

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

Input: $\omega = \langle \phi_1, \cdots, \phi_n \rangle$, ψ, r
Output: $\omega' = \langle \phi'_1, \cdots, \phi'_n \rangle$, ψ
1 $x \leftarrow 1$;
2 CAC $\leftarrow A$;
3 **repeat**
4   $\psi \leftarrow \mathcal{F}(\psi, r, l_x, \text{CAC})$;
5   CHMC $\leftarrow chmc(\phi_{l_x}, m^r_{l_x})$;
6   **if** CAC = A **then**
7   **if** CHMC = *AH* **then** CAC $\leftarrow N$;
8   **else if** CHMC = *AM* **then** CAC $\leftarrow A$;
9   **else** CAC $\leftarrow U$;
10   **else if** CAC = U **then**
11   **if** CHMC = *AH* **then** CAC $\leftarrow N$;
12   **else** CAC $\leftarrow U$;
13   **else** CAC is unchanged (i.e. *N*) ;
14   $x \leftarrow x + 1$;
15 **until** $x > n$;
16 $x \leftarrow n$;
17 $\omega' = \omega$;
18 **repeat**
19   CAC $\leftarrow \psi(r)(l_x)$;
20   **if** CAC = A **then** $\phi'_{l_x} \leftarrow \mathcal{U}(\phi'_{l_x}, m^r_{l_x})$;
21   **else if** CAC = N **then** $\phi'_{l_x} \leftarrow \phi'_{l_x}$;
22   **else** $\phi'_{l_x} \leftarrow \mathcal{J}(\mathcal{U}(\phi'_{l_x}, m^r_{l_x}), \phi'_{l_x})$
23   **if** $inc(l_x)$ **then**
24     $PO \leftarrow pout(\phi'_{l_x})$;
25   **foreach** $1 \leq y < x$ **do**
26       $PO' \leftarrow \emptyset$;
27   **foreach** $m_{l_x} \in PO$ **do**
28         $PO' \leftarrow PO' \cup \{m_{l_y} \in M_{l_y} | m_{l_y} \subseteq m_{l_x}\}$;
29       $\phi'_{l_y} \leftarrow \mathcal{I}(\phi'_{l_y}, PO')$;
30   $x \leftarrow x - 1$;
31 **until** $x = 0$;

level state updating (lines 20 – 22) uses the traditional method described in [9]. After updating the state, it checks whether l_x is an inclusive level, and tries to invalidate the memory blocks in the abstract cache states of the levels located above l_x if it is inclusive (lines 23 – 29). Line 24 ensures that the set of memory blocks that cause invalidations at upper levels is over-approximated, and lines 26 – 28 extracts all the sub-blocks *w.r.t.* the cache block size at an upper level. This loop abstracts the behavior of step (3) and partial behavior of step (6) of the concrete operational semantics. The rest of the behavior of the steps is abstracted in the $\mathcal{U}$ and $\mathcal{I}$ functions.

C. Abstract Domain for A Cache Level

According to the conditions introduced in Section V-B for the abstract cache level state domain, it is actually to perform the three analyses (i.e. the *may*, *must*, and *persistence*) on each level at the same time. Therefore, we define the abstract cache state domain Φ_{l_x} for a cache level l_x as:

$$\Phi_{l_x} = \Theta^{may}_{l_x} \times \Theta^{must}_{l_x} \times \Theta^{pers}_{l_x} \times \Delta_{l_x}$$

where $\Theta^{may}_{l_x}$ is the set of all the ACSs in terms of *may* analysis of the cache level l_x in isolation; likewise, $\Theta^{must}_{l_x}$ and $\Theta^{pers}_{l_x}$ are that in terms of *must* and *persistence* analysis, respectively. Note that the $\Theta^{may}_{l_x}$ and $\Theta^{must}_{l_x}$ we use are consistent with the ones defined in [6], namely:

$$\Theta^{may}_{l_x} = \Theta^{must}_{l_x} = \{1, \cdots, d_{l_x}\} \rightarrow \left(\{1, \cdots, k_{l_x}\} \rightarrow 2^{M_{l_x}}\right)$$

However, there are different approaches to safe cache *persistence* analysis, in which different abstract domains are used [5], [11]. No matter which approach is used, for a cache set i, there is an additional logical position $\top_{l_x} \equiv k_{l_x} + 1$. Given a $\theta^{pers}_{l_x} \in \Theta^{pers}_{l_x}$, $\theta^{pers}_{l_x}(i)(\top_{l_x})$ is an over-approximated set of memory blocks that are possibly evicted after being loaded into this cache set. The last component domain Δ_{l_x} is defined as:

$$\Delta_{l_x} = \{1, \cdots, d_{l_x}\} \rightarrow \{1, \cdots, k_{l_x}, \top_{l_x}\}$$

An element in the domain Δ_{l_x} is used to capture the minimum logical position in which there may be an invalidated memory block for each cache set.

For simplicity, let us use Φ directly in the following without its subscript l_x. Three semantic functions need to be defined as described in Section V-B.

Given two abstract states $\phi_1 = \langle \theta^{may}_1, \theta^{must}_1, \theta^{pers}_1, \delta_1 \rangle \in \Phi$ and $\phi_2 = \langle \theta^{may}_2, \theta^{must}_2, \theta^{pers}_2, \delta_2 \rangle \in \Phi$, the *join* function $\mathcal{J} : \Phi \times \Phi \rightarrow \Phi$ is defined as:

$$\mathcal{J}(\phi_1, \phi_2) = \langle \mathcal{J}^{may}(\theta^{may}_1, \theta^{may}_2), \mathcal{J}^{must}(\theta^{must}_1, \theta^{must}_2), \mathcal{J}^{pers}(\theta^{pers}_1, \theta^{pers}_2), \delta_\perp \rangle$$

where $\mathcal{J}^{may}$ is the traditional *join* function used for single-level cache *may* analysis (so are $\mathcal{J}^{must}$ and $\mathcal{J}^{pers}$), and $\delta_\perp \in \Delta$ is defined as $\forall i \in \{1, \cdots, d_{l_x}\} : \delta_\perp(i) = \top_{l_x}$.

Given a set PO of possibly invalidated memory blocks, the *invalidate* function $\mathcal{I} : \Phi \times 2^M \rightarrow \Phi$ is defined in Algorithm 2. It (lines 7 – 10) removes any possibly invalidated memory block from the *must* and *persistence* components and puts the removed block into the corresponding $\top$ position. It (lines 4 – 6) also captures the minimum logical position in which there may be an invalidated memory block for each cache set. Note that we do not remove any block from the *may* component since a block in PO may not cause invalidation.

Algorithm 2: Definition of the *invalidate* function $\mathcal{I}$

Input: $\phi = \langle \theta^{may}, \theta^{must}, \theta^{pers}, \delta \rangle$, PO
Output: $\phi' = \langle \theta^{may'}, \theta^{must'}, \theta^{pers'}, \delta' \rangle$
1 $\phi' \leftarrow \phi$;
2 $\delta' \leftarrow \delta_\perp$;
3 **foreach** $m \in PO$ **do**
4   **if** $m \in \theta^{may'}(i)(j)$ **then**
5   **if** $j < \delta'(i)$ **then**
6       $\delta'(i) = j$;
7   **if** $m \in \theta^{must'}(i)(j)$ **then**
8   remove m from $\theta^{must'}(i)(j)$;
9   **if** $m \in \theta^{pers'}(i)(j)$ **then**
10   remove m from $\theta^{pers'}(i)(j)$ and put it into $\theta^{pers'}(i)(\top)$;

According to the concrete operational semantics, we know that if a memory block in the logical position p of a cache set is invalidated, loading a new block into this cache set will not increment the logical positions of the blocks which are greater than p. When updating an abstract state, if we do not consider this behavior, it will not affect the safety but only the precision of the *must* and *persistence* analyses (both analyses stay safe as long as the logical positions of memory blocks are not underestimated). However, without considering this behavior

we can have an unsafe *may* analysis, since it is required not to overestimate the logical positions of memory blocks in the *may* analysis. Therefore, we extend the traditional *update* function $\mathcal{U}^{may}$ to $\overline{\mathcal{U}}^{may} : \Theta^{may} \times M \times \Delta \to \Theta^{may}$ for the *may* analysis to take into account any possible invalidation behavior:

$$\overline{\mathcal{U}}^{may}(\theta^{may}, m, \delta) = \begin{cases} \mathcal{U}^{may}(\theta^{may}, m) & \text{if } p \geq j \\ \theta^{may}[set(m) \mapsto \epsilon] & \text{otherwise} \end{cases}$$

where

$$p = \delta(set(m)) \bigwedge j = \begin{cases} h & \text{if } m \in \theta^{may}(set(m))(h) \\ \top & \text{otherwise} \end{cases} \bigwedge$$

$$\epsilon = \begin{array}{l} \big[\iota_1 \mapsto \{m\}, \\ \quad \iota_i \mapsto \theta^{may}(set(m))(\iota_{i-1}) | i = 2 \cdots p, \\ \quad \iota_{p+1} \mapsto \theta^{may}(set(m))(\iota_p) \cup (\theta^{may}(set(m))(\iota_{p+1}) \backslash \{m\}), \\ \quad \iota_i \mapsto \theta^{may}(set(m))(\iota_i) \backslash \{m\} | i = p+2 \cdots k\big] \end{array}$$

If there is no invalidation possible in this cache set (i.e. $p = \top$), or if m is in θ^{may} and its logical position is not behind the minimum position p where an invalidated memory block may reside, using the traditional $\mathcal{U}^{may}$ will not overestimate the logical positions of memory blocks. Otherwise, we need to consider there may be a "hole" behind p (including p) that needs to be filled first; so we can only increment the logical positions of the memory blocks until p, and keep the positions of other blocks unchanged (excluding m which will be moved to the first logical position if it is in the current θ^{may}). Based on this, given an abstract state $\phi = \langle \theta^{may}, \theta^{must}, \theta^{pers}, \delta \rangle \in \Phi$, we define the *update* function $\mathcal{U} : \Phi \times M \to \Phi$ as:

$$\mathcal{U}(\phi, m) = \langle \overline{\mathcal{U}}^{may}(\theta^{may}, m, \delta), \mathcal{U}^{must}(\theta^{must}, m), \mathcal{U}^{pers}(\theta^{pers}, m), \delta_{\perp} \rangle$$

where $\mathcal{U}^{must}$ and $\mathcal{U}^{pers}$ are the traditional *update* functions for *must* and *persistence* analysis, respectively.

In [22], an abstract domain called aging barrier is proposed to improve precision. Therefore, Φ can be extended by including that domain as a component domain (and also extend the *update* functions for *must* and *persistence* analyses). However, there are two reasons why we do not include this domain: (1) it is not necessary for safety of the analysis; (2) it may not be useful in the absence of data references.

D. Concretization

The set of concrete cache hierarchy states represented by an abstract cache hierarchy state $\omega = \langle \phi_{l_1}, \cdots, \phi_{l_n} \rangle$ where $\phi_{l_x} = \langle \theta^{may}_{l_x}, \theta^{must}_{l_x}, \theta^{pers}_{l_x}, \delta_{l_x} \rangle$ $(1 \leq x \leq n)$ can be derived by a concretization function $con_{\Omega} : \Omega \to 2^{\bar{S}}$. This concretization function is defined as:

$$con_{\Omega}(\omega) = \cdots \times \left(\gamma^{may}_{l_x}(\theta^{may}_{l_x}) \cap \gamma^{must}_{l_x}(\theta^{must}_{l_x}) \cap \gamma^{pers}_{l_x}(\theta^{pers}_{l_x}) \right) \times \cdots$$

where γ^{may}, γ^{must}, γ^{pers} are the well-defined concretization functions in [6] for the *may*, *must*, and *persistence* abstract states respectively. For an abstract cache level state, the set of concrete cache level states is derived independently of other levels. A concrete cache level state should satisfy the *may*, *must*, and *persistence* meanings of the abstract cache level state, so set intersection is used to guarantee this.

The meaning of the global abstract object $\psi \in \Psi$ is given by another concretization function $con_{\Psi} : \Psi \to \left(R \to 2^L\right)$ that is defined as:

$$con_{\Psi}(\psi) = \left[r \in R \mapsto \{l \in L | \psi(r)(l) \neq N\}\right]$$

For each reference r, this concretization function determines a set of cache levels that may be accessed by r. Given a cache level l, if we have $\psi(r)(l) \neq N$ (i.e. *A* or *U*), r may access l and l is in the derived set of cache levels for r.

E. Discussion on Data References

Since most of the inclusive caches are unified caches, data references need to be taken into account eventually. As always, one difficulty related to data references is to precisely derive a set of possibly referenced memory addresses for each dynamic load/store instruction [3], [11]. In addition, we need to consider which write policy is used in the cache hierarchy. For example, if write-back policy is used, a dirty block needs to be written into a lower level when it is evicted, which changes the cache state at that level. Thus, we need to safely classify whether a memory block is dirty (like *always*, *never*, or *uncertainly* dirty) and take into account the effect of eviction of dirty blocks on lower cache level states.

VI. Evaluation

In this section, we evaluate the approach proposed in this paper and compare with state-of-the-art approaches to multi-level inclusive cache analysis proposed in [10] and [22]. First, we describe the evaluation setup and also some assumptions in the experiments which nevertheless do not affect the validity of the evaluation. Then, we present the evaluation results showing the proposed approach improves precision, and we also show the computational overhead associated with the evaluation.

A. Evaluation Setup

We have developed a research prototype tool that implements the three approaches that are proposed in this paper, [10], and [22]. The tool takes MIPS R3000 compliant binaries and reconstructs CFGs from them. It also computes context-sensitive call graphs to improve the analysis precision. The evaluations are carried out on a set of benchmarks (as shown in Tab. II) maintained by the Mälardalen WCET research group [7], and they are compiled for MIPS R3000 using gcc-3.4.4.

Due to the limitations of our current tool, we only take into account the timing effects of multi-level inclusive instruction caches and do not consider data references for now as argued in Section III. We also do not consider the effects of other micro-architectural features like pipelines and branch predictors, so we assume there are no timing anomalies. Therefore, a memory reference that is classified as *NC* can be safely treated as a *AM* when used to estimate the WCET.

The IPET (Implicit Path Enumeration Technique) is used to calculate the WCET bound [16]. It uses a set of integer linear constraints to combine the flow information and timing effects of multi-level caches [10], [14]. In terms of the flow information, the structural constraints are generated automatically, but

currently the loop bounds are determined and input manually. The CPLEX solver is used to solve the generated ILP (Integer Linear Programming) problems.

We carry out the experiments on a two-level cache hierarchy and set L2 as inclusive. The fixed parameters of the hierarchy are shown in Tab. I. We also assume every needed information can be found in the main memory with a 100-cycle latency.

TABLE I
FIXED PARAMETERS OF TWO-LEVEL CACHE HIERARCHY

Level	Block Size	Associativity	Latency
L1	8B	2-way	1-cycle
L2	16B	4-way	10-cycle

Three experiments are performed for each benchmark under different cache capacity configurations. Let $size(L1)$ denote the capacity of L1 cache, and let $size(L2)$ denote the capacity of L2 cache. Let us assume that L2 cache size is always four times bigger than that of L1 cache, namely we have $size(L1) = \frac{size(L2)}{4}$. For a benchmark bm, let $size(bm)$ denote the code size of this benchmark. The cache size configurations for each benchmark are shown in Tab. II, whose criteria are as follows:

Large: L2 cache size is not smaller than the code size, and it satisfies $size(L2) \geq 2 \times size(bm) > \frac{size(L2)}{2}$.

Medium: L2 cache size is not bigger than the code size, and it satisfies $size(L2) \leq size(bm) < 2 \times size(L2)$.

Small: L2 cache size is not bigger than half the code size, and it satisfies $size(L2) \leq \frac{size(bm)}{2} < 2 \times size(L2)$.

TABLE II
LARGE, MEDIUM, AND SMALL SIZE CONFIGS FOR EACH BENCHMARK

Benchmark	Code Size	Large		Medium		Small	
		L1	L2	L1	L2	L1	L2
bs	320B	256B	1KB	64B	256B	32B	128B
insertsort	440B	256B	1KB	64B	256B	32B	128B
janne	324B	256B	1KB	64B	256B	32B	128B
cnt	944B	512B	2KB	128B	512B	64B	256B
expint	888B	512B	2KB	128B	512B	64B	256B
fir	600B	512B	2KB	128B	512B	64B	256B
ns	588B	512B	2KB	128B	512B	64B	256B
prime	556B	512B	2KB	128B	512B	64B	256B
qurt	1328B	1KB	4KB	256B	1KB	128B	512B
select	1580B	1KB	4KB	256B	1KB	128B	512B
compress	3564B	2KB	8KB	512B	2KB	256B	1KB
edn	3576B	2KB	8KB	512B	2KB	256B	1KB
jfdctint	2580B	2KB	8KB	512B	2KB	256B	1KB
lms	2588B	2KB	8KB	512B	2KB	256B	1KB
ludcmp	2276B	2KB	8KB	512B	2KB	256B	1KB
minver	3052B	2KB	8KB	512B	2KB	256B	1KB
ndes	3392B	2KB	8KB	512B	2KB	256B	1KB
adpcm	7612B	4KB	16KB	1KB	4KB	512B	2KB
statemate	10296B	8KB	32KB	2KB	8KB	1KB	4KB
nsichneu	40036B	32KB	128KB	8KB	32KB	4KB	16KB

B. Experimental Results

Tab. III shows the experimental results. For a benchmark, WCET_1 is derived by the approach proposed in [10], WCET_2 is derived by the approach proposed in [22], and WCET_3 is derived by the approach proposed in this paper. The estimates are reported in clock cycles. The precision improvement compared to the approach proposed in [10] and [22] is calculated by $\frac{\text{WCET}_1}{\text{WCET}_3} - 1$ and $\frac{\text{WCET}_2}{\text{WCET}_3} - 1$ respectively. 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.

As shown by the results, the proposed approach dominates the other two approaches in most cases. Under the large cache size configuration, the proposed approach performs almost the same as the approach proposed in [22]. For only a few benchmarks, WCET_3 is strictly lower than WCET_2 but the precision improvement is only within 3%. However, both the approaches perform better than the approach proposed in [10]. This is expected and reasonable, since the approach proposed in [10] does not try to classify any access as *A* or *N* instead of U at a lower level than L1.

A striking difference appears under the medium and small configurations. Under both configurations, the approach proposed in this paper gives above 10% improvement in most cases. For some benchmarks, the precision improvement is very significant (over 100%). Due to this large improvement, we also analyze some small benchmarks by hand to find out the reasons. For example, under the medium configuration, for *insertsort*, the approach proposed in this paper can achieve more than 170% improvement compared to the other two approaches. The reason for this is explained as follows: *insertsort* has two nested loops, and the total size of the two nested loops is 228 bytes which is smaller than the L2 cache size under the medium configuration which is 256 bytes; thus, if a reference in the loops cannot be classified as L1 *AH*, it should at least be classified as L2 *PS* if not L2 *AH*. While the approach proposed in this paper achieves this (i.e. it gives L2 *AH* or L2 *PS* to the references in the loops if they cannot be classified as L1 *AH*), the other two approaches fails to give such L2 classifications to the references in the loops (they assign L2 *NC* to them).

When applying the proposed approach, an interesting phenomenon is that a *smaller* cache size configuration may result in a *tighter* estimate (e.g. *bs*, *qurt*, *select*, *compress*, *statemate*). As observed from the results of *bs* benchmark, the calculated bound under the large cache size configuration (4027 clock cycles) is higher than that under the medium one (3757 clock cycles). The reason for such a phenomenon is as follows: The code size of *bs*'s loop is 208 bytes which is smaller than 256 bytes, so most of the blocks of the loop are at least persistent if they are not always in both L1 and L2 caches under the large configuration; whereas, most of the blocks are persistent in L2 cache under the medium configuration, but their sub-blocks will be evicted from L1 cache along with the loop iterations (since L1 cache size under the medium configuration is only 64 bytes and the shortest path in the loop involves 128 bytes). Thus, there are several references which are classified as L1 *PS* & L2 *PS* under the large configuration, and are classified as L1 *AM* & L2 *PS* under the medium configuration. Given such a reference under the large configuration, L1 *PS* & L2 *PS* means that the corresponding L2 block is not in the $\theta_{l_2}^{must}$ and its L2 *CAC* is *U*; and it cannot bring the corresponding L2 block into the $\theta_{l_2}^{must}$; thus, its subsequent references to the same L2 block can only be classified as L2 *PS* (if not L1 *AH*) but not L2 *AH*; namely, such subsequent ones can add additional main memory access latencies (100 clock cycles)

TABLE III
EXPERIMENTAL RESULTS: WCET ESTIMATES AND COMPUTATION TIME OVERHEADS

Benchmark	Configuration	Approach in [10]		Approach in [22]		Approach proposed		$\frac{WCET_1}{WCET_3}$-1	$\frac{WCET_2}{WCET_3}$-1
		$WCET_1$	Overhead	$WCET_2$	Overhead	$WCET_3$	Overhead		
bs	Large	4827	0	4027	0.1	4027	0.1	19.87%	0.00%
	Medium	7357	0	6857	0	3757	0.1	95.82%	82.51%
	Small	10079	0	9579	0	6679	0	50.90%	43.42%
insertsort	Large	17229	0.1	15929	0.2	15929	0.2	8.16%	0.00%
	Medium	187559	0	186359	0.1	68959	0.1	171.99%	170.24%
	Small	187559	0	186359	0.1	110859	0.1	69.19%	68.10%
janne	Large	5863	0	4963	0.1	4963	0.1	18.13%	0.00%
	Medium	20577	0	20077	0.1	15077	0.1	36.48%	33.16%
	Small	33767	0	33267	0	24167	0	39.72%	37.65%
cnt	Large	27176	0.4	25176	0.8	25176	0.7	7.94%	0.00%
	Medium	362188	0.2	358308	0.4	277508	0.6	30.51%	29.12%
	Small	537178	0.1	533298	0.3	289198	0.3	85.75%	84.41%
expint	Large	30737	0.2	29537	0.4	29337	0.3	4.77%	0.68%
	Medium	183462	0.1	171262	0.2	109162	0.3	68.06%	56.89%
	Small	474762	0.1	473162	0.2	322262	0.2	47.32%	46.83%
fir	Large	15643	0.2	14443	0.4	14443	0.5	8.31%	0.00%
	Medium	196536	0.1	170586	0.2	136786	0.3	43.68%	24.71%
	Small	380736	0.1	379636	0.1	213936	0.1	77.97%	77.45%
ns	Large	33601	0.3	31601	0.4	31601	0.4	6.33%	0.00%
	Medium	164491	0.1	162591	0.2	162291	0.2	1.36%	0.18%
	Small	1473161	0.1	1472305	0.2	972305	0.2	51.51%	51.42%
prime	Large	38044	0.3	36644	0.5	36644	0.4	3.82%	0.00%
	Medium	190664	0.1	189164	0.2	185064	0.3	3.03%	2.22%
	Small	1690694	0.1	1689194	0.2	612594	0.1	175.99%	175.74%
qurt	Large	43705	3.3	40905	6.2	39897	5.4	9.54%	2.53%
	Medium	175156	1.3	168176	2.5	150276	2.8	16.56%	11.91%
	Small	182356	1.0	179556	2.0	114056	1.5	59.88%	57.43%
select	Large	43264	3.0	42264	5.5	41064	6.6	5.36%	2.92%
	Medium	237114	1.0	236194	1.8	168594	3.2	40.64%	40.10%
	Small	237114	0.5	202594	1.1	153394	1.7	54.58%	32.07%
compress	Large	217433	13.0	213333	25.2	212933	26.7	2.11%	0.19%
	Medium	1624044	5.1	1614444	9.5	1472344	11.7	10.30%	9.65%
	Small	1624044	2.9	1614444	5.5	1299144	6.7	25.01%	24.27%
edn	Large	217583	19.1	211483	25.8	211483	35.6	2.88%	0.00%
	Medium	740873	5.7	734663	9.4	602663	9.8	22.93%	21.90%
	Small	2360263	2.8	2354263	5.3	2254963	5.3	4.67%	4.40%
jfdctint	Large	42825	11.9	42225	14.7	42125	20.9	1.66%	0.24%
	Medium	164895	3.2	164295	4.8	48195	4.1	242.14%	240.90%
	Small	265895	1.9	197895	3.2	107895	1.8	146.44%	83.41%
lms	Large	480987	8.9	479187	18.0	479187	16.2	0.38%	0.00%
	Medium	10438520	4.7	10414720	8.8	10410720	10.4	0.27%	0.04%
	Small	25358756	2.7	25047856	5.5	21404296	8.7	18.48%	17.02%
ludcmp	Large	40885	3.9	39285	7.2	39285	6.8	4.07%	0.00%
	Medium	61245	2.4	59645	4.3	59645	4.5	2.68%	0.00%
	Small	293163	1.4	292263	2.8	290063	3.8	1.07%	0.76%
minver	Large	56314	6.7	53914	11.9	53914	12.9	4.45%	0.00%
	Medium	73759	3.4	71159	6.1	70759	7.3	4.24%	0.57%
	Small	228073	2.1	224333	3.9	215233	5.7	5.97%	4.23%
ndes	Large	204676	12.1	202376	21.6	202076	25.4	1.29%	0.15%
	Medium	2967930	5.6	2926910	10.0	2926010	14.4	1.43%	0.03%
	Small	5615987	3.2	5497527	6.4	4845877	11.0	15.89%	13.45%
adpcm	Large	388500	105.1	385600	199.5	384300	233.0	1.09%	0.34%
	Medium	1262808	37.4	1251888	71.3	1127392	111.2	12.01%	11.04%
	Small	1640818	24.9	1600638	49.9	1524568	45.3	7.63%	4.99%
statemate	Large	102633	124.0	96733	230.9	96533	233.6	6.32%	0.21%
	Medium	159710	54.5	153150	93.9	133530	143.8	19.61%	14.69%
	Small	160220	33.1	153560	58.5	119560	112.0	34.01%	28.44%
nsichneu	Large	610498	3023.7	610198	6327.7	608598	8880.6	0.31%	0.26%
	Medium	1096788	897.3	1096688	1491.0	1088088	5675.8	0.80%	0.79%
	Small	1144988	528.6	1144888	887.3	1138288	3725.9	0.59%	0.58%

to the total estimate[6]. On the contrary, such a reference under the medium configuration is classified as L1 *AM* & L2 *PS*, so its L2 *CAC* is *A* which enables it to bring the corresponding L2 block into the $\theta_{l_2}^{must}$; some of its subsequent references to the same L2 block can be classified as L2 *AH* instead of L2 *PS*. Thus, the number of L1 misses under the medium one is proportional to the number of the iterations and can be much more than that under the large one; however, the number of overestimated L2 misses under the medium one can be less than that under the large one. Given the loop bound is only 4 and the difference between the latency of L2 access and main memory access is big, this phenomenon occurs. This phenomenon also shows the proposed approach can tightly analyze multi-level inclusive caches.

Compared to the other two approaches, the proposed one

[6]If a reference is not classified as L1 *AH* and is classified as L2 *PS*, it can contribute at most one main memory access latency even it is in a loop.

requires more analysis time, especially in the case of *nsichneu* benchmark (the biggest one we use). This is because the proposed approach needs to perform more iterations due to the CACs for many references at L2 changing from *A* or *N* to *U*.

VII. Related Work

The first multi-level (non-inclusive) cache analysis is proposed in [18], which is an extension to a well-established single-level cache analysis method called static cache simulation [19]. Later, in [9], it is pointed out that this method is actually unsafe for analyzing multi-level set associative caches, and it is proposed to use *cache access classification* (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 [9] which does not take into account data caches, a method for analyzing multi-level non-inclusive data caches is proposed in [14], and a method for analyzing non-inclusive cache hierarchies with unified caches is proposed in [3]. In [20], 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 [21], a dual-core processor with a shared L2 cache model is considered. In [15], task lifetime information is computed and utilized to refine possible interferences. In [8], a method for identifying and bypassing the static single usage memory blocks so as to reduce the number of interferences is proposed. In [17], abstract interpretation based cache analysis is combined with model checking based bus analysis to achieve more precise interference analysis. In [2], 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 [10], 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. In [22], an approach is proposed to improve the tightness of the estimation when analyzing cache hierarchies that contain inclusive cache levels. It first analyzes all the inclusive levels in a *bottom-up* direction, and then analyzes the rest of the levels in a *top-down* direction. By doing this bidirectional analysis, partial invalidation behavior caused by an inclusive level can become visible when analyzing any of its upper levels.

In order to avoid using too pessimistic estimation, probabilistic timing analysis (PTA) techniques have been proposed to produce multiple estimations with the probabilities that they can be exceeded. In [13], a measurement-based PTA approach is proposed to estimate probabilistic WCET in the presence of multi-level caches. In this paper, we want to guarantee safety, and consider PTA as a complementary methodology.

VIII. Conclusion

In this paper, we propose an approach that can safely and more precisely analyze multi-level inclusive caches for WCET estimation. We first define a concrete operational semantics for multi-level inclusive caches. Based on this concrete semantics, the proposed approach analyze a multi-level inclusive cache as a whole by integrating three analyses together. We evaluate the proposed approach using a set of benchmarks. From the experimental results, we can observe the proposed approach can significantly tighten the WCET bound under relatively small cache size configurations, compared to other approaches.

Acknowledgment

This work is supported in part by the NSF (CNS-1035655).

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] 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.

[3] S. Chattopadhyay and A. Roychoudhury. Unified cache modeling for wcet analysis and layout optimizations. In *RTSS '09*, pages 47–56, 2009.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*, pages 238–252, 1977.

[5] C. Cullmann. Cache persistence analysis: Theory and practice. *ACM Trans. Embed. Comput. Syst.*, 12(1s):40:1–40:25, Mar. 2013.

[6] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-timesystems. *Real-Time Syst.*, 17(2-3):131–181, Dec. 1999.

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

[8] 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.

[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] B. K. Huynh, L. Ju, and A. Roychoudhury. Scope-aware data cache analysis for wcet estimation. In *RTAS '11*, pages 203–212, 2011.

[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] L. Kosmidis, J. Abella, E. Quiones, and F. J. Cazorla. Multi-level unified caches for probabilistically time analysable real-time systems. In *RTSS '13*, pages 360–371, 2013.

[14] B. Lesage, D. Hardy, and I. Puaut. WCET Analysis of Multi-Level Set-Associative Data Caches. In *WCET '09*, pages 1–12, 2009.

[15] 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.

[16] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC '95*, pages 456–461, 1995.

[17] 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.

[18] F. Mueller. Timing predictions for multi-level caches. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 29–36, 1997.

[19] F. Mueller. Timing analysis for instruction caches. *Real-Time Syst.*, 18(2/3):217–247, May 2000.

[20] 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.

[21] J. Yan and W. Zhang. Wcet analysis for multi-core processors with shared l2 instruction caches. In *RTAS '08*, pages 80–89, 2008.

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

APPENDIX

Theorem 1. *The proposed approach to multi-level inclusive cache analysis is safe.*

Proof: Given a reference r, the collecting semantics cs where $cs(r) = \langle L_r, \bar{S}_r \rangle$, the derived global abstract object ψ, and the derived abstract cache hierarchy state ω at r, if the analysis is safe, it should satisfy the following condition:

$$L_r \subseteq con_\Psi(\psi)(r) \bigwedge \bar{S}_r \subseteq con_\Omega(\omega)$$

We prove this by mathematical induction.

Base case: At the beginning of an execution, no memory block is loaded and the first reference needs to access all the cache levels. The abstract cache hierarchy state at the entry point derived by the analysis is empty. By the definition of the function $\mathscr{U}$ the first reference is certainly classified as *AM* at each cache level, so the ψ maps the first reference to *A* at each cache level. The condition holds.

Inductive hypothesis: Let T be the set of all the predecessor references of r. In the last iteration of the analysis, we have $\forall t \in T : L_t \subseteq con_\Psi(\psi)(t) \bigwedge \bar{S}_t \subseteq con_\Omega(\omega')$ where $cs(t) = \langle L_t, \bar{S}_t \rangle$. For simplicity, let us assume r only has one predecessor t, so we have $\omega = \mathscr{U}(\omega', \psi, t)$. If r has more than one predecessor, we can use the $\mathscr{J}$ function, which is safe by construction, to combine the abstract cache hierarchy states. Therefore, we need to prove ω' updated by t is safe and ψ updated by r is also safe.

Inductive step: When updating ω', since the CAC for t at each level is safe (given by the hypothesis $L_t \subseteq con_\Psi(\psi)(t)$), updating the components of the last level in ω' will be safe. Let us assume the last level is inclusive: Since its components are safely updated, the set of possibly evicted blocks will be overestimated. According to the definition of the $\mathcal{I}$ function which remove blocks in the *must* and *persistence* state components, over-removal in these two state components made by the $\mathcal{I}$ function will not make the analysis unsafe. As described in Section V-C, the *update* function $\mathcal{U}$ can safely update an abstract cache level state. By using mathematical induction on levels, we can prove the component for each level in ω' can be safely updated. Therefore, the ω' updated by t is also safe, i.e. the abstract cache hierarchy state ω is safe. Thus, we have $\bar{S}_r \subseteq con_\Omega(\omega)$. When updating $\psi(r)$, r's CHMC at each level is derived from the ω which is safe as proven above. Since we have $\bar{S}_r \subseteq con_\Omega(\omega)$, the CHMC for r at each level is safe, so is the deduced CAC. The $\mathcal{F}$ function uses the least upper bound operator to derive the reference's possible CAC. Thus, the ψ updated by r is safe, i.e. $L_r \subseteq con_\Psi(\psi)(r)$. ■

In order to prove the proposed approach terminate, we need a well-defined partial ordering on each abstract domain. Given two abstract cache hierarchy states $\omega_1 = \langle \phi_{1,l_1}, \cdots, \phi_{1,l_n} \rangle$ and $\omega_2 = \langle \phi_{2,l_1}, \cdots, \phi_{2,l_n} \rangle$, the partial ordering $\preceq_\Omega$ on the the abstract cache hierarchy state domain Ω is defined as:

$$\omega_1 \preceq_\Omega \omega_2 \iff \phi_{1,l_1} \preceq_\Phi \phi_{2,l_1} \bigwedge \cdots \bigwedge \phi_{1,l_n} \preceq_\Phi \phi_{2,l_n}$$

where $\preceq_\Phi$ on the abstract cache level state domain Φ is defined naturally as the conjunction of the orders on the corresponding components in the Θ^{may}, Θ^{must}, Θ^{pers}, and Δ respectively. The partial orderings on the Θ^{may}, Θ^{must} and Θ^{pers} abstract cache state domains are already well-defined, so we define the partial ordering $\preceq_\Delta$ on the domain Δ. Given two elements δ_1 and δ_2 of the domain Δ, the partial ordering $\preceq_\Delta$ on the domain Δ is defined as:

$$\delta_1 \preceq_\Delta \delta_2 \iff \forall i \in \{1, \cdots, d\} : \delta_1(i) \geq \delta_2(i)$$

We also need to define the partial ordering on the domain Ψ. Given two elements ψ_1 and ψ_2 of the domain Ψ, the partial ordering $\preceq_\Psi$ on the domain Ψ is defined as:

$$\psi_1 \preceq_\Psi \psi_2 \iff \forall r \in R, \forall l \in L : \psi_1(r)(l) \preceq_{CAC} \psi_2(r)(l)$$

where $\preceq_{CAC}$ on the CAC domain is defined by the lattice as shown in Fig. 3.

Theorem 2. *The proposed approach to multi-level inclusive cache analysis terminates in finite iterations.*

Proof: The abstract cache hierarchy state domain Ω and the Ψ domain are finite and partially ordered, so if both $\mathscr{J}$ and $\mathscr{U}$ functions are monotone, the proposed analysis is guaranteed to terminate in finite iterations.

The $\mathscr{J}$ function only applies the monotone *join* functions of the *may*, *must*, and *persistence* analyses independently to the corresponding components of two abstract cache hierarchy states, so it is monotone by construction.

The $\mathscr{U}$ function is composed of four functions $\mathcal{F}$, $\mathcal{I}$, $\mathcal{U}$, and $\mathcal{J}$: (1) For any reference, the $\mathcal{F}$ function uses the $\sqcup$ operator on the reference's all possible CACs to derive a least upper bound. Thus, given $\psi_1 \preceq_\Psi \psi_2$, for any reference r at a cache level l with its CAC c, we have $\mathcal{F}(\psi_1, r, l, c) \preceq_\Psi \mathcal{F}(\psi_2, r, l, c)$. (2) The $\mathcal{I}$ function removes blocks only from θ^{must} and θ^{pers} of an abstract cache level state. Thus, given $\phi_1 \preceq_\Phi \phi_2$ and $PO_1 \subseteq PO_2$, we can easily verify the resultant θ_1^{must} from $\mathcal{I}(\phi_1, PO_1)$ and the resultant θ_2^{must} from $\mathcal{I}(\phi_2, PO_2)$ have the relation: $\theta_1^{must} \preceq^{must} \theta_2^{must}$; also the resultant θ_1^{pers} and θ_2^{pers} have the relation: $\theta_1^{pers} \preceq^{pers} \theta_2^{pers}$. Since the θ_1^{may} and θ_2^{may} are not changed by the $\mathcal{I}$ function (i.e. $\theta_1^{may} \preceq^{may} \theta_2^{may}$ still holds) and $PO_1 \subseteq PO_2$, we can also easily verify the resultant δ_1 and δ_2 have the relation: $\delta_1 \preceq_\Delta \delta_2$. (3) The $\mathcal{U}$ function is composed of the $\overline{\mathcal{U}}^{may}$ function and the well-defined monotone *update* functions $\mathcal{U}^{must}$ and $\mathcal{U}^{pers}$. Thus, the $\mathcal{U}$ function is monotone as long as the $\overline{\mathcal{U}}^{may}$ function is monotone. Given a memory block m, $\theta_1^{may} \preceq^{may} \theta_2^{may}$, and $\delta_1 \preceq_\Delta \delta_2$, we can verify $\overline{\mathcal{U}}^{may}(\theta_1^{may}, m, \delta_1) \preceq^{may} \overline{\mathcal{U}}^{may}(\theta_2^{may}, m, \delta_2)$, since $\delta_1 \preceq_\Delta \delta_2$ means we have $\delta_1(set(m)) \geq \delta_2(set(m))$ such that age increasing in θ_2^{may} is more conservative by the definition of the $\overline{\mathcal{U}}^{may}$ function. (4) The $\mathscr{J}$ function is monotone as shown above. Since the functions $\mathcal{F}$, $\mathcal{I}$, $\mathcal{U}$, and $\mathcal{J}$ are all monotone, the $\mathscr{U}$ function is monotone. ■