THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Computing 1, 13 Computing Drive, Singapore 117417

TR20/10

Scope-aware Data Cache Analysis for WCET Estimation

Bach Khoa Huynh, Lei Ju and Abhik Roychoudhury

October 2010

T e c h n i c a l R e p o r t

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

OOI Beng Chin
Dean of School

Scope-aware Data Cache Analysis for WCET Estimation

Bach Khoa Huynh Lei Ju Abhik Roychoudhury
National University of Singapore
E-mail: {huynhbac, julei, abhik}@comp.nus.edu.sg

***Abstract*—Caches are widely used in modern computer systems to bridge the increasing gap between processor speed and memory access time. On the other hand, the presence of caches, especially data caches, complicates the static worst case execution time (WCET) analysis. Correctness and tightness of WCET estimates are of crucial importance for system level design of embedded systems. In this report, we show that the originally proposed persistence analysis is both unsafe and pessimistic for worst-case cache behavior modeling. We propose a new update and join functions for persistence analysis and prove their soundness. Furthermore, we extend the semantics of memory block persistence, and propose a scope-aware persistence analysis which combines access pattern analysis and abstract interpretation. The dynamic behavior of a memory access is captured by its temporal scope (the loop iterations where a given memory block is accessed for a given data reference) during address analysis. Temporal scopes as well as loop hierarchy structure (the static scopes) are integrated and utilized to achieve a more precise abstract cache state modeling. We also prove the correctness of the proposed new persistence analysis.**

I. Introduction

Worst-case Execution Time (WCET) is a key metric for real-time embedded software. In hard real-time systems, WCET is an essential parameter for system level schedulability analysis, which ensures a set of tasks will always meet their deadlines. Static WCET analysis provides a safe bound on the maximum execution time of a program on a target platform over all possible program inputs. For cost-sensitive domains like automotive electronics, the WCET estimation must be tight for cost-effective design and resource dimensioning. On the other hand, modern processors contain performance enhancing features such as caches and pipeline whose run-time timing behavior is hard to predict statically. This makes micro-architectural modeling (building timing models for micro-architectural features such as caches) a key component of WCET analysis.

Timing models of instruction caches for WCET analysis have been well-studied [17]. However, static timing analysis of data cache behavior remains a major challenge for WCET analysis methods and tools. Accurate data cache modeling is of paramount importance for tight WCET analysis of data-intensive routines. However, the run-time computed access address (which data locations are accessed by different instances of an instruction) and dynamic cache behavior make it difficult to develop a tight yet flexible and scalable static analysis. Conservatively assuming that every memory access results in a cache miss yields a safe but pessimistic WCET estimate.

Different static data cache analysis techniques have been developed so far. Access pattern-based techniques (e.g., cache miss equation framework in [11]) achieve tight estimation, but are applicable to programs that contain *only* regular accesses with predictable patterns. On the other hand, abstract interpretation-based data cache analysis techniques ([10], [15]) work on general programs but suffer from large over-estimation. In this report, we first show that the original persistence analysis proposed in [9] and [10] is unsafe, i.e., the abstract cache state maintained in persistence analysis may *under-estimate* the worst case behaviors in a reachable concrete cache state. We show the safety issue can be fixed with our proposed *update* and *join* function with necessary proofs. Furthermore, we observe that the over-estimation in existing data cache persistence analysis ([10]) stems from the globally defined abstract domain. In particular, a coarse-grained address analysis is adopted to compute a set of memory blocks possibly referenced by a memory access, while temporal property of the access is ignored (e.g., a memory block can be accessed in only certain iterations of a loop execution). The approximation in the address analysis causes substantial over-estimation in WCET estimates. Moreover, traditionally the abstract interpretation computes fixed point of the abstract cache state conservatively for the entire program execution (disregarding cache behavior in specific program scopes), leading to large over-estimation. We propose a multi-level scope-aware persistence analysis that overcomes the pessimism and achieve tighter WCET estimation. In this technical report, we focus on proving the soundness of the proposed analysis.

II. Assumptions and Notations

In our cache analysis, we consider a memory hierarchy containing separated L1 instruction and data caches. We use the following notations to represent the instruction/data cache configuration and accessibility.

- Capacity C: size of the cache in number of bytes
- Block (line) size B: number of contiguous bytes to be loaded from memory to cache on each memory access.
- Associativity A: A-way set associative cache means that information stored at some addresses in memory could be loaded into any of A locations in the cache (depends on the cache replacement policy).

- Cache set $F = \langle f_1, \ldots, f_{(C/B)/A} \rangle$: A cache set f_i is a sequence of cache blocks (lines) $CL = \langle l_1, \ldots, l_A \rangle$ which contains all the A ways that can be addressed with the same index. $set(m)$ returns the cache set memory block m maps to.

We assume LRU (Least Recently Used) replacement policy is used to determine relative age of a memory block in the A-way associative cache set. Among common cache replacement policies, LRU is the most predictable policy thus more suitable for safety critical real-time systems [7]. Given a concrete cache state c at a program point p, the concrete set state s_i describes the state of cache set $c[f_i]$ at p. If $s_i(l_x) = m$, memory block m has a relative age x in $c[f_i]$ $(1 \leq x \leq A)$.

We assume write-through with no-write-allocate policy for a memory store instruction in our discussion of data cache analysis. However, our data cache analysis framework is applicable to different write policies with minor amendments in the analysis (discussed in Section VIII-A). We consider the static and temporal scope information of data references at the assembly code level in our analysis. Finally, we would like to clarify that our proposed persistence analysis (Section VIII) is "multi-level" in the sense that an independent analysis is performed at each loop nesting level (also referred as the static scope), which should not be confused with analysis of the multi-level caches (e.g., the L2, L3 caches).

III. Persistence Analysis

A. Overview

Persistence analysis determines if a memory block m is persistent: once loaded, it will not be evicted out of the cache in any possible execution. Therefore, the first access to a persistent memory block m *may* encounter a miss. However, all subsequent accesses are guaranteed to result in cache hits.

To determine if a memory block m is persistent at a *program point* p, the persistence analysis [9], [10] computes an *abstract cache state (ACS)* to determine *maximum relative age* x for each memory block m which *may* be in the cache when the program control reaches p. If x is not higher than cache associativity A, once loaded, m is not evicted from the cache at program point p in all possible executions. As a result, m is classified as persistent.

An ACS $\hat{c} = \langle \hat{s}_1, ..., \hat{s}_{n/A} \rangle$ at a program point p models an A-way set associative cache with n cache lines, n/A cache sets. Each *abstract set state* $\hat{s}_k = \langle l_1, ..., l_A, l_\top \rangle$ consists of A cache lines $l_1, ..., l_A$ and an additional evicted cache line $l_\top$ to record evicted memory blocks. For each memory block m, $\hat{s} = \hat{c}[set(m)]$ returns the abstract set state $\hat{s}$ in ACS $\hat{c}$ where m is mapped to. If $m \in \hat{s}(l_x)$, m has maximal relative age x in all possible concrete cache states when program control reaches p. If m is in evicted line $\hat{s}(l_\top)$, the maximum relative age of m is greater than cache associativity A, so it may be evicted from the cache in some executions.

Persistence analysis can be performed on the control flow graph (CFG). A CFG consists of a *set of node* $V = \{n_1, ..., n_k\}$ connected by directed edges. Each *control flow node* n_k is a basic block where the program execution is strictly sequential without any jump or jump target. At basic block n_k with incoming ACS $\hat{c}^{in}$, if the program accesses memory block m, the *cache update function* $\hat{\mathcal{U}}_{\hat{\mathcal{C}}}$ computes the output ACS $\hat{c}^{out}$ after accessing m. If a basic block n_k has two or more incoming ACSs, the *cache join function* $\hat{\mathcal{J}}_{\hat{\mathcal{C}}}$ combines upper bound of all incoming ACSs into the representative input ACS $\hat{c}^{in}$ of node n. The persistence analysis repeatedly traverses through the CFG and performs these computations until the input ACSs of all nodes reach fixed-point.

Given an accessed to memory block m and a concrete cache state c, the updating of A-way set associative cache is modeled using the concrete cache update function $\mathcal{U}_{\mathcal{C}}$ [9] as follows:

$$\mathcal{U}_{\mathcal{C}}(c, m) = c[set(m) \mapsto \mathcal{U}_{\mathcal{S}}(c[set(m)], m)]$$

The concrete cache update function $\mathcal{U}_{\mathcal{C}}$ models the change in cache set $s = set(m)$ where referenced memory block m is mapped to using concrete set update function $\mathcal{U}_{\mathcal{S}}$

$$\mathcal{U}_{\mathcal{S}}(s, m) = \begin{cases} l_1 \mapsto \{m\}, \\ l_i \mapsto s(l_{i-1}) | i = 2...h \\ l_i \mapsto s(l_i) | i = h+1...A \\ \qquad\qquad if \exists h \in \{1..A\}, m \in s(l_h) \\ l_1 \mapsto \{m\}, \\ l_i \mapsto s(l_{i-1}) | i = 2...A \\ \qquad\qquad otherwise \end{cases}$$

From the concrete update function, Ferdinand and Wilhelm [10] proposes an abstract cache update function $\hat{\mathcal{U}}_{\hat{\mathcal{C}}}$ to compute the ACS after an access to memory block m as follows:

$$\hat{\mathcal{U}}_{\hat{\mathcal{C}}}(\hat{c}, m) = \hat{c}[set(m) \mapsto \hat{\mathcal{U}}_{\hat{\mathcal{S}}}(\hat{c}[set(m)], m)]$$

$$\hat{\mathcal{U}}_{\hat{\mathcal{S}}}(\hat{s}, m) = \begin{cases} l_1 \mapsto \{m\}, \\ l_i \mapsto \hat{s}(l_{i-1}) | i = 2...h-1 \\ l_h \mapsto \hat{s}(l_h) \cup \hat{s}(l_{h-1}) \setminus \{m\} \\ l_i \mapsto \hat{s}(l_i) | i = h+1...A, \top \\ \qquad\qquad if \exists h \in \{1..A\}, m \in \hat{s}(l_h) \\ l_1 \mapsto \{m\}, \\ l_i \mapsto \hat{s}(l_{i-1}) | i = 2...A \\ l_\top \mapsto \hat{s}(l_\top) \cup \hat{s}(l_A) \setminus \{m\} \\ \qquad\qquad otherwise \end{cases}$$

The abstract set update function $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$ computes the change in abstract state set state $\hat{s} = \hat{c}[set(m)]$ after accessing m. It brings (or renews) the newly accessed memory block m to youngest cache line l_1. If $m \notin \hat{s}$, $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$ ages all memory blocks m' currently in $\hat{s}$. If $m \in \hat{s}(l_h)$, for each $m' \in \hat{s}(l_k)$,

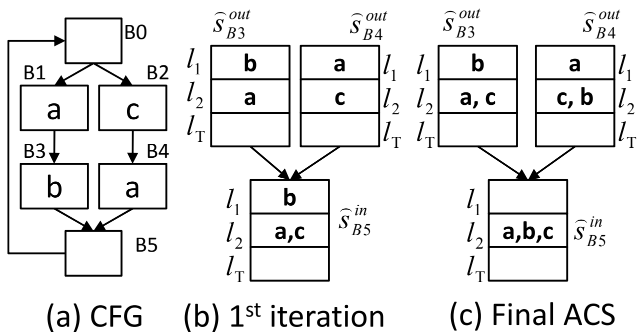


Figure 1. Running example and analysis result of original persistence analysis

if m' is younger than m in the ACS ($k < h$), m will age m' to $\hat{s}(l_{k+1})$. Otherwise ($k \geq h$), m' remains in $\hat{s}(l_k)$.

If a CFG node n has two immediate predecessors n_1 and n_2, a join function $\mathcal{J}_{\hat{\mathcal{C}}}$ combines the output ACSs of n_1 and n_2 to form the input ACS of n. The new relative age of a memory block m is equal to the maximum age of its existences in all output ACSs of the predecessor nodes of n. Let $\hat{c}_1, \hat{c}_2$ be the output ACS of predecessors n_1, n_2, join function $\mathcal{J}_{\hat{\mathcal{C}}}$ computes the input ACS $\hat{c}$ of node n as follows:

$$\begin{aligned}
&\mathcal{J}_{\hat{\mathcal{C}}}(\hat{c}_1, \hat{c}_2) = \hat{c}[s_i \mapsto \mathcal{J}_{\hat{\mathcal{S}}}(\hat{c}_1[s_i], \hat{c}_2[s_i])] \\
&\mathcal{J}_{\hat{\mathcal{S}}}(\hat{s}_1, \hat{s}_2) = \hat{s} \text{ where:} \\
&\hat{s}(l_x) = \{m | m \in \hat{s}_1(l_a) \wedge m \in \hat{s}_2(l_b), x = max(a, b)\} \\
&\qquad \cup \{m | m \in \hat{s}_1(l_x) \wedge m \notin \hat{s}_2\} \\
&\qquad \cup \{m | m \notin \hat{s}_1 \wedge m \in \hat{s}_2(l_x)\}
\end{aligned}$$

B. Safety issue

It has been pointed out that the original persistence analysis proposed in Ferdinand [9], [10] is unsafe. Figure 1 illustrates an unsafe scenario of the original persistence analysis. The CFG in Figure 1(a) with six nodes $B0$,...,$B5$ in a loop. The program accesses memory block a in $B1$ and $B4$, b in $B3$, and c in $B2$. Assume a, b, c are all mapped to cache set s with associativity $A = 2$. Figure 1(b) shows the abstract set state $\hat{s}^{out}_{B3}$ of $B3$, $\hat{s}^{out}_{B4}$ of $B4$, and $\hat{s}^{in}_{B5}$ of $B5$ after the first iteration through the loop. According to the update function $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$ described above, since memory block a is in the ACS and c is not younger than a, an access to a will not increase the maximal relative age of c, and similarly for c. Since all memory blocks a, b, and c are the in ACS, all accesses to a, b, c will not increase their maximal relative age from l_2 to $l_\top$.

Figure 1(c) gives the ACS at fixed-point. The input ACS of $B5$ at fixed point ($\hat{s}^{in}_{B5}$ in Figure 1(c)) shows that memory block c is persistent in the loop. However, in the path $B0 \rightarrow B2 \rightarrow B4 \rightarrow B5$, then $B0 \rightarrow B1 \rightarrow B3$, we see that c is evicted by accesses to a and b. Therefore, c is not persistent at $B5$, and the persistence analysis in [10] is unsafe.

The incorrectness is due to an error of the update function $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$. It wrongly assumes that if memory block $b \in \hat{s}^{in}_{B5}$ (Figure 1(c)), b is in concrete set s^{in}_{B5} in all possible execution paths. Consequently, the update function does not age memory blocks with relative age equal or older than b in $\hat{s}^{in}_{B5}$ such as a or c. However, when $b \in \hat{s}^{in}_{B5}$, b just *may* be in concrete set state s^{in}_{B5}. As a result, there exists concrete set states s^{in}_{B5} that do not contain b (e.g. only a and c are in s^{in}_{B5} of path $B0 \rightarrow B2 \rightarrow B4 \rightarrow B5$). In that case, b will age both a and c in s^{in}_{B5}, and the original persistence analysis [9] will underestimate the relative age of a and c.

Let $conc_{\hat{\mathcal{C}}}(\hat{c}^{in})$ be the set of all possible concrete cache states represented by ACS $\hat{c}^{in}$ at program point p, the unsafe scenario when accessing a memory block $m_a \in \hat{c}$ can be formulated mathematically as follows:

$$\begin{aligned}
&\hat{s}^{in} = \hat{c}^{in}[set(m_a)] \wedge m_a \in \hat{s}^{in}(l_h) \\
&\rightarrow \exists c^{in} \in conc_{\hat{\mathcal{C}}}(\hat{c}^{in}), s^{in} = c^{in}[set(m_a)] \wedge m_a \notin s^{in} \\
&\quad \wedge \exists m, m \in \hat{s}^{in}(l_h) \wedge m \in s^{in}(l_h) \\
&\quad \wedge h > 1 \wedge h \leq A
\end{aligned}$$

Let $s^{out} = \mathcal{U}_{\mathcal{S}}(s^{in}, m_a)$ and $\hat{s}^{out} = \hat{\mathcal{U}}_{\hat{\mathcal{S}}}(\hat{s}^{in}, m_a)$ be the output concrete set state s^{out} and abstract set state $\hat{s}^{out}$ after the cache update. The relative age of memory block m in the output concrete set s^{out} and abstract set $\hat{s}^{out}$ are as follows

$$\begin{aligned}
&m \in s^{in}(l_h) \wedge m_a \notin s^{in}, \\
&\qquad s^{out} = \mathcal{U}_{\mathcal{S}}(s^{in}, m_a) \rightarrow m \in s^{out}(l_{h+1})
\end{aligned}$$

$$\begin{aligned}
&m \in \hat{s}^{in}(l_h) \wedge m_a \in \hat{s}^{in}(l_h) \\
&\qquad \hat{s}^{out} = \hat{\mathcal{U}}_{\hat{\mathcal{S}}}(\hat{s}^{in}, m_a) \rightarrow m \in \hat{s}^{out}(l_h)
\end{aligned}$$

Because m_a is not in s^{in}, m_a ages m in line l_h to l_{h+1}. On the other hand, m_a is in $\hat{s}^{in}(l_h)$, so update function $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$ does not age m from l_h to l_{h+1}. Therefore, $m \in \hat{s}^{out}(l_h)$ but $m \in s^{out}(l_{h+1})$, the abstract set state $\hat{s}^{out}$ underestimate the maximum relative age of m in concrete set state s^{out}.

C. Correcting the persistence analysis

As demonstrated above, we cannot use the maximum relative age of memory block m_a in ACS $\hat{c}$ to determine if an access to m_a would further age other memory blocks in $\hat{c}$. Given abstract set state $\hat{s}$ with $m_a \in \hat{s}(l_h)$ and $m \in \hat{s}(l_k)$, an access to m_a could still increase maximum relative age k of memory block m even when m has older maximum relative age ($k \geq h$). As a result, we propose to track the set of memory blocks that may be more recently used (younger) than memory block m in the ACS. An access to memory block m_a will increase the maximum relative age of m only if m_a is not in the current younger set of m. Otherwise, m_a is already counted as a possible younger memory block than m. Therefore according to LRU policy, it will not further increase the maximum relative age of memory block m. We define the Younger Set ($\mathcal{YS}$) as follows.

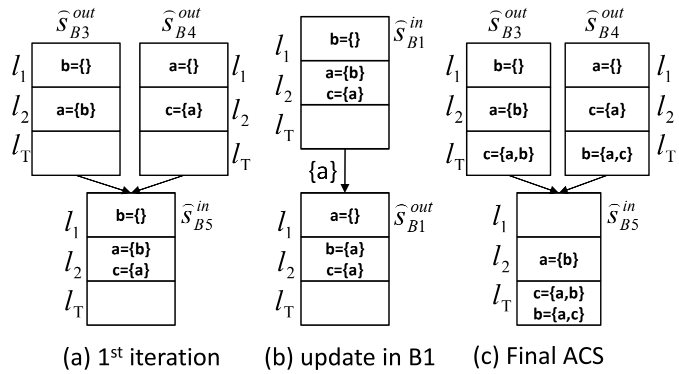


Figure 2. Analysis result of with proposed update and join function

Definition 1: **(Younger Set)**: For an abstract set state $\hat{s}$ at program point p, the younger set $\mathcal{YS}(\hat{s}, m)$ of m captures a *superset* of all memory blocks that may have smaller relative ages (younger) than m at p in some possible program execution that reaches p. □

In LRU replacement policy, the relative age of memory block m is determined by the number of memory blocks more recently used (younger) than m in the same cache set. Consequently, the maximum relative age x of m in $\hat{s}$ should be larger than the number of memory blocks possibly younger than m, i.e. the size of younger set $\mathcal{YS}(\hat{s}, m)$ ($x = |\mathcal{YS}(\hat{s}, m)| + 1$). If maximum relative age x is not greater than cache associativity A, memory block m is guaranteed to remain in the cache once it has been accessed.

To optimize analysis performance, we stop tracking younger set $\mathcal{YS}(\hat{s}, m)$ of m once it has more memory blocks than cache associativity A (hence m is not persistent). For cache using LRU replacement, A is usually small (e.g. $A \leq 4$). Therefore, the younger set $\mathcal{YS}(\hat{s}, m)$ is generally small and easy to track.

Figure 2(a) illustrates the younger set of each memory blocks a, b, c in ACS of $B3$, $B4$, $B5$ in the first loop iteration. In $B3$, b is just accessed so b is brought to the youngest line $\hat{s}^{out}_{B3}(l_1)$ with no younger memory block. a is older than b, so a is in $\hat{s}^{out}_{B3}(l_2)$ with younger set $\mathcal{YS}(\hat{s}^{out}_{B3}, a) = \{b\}$. Similarly in $B4$, a is just accessed so a is in the newest cache line $\hat{s}^{out}_{B4}$, and the younger set $\mathcal{YS}(\hat{s}^{out}_{B4}, a)$ is empty. c is older than a, so $\mathcal{YS}(\hat{s}^{out}_{B4}, c) = \{a\}$. In $B5$, b has no younger memory block in both incoming block $B3$ and $B4$, so it has no younger memory block in $B5$. a has younger memory block b in incoming block $B3$ and none in $B4$, so the younger set $\mathcal{YS}(\hat{s}^{in}_{B5}, a) = \{b\}$. Similarly, c has only one younger memory block a in $B4$, so the younger set $\mathcal{YS}(\hat{s}^{in}_{B5}, c) = \{a\}$.

Notice that from the younger set, we know that in first iteration, memory block b is not a possible younger memory block of c in any concrete cache state at $B5$ even though the maximum relative age of b is smaller than the maximum relative age of c in $\hat{s}^{in}_{B5}$. Therefore, we know that a subsequent access to b will increase the maximum relative age of c. Consequently, our proposed younger set notion helps avoid the incorrectness of original persistence analysis in [10] (Figure 2(c)).

We propose a new update and join function to track and use younger set notion in ACS computation as follows.

New update function: Given a program point p with ACS $\hat{c}^{in}$, if the program accesses memory block m_a at p, our cache update function $\hat{\mathcal{U}}_{\hat{\mathcal{C}}}$ updates the state of cache set $set(m_a)$ using the set update function $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$

$$\hat{\mathcal{U}}_{\hat{\mathcal{C}}}(\hat{c}^{in}, m_a) = \hat{c}^{out}[set(m_a) \mapsto \hat{\mathcal{U}}_{\hat{\mathcal{S}}}(\hat{c}^{in}[set(m_a)], m_a)]$$

Given the accessed memory block m_a and the input abstract set state $\hat{s}^{in}$ where m_a is mapped to, the update function $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$ computes the output abstract set state $\hat{s}^{out}$ and calculate the younger set $\mathcal{YS}(\hat{s}^{out}, m)$ for each memory block m in $\hat{s}^{out}$ as follows:

$$\hat{\mathcal{U}}_{\hat{\mathcal{S}}}(\hat{s}^{in}, m_a) = \hat{s}^{out} \text{ with}$$
$$\hat{s}^{out}(l_x) = \{m | m \in \hat{s}^{in} \cup \{m_a\}, \quad x = \min(|\mathcal{YS}(\hat{s}^{out}, m)| + 1, \top)\}$$

Where $\forall m \in \hat{s}^{in} \cup \{m_a\}$,

$$\mathcal{YS}(\hat{s}^{out}, m) = \begin{cases} \mathcal{YS}(\hat{s}^{in}, m) \cup \{m_a\} & \text{if } m \neq m_a \\ \emptyset & \text{if } m = m_a \end{cases}$$

When m_a is accessed, for each memory block m in $\hat{s}^{in}$, if $m \neq m_a$, m_a becomes a more recently used memory block than m. Therefore, update function $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$ adds m_a to the younger set $\mathcal{YS}(\hat{s}^{out}, m)$ and changes maximum relative age of m accordingly. If $m = m_a$, m is accessed and becomes the youngest memory block in set $\hat{s}^{out}$. As a result, update function $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$ brings m to $\hat{s}^{out}(l_1)$ and set its younger set $\mathcal{YS}(\hat{s}^{out}, m)$ to empty.

Figure 2(b) shows our update function at $B1$ after the first iteration described in Figure 2(a). $\hat{s}^{in}_{B1}$ contains memory block b in cache line l_1, a and c in cache line l_2. As seen in Figure 2(a), after the first iteration, b is the youngest memory block. Therefore, $\mathcal{YS}(\hat{s}^{in}_{B1}, b)$ is empty. a is aged by b in $B3$ so $\mathcal{YS}(\hat{s}^{in}_{B1}, a) = \{b\}$. And similarly, c is aged by a in $B4$ so $\mathcal{YS}(\hat{s}^{in}_{B1}, c) = \{a\}$. At $B1$, the program accesses memory block a. Consequently, a is renewed to youngest line $\hat{s}^{in}_{B1}(l_1)$ and younger set $\mathcal{YS}(\hat{s}^{out}_{B1}, a)$ is set to empty. a becomes a new younger block of b so $\mathcal{YS}(\hat{s}^{out}_{B1}, b) = \{a\}$. With one possible younger memory block, b has maximal relative age $x = 2$. Because c already has a in its younger set $\mathcal{YS}(\hat{s}^{in}_{B1}, c)$, it keeps the same maximal relative age and younger set.

New join function: Given a program point p with two incoming edges from p_1 and p_2 having ACS $\hat{c}_1$ and $\hat{c}_2$, the join function $\mathcal{J}_{\hat{\mathcal{C}}}$ computes the joined ACS $\hat{c}$ as combined upper bound of incoming ACSs

$$\mathcal{J}_{\hat{\mathcal{C}}}(\hat{c}_1, \hat{c}_2) = \hat{c}[s_i \mapsto \mathcal{J}_{\hat{\mathcal{S}}}(\hat{c}_1[s_i], \hat{c}_2[s_i])]$$

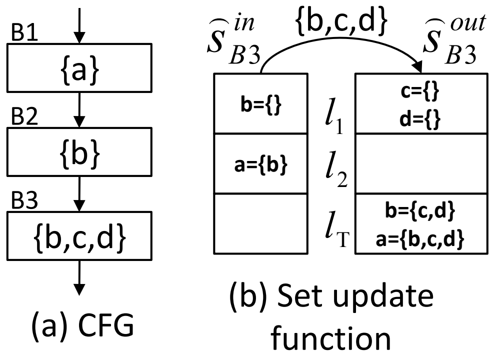


Figure 3. Cache update for set of possible access addresses

Given two incoming abstract set state $\hat{s}_1$ and $\hat{s}_2$, we propose a new join function to compute combined abstract set state $\hat{s}$ and track the younger set for each memory block $m \in \hat{s}$ as follows:

$$\mathcal{J}_{\hat{\mathcal{S}}}(\hat{s}_1, \hat{s}_2) = \hat{s} \text{ with:}$$
$$\hat{s}(l_x) = \{m | m \in \hat{s}_1 \cup \hat{s}_2, x = \min(|\mathcal{YS}(\hat{s}, m)| + 1, \top)\}$$
$$\text{where } \forall m \in \hat{s}_1 \cup \hat{s}_2$$
$$\mathcal{YS}(\hat{s}, m) = \begin{cases} \mathcal{YS}(\hat{s}_1, m) \cup \mathcal{YS}(\hat{s}_2, m) \\ \qquad \text{if } m \in \hat{s}_1 \wedge m \in \hat{s}_2 \\ \mathcal{YS}(\hat{s}_1, m) \\ \qquad \text{if } m \in \hat{s}_1 \wedge m \notin \hat{s}_2 \\ \mathcal{YS}(\hat{s}_2, m) \\ \qquad \text{if } m \notin \hat{s}_1 \wedge m \in \hat{s}_2 \end{cases}$$

The joined abstract set state $\hat{s}$ is a set union of $\hat{s}_1$ and $\hat{s}_2$. Moreover, the younger set $\mathcal{YS}(\hat{s}, m)$ of each memory block m in $\hat{s}$ is also the set union of younger set of m in $\hat{s}_1$ and $\hat{s}_2$ if there is. The relative age of m in $\hat{s}$ is then set according the size of its younger set. Because the younger set $\mathcal{YS}(\hat{s}, m)$ always contain all younger memory blocks of m in $\hat{s}_1$ and $\hat{s}_2$, it safely estimates the possible memory blocks younger than m in $\hat{s}$ in all possible executions.

Figure 2(c) illustrates our join function. In $B3$, memory block b has no younger memory block but in $B4$, b has two younger memory blocks a and c, so $\mathcal{YS}(\hat{s}^{in}_{B5}, b) = \{a, c\}$ in combined abstract set state $\hat{s}^{in}_{B5}$ of $B5$. Similarly, $\mathcal{YS}(\hat{s}^{in}_{B5}, c) = \{a, b\}$ and $\mathcal{YS}(\hat{s}^{in}_{B5}, a) = \{b\}$. Our proposed persistence analysis accurately points out that a is persistent at $B5$. However, b and c have up to two possible younger memory blocks so they may be evicted.

New update function for set: Unlike instruction references, a data reference D can access a set of possible different data addresses $Addr(D)$. Therefore, cache update function $\hat{\mathcal{U}}_{\hat{\mathcal{C}}}$ need to handle sets of possibly referenced memory blocks, as in [10]. We propose a new update function for set to update the change in ACS $\hat{c}$ and track the younger set after an access of data reference D as follows:

$$\hat{\mathcal{U}}_{\hat{\mathcal{C}}}(\hat{c}, Addr(D)) = \hat{c}[f_i \mapsto \hat{\mathcal{U}}_{\hat{\mathcal{S}}}(\hat{c}[f_i], X_{f_i})]$$
$$\text{for all} f_i \in \{f = set(m) | m \in Addr(D)\}$$
$$\text{where } X_{f_i} = \{m_y | m_y \in Addr(D), set(m_y) = f_i\},$$

Given a set of possible access addresses $Addr(D)$ of data reference D, the abstract cache update function $\hat{\mathcal{U}}_{\hat{\mathcal{C}}}$ divides it into X_{f_i}, the set of possible access addresses in $Addr(D)$ corresponds to cache set f_i. Our new abstract set update function $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$ compute the output abstract set state $\hat{s}^{out}$ from the input abstract set state $\hat{s}^{in}$ and the set X_{f_i} of $Addr(D)$ mapped to this cache set as follows

$$\hat{\mathcal{U}}_{\hat{\mathcal{S}}}(\hat{s}^{in}, X_{f_i}) = \hat{s}^{out}$$
$$\text{with } \hat{s}^{out}(l_x) = \{m | m \in \hat{s}^{in} \cup X_{f_i},$$
$$x = \min(|\mathcal{YS}(\hat{s}^{out}, m)| + 1, \top)\}$$

Where $\forall m \in \hat{s}^{in} \cup X_{f_i}$

$$\mathcal{YS}(\hat{s}^{out}, m) = \begin{cases} \mathcal{YS}(\hat{s}^{in}, m) \cup X_{f_i} \setminus \{m\} & \text{if } m \in \hat{s}^{in} \\ \emptyset & \text{otherwise} \end{cases}$$

Because no memory block $m_a \in Addr(D)$ is guaranteed to be accessed, we cannot renew $m_a \in \hat{s}^{in}$ even though $m_a \in Addr(D)$. However, any $m_a \in X_{f_i}$ could possibly become a new younger memory block of all memory block m currently in $\hat{s}^{in}$. Therefore, the update function $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$ adds X_{f_i} to the younger set $\mathcal{YS}(\hat{s}, m)$ of m. If a memory block $m_a \in X_{f_i}$ and $m_a \notin \hat{s}$, m_a may be a newly accessed memory block in $\hat{s}^{out}$. Therefore, update function $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$ adds m_a to the abstract set state $\hat{s}^{out}$ as a youngest memory block with empty younger set.

Figure 3(a) illustrates such scenario. A data reference D in $B3$ may access a set of possible memory block $\{b, c, d\}$ mapped to $\hat{s}^{in}_{B3}$. Figure 3(b) shows the input abstract set state $\hat{s}^{in}_{B3}$ and the resulting abstract set state $\hat{s}^{out}_{B3}$ after the memory access. As all of $\{b, c, d\}$ could be accessed, the set update function adds all of them to the younger set of memory block a and b in $\hat{s}^{in}_{B2}$. Therefore, a is aged to evicted line $l_\top$ because it has $\{b, c, d\}$ as possible younger blocks. b is also evicted to $l_\top$ because it has two possible younger blocks c, d. c and d are added to $\hat{s}^{out}_{B2}(l_1)$ as most recently used memory blocks with no younger memory block.

IV. Safety Proofs of Corrected Persistence Analysis

In this section, we will prove the safety and termination of our proposed persistence analysis.

In our persistence analysis and the proofs, we consider a program point before and after each program instruction. Note that for data cache analysis, it is possible that there is no data memory references between two program points if the instruction does not access data memory.

For each memory block m, the relative age of m in the cache is determined by the number of more recently used (younger) memory blocks in the same cache set. At program point p, given a execution path pa that reaches p with concrete cache state c. Memory block m in cache set $s = c[set(m)]$ will have relative age y ($m \in s(l_y)$) if there are $y - 1$ younger memory blocks in s (from $s(l_1)$ to $s(l_{y-1})$). We define the concrete younger set of memory block m as follows:

Definition 2: **(Concrete younger set)** Concrete younger set $ys(s, m)$ of memory block m is the set of memory blocks more recently used (younger) than m in concrete set state s of cache set where m is mapped to. □

$$m \in s(l_y) \to ys(s, m) = s(l_1) \cup ... \cup s(l_{y-1}) \\ \land y = |ys(s, m)| + 1$$

In our proposed persistence analysis, at program point p with ACS $\hat{c}$ at fixed point, we determine the maximum relative age x of memory block m by the younger set $\mathcal{YS}(\hat{s}, m)$, the set of all memory blocks possibly younger (more recently used) than m in the abstract set state $\hat{s} = \hat{c}[set(m)]$, i.e. $x = |\mathcal{YS}(\hat{s}, m)| + 1$. To prove the safety of our persistence analysis, we prove that from our proposed update and join function, the younger set $\mathcal{YS}(\hat{s}, m)$ is the superset of concrete younger set $ys(s, m)$ in concrete set state $s = c[set(m)]$ at p in any execution path that reaches p, captured by the younger set property.

Definition 3: **(YS property)**: Given an arbitrary path pa from start of execution to program point p which results in concrete cache state c. Let $\hat{c}$ be the computed fixed point ACS at p. For each memory block $m \in c$, let $\hat{s} = \hat{c}[set(m)]$ and $s = c[set(m)]$ be the abstract and concrete state of cache set where m is mapped to, the younger set $\mathcal{YS}(\hat{s}, m)$ is the superset of the concrete younger set $ys(s, m)$. □

$$\forall m \in c, s = c[set(m)], \hat{s} = \hat{c}[set(m)], \\ ys(s, m) \subseteq \mathcal{YS}(\hat{s}, m)$$

If the younger set $\mathcal{YS}(\hat{s}, m)$ is the superset of concrete younger set $ys(s, m)$, the maximum relative age x of m in $\hat{s}$ computed by our analysis ($x = |\mathcal{YS}(\hat{s}, m)| + 1$) is always greater or equal than the concrete relative age y of m in s ($y = |ys(s, m)| + 1$). Hence if maximum relative age x is less than or equal cache associativity A, m is not evicted out of the cache for any concrete cache set s at p. Therefore, our persistence analysis is safe.

A. Structure of the proof

We prove by induction that the YS property holds in all possible execution paths in the program.

- Because the concrete cache state c is empty at the start of the execution, YS property is trivially true initially.
- Assume YS property holds at p^{in}, before program point p. If at p, the program accesses memory block m_a (or a set of possible memory blocks $Addr(D) = \{m_1...m_k\}$ of data reference D), we prove that YS property holds at p^{out}, after program point p by proving the correctness of our update function $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$ (Section IV-B and Section IV-D).
- Assume YS property holds at p^{out}, after program point p, we prove that YS property holds at p_n^{in}, before the next program point p_n by proving the correctness of our join function $\hat{\mathcal{J}}_{\hat{\mathcal{S}}}$ (Section IV-C)

As YS property is true at the start of the execution, before and after each program point, and from one program point to another, YS property holds for all possible executions of the program. Therefore, given fixed-point ACS $\hat{c}$ at program point p, in any execution path that reaches p with concrete cache state c, let $\hat{s} = \hat{c}[set(m)]$ and $s = c[set(m)]$, the younger set $\mathcal{YS}(\hat{s}, m)$ is the superset of the concrete younger set $ys(s, m)$ of m in s. Consequently, the maximal relative age x of m in $\hat{s}$ ($x = |\mathcal{YS}(\hat{s}, m)| + 1$) is always greater or equal than the relative age y of m in s ($y = |ys(s, m)| + 1$). As a result, if the maximal relative age x is less than or equal to cache associativity A, m is persistent when the program control reaches p in all executions.

B. Safety of update function

We prove our update function preserves the YS property. If the program accesses m_a at program point p, assume YS property holds at p^{in}, we prove YS property holds at p^{out}.

Given a path pa having concrete cache state c^{in} at p^{in}, before program point p. Let $\hat{c}^{in}$ be the fixed-point ACS at p^{in}. Assume YS property holds at p^{in}, we have

$$\forall m \in c^{in}, s^{in} = c^{in}[set(m)], \hat{s}^{in} = \hat{c}^{in}[set(m)], \\ ys(s^{in}, m) \subseteq \mathcal{YS}(\hat{s}^{in}, m) \quad \text{[B.1]}$$

If the program accesses memory block m_a at program point p, let c^{out} be the concrete cache state of path pa at p^{out}, after program point p. Let $\hat{c}^{out}$ be the fixed-point ACS at p^{out}. We prove YS property holds at p^{out}

$$\forall m \in c^{out}, s^{out} = c^{out}[set(m)], \hat{s}^{out} = \hat{c}^{out}[set(m)], \\ ys(s^{out}, m) \subseteq \mathcal{YS}(\hat{s}^{out}, m) \quad \text{[B.2]}$$

Case 1: $set(m) \neq set(m_a)$

Because $set(m) \neq set(m_a)$, the cache state of m is unaffected by the access to memory block m_a. As a result, there is no change in the concrete set state, $s^{out} = s^{in}$, so $ys(s^{out}, m) = ys(s^{in}, m)$. Similarly, there is no change in the abstract set state, $\hat{s}^{out} = \hat{s}^{in}$, so $\mathcal{YS}(\hat{s}^{out}, m) = \mathcal{YS}(\hat{s}^{in}, m)$. Therefore, YS property continues to hold from p^{in} to p^{out}.

Case 2: $set(m) = set(m_a)$

As m and m_a are mapped to the same cache set, if $m \neq m_a$, m_a becomes a new younger memory block of

m. Otherwise ($m_a = m$), m is accessed so it is brought (or renewed) to youngest line l_1.

$$ys(s^{out}, m) = \begin{cases} ys(s^{in}, m) \cup \{m_a\} & \text{if } m \neq m_a \\ \emptyset & \text{if } m = m_a \end{cases} \quad \text{[B.3]}$$

From our proposed update function $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$, the new younger set of each memory block in $\hat{s}^{in}$ is computed as follows.

$$\mathcal{YS}(\hat{s}^{out}, m) = \begin{cases} \mathcal{YS}(\hat{s}^{in}, m) \cup \{m_a\} & \text{if } m \neq m_a \\ \emptyset & \text{if } m = m_a \end{cases} \quad [\hat{\mathcal{U}}_{\hat{\mathcal{S}}}]$$

As a result, we have

$$\text{[B.1]} \rightarrow ys(s^{in}, m) \subseteq \mathcal{YS}(\hat{s}^{in}, m)$$

$$\text{[B.3]} \rightarrow ys(s^{out}, m) = \begin{cases} ys(s^{in}, m) \cup \{m_a\} & \text{if } m \neq m_a \\ \emptyset & \text{if } m = m_a \end{cases}$$

$$[\hat{\mathcal{U}}_{\hat{\mathcal{S}}}]\ \mathcal{YS}(\hat{s}^{out}, m) = \begin{cases} \mathcal{YS}(\hat{s}^{in}, m) \cup \{m_a\} & \text{if } m \neq m_a \\ \emptyset & \text{if } m = m_a \end{cases}$$

$$\text{[B.1],[B.3]}, [\hat{\mathcal{U}}_{\hat{\mathcal{S}}}] \rightarrow \begin{cases} \text{if } m = m_a \\ \quad ys(s^{out}, m) = \emptyset \subseteq \mathcal{YS}(\hat{s}^{out}, m) \\ \text{if } m \neq m_a \\ \quad ys(s^{out}, m) = ys(s^{in}, m) \cup \{m_a\} \\ \quad \mathcal{YS}(\hat{s}^{out}, m) = \mathcal{YS}(\hat{s}^{in}, m) \cup \{m_a\} \\ \quad ys(s^{in}, m) \subseteq \mathcal{YS}(\hat{s}^{in}, m) \\ \quad \rightarrow ys(s^{out}, m) \subseteq \mathcal{YS}(\hat{s}^{out}, m) \end{cases}$$

Therefore, YS property holds at p^{out}, after the execution of step p.

C. Safety of join function

Assume YS property holds at p^{out}, after program point p, we prove that YS property holds at p_n^{in}, before the immediate program point p_n by proving the correctness of our join function $\hat{\mathcal{J}}_{\hat{\mathcal{S}}}$.

Given a path pa having concrete cache state c^{out} at p^{out}. Let $\hat{c}^{out}$ be the fixed-point ACS at p^{out}. Assume YS property holds at p^{out}, we have

$$\forall m \in c^{out}, s^{out} = c^{out}[set(m)], \hat{s}^{out} = \hat{c}^{out}[set(m)], \\ ys(s^{out}, m) \subseteq \mathcal{YS}(\hat{s}^{out}, m) \quad \text{[C.1]}$$

Let c_n^{in} be the concrete cache state of path pa at p_n^{in}, before the next program point p_n. Let $\hat{c}_n^{in}$ be the fixed-point ACS at p_n^{in}. We prove YS property holds at $\hat{c}_n^{in}$

$$\forall m \in c_n^{in}, s_n^{in} = c_n^{in}[set(m)], \hat{s}_n^{in} = \hat{c}_n^{in}[set(m)], \\ ys(s_n^{in}, m) \subseteq \mathcal{YS}(\hat{s}_n^{in}, m) \quad \text{[C.2]}$$

From our proposed join function $\hat{s} = \hat{\mathcal{J}}_{\hat{\mathcal{S}}}(\hat{s}_1, \hat{s}_2)$, younger set $\mathcal{YS}(\hat{s}, m)$ of m at p_n^{in} is the union of all younger sets of incoming edges of p_n^{in}. As p^{out} is one of the incoming edge, we have

$$\mathcal{YS}(\hat{s}^{out}, m) \subseteq \mathcal{YS}(\hat{s}_n^{in}, m) \quad [\hat{\mathcal{J}}_{\hat{\mathcal{S}}}]$$

Because program point p_n^{in} is immediately after p^{out}, no new memory block is accessed, so the concrete set state remains the same, $s_n^{in} = s^{out}$. As a result, the concrete younger set for each memory block m also remains the same

$$ys(s_n^{in}, m) = ys(s^{out}, m) \quad \text{[C.3]}$$

In summary

$$\begin{aligned} &\text{[C.1]} \rightarrow ys(s^{out}, m) \subseteq \mathcal{YS}(\hat{s}^{out}, m) \\ &[\hat{\mathcal{J}}_{\hat{\mathcal{S}}}] \rightarrow \mathcal{YS}(\hat{s}^{out}, m) \subseteq \mathcal{YS}(\hat{s}_n^{in}, m) \\ &\text{[C.3]} \rightarrow ys(s_n^{in}, m) = ys(s^{out}, m) \\ &\rightarrow ys(s_n^{in}, m) \subseteq \mathcal{YS}(\hat{s}_n^{in}, m) \end{aligned}$$

So the younger set $\mathcal{YS}(\hat{s}_n^{in}, m)$ always contains all possible memory blocks younger than m in $set(m)$ of c^{in} at p_n^{in}. Therefore the YS property holds at next program point p_n^{in}.

D. Safety of set update function

A data reference D can access a set of possible different data addresses $Addr(D) = \{m_1 ... m_k\}$. Therefore, cache update function $\hat{\mathcal{U}}_{\hat{\mathcal{C}}}$ need to handle sets of possibly referenced memory blocks, as in [10]. We prove our set update function preserves the YS property. If the program may access any $m_a \in Addr(D) = \{m_1 ... m_k\}$ at p, assume YS property holds at p^{in}, before program point p, we prove YS property holds at p^{out}, after the data memory access at program point p.

Given a path pa having concrete cache state c^{in} at p^{in}. Let $\hat{c}^{in}$ be the fixed-point ACS at p^{in}. Assume YS property holds at p^{in}, we have

$$\forall m \in c^{in}, s^{in} = c^{in}[set(m)], \hat{s}^{in} = \hat{c}^{in}[set(m)], \\ ys(s^{in}, m) \subseteq \mathcal{YS}(\hat{s}^{in}, m) \quad \text{[D.1]}$$

Let c^{out} be the concrete cache state of path pa at p^{out}, after the memory access at p. Let $\hat{c}^{out}$ be the fixed-point ACS at p^{out}. We prove YS property holds at p^{out}

$$\forall m \in c^{out}, s^{out} = c^{out}[set(m)], \hat{s}^{out} = \hat{c}^{out}[set(m)], \\ ys(s^{out}, m) \subseteq \mathcal{YS}(\hat{s}^{out}, m) \quad \text{[D.2]}$$

For each memory block m in the cache set s^{in}, let X_{f_i} be the set of memory blocks in $Addr(D)$ mapped to s^{in}. The data reference D can access any memory block $m_a \in X_{f_i}$. If $m \neq m_a$, m_a becomes a new younger memory block of memory block m. Otherwise ($m = m_a$), m is renewed to the youngest cache line and has no younger memory block.

$$ys(s^{out}, m) = \begin{cases} ys(s^{in}, m) \cup \{m_a\}, \forall m_a \in X_{f_i} \\ \qquad \text{if } m \in s^{in} \wedge m \neq m_a \\ \emptyset \\ \qquad \text{Otherwise} \end{cases} \quad \text{[D.3]}$$

Our proposed set update function calculates new possible younger set of m in $\hat{s}^{in}$ when accessed by set X_{f_i} as follow

$$\mathcal{YS}(\hat{s}_o, m) = \begin{cases} \mathcal{YS}(\hat{s}_i, m) \cup X_{f_i} \setminus \{m\} & \\ \quad \text{if } m \in \hat{s}_i & [\hat{\mathcal{U}}_{\hat{\mathcal{S}}}] \\ \emptyset \quad \text{otherwise} & \end{cases}$$

In summary

[D.1], [D.3], $[\hat{\mathcal{U}}_{\hat{\mathcal{S}}}] \rightarrow$

if $m \neq m_a$

$$ys(s^{out}, m) = ys(s^{in}, m) \cup \{m_a\}, \forall m_a \in X_{f_i}$$
$$\mathcal{YS}(\hat{s}^{out}, m) = \mathcal{YS}(\hat{s}^{in}, m) \cup X_{f_i} \setminus \{m\}$$
$$ys(s^{in}, m) \subseteq \mathcal{YS}(\hat{s}^{in}, m)$$
$$\rightarrow ys(s^{out}, m) \subseteq \mathcal{YS}(\hat{s}^{out}, m)$$

if $m = m_a$

$$ys(s^{out}, m) = \emptyset \rightarrow ys(s^{out}, m) \subseteq \mathcal{YS}(\hat{s}^{out}, m)$$

So $\mathcal{YS}(\hat{s}^{out}, m)$ contains all possible memory blocks younger than m in $c^{out}[set(m)]$ at p^{out} after the access of data reference D. As a result, the YS property holds at program point p^{out}, after the data access in p.

E. Termination of the analysis

The number of memory blocks in a program and the number of cache lines are finite. Therefore, the abstract domain $\hat{c} : L \mapsto 2^S$ is finite. Moreover, the cache update function $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$, and join function $\hat{\mathcal{J}}_{\hat{\mathcal{S}}}$ are monotonic. Therefore, our analysis will always terminate.

V. Scope-aware Persistence Analysis

VI. Motivations

Current persistence analysis (proposed by Ferdinand [9], [10], corrected in the above chapter) determines if once loaded, a memory block m will not be evicted out of the cache under all circumstances. However, a data memory block m remains in the cache under all circumstances only when the data cache is large enough to hold all possible data addresses. Otherwise, memory block m could be evicted hence it cannot be classified as persistence. Consequently, all data accesses to unclassified m are conservatively treated as all miss.

However, we notice that for each loop L, a data reference D may access memory block m only in a limited interval $[lw, up]$ of L's iterations (from iteration lw to iteration up of loop L). In this interval, if memory block m is guaranteed to remain in the cache once loaded, the first time D accesses m may causes one cache miss, but all subsequent accesses to m must result in cache hit. Moreover, outside this interval, memory block m is not accessed by data reference D, so it causes no cache miss to D. As a result, if memory block m is persistent (not evicted out of the cache once loaded) in the interval $[lw, up]$ of loop L's iterations, it causes at most one cache miss to D each time loop L is executed. Therefore, by capturing the persistence of memory block m in a smaller scope (i.e. interval $[lw, up]$ of loop L), we could guarantee a tighter worst-case performance of data cache.

Figure 4(a) presents our motivating example with four array references in two nested loop $L1$ and $L2$. The unpredictable array reference $A[x]$ could access any memory block in address set $Addr(A) = \{m_0, m_1\}$ (assume $A[x]$ always accesses within address range of array A). Similarly, the array reference $B[i][j]$ and $C[i][j]$ could access any memory block in address set $Addr(B) = \{m_2...m_9\}$ and $Addr(C) = \{m_{12}...m_{15}\}$ respectively. And $D[0]$ accesses only memory block m_{10}. Figure 4(b) shows the CFG and possible memory addresses of each data references. Assume a 2-way associative cache with four cache sets $\{f_0...f_3\}$, Figure 4(d) gives the possible cache conflicts within the loop nest. Because no memory block is persistent throughout the program execution, all data accesses are conservatively treated as all-miss in worst case according to the existing persistence analysis framework.

However, Figure 4(c) describes the access pattern for each data reference in the running example. As $A[x]$ is an unpredictable data access, it could access either m_0 or m_1 in any iteration of loop $L1$. On the other hand, $B[i][j]$ and $C[i][j]$ are loop-affine array access with statically predictable access pattern. When $i = 2$ and $j = 0..7$, $B[i][j]$ only accesses m_6. Therefore, if m_6 is not evicted in the scope $\{L1 \mapsto [2, 2], L2 \mapsto [0, 7]\}$ (interval $[0, 7]$ of $L2$'s iterations, for each $L2$'s execution in interval $[2, 2]$ of $L1$'s iterations), $B[i][j]$ has at most one cache miss for 8 accesses. Similarly, if m_{15} is persistent in the scope $\{L1 \mapsto [3, 3], L2 \mapsto [0, 15]\}$, $C[i][j]$ has at most one cache miss for 16 accesses. As a result, by capturing the persistence of memory block in those scopes, we could obtain a much tighter data cache performance estimation.

VII. Temporal Scope and Address Analysis

Central to our scope-aware data cache analysis is the notion of temporal scope that characterizes the behavior of a data reference over different loop iterations. Furthermore, we parameterize the definition and operations of temporal scopes with the static scope information on loop nesting. We will discuss how our proposed persistence analysis can utilize such information for more accurate abstract domain construction in Section VIII.

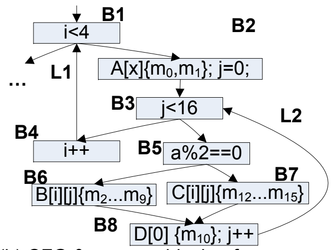
Definition 4: **(Temporal scope)** A temporal scope $\overline{m}^D$ of memory block m which may be accessed by a data reference D is defined as

$$\overline{m}^D = \{L_i \mapsto [lw, up] | \forall L_i \in reside(D)\}$$

where $reside(D)$ is the set of loops where D resides in. To simplify the presentation, we use $\overline{m}$ to denote $\overline{m}^D$ when there is no ambiguity about the data reference. For each of such loops L_i, temporal scope $\overline{m}$ (or $\overline{m}^D$) maintains a

```
int A[16];  int B[4][16];
int D[4]; short int C[4][16];
for ( i=0; i<4; i++) { //L1
    a = A[x];
    for (j=0; j<16; j++) {//L2
        if (a%2==0) b = B[i][j];
        else b = C[i][j];
        sum += D[0] + b;
    }
}
```

(a) Code fragment



(b) CFG & memory block references

	i / j	0	1	2	3
A[x]	0..15	m_0, m_1			
B[i][j]	0..7	m_2	m_4	m_6	m_8
B[i][j]	8..15	m_3	m_5	m_7	m_9
C[i][j]	0..15	m_{12}	m_{13}	m_{14}	m_{15}
D[0]	0..15	m_{10}			

(c) Memory block accessed according to loop iterations of L1 and L2

f_0	m_0, m_4, m_8, m_{12}
f_1	m_1, m_5, m_9, m_{13}
f_2	m_2, m_6, m_{10}, m_{14}
f_3	m_3, m_7, m_{15}

(d) Cache mapping

Figure 4. Motivating example

mapping between L_i and $\overline{m}[L_i]$, a closed interval $[lw, up]$ of L_i's iterations where D may access m. □

For a data reference D, address analysis calculates set of memory blocks possibly accessed by D. We follow the register expansion framework in [18] to identify address expression for each data reference at binary-code level. For each register used to specify address of load/store instruction, we perform register expansion to trace the source registers and the computation performed. We recursively expand a source register until it traces back to a defined constant c, an unpredictable value $\perp$, or a loop induction variable V. Readers are referred to White et al. [18] for details of address expression detection.

Given the address expression of a data reference D, set of possibly accessed memory blocks and their corresponding temporal scopes are automatically derived as follows.

- In case the address expression is a constant, it corresponds to a scalar access to a fixed memory block m. Data reference D will access m in all loop iterations. Therefore, the temporal scope $\overline{m}^D$ covers all iterations of each loop L where D resides in. In Figure 5(a), address expression of $D[0]$ is evaluated to $BaseD$, which corresponds to m_{10}. Because $D[0]$ will access m_{10} in all iterations of loop $L1$ and $L2$ where it resides in, the temporal scope $\overline{m}_{10} = \{L1 \mapsto [0,3], L2 \mapsto [0,15]\}$.
- If the address expression contains unpredictable value $\perp$, the corresponding array access may reference any of the memory blocks contained in the array. For example in Figure 5, $A[x]$ is an unpredictable access which may reference m_0 or m_1 in any iteration of $L1$. Therefore, the temporal scope $\overline{m}_0 = \{L1 \mapsto [0,3]\}$. Similarly, temporal scope $\overline{m}_1 = \{L1 \mapsto [0,3]\}$.
- If the address expression contains linear expression of loop-induction variables, it corresponds to loop-affine access with predictable access pattern, such as $B[i][j]$ in Figure 5(a). By enumerating possible values of the loop induction variables i and j, temporal scope of each memory block that is possibly accessed by $B[i][j]$ can be automatically calculated. For example, when $i = 2$ and $0 \leq j \leq 7$, value of the address expression for $B[i][j]$ is evaluated to $[128 + BaseB, 128 + 28 + BaseB]$, where $BaseB$ is the base address of $B[i][j]$. Given our assumption that $BaseB$ corresponds to memory block m_2 and memory block size is 32-Byte, the address range $[128 + BaseB, 128 + 28 + BaseB]$ corresponds to m_6, so the temporal scope $\overline{m}_6 = \{L1 \mapsto [2,2], L2 \mapsto [0,7]\}$.

	Address Expression
A[x]	$\perp \times 4 + m_0$ (BaseA)
B[i][j]	$16 \times i \times 4 + j \times 4 + m_2$ (BaseB)
C[i][j]	$16 \times i \times 2 + j \times 2 + m_{12}$ (BaseC)
D[0]	m_{10} (BaseD)

(a) Address expressions

$\overline{m}_0$	{ L1→[0,3] }
$\overline{m}_6$	{ L1→[2,2], L2→[0,7] }
$\overline{m}_7$	{ L1→[2,2], L2→[8,15] }
$\overline{m}_{15}$	{ L1->[3,3], L2→[0,15] }
$\overline{m}_{10}$	{ L1→[0,3], L2→[0,15] }

(b) Temporal scopes

Figure 5. Address expressions and temporal scopes

Given two memory blocks m_i and m_j accessed in temporal scope $\overline{m}_i$ and $\overline{m}_j$ respectively. An access to m_i in scope $\overline{m}_i[L]$ will increase the relative age of m_j in scope $\overline{m}_j[L]$ only if m_i and m_j are mapped to the same cache set and their temporal scopes overlap during execution of L. We define the overlapping between two temporal scope $\overline{m}_i$ and $\overline{m}_j$ in loop L as follows

Definition 5: **(Scope overlap)** The overlapping between two temporal scope $\overline{m}_i$ and $\overline{m}_j$ in loop L is recursively defined as

$$overlap(\overline{m}_i, \overline{m}_j, L) \iff m_i \neq m_j \wedge (\overline{m}_i[L] \cap \overline{m}_j[L]) \neq \emptyset \wedge overlap(\overline{m}_i, \overline{m}_j, outer(L)) \quad (1)$$

where $outer(L)$ is the immediate outer loop of L. Thus, two temporal scopes overlap at loop level L only if the access intervals for loop L and all outer loops containing L are *not* mutually exclusive.

In Figure 5(b), since $\overline{m}_6[L2]$ and $\overline{m}_7[L2]$ refer to interval $[0,7]$ and $[8,15]$ of $L2$'s iterations, they do not overlap. In an other example, $\overline{m}_{15}[L2]$ and $\overline{m}_6[L2]$ overlap in interval $[0,7]$ of L2's iterations. However, in the parent loop $L1$, $\overline{m}_{15}[L1]$ refers to interval $[3,3]$ while $\overline{m}_6[L1]$ refers to a separated interval $[2,2]$ of loop $L1$'s iterations. Therefore, the scope $\overline{m}_{15}[L2]$ and $\overline{m}_6[L2]$ do not overlap because they belong to $L2$'s executions in separated intervals of $L1$.

To capture the persistence of a data memory in a scope for more accurate WCET analysis, we integrate access pattern analysis into the abstract interpretation framework. In our analysis, we extend the definition of memory block persistence in [10], and utilize the computed temporal scope information for a scope-aware analysis. The proposed framework is built on our correct version of persistence analysis as described in Section III-C. The soundness proofs are presented in Section IX.

VIII. Scope-aware Persistence Analysis

The basic idea of our scope-aware persistence analysis is to categorize the persistence of memory blocks in the calculated temporal scopes (Section VII), instead of the globally defined persistence in [10]. For a data reference D, the temporal scope $\overline{m}^D$ identifies a mapping between loop L where D resides in and L's iteration interval $\overline{m}^D[L]$ where D may access m. The scope-aware analysis approach allows us to integrate access pattern into the abstract interpretation framework, and determine the local behavior of data cache. In particular, our scope-aware persistence analysis computes memory block persistence within its temporal scope for each static scope (loop hierarchy) it may get accessed.

Definition 6: **(Scope persistence)** Let $\overline{m}^D$ defines the loop interval $[\overline{m}^D[L].lw, \overline{m}^D[L].up]$ where data reference D may access memory block m in an execution of loop L (between L's entry and exit). The temporal scope $\overline{m}^D$ is persistent at loop level L if and only if within interval $\overline{m}^D[L]$, m is guaranteed to remain in the cache after the first time it is loaded into cache by D. □

Given the above definition of scope persistence, for memory block m to cause only one cache miss to data reference D in one complete execution of loop L, it does not need to stay in the cache for all iterations of L. In loop L, the temporal scope $\overline{m}^D$ (or $\overline{m}$ for short) defines an interval $\overline{m}[L]$ (from iteration $\overline{m}[L].lw$ to iteration $\overline{m}[L].up$ of loop L) where D may access m. If once loaded, memory block m is not evicted out of the cache in any execution within the interval $\overline{m}[L]$, all data accesses to m from D cause at most *one* cache miss for each complete execution of L.

To capture the scope persistence in the abstract domain of the persistence analysis framework, we define our scope-aware abstract set state and abstract cache state as follows.

Definition 7: **(Scope-aware abstract cache state)** In analysis at loop level L, abstract cache state $\hat{c}[L]$: $F \rightarrow \hat{S}$ maps cache sets to abstract set states. □

Definition 8: **(Scope-aware abstract set state)** An abstract set state $\hat{s}$: $\{l_1 \dots l_A\} \cup \{l_\top\} \rightarrow 2^{TS}$ maps cache lines (including the evicted line $l_\top$) to set of all temporal scopes TS (refer to Figure 6(c) for an example). $\hat{S}$ denotes the set of all abstract set states. □

In our scope-aware ACS $\hat{c}[L]$ of loop L, if temporal scope $\overline{m}$ is in $\hat{s}(l_x)$, once loaded to the cache in scope $\overline{m}[L]$, memory block m reaches maximum relative age x in any possible execution from iteration $\overline{m}[L].lw$ to iteration $\overline{m}[L].up$ of loop L.

We have re-designed the *update function* $\hat{\mathcal{U}}_{\hat{\mathcal{C}}}$ and *join function* $\hat{\mathcal{J}}_{\hat{\mathcal{C}}}$ to utilize the scope information when modeling cache conflicts in the ACS. By capturing such fine-grained persistence properties, our analysis can accurately model the local behavior of data cache for WCET estimation.

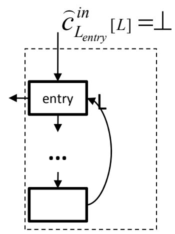
A. Overall framework

We adopt the multi-level persistence framework for instruction cache analysis from [2], and extend it for our data cache analysis. As shown in Figure 6(a), for each loop L, we perform a separate persistence analysis on the CFG fragment within L, with empty initial ACS $\hat{c}^{in}_{L_{entry}}[L] = \bot$ as input ACS of the L's entry node L_{entry}. Consequently, the analysis will consider only paths and data accesses within loop L. As a result, we can determine the local persistence of a memory block in different loop levels. In Figure 6 we show the estimation results of our analysis for the motivating example presented in Figure 4, and a detailed discussion will be given in Section VIII-C.

Algorithm 1 MPA(L) — Multi-level Persistence Analysis Algorithm. L denotes a loop (or the `main` procedure) under analysis.

1: $\hat{c}^{in}_{L_{entry}}[L] = \bot$;
2: $Queue.insert(L_{entry})$;
3: **while** !Queue.empty() **do**
4: n = Queue.remove();
5: $\hat{c}^{in}_{n}[L] = \hat{\mathcal{J}}_{\hat{\mathcal{C}}}(\{\hat{c}^{out}_{n'}[L] | \forall n' \in Pred(n) \wedge n' \in L\})$;
6: **if** reached_fixed_point($\hat{c}^{in}_{n}[L]$) **then** continue;
7: $\hat{c}^{out}_{n}[L] = \hat{c}^{in}_{n}[L]$;
8: **for** each data reference D in n **do**
9: $\hat{c}^{out}_{n}[L] = \hat{\mathcal{U}}_{\hat{\mathcal{C}}}(\hat{c}^{out}_{n}[L], TS^D, L)$;
10: **end for**
11: $Queue.insert(\{n' | \forall n' \in Succ(n) \wedge n' \in L\})$;
12: **end while**

Algorithm 1 describes the multi-level persistence analysis algorithm to analyze loop L. $\hat{c}^{in}_{n}[L]$ and $\hat{c}^{out}_{n}[L]$ denote the input and output ACSs of a node n for analysis at loop level L. $Pred(n)$ and $Succ(n)$ refer to the sets of predecessors and successors of n within the CFG of loop L currently being analyzed. We perform a standard fixed-point computation of the ACSs. The analysis initializes the input ACS of loop entry node L_{entry} to empty (line 1) because initially no memory block has been accessed in this loop. The processing queue $Queue$ starts with the loop entry node (line 2). For each node n, we compute the input ACS $\hat{c}^{in}_{n}[L]$ by joining all the output ACSs of its predecessors within L (line 5). The *scope-aware join function* $\hat{\mathcal{J}}_{\hat{\mathcal{C}}}$ computes the joined ACS as the union of all input ACSs. If the input ACS $\hat{c}^{in}_{n}[L]$ has reached fixed point, the analysis continue to process the next node in $Queue$ (line 6). Otherwise, for each memory reference D in node n, we compute $\hat{c}^{out}_{n}[L]$ from its input ACS and the set TS^D of temporal scopes of D as computed in Section VII (line 7-10). In case where no-write-allocate is used (in write-through or write-back policy), a store instruction does not modify the cache state. We consider only load instructions in the cache analysis. Otherwise for write-allocate policy, all load and store instructions will be considered in the ACS calculation.



(a) Inner loop analysis

$\hat{c}^{out}_{B4}$

	l_1	l_2	l_T
f_0			m_0, m_4, m_8, m_{12}
f_1			m_1, m_5, m_9, m_{13}
f_2	m_{10}		m_2, m_6, m_{14}
f_3			m_3, m_7, m_{15}

(b) Original persistence analysis

$\hat{c}^{out}_{B4}$[L1]

	l_1	l_2	l_T
f_0		$\overline{m}_4$ $\overline{m}_8$ $\overline{m}_{12}$	$\overline{m}_0$
f_1		$\overline{m}_9$	$\overline{m}_1$ $\overline{m}_5$ $\overline{m}_{13}$
f_2	$\overline{m}_{10}$	$\overline{m}_2$	$\overline{m}_6$ $\overline{m}_{14}$
f_3	$\overline{m}_3$ $\overline{m}_7$ $\overline{m}_{15}$		

(c) Scope-aware persistence analysis for L1

$\hat{c}^{out}_{B8}$[L2]

	l_1	l_2	l_T
f_0	$\overline{m}_4$ $\overline{m}_8$ $\overline{m}_{12}$		
f_1	$\overline{m}_9$	$\overline{m}_5$ $\overline{m}_{13}$	
f_2	$\overline{m}_{10}$	$\overline{m}_2$	$\overline{m}_6$ $\overline{m}_{14}$
f_3	$\overline{m}_3$ $\overline{m}_7$ $\overline{m}_{15}$		

(d) Scope-aware persistence analysis for L2

Figure 6. Multi-level analysis and results for the motivating example in Figure 4

Finally, all successors of n within L are inserted into $Queue$ to capture the possible changes in $\hat{c}_n^{out}[L]$ (line 11).

B. Scope-aware update and join functions

Scope-aware update function:

Given a data reference D which accesses a set of possible addresses $Addr(D) = \{m_1...m_k\}$ in loop L, the scope-aware update function $\hat{\mathcal{U}}_{\hat{\mathcal{C}}}$ calculate the change in ACS $\hat{c}[L]$ after a data reference of D (line 9 in Algorithm 1). For each memory block $m_a \in Addr(D)$, the temporal scope $\overline{m}_a^D$ (or $\overline{m}_a$ for short) identify the loop intervals where D may access m_a. An access to m_a in scope $\overline{m}_a[L]$ (from iteration $\overline{m}_a[L].lw$ to iteration $\overline{m}_a[L].up$) does not affect the maximum relative age (and the scope persistence) of a memory block m in scope $\overline{m}[L]$ if $\overline{m}_a$ and $\overline{m}$ do not overlap in loop L (refer to Equation 1 in Section VII). Therefore, our proposed scope-aware update function $\hat{\mathcal{U}}_{\hat{\mathcal{C}}}$ only considers memory block m_a as conflict with memory block m in scope $\overline{m}[L]$ when the temporal scope $\overline{m}_a$ and $\overline{m}$ overlap in L.

$$\hat{\mathcal{U}}_{\hat{\mathcal{C}}}(\hat{c}, TS^D = \{\overline{m}_1...\overline{m}_k\}, L) = \hat{c}[f_i \mapsto \hat{\mathcal{U}}_{\hat{\mathcal{S}}}(\hat{c}[f_i], X_{f_i}, L)]$$
$$\text{for all } f_i \in \{set(m_1)..set(m_k)\}$$
$$\text{where } X_{f_i} = \{\overline{m}_y | \overline{m}_y \in \{\overline{m}_1...\overline{m}_k\}, set(m_y) = f_i\}$$

Given data reference D and its set of possible addresses $Addr(D)$, our scope-aware cache update function $\hat{\mathcal{U}}_{\hat{\mathcal{C}}}$ computes the change in cache set f_i possibly affected by the data access using our scope-aware set update function $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$. For each input abstract set state $\hat{s}^{in}$, the set update function computes the output abstract set state $\hat{s}^{out}$ and tracks the Younger Set of each temporal scope $\overline{m} \in \hat{s}^{in}$ as follows.

$$\hat{\mathcal{U}}_{\hat{\mathcal{S}}}(\hat{s}^{in}, X_{f_i}, L) = \hat{s}^{out} \text{ with :}$$
$$\hat{s}^{out}(l_x) = \{\overline{m} | \overline{m} \in \hat{s}^{in} \cup X_{f_i}, x = \min(|\mathcal{YS}(\hat{s}^{out}, \overline{m})| + 1, \top)\}$$

where $\forall \overline{m} \in \hat{s}^{in} \cup X_{f_i}, \mathcal{YS}(\hat{s}^{out}, \overline{m}) =$

$$\begin{cases} \emptyset & \text{if } \overline{m} \notin \hat{s}^{in} \\ \emptyset & \text{else if } \overline{m} \in X_{f_i} \wedge \neg overlap(\overline{m}, \overline{m}_a, L), \forall \overline{m}_a \in TS^D \\ \mathcal{YS}(\hat{s}^{in}, \overline{m}) \cup \{m_a | \overline{m}_a \in X_{f_i} \wedge overlap(\overline{m}, \overline{m}_a, L)\} & \text{Otherwise.} \end{cases}$$

where $overlap(\overline{m}, \overline{m}_a, L)$ is true when the temporal scopes $\overline{m}$ and $\overline{m}_a$ overlap in loop level L according to Equation 1.

The update function $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$ determines the maximum relative age x of temporal scope $\overline{m}$ in output abstract set state $\hat{s}^{out}$ by computing the younger set $\mathcal{YS}(\hat{s}^{out}, \overline{m})$. In our scope-aware ACS, the younger set $\mathcal{YS}(\hat{s}^{out}, \overline{m})$ identifies the set of all possible memory blocks that could be younger than m in all executions in scope $\overline{m}[L]$ after the first access to m in this scope. To determine the younger set $\mathcal{YS}(\hat{s}^{out}, \overline{m})$, we have the following scenarios:

- If temporal scope $\overline{m}$ is not in $\hat{s}^{in}$, memory block m has not been accessed the first time in scope $\overline{m}[L]$ in any execution. If the data reference D accesses m, m will be brought to youngest cache line l_1 with no younger memory block. Otherwise, memory block m remains not accessed. Since our scope-aware persistence analysis only captures the maximum relative age of m after the first access to m in scope $\overline{m}[L]$, our scope-aware update function $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$ adds $\overline{m}$ to $\hat{s}^{out}$ as youngest memory block with empty younger set.
- If temporal scope $\overline{m} \in \hat{s}^{in}$, memory block m may have been accessed in scope $\overline{m}[L]$. In scope $\overline{m}[L]$, for any memory block $m_a \in Addr(D)$, the data reference D may access m_a if temporal scopes $\overline{m}$ and $\overline{m}_a$ overlap in loop L. Therefore, if exists $m_a \in Addr(D)$ where $overlap(\overline{m}, \overline{m}_a, L)$, data reference D may access m_a and not renew m. Otherwise if $m \in Addr(D)$, all data accesses of D in scope $\overline{m}[L]$ will definitely access and renew m. Consequently, we can guarantee that data reference D will indeed access m in scope $\overline{m}[L]$ and renew $\overline{m}$ to youngest cache line l_1.
- Otherwise, in scope $\overline{m}[L]$, the data reference D may access any memory block m_a ($m_a \neq m$) if temporal scope $\overline{m}_a$ overlaps with $\overline{m}$ in loop L. If m_a is mapped to cache set f_i ($m_a \in X_{f_i}$), it can be accessed and become a new younger memory block of m in scope $\overline{m}[L]$. Therefore, our scope-aware update $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$ function adds all those memory blocks to the younger set $\mathcal{YS}(\hat{s}^{out}, \overline{m})$ of $\overline{m}$, and set its maximal relative age accordingly.

Figure 7(a) illustrates our scope-aware persistence analysis in loop $L2$ of the running example in Figure 4. While m_4, m_8, and m_{12} are all mapped to cache set f_0, the temporal

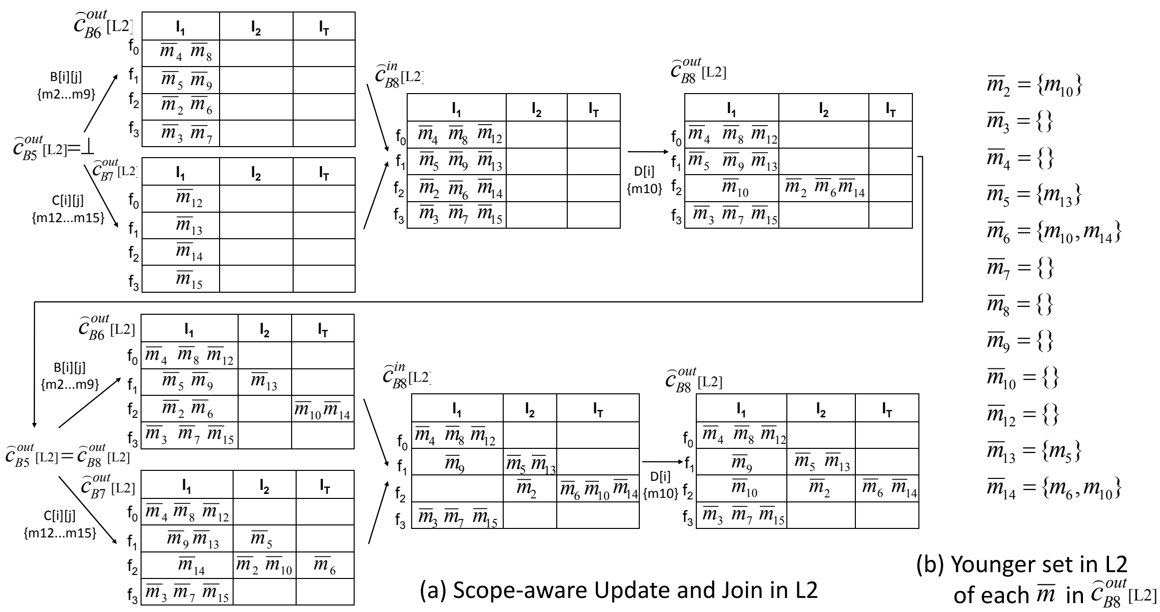


Figure 7. Scope-aware ACS computation for L2 of the motivating example in Figure 4

scopes $\overline{m}_4$, $\overline{m}_8$, and $\overline{m}_{12}$ do not overlap in loop $L2$. As a result, they do not affect the scope-persistence of each other. On the other hand, in cache set f_1, $B[i][j]$ accesses m_5 when $i = 1$ and $j = 8..15$, while $C[i][j]$ accesses m_{13} when $i = 1$ and $j = 0..15$. Therefore, the temporal scope $\overline{m}_5$ overlaps with $\overline{m}_{13}$ in loop $L2$. Hence m_{13} will age m_5 and become a younger memory block of m_5 in scope $\overline{m}_5[L2]$. Therefore, the scope-aware update function adds m_{13} to the younger set of $\overline{m}_5$, as shown in Figure 7(b).

Scope-aware join function:

At any program point p in loop level L, the join function $\hat{\mathcal{J}}_{\hat{\mathcal{C}}}$ (line 5 in Algorithm 1) computes an ACS from all the output ACSs of p's control flow predecessors. It can be done by pair-wise joining of two output ACSs $\hat{c}_1[L]$ and $\hat{c}_2[L]$ into a representative ACS $\hat{c}[L]$ at p using the the scope-aware join function $\mathcal{J}_{\hat{\mathcal{C}}}$. Formally, our scope-aware join function is defined as follows.

$\mathcal{J}_{\hat{\mathcal{C}}}(\hat{c}_1, \hat{c}_2) = \hat{c}[s_i \mapsto \mathcal{J}_{\hat{\mathcal{S}}}(\hat{c}_1[s_i], \hat{c}_2[s_i])]$

$\mathcal{J}_{\hat{\mathcal{S}}}(\hat{s}_1, \hat{s}_2) = \hat{s}$ with:

$\hat{s}(l_x) = \{\overline{m} | \overline{m} \in \hat{s}_1 \cup \in \hat{s}_2, x = \min(|\mathcal{YS}(\hat{s}, \overline{m})| + 1, \top)\}$

where $\forall \overline{m} \in \hat{s}_1 \cup \hat{s}_2$

$$\mathcal{YS}(\hat{s}, \overline{m}) = \begin{cases} \mathcal{YS}(\hat{s}_1, \overline{m}) \cup \mathcal{YS}(\hat{s}_2, \overline{m}) \\ \qquad \text{if } \overline{m} \in \hat{s}_1 \wedge \overline{m} \in \hat{s}_2 \\ \mathcal{YS}(\hat{s}_1, \overline{m}) \\ \qquad \text{if } \overline{m} \in \hat{s}_1 \wedge \overline{m} \notin \hat{s}_2 \\ \mathcal{YS}(\hat{s}_2, \overline{m}) \\ \qquad \text{if } \overline{m} \notin \hat{s}_1 \wedge \overline{m} \in \hat{s}_2 \end{cases}$$

For each temporal scope $\overline{m}$, the scope-aware join function $\mathcal{J}_{\hat{\mathcal{C}}}$ unionizes the younger set of $\overline{m}$ in both output ACSs from the control flow predecessors to form the younger set $\mathcal{YS}(\hat{s}, \overline{m})$ of m in abstract set state $\hat{s} = \hat{c}[set(m)]$ at p. Therefore, $\mathcal{YS}(\hat{s}, \overline{m})$ always contains all possible younger memory blocks of m in scope $\overline{m}$ at p.

C. ACS computation of the motivating example

Figure 6(b), (c) and (d) shows the fixed-point ACSs computed by the original persistence analysis (at basic block $B4$, exit of $L1$), our multi-level analysis for $L1$ (at $B4$) and $L2$ (at basic block $B8$, exit of $L2$), respectively. Given 2-way associative cache with 4 cache sets, no memory block accessed by $B[i][j]$ and $C[i][j]$ can be categorized as persistent in the original persistence analysis. On the other hand, our multi-level scope-aware persistence analysis produces much tighter estimation results on the worst-case cache behavior. For example, m_4 accessed by $B[i][j]$ is guaranteed to be scope persistent at both loop levels, resulting in at most 1 cold miss globally. m_5 is scope persistent only in $L2$. Thus, accesses to m_5 in each complete execution of $L2$ (between entry to exit) incurs at most 1 cold miss.

IX. Safety proofs of scope-aware persistence analysis

In this section, we will prove the safety of our proposed scope-aware persistence analysis framework.

In a concrete cache state c, for LRU replacement policy, the relative age of memory block m is determined by the number of memory blocks more recently used (younger) than m in the same cache set. Let $s = c[set(m)]$ be the concrete set state of the cache set where memory block m is mapped to, and concrete younger set $ys(s, m)$ be the set of memory blocks more recently used (younger) than m in set s (as in Definition 2), we have

$$m \in s(l_y) \rightarrow ys(s,m) = s(l_1) \cup ... \cup s(l_{y-1}) \\ \wedge\, y = |ys(s,m)| + 1$$

A memory block m is persistent in the scope $\overline{m}[L]$ (from iteration $\overline{m}[L].lw$ to iteration $\overline{m}[L].up$ of loop L) if once m has been loaded to the cache the first time in this scope, it will not be evicted out of the cache in any possible execution before the program exists the scope (i.e. finishes iteration $\overline{m}[L].up$ of loop L). In our ACS semantic, given ACS $\hat{c}[L]$ of analysis in loop L and $\hat{s} = \hat{c}[L][set(m)]$, if temporal scope $\overline{m} \in \hat{s}(l_x)$, once loaded to the cache in scope $\overline{m}[L]$, memory block m has maximum relative age x in all possible executions in the scope. Our scope-aware persistence analysis computes the maximum relative age x by tracking the younger set $\mathcal{YS}(\hat{s},\overline{m})$, the set all memory blocks which are possibly younger than m in the scope $\overline{m}[L]$ after m is loaded to the cache. As the relative age of memory block m is determined by the number of memory blocks more recently used (younger) than m in the same cache set, the maximum relative age of m in scope $\overline{m}[L]$ should greater than the size of younger set $\mathcal{YS}(\hat{s},\overline{m})$, i.e. $x = |\mathcal{YS}(\hat{s},\overline{m})| + 1$. If memory block m has less than A possibly younger memory blocks in scope $\overline{m}[L]$, once loaded, it will not be evicted out of the cache and is persistent in scope $\overline{m}[L]$.

To prove the safety of our scope-aware persistence analysis, we prove that for any execution path pa that reaches program point p in the scope $\overline{m}[L]$ with concrete cache state c, *if* path pa has accessed memory block m in this scope, the younger set $\mathcal{YS}(\hat{s},\overline{m})$ contain all memory blocks in concrete younger set $ys(s,m)$, the set of memory blocks younger than m in cache set $s = c[set(m)]$. Consequently, the maximum relative age x determined by our analysis ($x = |\mathcal{YS}(\hat{s},\overline{m})| + 1$) will always greater or equal than the relative age y of memory block m in concrete cache set s ($y = |ys(s,m)|+1$). Therefore, our scope-aware persistence analysis is safe.

Note that our scope-aware persistence analysis computes the maximum relative age x of memory block m only after the first time memory block m has been loaded to the cache in scope $\overline{m}[L]$. We do not consider the relative age of memory block m before its first access in this scope, as we conservatively assume the first access to m in the scope $\overline{m}[L]$ always results in a cache miss.

A. *Structure of the proof*

We prove by induction that for each temporal scope $\overline{m}$ in ACS $\hat{c}[L]$, the ScopeYS property holds in all possible execution paths in scope $\overline{m}[L]$.

Definition 9: **(ScopeYS property)**: Given an arbitrary path pa from the start of execution to program point p in scope $\overline{m}[L]$ of loop L which results in concrete cache state c, and $\hat{c}[L]$ be the computed fixed point ACS of loop L at p. For each memory block $m \in s = c[set(m)]$ and its corresponding temporal scope $\overline{m} \in \hat{s} = \hat{c}[L][set(m)]$, if path pa has accessed memory block m in scope $\overline{m}[L]$, the younger set $\mathcal{YS}(\hat{s},\overline{m})$ will contain all memory blocks in concrete younger set $ys(s,m)$.□

$$\forall m \in c, s = c[set(m)], \hat{s} = \hat{c}[L][set(m)], \\ \neg Accessed(m,\overline{m}[L],s) \vee ys(s,m) \subseteq \mathcal{YS}(\hat{s},\overline{m})$$

where $Accessed(m,\overline{m}[L],s)$ indicates if memory block m has been accessed in scope $\overline{m}[L]$ for concrete set state s.

We prove by induction that for each memory block m and its corresponding temporal scope $\overline{m}$, the ScopeYS property holds in all possible execution paths in scope $\overline{m}[L]$ (from iteration $\overline{m}[L].lw$ to iteration $\overline{m}[L].up$ of loop L)

- If memory block m has not been accessed in scope $\overline{m}[L]$ ($\neg Accessed(m,\overline{m}[L])$), our ScopeYS property is trivially true. We do not consider the relative age of memory block m before its first access in scope $\overline{m}[L]$, as we conservatively assume the first access to m in the scope results is a miss.
- At the first access to m in scope $\overline{m}[L]$, memory block m is brought to concrete set state s at youngest line $s(l_1)$. Consequently, $ys(s,m) = \emptyset$, so $ys(s,m) \subseteq \mathcal{YS}(\hat{s},\overline{m})$. Therefore the ScopeYS property is true immediately after the first access to m in scope $\overline{m}[L]$.
- Assume ScopeYS property holds at p^{in}, before the program point p. If at p, a data reference D accesses a set of possible memory blocks $\{m_1...m_k\}$ in their respective temporal scopes $\{\overline{m}_1...\overline{m}_k\}$, we prove the ScopeYS property holds at p^{out}, after program point p by proving the correctness of our scope-aware update function (Section IX-B).
- Assume ScopeYS property holds at p^{out}, we prove ScopeYS property holds at p_n^{in}, before the next program point p_n, by proving the correctness of our scope-aware join function (Section IX-C).

For each memory block m in scope $\overline{m}[L]$, we prove that ScopeYS property holds before and immediately after the first access to m in scope $\overline{m}[L]$. In subsequent executions within the scope, ScopeYS property holds after each data access, and from one program point to another. Therefore, ScopeYS property holds for any arbitrary path pa in the scope $\overline{m}[L]$. Consequently, at any program point p in scope $\overline{m}[L]$ with concrete cache state c, the younger set $\mathcal{YS}(\hat{s},\overline{m})$ contains all memory blocks in concrete younger set $ys(s,m)$ of m in the set $s = c[set(m)]$. As a result, the maximum relative age x of memory block m in scope $\overline{m}[L]$ determined by our ACS $\hat{c}[L]$ ($x = |\mathcal{YS}(\hat{s},\overline{m})|+1$) is always greater than or equal to the relative age y of m in set s ($y = |ys(s,m)+1$). Therefore, our analysis safely estimates the maximum relative age and the persistence of m in scope $\overline{m}[L]$.

B. Safety proof of scope-aware update function

At program point p in loop L, a data reference D accesses a set of possible memory blocks $Addr(D) = \{m_1 ... m_k\}$ in their respective temporal scopes $\{\overline{m}_1 ... \overline{m}_k\}$. The scope-aware update function computes the change in ACS $\hat{c}[L]$, and tracks the younger set $\mathcal{YS}(\hat{s}, \overline{m})$ of each temporal scope $\overline{m}$ after the data access. We prove our scope-aware update function preserves the ScopeYS property. Assume ScopeYS property holds at p^{in}, before program point p, we prove ScopeYS property holds at p^{out}, after program point p.

Given the concrete cache state c^{in} of path pa at p^{in}, and $\hat{c}^{in}[L]$ is the computed ACS of loop L at p^{in}. Assume ScopeYS property holds at p^{in}, we have

$$\begin{aligned}&\forall m \in c^{in}, s^{in} = c^{in}[set(m)], \hat{s}^{in} = \hat{c}^{in}[set(m)],\\ &\quad \neg Accessed(m, \overline{m}[L], s^{in})\\ &\quad\quad \vee\, ys(s^{in}, m) \subseteq \mathcal{YS}(\hat{s}^{in}, \overline{m}) \qquad \text{[B.1]}\end{aligned}$$

Given concrete cache state c^{out} of path pa at p^{out}, and $\hat{c}^{out}[L]$ is the computed ACS of loop L at p^{out}. We prove ScopeYS property holds at p^{out}:

$$\begin{aligned}&\forall m \in c^{out}, s^{out} = c^{out}[set(m)], \hat{s}^{out} = \hat{c}^{out}[L][set(m)],\\ &\quad \neg Accessed(m, \overline{m}[L], s^{out})\\ &\quad\quad \vee\, ys(s^{out}, m) \subseteq \mathcal{YS}(\hat{s}^{out}, \overline{m}) \qquad \text{[B.2]}\end{aligned}$$

At program point p in loop L, given a data reference D and input abstract set state $\hat{s}^{in}$, our scope-aware update function $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$ computes the output abstract set state $\hat{s}^{out}$ and the updated younger set $\mathcal{YS}(\hat{s}^{out}, \overline{m})$ as follow:

$$\begin{aligned}&\hat{\mathcal{U}}_{\hat{\mathcal{S}}}(\hat{s}^{in}, X_{f_i}, L) = \hat{s}^{out} \text{ with :}\\ &\quad \hat{s}^{out}(l_x) = \{\overline{m} | \overline{m} \in \hat{s}^{in} \cup X_{f_i},\\ &\quad\quad x = \min(|\mathcal{YS}(\hat{s}^{out}, \overline{m})| + 1, \top)\}\end{aligned}$$

where $\forall \overline{m} \in \hat{s}^{in} \cup X_{f_i}, \mathcal{YS}(\hat{s}^{out}, \overline{m}) =$

$$\begin{cases} \emptyset & \text{if } \overline{m} \notin \hat{s}^{in} \\ \emptyset & \text{else if } \overline{m} \in X_{f_i} \wedge \neg overlap(\overline{m}, \overline{m}_a, L), \\ & \qquad \forall \overline{m}_a \in TS^D \\ \mathcal{YS}(\hat{s}^{in}, \overline{m}) \cup \{m_a | \overline{m}_a \in X_{f_i} \wedge overlap(\overline{m}, \overline{m}_a, L)\} & \\ & \text{Otherwise.} \end{cases}$$

We prove the correctness of our scope-aware update function $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$ by dividing access scenarios into two cases:

Case 1: m has not been accessed in scope $\overline{m}[L]$

- Case 1.1: D does not access m at program point p
  As D does not access m at p, m remains not accessed at p^{out}. Therefore we have

  $$\begin{aligned}&\neg Accessed(m, \overline{m}[L], s^{in}) \wedge D \text{ does not access } m\\ &\quad \rightarrow \neg Accessed(m, \overline{m}[L], s^{out}) \quad \text{([B.2] proven)}\end{aligned}$$

- Case 1.2: D accesses m at program point p
  Since data reference D accesses m, m becomes the most recently used memory block in cache line l_1. Consequently, m has no younger memory block.

  $$\begin{aligned}&ys(s^{out}, m) = \emptyset\\ &\rightarrow ys(s^{out}, m) \subseteq \mathcal{YS}(\hat{s}^{out}, \overline{m}) \quad \text{([B.2] proven)}\end{aligned}$$

Case 2: m has been accessed in scope $\overline{m}[L]$

Since memory block m has been accessed in scope $\overline{m}[L]$ and ScopeYS holds at p^{in}, we have:

$$\begin{aligned}&[B.1] \wedge Accessed(m, \overline{m}[L], s^{in})\\ &\quad \rightarrow ys(s^{in}, m) \subseteq \mathcal{YS}(\hat{s}^{in}, \overline{m}) \qquad [1]\end{aligned}$$

In scope $\overline{m}[L]$, D may access memory block m_a only if temporal scope $\overline{m}_a$ overlaps with $\overline{m}$ in loop L ($overlap(\overline{m}, \overline{m}_a, L)$. Moreover, m_a will become a younger memory block of m in s^{out} if $m_a \neq m$ and they are mapped to the same cache set ($m_a \in X_{f_i}$). As a result, we have

$$[2] \quad ys(s^{out}, m) = \begin{cases} \emptyset & \text{if } m_a = m \\ ys(s^{in}, m) \cup \{m_a\} & \\ & \text{if } m_a \neq m \wedge m_a \in X_{f_i} \\ ys(s^{in}, m) & \text{Otherwise} \end{cases}$$

$$\text{where } \overline{m}_a \in X_{f_i} \wedge overlap(\overline{m}, \overline{m}_a, L)$$

$$\begin{aligned}&[3] \quad \mathcal{YS}(\hat{s}^{out}, \overline{m}) =\\ &\mathcal{YS}(\hat{s}^{in}, \overline{m}) \cup \{m_a | \overline{m}_a \in X_{f_i} \wedge overlap(\overline{m}, \overline{m}_a, L)\}\\ &[1][2][3] \rightarrow ys(s^{out}, m) \subseteq \mathcal{YS}(\hat{s}^{out}, \overline{m}) \qquad \text{([B.2] proven)}\end{aligned}$$

As a result, in all cases, either memory block m has not been accessed, or $\mathcal{YS}(\hat{s}^{out}, \overline{m})$ contains all possible temporal scopes of memory blocks accessed within scope $\overline{m}[L]$ which may be younger than m. Therefore, the ScopeYS property holds at p^{out}.

C. Safety proof of scope-aware join function

Assume ScopeYS property holds at p^{out}, after program point p, we prove that ScopeYS property holds at p_n^{in}, before the next program point p_n by proving the correctness of our scope-aware join function $\hat{\mathcal{J}}_{\hat{\mathcal{S}}}$.

Given concrete cache state c^{out} of path pa at p^{out}, and $\hat{c}^{out}[L]$ is the computed ACS of loop L at p^{out}. Assume ScopeYS property holds at p^{out}, we have

$$\begin{aligned}&\forall m \in c^{out}, s^{out} = c^{out}[set(m)], \hat{s}^{out} = \hat{c}^{out}[L][set(m)],\\ &\quad \neg Accessed(m, \overline{m}[L], s^{out})\\ &\quad\quad \vee\, ys(s^{out}, m) \subseteq \mathcal{YS}(\hat{s}^{out}, \overline{m}) \qquad \text{[C.1]}\end{aligned}$$

Let c_n^{in} be the concrete cache state of path pa at p_n^{in}, and $\hat{c}_n^{in}[L]$ is the computed ACS of loop L at p_n^{in}. We prove ScopeYS property holds at p_n^{in}:

$$\begin{aligned}&\forall m \in c_n^{in}[L], s_n^{in} = c_n^{in}[set(m)], \hat{s}_n^{in} = \hat{c}_n^{in}[L][set(m)],\\ &\quad \neg Accessed(m, \overline{m}[L], s_n^{in})\\ &\quad\quad \vee\, ys(s_n^{in}, m) \subseteq \mathcal{YS}(\hat{s}_n^{in}, \overline{m}) \qquad \text{[C.2]}\end{aligned}$$

From our proposed scope-aware join function $\hat{s} = \hat{\mathcal{J}}_{\hat{\mathcal{S}}}(\hat{s}_1, \hat{s}_2)$, younger set $\mathcal{YS}(\hat{s}, \overline{m})$ of m at p_n^{in} is the union of all younger sets of incoming edges of p_n^{in}. As p^{out} is one of the incoming edge of p_n^{in}, we have

$$\mathcal{YS}(\hat{s}^{out}, \overline{m}) \subseteq \mathcal{YS}(\hat{s}_n^{in}, \overline{m}) \qquad [\hat{\mathcal{J}}_{\hat{\mathcal{S}}}]$$

Because p_n^{in} is immediately after p^{out}, no new memory block is accessed. Therefore the concrete set state s_n^{in} is exactly the same as concrete set state s^{out}, and the concrete younger set remains the same:

$$ys(s_n^{in}, m) = ys(s^{out}, m) \quad \text{[C.3]}$$

If m has not been accessed in scope $\overline{m}[L]$ at p^{out}, m remains not accessed at p_n^{in}. The ScopeYS property will hold at p_n^{in}.

Otherwise, if m has been accessed in scope $\overline{m}[L]$ at p^{out}, we have

$$\begin{array}{ll} \text{[C.1]} & ys(s^{out}, m) \subseteq \mathcal{YS}(\hat{s}^{out}, \overline{m}) \\ [\hat{\mathcal{J}}_{\hat{\mathcal{S}}}] & \mathcal{YS}(\hat{s}^{out}, \overline{m}) \subseteq \mathcal{YS}(\hat{s}_n^{in}, \overline{m}) \\ \text{[C.3]} & ys(s_n^{in}, m) = ys(s^{out}, m) \\ \multicolumn{2}{l}{\rightarrow ys(s_n^{in}, m) \subseteq \mathcal{YS}(\hat{s}_n^{in}, \overline{m}) \qquad \text{([C.2] proven)}} \end{array}$$

The younger set $\mathcal{YS}(\hat{s}_n^{in}, \overline{m})$ contains all possible memory blocks younger than m in $set(m)$ of s_n^{in} at p_n^{in}. Therefore the ScopeYS property holds at p_n^{in}.

According to the proof structure outlined in Section IX-A, the ScopeYS property holds before and immediately after memory block m is first accessed in scope $\overline{m}[L]$. Then ScopeYS property holds before and after memory access at each program point p, and from p to the next program point p_n. As a result, the maximum relative age x of memory block m in scope $\overline{m}[L]$ determined by our scope-aware persistence analysis (i.e. $x = |\mathcal{YS}(\hat{s}, \overline{m})| + 1$) is always greater or equal to the relative age of m in concrete set state $s = c[set(m)]$ in arbitrary path pa after the first access of m in scope $\overline{m}[L]$. Therefore, our scope-aware persistence analysis is safe.

X. Cache Miss Computation

In abstract interpretation-based approaches, the cache analysis results are used to classify the cache behavior of each data reference D in the program. Typical worst case categories are (1) *All Hit (AH)*: all data accesses of D result in cache hit; (2) *All Miss (AM)*: all data accesses of D result in cache miss; (3) *Persistent (PS)*: all possible accessed memory blocks of D are persistent (D has at most one cold miss for each persistent memory block); and (4) *Non Classified (NC)*: the cache behavior of D could not be classified (all accesses of D are considered to be misses).

In the presence of data cache, different executions of the same data reference may access various memory blocks and result in different cache behavior. In our motivating example shown in Figure 4, data reference $B[i][j]$ may access m_4, m_5, and m_6 in the temporal scopes $\overline{m}_4$, $\overline{m}_5$, and $\overline{m}_6$ respectively. As illustrated in Figure 6(c) and Figure 6(d), memory blocks may have distinct cache behaviors in different loop nesting levels. Scope persistence of the above-mentioned memory blocks are shown in Figure 8. In Figure 6, because temporal scope $\overline{m}_4$ is not aged to evicted line $l_\top$ in both $L1$ and $L2$, m_4 is persistent in both scope $\overline{m}_4[L1]$ and $\overline{m}_4[L2]$. Therefore, we annotate the iterations of $L1$ and $L2$ bounded by $\overline{m}_4$ with PS. On the other hand, $\overline{m}_5$ is not persistent in outer loop $L1$ (annotated as $\neg PS$) but is persistent in inner loop $L2$, so m_5 is persistent in scope $\overline{m}_5[L2]$ but not $\overline{m}_5[L1]$. $\overline{m}_6$ is not persistent in any of the loop levels. Pessimistically categorizing all data accesses from $B[i][j]$ as Non Classified (as in the original persistence analysis) introduces significant over-estimation on the total number of data misses, which can be avoided in our scope-aware data cache analysis.

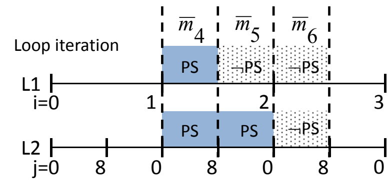


Figure 8. Temporal scopes and loop iterations

Our multi-level analysis computes a fixed-point abstract cache states $\hat{c}_n^{in}[L]$ ($\hat{c}_n^{out}[L]$) for entry (exit) of each CFG node n in each loop level L. If m is persistent in scope $\overline{m}[L]$ (or $\overline{m}^D[L]$) of loop level L, accesses to m by data reference D incurs only one cold miss for each complete execution of L (between entry and exit). Let L_{ps} be the outer-most loop level where $\overline{m}$ is persistent. Hence, accesses to m incur 1 cold miss for each execution of L_{ps} (including all its inner loops). The following function $blockMiss(D, m)$ computes the maximum number of cache misses D may incur due to accesses of m during the entire program execution.

$$blockMiss(D, m) = \begin{cases} \prod(\overline{m}[L_i].up - \overline{m}[L_i].lw + 1) \\ \quad \forall L_i \in reside(D), \text{ if } L_{ps} == \emptyset \\ 1 \quad \text{if } outer(L_{ps}) == \emptyset \\ \prod(\overline{m}[L_i].up - \overline{m}[L_i].lw + 1) \\ \quad \forall L_i \in outer(L_{ps}), \text{ otherwise.} \end{cases}$$
$$\text{with } \overline{m} = \overline{m}^D$$

where $outer(L_{ps})$ is the set of all outer loops of L_{ps}. In other words, $blockMiss(D, m)$ computes the number of times L_{ps} executed (in its outer loops) given the temporal scope where m may get accessed by D. In case $\overline{m}$ is not persistent in any loop level ($L_{ps} == \emptyset$), each access to m within its temporal scope results into 1 miss. On the other hand, if L_{ps} is outer-most loop of the program (globally persistent), all accesses to m incur only 1 cold miss.

As illustrated in Figure 8, $L1$ is the outer most loop where $\overline{m}_4$ is persistent. Since $L1$ is the outermost loop, m_4 causes at most one cold miss globally. $\overline{m}_5$ is only persistent in

Table I
BENCHMARK DESCRIPTIONS AND WCET ESTIMATION RESULT

Benchmark	Benchmark description	Array Size	Simulation (cycle)	Our Analysis (cycle)	Analysis Time
Edn	Finite Impulse Response (FIR) filter calculations.	2048	2,542,444	2,628,150	0.28s
Fdct	Fast Discrete Cosine Transform.	2048	917,636	926,468	0.92s
Cnt	Counts non-negative numbers in a matrix.	32×32	21,611	22,826	0.02s
Matmult	Matrix multiplication.	24×24	374,887	441,916	0.04s
Bsort100	Bubblesort program.	1024	15,945,200	17,350,300	0.02s
InsertSort	Insertion sort on a reversed array.	1024	14,900,732	16,279,600	0.58s
Jfdctint	Discrete-cosine transformation of pixel blocks.	256×64	1,485,075	1,497,910	2.62s
Lms	LMS adaptive signal enhancement.	1024	1,425,585	1,580,200	0.04s
Adpcm	Adaptive pulse code modulation algorithm.	2048	193,525	298,632	0.14s

$L2$. Therefore, accesses to $m5$ from $B[i][j]$ causes one cold miss for each iteration of $L1$ in the interval $[1,1]$ defined by $\overline{m}_5[L1]$. $\overline{m}_6$ is not persistent in any level, so all occurrences of $B[i][j]$ in the scope result in cache misses. The temporal scope $\overline{m}_6$ covers interval $[2,2]$ of $L1$ and $[0,7]$ of $L2$, so m_6 causes at most $1 \times 1 \times 8 = 8$ misses to $B[i][j]$.

Finally, the maximal possible cache misses incurred by D, $miss(D)$, is the summation of $blockMiss(D,m)$ over all memory blocks in $AddrSet(D)$ which D may access.

$$miss(D) = \sum blockMiss(D,m), \forall m \in AddrSet(D)$$

In our motivating example, $B[i][j]$ accesses 8 memory blocks ($\{m_2, \ldots, m_9\}$). According to our scope-aware analysis results shown in Figure 6, m_6 is non-persistent in both $L1$ and $L2$, m_5 is persistent only in $L2$, and other 6 memory blocks are persistent in both loops. According to our cache miss estimation, maximal number of cache misses from $B[i][j]$ is $8 + 1 + 1 \times 6 = 15$ misses, compared to the original pessimistic analysis which considers all accesses to $B[i][j]$ lead to totally 64 cache misses.

XI. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of our proposed scope-based persistence analysis using the data-intensive routines taken from the WCET Benchmarks ([1]). We assume the benchmarks are executed on a processor architecture with 5-stage pipeline, in-order execution, perfect branch prediction, separate L1 instruction cache and data cache. Both instruction and data caches have cache size 2 KB , block size 32 B, cache associativity 2, and perfect LRU replacement policy. Cache hit latency is 1 cycle, and cache miss latency is 6 cycles. We use SimpleScalar tool ([3]) to obtain simulation results. We extend SimpleScalar simulation to make it consistent with the assumptions made in our analysis. The cache analysis results on maximum number of data cache misses for each data reference are integrated as linear constraints into Chronos ([8]), an ILP-based WCET analysis tool for static WCET estimation. In our current implementation, we assume a processor architecture without timing anomalies [6]. However, it is possible to use our persistence analysis framework in presence of timing anomalies. For each Non Classified data reference, we consider both cache hit and miss situations during the WCET analysis, which gives a latency interval for each Non Classified data reference. The resulted cache modeling can be integrated with pipeline analysis as presented in [13] for architectures with timing anomalies.

Table I shows the set of benchmarks used in our evaluation. We have enlarged the array sizes (and corresponding loop bounds) to introduce more data cache conflicts and amplify the effect of data cache performance on overall program execution time. *Array Size* shows the array size used in our simulation and analysis for each of the benchmarks. *Simulation* shows the observed WCET from SimpleScalar simulation in CPU clock cycles. Note that the simulation results may be smaller than the actual WCET values for benchmarks with input-dependent branches/accesses (e.g, Cnt, Bsort100, InsertSort and Adpcm). Finally, we report the WCET results obtained with our scope-aware persistence data cache analysis, as well as the time spent for the analysis (on a Intel(R) Xeon(TM) 2.20 Ghz with 2.5 GB RAM).

We have implemented the revised persistence analysis (Section III-C), multi-level persistence framework [2] (using the revised persistence analysis), and the must analysis with loop unrolling as proposed in [15] to compare with our proposed scope-aware analysis. Figure 9 shows the percentage of overestimation from various data cache analysis approaches, compared to the normalized observed WCET results from SimpleScalar simulation (shown in Table I). Given the array size in our experiment, since the entire array does not fit into the data cache for any of the benchmarks, no memory block can be categorized as persistent in the persistence analysis. Without the temporal scope information, multi-level persistence analysis [2] cannot give tighter estimation, except for the Lms benchmark, where only small arrays are accessed in different loop nesting levels. As a result, the estimated WCET results without temporal scope are up to 83% higher than the observed WCET (for *InsertSort*). We also compare the estimated WCET results using must analysis with 20% and 50% virtual unrolling of the loop nest ([15]), where the analysis is repeatedly performed for each unrolled loop iteration. As shown in

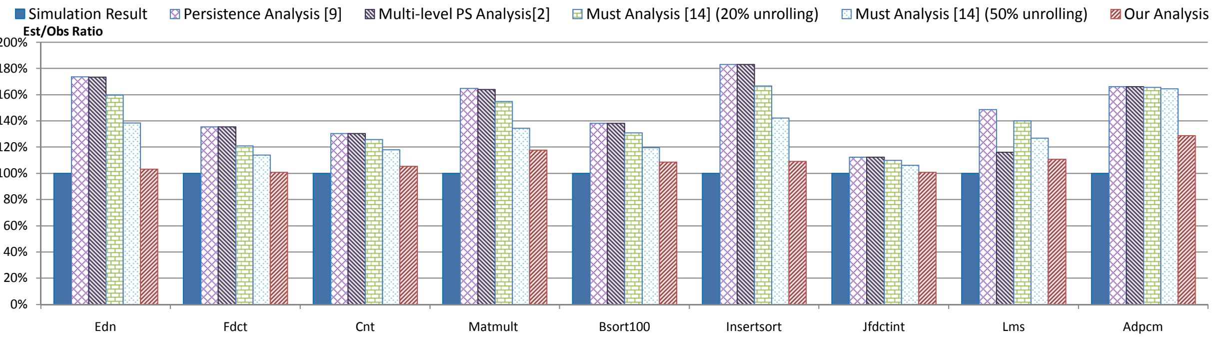


Figure 9. WCET estimation results from different analyses

Figure 9, even when 50% of the loop nest is unrolled, must analysis [15] still reports up to 65% higher WCET estimate compared to the observed simulation time (for *Adpcm*). In particular, must analysis requires loop unrolling to bring memory blocks to the data cache and to capture subsequent cache reuse. As a result, for the remaining portion of the loop nest where unrolling is not applied, they can not capture any cache reuse.

On the other hand, our proposed analysis always obtains tighter WCET estimates compared to existing approaches. In most of the benchmarks, our WCET estimates are less than 10% higher than the simulation results (except for *Matmult* and *Adpcm*). We observe that many data references in these benchmarks have sequential array access patterns. They traverse array elements in sequential order, according to the row-major arrangement of array in the memory. Our scope-aware approach fully captures the temporal locality of such data accesses to bound the worst-case data cache performance. Our proposed analysis achieves 5% to 74% tighter WCET estimates compared to the original persistence analysis without temporal scope information, and 5% to 35% compared to must analysis with 50% unrolling.

Matmult contains a column array access in addition to sequential array accesses. In our analysis, a temporal scope captures the lower and upper bound of loop iterations where a memory block may get accessed. For column array access, array elements contained in a single memory block are usually accessed in non-contiguous loop iterations, which leads to over-estimation in the computed temporal scopes. However, as shown in Figure 9, our estimated WCET is only 25% higher than the observed WCET, and is 10% to 40% tighter than other approaches.

Adpcm is a complex benchmark with input-dependent branches and accesses, so our simulation result may underestimate the real WCET. Due to the presence of input-dependent branches and accesses, must analysis cannot guarantee a memory block to be loaded into the cache for subsequent reuse even with unrolling. In our scope-aware persistence analysis, by guaranteeing the scope persistence of memory blocks, we can achieve 30% tighter WCET estimate compared to must analysis (with 50% loop unrolling).

XII. Related Work

Abstract interpretation methods have been successfully applied to instruction cache analysis for WCET estimation [17], [2]. A globally defined abstract cache state (ACS) is calculated via fixed-point computation, which conservatively captures the worst-case cache behavior at each program point (e.g., basic block boundary). However, existing approaches using abstract interpretation for data cache analysis (e.g., must analysis [15] and persistence analysis [10]) suffer from significant over-estimation. The major source of the over-estimation arises from the fact that the definition and computation of ACS are insensitive to local program behavior. In particular, an array reference may access different memory blocks in different loop iterations, which must be captured in the analysis for a tight estimation. To overcome this problem, Sen and Srikant [15] proposes virtual loop unrolling, which makes the analysis computationally expensive. Moreover, in the presence of input-dependent branches, even with loop unrolling, no memory block can be guaranteed to be loaded to the cache for later reuse by must analysis. Lesage, Hardy and Puaut [12] applies persistence analysis to multi-level data caches.

In many real programs the access pattern of an array follows an uniform affine pattern. The cache miss equation (CME) framework [11] and Presburger Arithmetic formulation [4] have been applied to analyze array access patterns for data cache analysis. The CME framework computes the reuse vector of affine accesses and generates a set of Diophantine equations to characterize whether a reuse can be realized, or interfered with due to cache conflict. The solutions of this equation set are the possible conflict points. White et al. [18] proposes a framework to detect loop-affine array accesses at binary code level. Ramaprasad and Mueller [14] extends the CME framework to analyze scalar accesses and more general loop-nest. The data cache analysis with Presburger Arithmetic framework is exact and can handle certain non-linear access pattern; however, it has super-exponential complexity in the worst case. Furthermore, these approaches *cannot* handle programs with input-dependent branches and unpredictable data accesses. It is also hard to combine such frameworks into a comprehensive WCET analysis considering other micro-architecture features, such as instruction cache [17] or unified cache analysis [5].

Staschulat and Ernst [16] identifies single data sequence (SDS) where both control flow and accessed memory blocks are input independent. In such cases, cache performance can be determined by simple simulation and no analysis is needed. For non-SDS data references, persistence analysis is used to bound the worst-case cache conflicts. Similar to [10], the persistence analysis does not capture array access patterns and leads to very pessimistic analysis results.

XIII. Conclusion

In this technical report, we have presented a novel data cache modeling approach for static WCET analysis. Our analysis effectively exploits regular data access patterns, while retaining the strength and applicability of the abstract interpretation approach. We define temporal scopes to capture the local behavior of memory references (when a particular memory block is accessed). These temporal scopes are automatically calculated during address analysis.

Our scope-aware multi-level data cache analysis extends the cache persistence analysis framework to compute fine-grained scope-based persistence information, which leads to substantially tighter worst-case cache miss estimation. While we have presented our analysis for LRU based cache replacement policy, it can also be extended to handle other deterministic cache replacement policies like FIFO and MRU. In particular, the abstract cache update function has to be changed to cope with the chosen replacement policy. Finally, the proposed analysis has been integrated into the open-source Chronos WCET analyzer ([8] version 4.1).

XIV. Acknowledgements

This work was partially supported by NUS University Research Council grant R252-000-321-112.

References

[1] WCET Benchmarks. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[2] C. Ballabriga and H. Casse. Improving the First-Miss Computation in Set-Associative Instruction Caches. In *ECRTS*, 2008.

[3] D. Burger and T.M. Austin. The SimpleScalar tool set, version 2.0. *ACM SIGARCH*, 25(3), 1997.

[4] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *PLDI*, 2001.

[5] S. Chattopadhyay and A. Roychoudhury. Unified cache modeling for wcet analysis and layout optimizations. In *RTSS*, 2009.

[6] J. Reineke *et al.* A definition and classification of timing anomalies. In *In WCET Workshop*, 2006.

[7] Jan Reineke *et al.* Timing predictability of cache replacement policies. *Real-Time Systems*, 2007.

[8] X. Li *et al.* Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3):56–67, 2007, http://www.comp.nus.edu.sg/~rpembed/chronos.

[9] C. Ferdinand. *Cache behavior prediction for real-time systems*. PhD thesis, Saarland University, 1999.

[10] C. Ferdinand and R. Wilhelm. On predicting data cache behavior for real-time systems. In *LCTES*, 1998.

[11] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[12] B. Lesage, D. Hardy, and I. Puaut. WCET analysis of multi-level set-associative data caches. In *WCET Workshop*, 2009.

[13] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for software timing analysis. In *RTSS*, 2004.

[14] H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *RTAS*, 2005.

[15] R. Sen and Y.N. Srikant. WCET estimation for executables in the presence of data caches. In *EMSOFT*, 2007.

[16] J. Staschulat and R. Ernst. Worst case timing analysis of input dependent data cache behavior. In *ECRTS*, 2006.

[17] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2):157–179, 2000.

[18] R. T. White et al. Timing analysis for data and wrap-around fill caches. *Real-Time System*, 17(2-3):209–233, 1999.