THE NATIONAL UNIVERSITY
of SINGAPORE

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

TR20/10

Safety Proofs of Presistence Analysis

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

Safety Proofs of Persistence Analysis

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 [15]. 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 [10]) 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 ([9], [13]) work on general programs but suffer from large over-estimation. In this report, we first show that the original persistence analysis proposed in [8] and [9] 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 ([9]) 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. 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. Finally, we would like to clarify that our proposed persistence analysis (Section V-C) is "multi-level" in the sense that an independent analysis is performed at each loop nesting level (not to be confused with analysis of multi-level caches).

III. Persistence Analysis by Ferdinand [8]

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. As a result, the first access to a persistent memory block m encounters a cold miss. All subsequent accesses are guaranteed to be cache hits.

To classify a memory block m as persistence, at each *program point* p, the persistence analysis computes the *relative age* x of each memory block m that *may* be accessed at p using an *abstract cache state (ACS)*. If x is not higher than cache associativity A, m is not evicted from the cache in all possible executions at p. Therefore, m is classified as persistence.

An ACS $\hat{c} = \langle \hat{s}_1, ..., \hat{s}_{n/A} \rangle$ at a program point p models an A-way 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 line $l_1, ..., l_A$ 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 in ACS $\hat{c}$ where m is mapped to. If $m \in \hat{s}(l_x)$, m has maximal relative age x at p. If m is in evicted line $\hat{s}(l_\top)$, m could be possibly evicted in some execution paths.

Persistence analysis can be performed on the assembly code level 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 each memory reference to m with incoming ACS $\hat{c}$, the *cache update function* $\hat{\mathcal{U}}_{\hat{\mathcal{C}}}(\hat{c}, m)$ computes resulting ACS after accessing m. If p has multiple incoming ACS $\hat{c}_1, \hat{c}_2$, the *join function* $\hat{\mathcal{J}}_{\hat{\mathcal{C}}}(\hat{c}_1, \hat{c}_2)$ computes the upper bound ACS $\hat{c}$ representing both $\hat{c}_1, \hat{c}_2$. The persistence analysis traverses through the CFG and perform computation until the input ACSs of all nodes reach fixed-point. If a memory block $m \in \hat{s}(l_x)$ where $x <= A$ at ACS $\hat{c}$ of program point p, m is guaranteed to be persistent at p.

Given an accessed memory block m and a concrete set state $s^{in} = c[set(m)]$, the concrete update function $\mathcal{U}_{\mathcal{C}}(c, m)$ computes the resulting ACS after accessing m:

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

The concrete cache update function $\mathcal{U}_{\mathcal{C}}$ models the change in concrete set state $s = c[set(m)]$ when accessing memory block m using concrete set update function $\mathcal{U}_{\mathcal{S}}$

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

Ferdinand and Wilhelm [9] proposes an abstract cache update function $\hat{\mathcal{U}}_{\hat{\mathcal{C}}}(\hat{c}, m)$ to compute the ACS after an access to memory block m as follows

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

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

The abstract set update function $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}(\hat{s}, m)$ computes the new abstract state set $\hat{s} = \hat{c}[set(m)]$ after accessing m. It brings (or renews) the newly accessed memory block m to newest cache line l_1. If $m \notin \hat{s}$, $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}(\hat{s}, m)$ ages all memory blocks m' currently in $\hat{s}$. If $m \in \hat{s}(l_h)$, for each $m' \in \hat{s}(l_k)$, $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}(\hat{s}, m)$ ages m' to $\hat{s}(l_{k+1})$ if m' is younger than m ($k < h$). Otherwise ($k \geq h$), m' remains in $\hat{s}(l_k)$.

If a CFG node n has two incoming edges from $n1$ and $n2$, a join function is used to combine the output ACSs of the two predecessor nodes as the input ACS of n. The new relative age of a memory block m is equal to the maximum age of its existences in the two output ACSs of the predecessor nodes.

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

B. Safety Issue

Figure 1 illustrates an unsafe scenario of the original persistence analysis as proposed by [9]. The CFG in Figure 1(a) with five nodes $B0$,...,$B5$ in a loop. The program accesses memory block a in $B1, B4$, 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 output ACS $\hat{s}^{out}_{B3}$ of $B3$, $\hat{s}^{out}_{B4}$ of $B4$, and the input ACS $\hat{s}^{in}_{B5}$ of $B5$ after the first iteration through the loop. As all a, b, c are in the ACS, no memory block is aged further when the program loops back and accesses a, b, c again, according to update function $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$ described above. 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 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 a, b. So c is not persistent at $B5$, and the persistence analysis in [9] is unsafe.

The unsafe is due to an error of 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. Therefore, 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}. As a result, the original persistence analysis may underestimate the relative age of a and c.

Let $conc_{\hat{\mathcal{C}}}(\hat{c}^{in})$ is 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 can be formulated mathematically as follow:

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

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

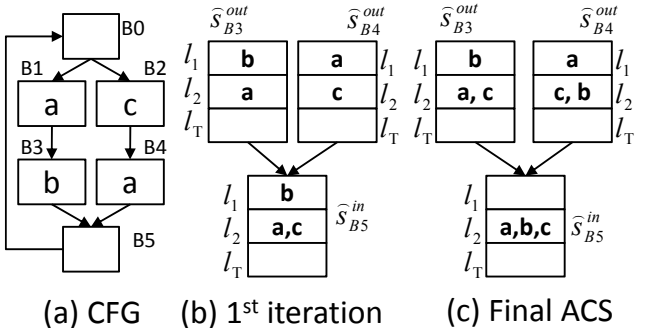


Figure 1. Running example and analysis result of original persistence analysis

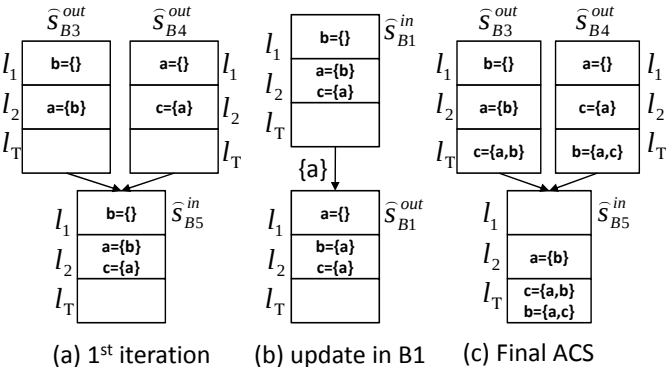


Figure 2. Analysis result of with proposed update and join function

s^{out} and $\hat{s}^{out}$ are as follows

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

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

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 at p for concrete set state s^{out}.

C. Fixing the Persistence Analysis

As demonstrated above, we cannot use relative age of m 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 relative age k of memory block m even when m has older relative age ($k \geq h$). As a result, we propose to keep track of the possible memory blocks that are more recently used (younger) than m in all executions. An access to m_a will increase relative age of m if m_a is not in the

current younger set of m. Otherwise, m_a is younger (more recently used) than m, so it will not age m according to LRU policy. We define the Younger Set ($\mathcal{YS}$) as follow

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

Since m can only be aged by memory blocks $m' \in \mathcal{YS}(\hat{s}, m)$ in any execution, the relative age x of m in $\hat{s}$ should be larger than number of possible younger memory blocks, i.e. $|\mathcal{YS}(\hat{s}, m)| + 1$.

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 b is not a younger memory block of c even though relative age of b is smaller than relative age of c in $\hat{s}^{in}_{B5}$. Therefore, we knows that a subsequent access to b will increase relative age of c and avoid the unsafe of original persistence analysis in [9] (Figure 2(c)). We propose a new update and join function to track and use younger set notion in ACS computation as follow.

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 $\hat{\mathcal{U}}_{\hat{\mathcal{C}}}(\hat{c}^{in}, m_a)$ update the state of $set(m_a)$ 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 input abstract set state $\hat{s}^{in}$ and the accessed memory block m_a, the set 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:

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

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, m_a becomes the youngest memory block. Therefore, $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$ sets the younger set$\mathcal{YS}(\hat{s}^{out}, m_a)$ of m_a to empty and brings (or renews) m_a to $\hat{s}^{out}(l_1)$. For each memory block m in $\hat{s}^{in}$, m_a is a possible "new" younger block. Therefore, $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$ adds m_a to the younger set $\mathcal{YS}(\hat{s}_o, m)$ and sets its relative age accordingly. Since no "new" younger block other than m_a, $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$ always safely estimates all younger blocks of m in $set(m)$.

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 $\mathcal{YS}(\hat{s}^{in}_{B1}, c) = \{a\}$.

The program accesses memory block a in $B1$. Therefore, 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 is a new younger block of b so $\mathcal{YS}(\hat{s}^{out}_{B1}, b) = \{a\}$. As b has a new younger memory block, relative age of b is increased to 2. Because c already has a in its younger set $\mathcal{YS}(\hat{s}^{in}_{B1}, c)$, it keeps the same relative age.

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 for ACS applies the new join function for abstract set state $\mathcal{J}_{\hat{\mathcal{S}}}$ to join each cache set

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

Given two abstract set state $\hat{s}_1$ and $\hat{s_2}$, our new join function computes the joined abstract set state $\hat{s}$ as follows

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

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

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

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

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 $B3$, b has no younger memory block but in $B4$, b has two younger memory blocks a and c, so in $B5$, $\mathcal{YS}(\hat{s}^{in}_{B5}, b) = \{a, c\}$. Similarly, c has younger memory block a in $B4$ and $\{a, b\}$ in $B3$, so $\mathcal{YS}(\hat{s}^{in}_{B5}, c) = \{a, b\}$. a has younger set $\{b\}$ in $B3$ and empty in $B4$, therefore $\mathcal{YS}(\hat{s}^{in}_{B5}, a) = \{b\}$. Our proposed persistence analysis accurately points out that a is never evicted at $B5$. However, b and c have up to two younger memory blocks so they may be evicted.

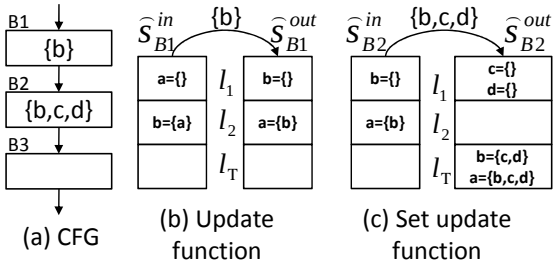


Figure 3. Cache update for set of possible access addresses

To optimize analysis performance, we stop tracking $\mathcal{YS}(\hat{c}[set(m)], m)$ if it has more memory blocks than cache associativity A. 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.

New update function for set: Unlike instruction references, a data reference D can access a set of possible different data addresses. Therefore, cache update function $\hat{\mathcal{U}}_{\hat{\mathcal{C}}}$ need to handle sets of possibly referenced memory blocks, as in [9]. We propose a new set update function with our younger set $\mathcal{YS}$ as follow

$$\hat{\mathcal{U}}_{\hat{\mathcal{C}}}(\hat{c}, \{m_1, .., m_x\}) = \hat{c}[f_i \mapsto \hat{\mathcal{U}}_{\hat{\mathcal{S}}}(\hat{c}[f_i], X_{f_i})]$$
$$\text{for all } f_i \in \{set(m_1)...set(m_x)\}$$
$$\text{where } X_{f_i} = \{m_y | m_y \in \{m_1, .., m_x\}, set(m_y) = f_i\}$$

Given a set of possible access addresses $\{m_1 \dots m_x\}$, the abstract cache update function $\hat{\mathcal{U}}_{\hat{\mathcal{C}}}$ divides it into X_{f_i}, the set of possible access addresses 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 possible accessed addresses mapped to this cache set.

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

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 the program may access any memory block in X_{f_i}, all of them could possibly become younger memory block for each 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 $B2$ may access a set of possible memory block $\{b, c, d\}$ mapped to $\hat{s}^{in}_{B2}$. Figure 3.b shows the input abstract set state $\hat{s}^{in}_{B2}$ and the resulting abstract set state $\hat{s}^{out}_{B2}$ after the memory access. As all $\{b, c, d\}$ 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

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. □

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

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 possible memory blocks more recently used (younger) 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 $\mathcal{YS}(\hat{s}, m)$ contains all possible memory blocks younger than m in concrete set state $s = c[set(m)]$ (the concrete younger set $ys(s, m)$) at p in any execution path that reaches p.

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 abstract set state $\hat{s} = \hat{c}[set(m)]$, the younger set $\mathcal{YS}(\hat{s}, m)$ contains all memory blocks in the concrete younger set $ys(s, m)$, the set of more recently used (younger) than m in the cache set $s = c[set(m)]$ where m is mapped to. □

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

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 ($y = |ys(s, m)| + 1$). 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.

- YS property is trivially true at the start of the execution because the concrete cache state c is empty.
- 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 $\{m_1...m_k\}$), 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 abstract set state $\hat{s}$ at program point p, for each memory block $m \in \hat{s}$, the younger set $\mathcal{YS}(\hat{s}, m)$ contains all possible memory blocks younger than m at p. The maximal relative age of m in $\hat{s}$ is determined by the number of possible younger memory blocks than m, i.e. $|\mathcal{YS}(\hat{s}, m)| + 1$. As a result, if the size of younger set $\mathcal{YS}(\hat{s}, m)$ is less than or equal to cache associativity A, m is persistent at p in any execution.

B. Safety of update function

We prove our update function preserves the YS property. If the program accesses m_a at 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},

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

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}

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

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 the program accesses m_a at p and m mapped to the same cache set with m_a, m_a becomes a new younger memory block of m if $m \neq m_a$. If $m_a = m$, m is brought (or renewed) to l_1 with empty younger set.

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

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 follow

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

As a result, we have

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

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

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

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

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 next 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},

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

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 $\hat{c}_n^{in}$

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

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

$$(\hat{\mathcal{J}}_{\hat{\mathcal{S}}}) \rightarrow \mathcal{YS}(\hat{s}_n^{in}, m) = \mathcal{YS}(\hat{s}^{out}, m) \cup \ldots$$
$$\rightarrow \mathcal{YS}(\hat{s}^{out}, m) \subseteq \mathcal{YS}(\hat{s}_n^{in}, m)$$

Because p_n^{in} is immediately after p^{out}, no new memory block is accessed, so the concrete set state remains the same, $s^{in} = s^{out}$. Therefore, for all memory blocks m:

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

In summary

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

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 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 [9]. We prove our set update function preserves the YS property. If the program accesses $Addr(D) = \{m_1 ... m_k\}$ 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},

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

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}

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

As the program accesses $\{m_1 ... m_k\}$ at p, for each memory block m in s^{in}, let X_{f_i} be the set of memory blocks in $\{m_1 ... m_k\}$ mapped to s^{in}. Any memory block $m_a \in X_{f_i}$ can become a new younger memory block of memory block m in $\hat{s}^{in}$ if $m \neq m_a$. Moreover, if $m_a \notin \hat{s}^{in}$, m_a can become a newly accessed memory block of the cache set. Therefore m_a will be loaded to $\hat{s}^{out}$ as youngest memory block.

$$ys(s^{out}, m) = \qquad \text{(D.3)}$$
$$\begin{cases} ys(s^{in}, m) \cup \{m_a\}, \text{for any } m_a \in X_{f_i} \\ \qquad\qquad \text{if } m \in \hat{s}^{in}, m \neq m_a \\ \\ \emptyset \qquad\qquad \text{Otherwise} \end{cases}$$

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

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

In summary

$$\text{[D.1]} \rightarrow ys(s^{in}, m) \subseteq \mathcal{YS}(\hat{s}^{in}, m)$$
$$\text{[D.3]}, [\hat{\mathcal{U}}_{\hat{\mathcal{S}}}] \rightarrow$$
$$\text{if } m \in \hat{s}_i$$
$$ys(s^{out}, m) = ys(s^{in}, m) \cup \{m_a\},$$
$$\text{for any } m_a \in X_{f_a}, m \neq m_a$$
$$\mathcal{YS}(\hat{s}^{out}, m) = \mathcal{YS}(\hat{s}^{in}, m) \cup X_{f_i} \setminus \{m\}$$
$$\text{[D.1]} \rightarrow ys(s^{out}, m) \subseteq \mathcal{YS}(\hat{s}^{out}, m)$$
$$\text{if } m \notin \hat{s}_i$$
$$ys(s^{out}, m) = \emptyset$$
$$\rightarrow ys(s^{out}, m) \subseteq \mathcal{YS}(\hat{s}^{out}, m)$$

So $\mathcal{YS}(\hat{s}^{out}, m)$ contains all possible memory blocks younger than m in $c^{out}[set(m)]$ at p^{out} after accessing $\{m_1 ... m_k\}$. As a result, the YS property holds at p^{out}, after the execution of step p.

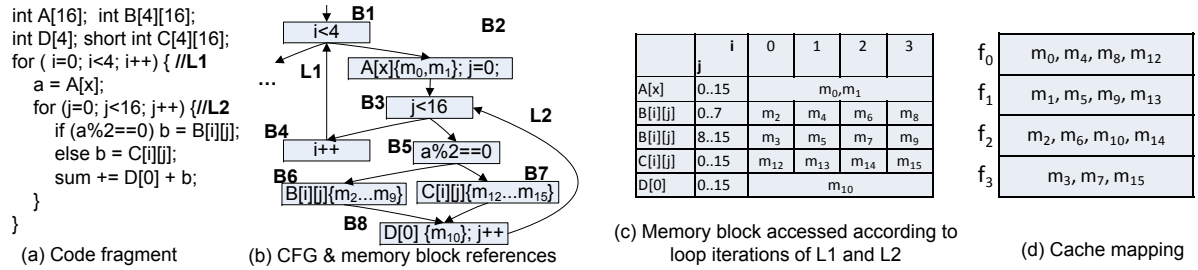
E. Termination of the analysis

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

A. Motivations

Figure 4.a presents our motivating example with four array references in two nested loop $L1$ and $L2$. Assume $A[x]$ always accesses within array A segment $Addr(A) = \{m_0, m_1\}$, the unpredictable data reference $A[x]$ could access either m_0 or m_1 in $Addr(A)$. The array reference $B[i][j]$ and $C[i][j]$ access $Addr(B) = \{m_2 ... m_9\}$ and $Addr(C) = \{m_{12} ... m_{15}\}$ respectively. And $D[i]$ accesses only m_{10}. Figure 4.b shows the CFG and possible memory



	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}			

Figure 4. Motivating example

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 original persistence analysis framework.

However, Figure 4.c describes the access pattern for each data reference in the running example. As $A[x]$ has unpredictable access address, it could access either m_0 or m_1 in any occurrence. However, $B[i][j]$ 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. If m_6 is not evicted in the scope $\{L1 \mapsto [2,2], L2 \mapsto [0,7]\}$, $B[i][j]$ 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 having a flexible persistence scope, we could obtain a much tighter data cache analysis.

B. Temporal scope & 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 V-C.

Definition 4: **(Temporal scope)** A temporal scope for memory block m which is possibly accessed by a data reference D is defined as

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

where $reside(D)$ is the set of loops where D resides in. For each of such loops L_i, temporal scope $TS_m^D[L_i]$ (or $\overline{m}[L_i]$) maintains a mapping between L_i and a closed interval $[lw, up]$ of L_i's iterations where D may access m. To simplify the presentation, we use $\overline{m}$ to denote TS_m^D when there is no ambiguity of the access context. □

For a data reference D, address analysis calculates set of memory blocks possibly accessed by D. We follow the register expansion framework in [16] 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 [16] 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. The same memory block is accessed in any loop iteration, so that its temporal scope covers all iterations. In Figure 5(a), address expression of $D[0]$ is evaluated to $BaseD$, which corresponds to m_{10}. So the temporal scope $\overline{m}_{10}$ is $\{L1 \mapsto [0,3], L2 \mapsto [0,15]\}$.
- 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 $L1$. So temporal scope $\overline{m}_0 = \{L1 \mapsto [0,3]\}$.
- 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, we obtain that temporal scope $\overline{m}_6 = \{L1 \mapsto [2,2], L2 \mapsto [0,7]\}$.

Given the access intervals defined by our temporal scopes, two memory blocks m_i and m_j conflict within a single complete execution of loop L (between entry and exit of L) only if they are mapped to the same cache set and their access intervals overlap during execution of L. The

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

(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

scope overlapping between two temporal scopes over L is recursively defined as

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

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 L and all outer loops containing L are *not* mutually exclusive.

In Figure 5, 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 $L1$, $\overline{m}_{15}[L1]$ and $\overline{m}_6[L1]$ are separated intervals. Therefore, the temporal scope $\overline{m}_{15}$ and $\overline{m}_6$ do not overlap because they belong to different iterations of the outer loop $L1$.

To reduce the pessimism mentioned above, we integrate access pattern analysis into the abstract interpretation framework for accurate WCET analysis. We extend the definition of memory block persistence in [9]. In our analysis, we capture memory block persistence at different loop-nest levels of the program execution, 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 VI.

C. Scope-aware Persistence Analysis

In [9], accesses to m are categorized to be *Non Classified* and result to all cache misses. Furthermore, [9] considers two memory blocks to be conflict with each other if they are mapped to the same cache set. However, two memory blocks accessed in disjoint program execution fragments (e.g., loop iterations) do *not* affect the persistence of each other.

The basic idea of our scope-aware persistence analysis is to categorize the persistence of memory blocks in the calculated temporal scopes (Section V-B), instead of the globally defined persistence in [9]. For a data reference D, the temporal scope TS^D_m identifies a set of loops (where D resides in) and a loop iteration interval for each of the loops 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 5: **(Scope persistence)** Let $Iter$ be the loop iterations bounded by $[TS^D_m[L].lw, TS^D_m[L].up]$ where data reference D may access memory block m during an execution of loop L (between L's entry and exit). The temporal scope TS^D_m is persistent at loop level L if and only if within iterations $Iter$ of L, m is guaranteed to remain in the cache after the first time it is loaded into cache by D. □

Given above definition of scope persistence, for a memory block m possibly referenced by data access D to be persistent within loop L, it does not need to stay in the cache for all iterations of L. If m is not evicted out from cache during the iteration interval defined by $[TS^D_m[L].lw, TS^D_m[L].up]$, all accesses to m from D cause at most *one* cache miss (the cold miss) within one 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 6: **(Scope-aware abstract set state)** An abstract set state $\hat{s}$: $\{l_1 \ldots l_A\} \cup \{l_\top\} \rightarrow 2^{TS}$ maps cache lines (including the specially introduced evicted line $l_\top$) to set of all temporal scopes TS. $\hat{S}$ denotes the set of all abstract set states. □

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. □

We have re-design the *update* function 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.

D. Overall Framework

We adopt the multi-level persistence framework from [2] for instruction cache analysis, 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 $\hat{c}^{in}_{L_{entry}}[L] = \perp$ as input ACS of the L's entry node L_{entry}. Consequently, the analysis will consider only paths and data accesses within 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 V-F.

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) and inserts

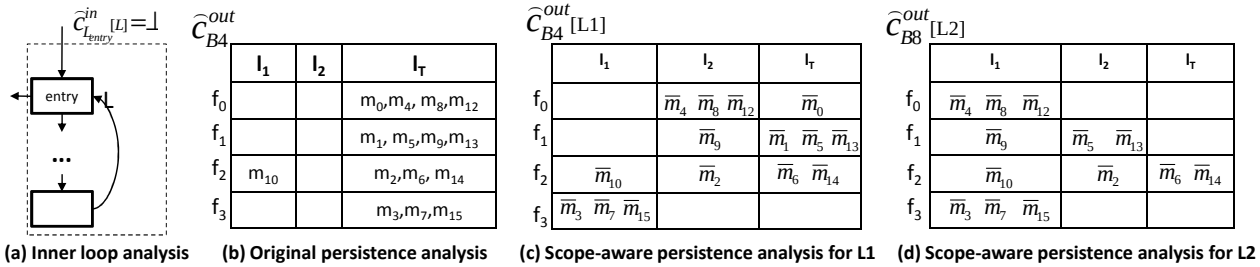


Figure 6. Multi-level analysis and results for the motivating example in Figure 4

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**

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

E. Scope-aware Update and Join Functions

At loop level L, given a data reference D which accesses a set of possible address $Addr(D) = \{m_1...m_k\}$. For each $m_a \in Addr(D)$, we compute the temporal scope $TS^D_{m_a}$ (or $\overline{m}_a$ for short) where D may access m_a. The access to m_a in scope $\overline{m}_a[L]$ does not age a memory block m in scope $\overline{m}[L]$ if their scopes do not overlap (refer to Equation 1 in Section V-B). Therefore, to avoid considering cache conflict outside the scope, the scope-aware update function only adds $\overline{m}_a$ to the younger set of $\overline{m}$ (as in Definition 3) when they overlap.

The scope-aware update function at loop level L for a given input ACS and set of temporal scopes accessed by D can be defined as

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

Given the set of temporal scopes $\{\overline{m}_1...\overline{m}_k\}$, the scope-aware update function divides it into X_{f_i}, the set of accessed temporal scopes for each cache set f_i. For each input abstract set state $\hat{s}^{in}$, the set update function computes the output abstract set state $\hat{s}^{out}$, via updating the Younger Set and the maximal relative age of each temporal scope $\overline{m} \in (\hat{s}^{in} \cup X_{f_i})$ 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 $\forall \overline{m} \in \hat{s}^{in} \cup X_{f_i}, \mathcal{YS}(\hat{s}^{out}, \overline{m}) =$

$$\begin{cases} \emptyset & \text{if } \overline{m} \notin \hat{s}^{in} \\ \emptyset & \text{else if } OpS(\overline{m}, X_{f_i} \setminus \overline{m}, L) = \emptyset \wedge \overline{m} \in X_{f_i} \\ \mathcal{YS}(\hat{s}^{in}, \overline{m}) \cup (OpS(\overline{m}, X_{f_i}, L) \setminus \overline{m}) & \\ & \text{Otherwise.} \end{cases}$$

where $OpS(\overline{m}, X_{f_i}, L)$ denotes the set of temporal scopes in X_{f_i} that overlaps with m in loop level L, which can be calculated as

$$OpS(\overline{m}, X_{f_i}, L) = \{\overline{m}' | \overline{m}' \in X_{f_i} \wedge overlap(\overline{m}, \overline{m}', L)\}$$

In our set update function, the maximal relative age of a memory block in the output abstract set state is set to be larger than the number of all possible younger memory blocks of it, i.e., $|\mathcal{YS}(\hat{s}^{out}, \overline{m})| + 1$. To find the younger set $\mathcal{YS}(\hat{s}^{out}, \overline{m})$, we have the following situations.

- If temporal scope $\overline{m}$ is not in $\hat{s}^{in}$, and m is newly accessed in X_{f_i}, $\overline{m}$ has no younger memory block and its maximal relative age is set to be 1.
- Assume $\overline{m}$ is in $\hat{s}^{in}$ and it is also accessed in X_{f_i}. If there is no other temporal scope in X_{f_i} overlaps with $\overline{m}$, then the data reference D accesses only m in the temporal scope defined by $\overline{m}$. As a result, data reference D renews the relative age of $\overline{m}$ in $\hat{s}^{in}$, and we can set its younger set to be empty.

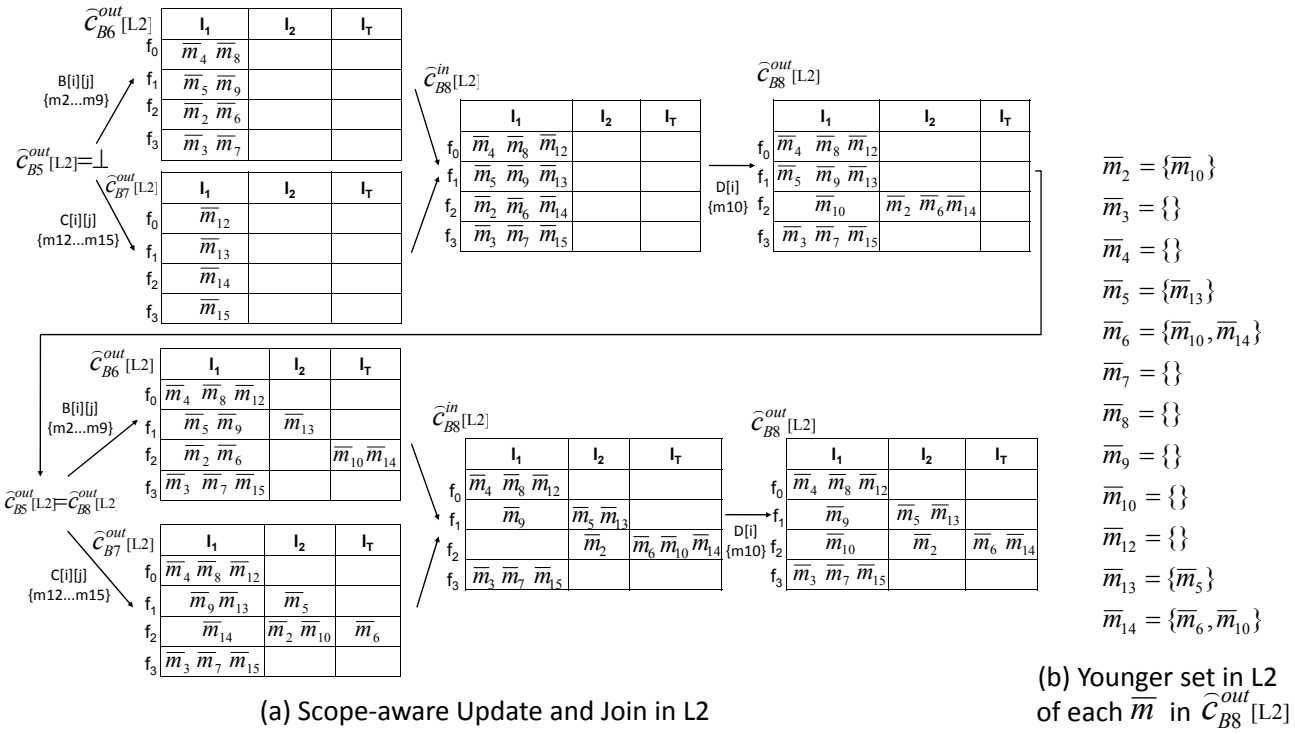


Figure 7. Scope-aware ACS computation for L2 of the motivating example in Figure 4

- Otherwise, the update function adds all possible temporal scopes in X_{f_i} that overlap with $\overline{m}$ to the younger set 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 scopes $\overline{m}_4$, $\overline{m}_8$, and $\overline{m}_{12}$ do not overlap. As a result, they do not age each other. On the other hand, in cache set f_1, as shown in Figure 4, $B[i][j]$ 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 $\overline{m}_{13}$. 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 $\overline{m}_{13}$ to the younger set of $\overline{m}_5$, as shown in Figure 7(b).

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 $\hat{c}_2[L]$ as follows. For each temporal scope $\overline{m}$ in $\hat{c}$, the scope-aware join function unionizes the younger set of m in both output ACSs from the control flow predecessors to form the younger set of 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. Formally, our scope-aware join function is defined as follows.

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

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

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

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

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

F. 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.

VI. Safety proof of the 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 concrete set state $s = c[set(m)]$ is 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 as defined by the temporal scope $\overline{m}$) 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. finish iteration $\overline{m}[L].up$ of loop L). In our ACS semantic, the maximum relative age of memory block m in the scope $\overline{m}[L]$ is the relative age of temporal scope $\overline{m}$ in ACS $\hat{c}[L]$. Given a program point p in loop L with the ACS $\hat{c}[L]$ and the abstract set state $\hat{s} = \hat{c}[L][set(m)]$ where memory block m is mapped to, our scope-aware persistence analysis computes the maximum relative age x of memory block m in scope $\overline{m}[L]$ by tracking the younger set $\mathcal{YS}(\hat{s}, \overline{m})$, the set of temporal scopes of all possible memory blocks which are younger than m and are accessed within the scope $\overline{m}[L]$. 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 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 the maximum relative age x of temporal scope $\overline{m}$ computed by our scope-aware persistence analysis is smaller or equal to the cache associativity A, memory block m will not be evicted out of the cache in scope $\overline{m}[L]$ once it is loaded to the cache, hence m 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 $\overline{m}[L]$, *if* path pa has accessed memory block m in this scope, the younger set $\mathcal{YS}(\hat{s}, \overline{m})$ contain all possible memory blocks younger than m in cache set $s = c[set(m)]$, where c is the concrete cache state of path pa at program point p. 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 of memory block m in concrete cache set s. 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 8: **(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 $m \in c$ 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 $\mathcal{YS}(\hat{s}, \overline{m})$ contains all possible temporal scopes of memory blocks in the cache set $s = c[set(m)]$ that are more recently used (younger) than m in s.□

$$\forall m \in c, s = c[set(m)], \hat{s} = \hat{c}[L][set(m)] \\ m \text{ has been accessed in scope } \overline{m}[L] \\ \rightarrow \forall m', m' \in ys(s, m) \rightarrow \overline{m}' \in \mathcal{YS}(\hat{s}, \overline{m})$$

Where concrete younger set $ys(s, m)$ is the set of memory blocks younger than m in cache set s (Definition 2).

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 $\overline{m}[L]$, 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 the ScopeYS property is trivially true.
- 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 VI-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 VI-C).

For each memory block m in scope $\overline{m}[L]$, as ScopeYS property is true at the first time the program accesses m in scope $\overline{m}[L]$, before and after each program point after that,

and from one program point to another. Therefore, ScopeYS property holds for all possible executions in the scope $\overline{m}[L]$ and the younger set $\mathcal{YS}(\hat{s}, \overline{m})$ contains all temporal scopes of memory blocks which are more recently used (younger) than m in concrete set state s of execution path pa in scope $\overline{m}[L]$. Consequently, the maximum relative age x of memory block m in scope $\overline{m}[L]$ determined by our ACS $\hat{c}[L]$ is always greater than or equal to the relative age of m in s. As a result, our analysis is safe.

B. Safety proof of scope-aware update function

We prove our scope-aware update function preserves the ScopeYS property. If the program accesses $\{m_1 ... m_k\}$ in their respective temporal scopes $\{\overline{m}_1 ... \overline{m}_k\}$ at program point p, 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

$$\begin{aligned}&\forall m \in c^{in}, s^{in} = c^{in}[set(m)], \hat{s}^{in} = \hat{c}^{in}[set(m)],\\ &\quad m \text{ has been accessed in scope } \overline{m}[L]\\ &\quad \rightarrow \forall m', m' \in ys(s^{in}, m) \rightarrow \overline{m}' \in \mathcal{YS}(\hat{s}^{in}, \overline{m}) \quad \text{[B.1]}\end{aligned}$$

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}:

$$\begin{aligned}&\forall m \in c^{out}, s^{out} = c^{out}[set(m)], \hat{s}^{out} = \hat{c}^{out}[L][set(m)],\\ &\quad m \text{ has been accessed in scope } \overline{m}[L]\\ &\quad \rightarrow \forall m', m' \in ys(s^{out}, m) \rightarrow \overline{m}' \in \mathcal{YS}(\hat{s}^{out}, \overline{m}) \quad \text{[B.2]}\end{aligned}$$

At program point p in loop L, a data reference D accesses $X_{f_i} = \{\overline{m}_1 ... \overline{m}_k\}$, a set of temporal scopes of all possible memory blocks mapped to cache set f_i. 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:

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

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

$$\begin{cases} \emptyset & \text{if } \overline{m} \notin \hat{s}^{in}\\ \emptyset & \text{else if } OpS(\overline{m}, X_{f_i} \setminus \overline{m}, L) = \emptyset \wedge \overline{m} \in X_{f_i}\\ \mathcal{YS}(\hat{s}^{in}, \overline{m}) \cup (OpS(\overline{m}, X_{f_i}, L) \setminus \overline{m}) & \\ \qquad \text{Otherwise.} & \end{cases}$$

where $OpS(\overline{m}, X_{f_i}, L)$ denotes the set of temporal scopes in X_{f_i} that overlaps with m in loop level L, which can be calculated as

$$OpS(\overline{m}, X_{f_i}, L) = \{\overline{m}' | \overline{m}' \in X_{f_i} \wedge overlap(\overline{m}, \overline{m}', L)\}$$

From iteration $\overline{m}[L].lw$ to iteration $\overline{m}[L].up$ of loop L, for each $\overline{m}_a \in X_{f_i}$, data reference D will only access m_a in scope $\overline{m}[L]$ if temporal scope $\overline{m}_a$ and $\overline{m}$ overlaps in loop L ($overlap(\overline{m}, \overline{m}_a, L)$). Therefore, in scope $\overline{m}[L]$, D can only access m_a when $\overline{m}_a \in OpS(\overline{m}, X_{f_i}, L)$.

Case 1: At program point p, data reference D does not access m and m is not yet accessed in scope $\overline{m}[L]$

The ScopeYS property is trivially true at p^{out} because we do not consider the relative age of memory block m before its first access in scope $\overline{m}[L]$. We assume the first access to m always result in cache miss.

Case 2: At program point p, data reference D accesses memory block m in scope $\overline{m}$ (if $\overline{m} \in X_{f_i}$)

$$\begin{aligned}&ys(s^{out}, m) = \emptyset \qquad (D \text{ accesses } m)\\ &\rightarrow \forall m', m' \in ys(s^{out}, m) \rightarrow \overline{m}' \in \mathcal{YS}(\hat{s}^{out}, \overline{m}) \quad \text{[B.2]}\end{aligned}$$

Since data reference D accesses memory block m at program point p, m is brought (or renewed) to the youngest cache line $s^{out}(l_1)$. Therefore, $ys(s^{out}, m) = \emptyset$, so ScopeYS property is true regardless of younger set $\mathcal{YS}(\hat{s}^{out}, \overline{m})$.

Case 3: At program point p, data reference D accesses memory block m_a, $m_a \neq m$, $set(m_a) = set(m)$, and m is already accessed in scope $\overline{m}[L]$

D may access memory block m_a in scope $\overline{m}[L]$ only if $\overline{m}_a \in X_{f_i}$ and the temporal scope $\overline{m}_a$ overlaps temporal scope $\overline{m}$ in loop L ($\overline{m}_a \in OpS(\overline{m}, X_{f_i} \setminus \overline{m}, L)$).

$$\begin{aligned}&[1] \quad ys(s^{out}, m) = ys(s^{in}, m) \cup \{m_a\}\\ &[2] \quad \overline{m}_a \in OpS(\overline{m}, X_{f_i} \setminus \overline{m}, L)\\ &\qquad\qquad \rightarrow OpS(\overline{m}, X_{f_i} \setminus \overline{m}, L) \neq \emptyset\\ &[B.1] \rightarrow \begin{cases} \forall m', m' \in ys(s^{in}, m) \rightarrow \overline{m}' \in \mathcal{YS}(\hat{s}^{in}, \overline{m})\\ \overline{m} \in \hat{s}^{in} \end{cases}\\ &[3] \quad \text{Since } OpS(\overline{m}, X_{f_i} \setminus \overline{m}, L) \neq \emptyset \wedge \overline{m} \in \hat{s}^{in}\\ &(\hat{\mathcal{U}}_{\hat{\mathcal{S}}}) \rightarrow \mathcal{YS}(\hat{s}^{out}, \overline{m}) = \mathcal{YS}(\hat{s}^{in}, \overline{m}) \cup OpS(\overline{m}, X_{f_i} \setminus \overline{m}, L)\\ &[B.1][1][2][3] \rightarrow\\ &\qquad \forall m', m' \in ys(s^{out}, m) \rightarrow \overline{m}' \in \mathcal{YS}(\hat{s}^{out}, \overline{m}) \quad \text{[B.2]}\end{aligned}$$

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

$$\forall m \in c^{out}, s^{out} = c^{out}[set(m)], \hat{s}^{out} = \hat{c}^{out}[L][set(m)],$$
$$m \text{ has been accessed in scope } \overline{m}[L]$$
$$\rightarrow \forall m', m' \in ys(s^{out}, m) \rightarrow \overline{m}' \in \mathcal{YS}(\hat{s}^{out}, \overline{m}) \quad \text{[C.1]}$$

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}:

$$\forall m \in c_n^{in}[L], s_n^{in} = c_n^{in}[set(m)], \hat{s}_n^{in} = \hat{c}_n^{in}[L][set(m)],$$
$$m \text{ has been accessed in scope } \overline{m}[L]$$
$$\rightarrow \forall m', m' \in ys(s_n^{in}, m) \rightarrow \overline{m}' \in \mathcal{YS}(\hat{s}_n^{in}, \overline{m}) \quad \text{[C.2]}$$

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 will not be 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

$$\text{[C.1]} \quad \forall m' \in ys(s^{out}, m), \overline{m}' \in \mathcal{YS}(\hat{s}^{out}, \overline{m})$$
$$[\hat{\mathcal{J}}_{\hat{\mathcal{S}}}] \quad \mathcal{YS}(\hat{s}^{out}, \overline{m}) \subseteq \mathcal{YS}(\hat{s}_n^{in}, \overline{m})$$
$$\text{[C.3]} \quad ys(s_n^{in}, m) = ys(s^{out}, m)$$
$$\rightarrow \forall m', m' \in ys(s_n^{in}, m) \rightarrow \overline{m}' \in \mathcal{YS}(\hat{s}_n^{in}, \overline{m}) \qquad \text{[C.2]}$$

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 VI-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.

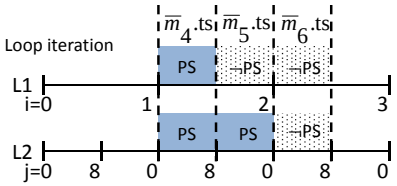


Figure 8. Temporal scopes and loop iterations

VII. 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.

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 $TS_m^D[L]$) 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.

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

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 $L2$. 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 is the summation of $blockMiss(D,m)$ over all memory blocks D may access ($AddrSet(D)$)

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

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.

VIII. 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 to support write-through with no-write-allocate policy and no write buffer in the simulation, to be 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 ([7]), 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 integrate our persistence analysis framework in presence of timing anomalies, that is, for each Non Classified data reference in the analysis result, we consider both cache hit and miss situations during the WCET analysis.

Table I shows the set of benchmarks used in our evaluation. We have enlarged 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. However, the simulation results may be smaller 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 processor with 2.5 GB of RAM).

We have implemented the must analysis with loop unrolling as proposed in [13], and the revised persistence analysis (Section III-C) 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 original persistence analysis of [9]. As a result, the estimated WCET results with original persistence analysis 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 ([13]), where the analysis is repeatedly performed for each unrolled loop iteration. As shown in Figure 9, even when 50% the loop nest is unrolled, [13] 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 scope-aware 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

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	2,631,312	0.25s
Fdct	Fast Discrete Cosine Transform.	2048	917,636	970,646	0.82s
Cnt	Counts non-negative numbers in a matrix.	32×32	21,611	22,826	0.02s
Matmult	Matrix multiplication.	24×24	374,887	467,116	0.02s
Bsort100	Bubblesort program.	1024	15,945,200	16,556,926	0.02s
InsertSort	Insertion sort on a reversed array.	1024	14,900,732	16,298,086	0.58s
Jfdctint	Discrete-cosine transformation of pixel blocks.	256×64	1,485,075	1,499,938	2.05s
Lms	LMS adaptive signal enhancement.	1024	1,425,585	1,527,952	0.02s
Adpcm	Adaptive pulse code modulation algorithm.	2048	193,525	278,495	0.03s

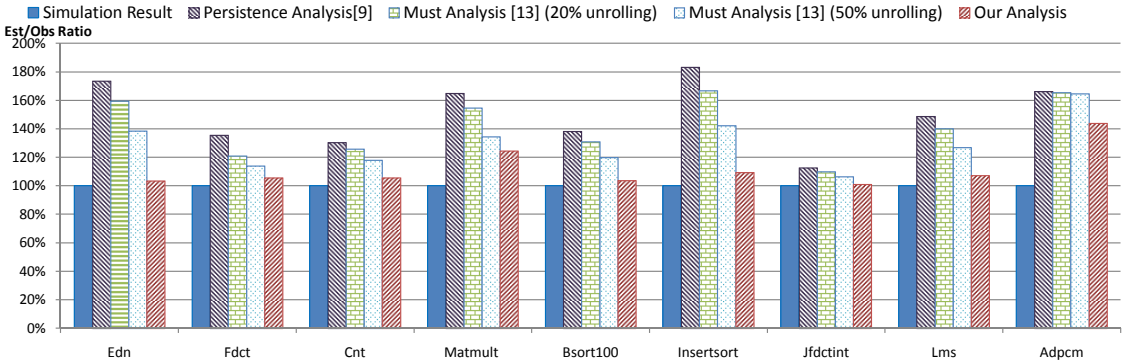


Figure 9. WCET estimation results from different analyses

(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 scope-aware persistence analysis achieves 12% to 74% tighter WCET estimates compared to original persistence analysis, 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 20% tighter WCET estimate compared to must analysis (with 50% loop unrolling).

IX. RELATED WORK

Abstract interpretation methods have been successfully applied to instruction cache analysis for WCET estimation [15], [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 [13] and persistence analysis [9]) 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, [13] 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. In [11], persistence analysis is applied 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 [10] 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. [16] proposes a framework to detect loop-affine array accesses at binary code level. [12] 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 [15] or unified cache analysis [5].

[14] 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 [9], the persistence analysis does not capture array access patterns and leads to very pessimistic analysis results.

X. Conclusion

In this technical report, we have revised and corrected the persistence analysis as proposed in [9], and 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 proposed scope-aware multi-level data cache analysis extends the cache persistence analysis framework to compute fine-grained scope-based persistence information to tightly capture the worst case performance of data cache.

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 Computer Architecture News*, 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] 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.

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

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

[10] 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.

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

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

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

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

[15] 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.

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