Scope-aware Data Cache Analysis for WCET Estimation

Bach Khoa Huynh
National University of Singapore
huynhbac@comp.nus.edu.sg

Lei Ju
Shandong University
julei@sdu.edu.cn (Corresponding author)

Abhik Roychoudhury
National University of Singapore
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, presence of caches, especially data caches, complicates the static worst case execution time (WCET) analysis. Access pattern analysis (e.g., cache miss equations) are applicable to only a specific class of programs, where all array accesses must have predictable access patterns. Abstract interpretation-based methods (must/persistence analysis) determines possible cache conflicts based on coarse-grained memory access information from address analysis, which usually leads to significantly pessimistic estimation. In this paper, we first present a refined persistence analysis method which fixes the potential underestimation problem in the original persistence analysis. Based on our new persistence analysis, we propose a framework to combine access pattern analysis and abstract interpretation for accurate data cache analysis. We capture the dynamic behavior of a memory access by computing 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. Experimental results shows that our proposed analysis obtains up to 74% reduction in the WCET estimates compared to existing data cache analysis.**

I. Introduction

Worst-case Execution Time (WCET) is a key metric for real-time embedded software. 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. However, 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 [18]. On the other hand, 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], [16]) work on general programs but suffer from large over-estimation. In this paper, we seek to combine the strengths of these two approaches. We observe that the over-estimation in existing abstract interpretation-based data cache analysis 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. Furthermore, 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.

In this work, we propose a general and accurate static data cache analysis method by combining access pattern analysis and abstract interpretation. For abstract cache state computation, we extend the cache behavior categorization of "persistence" as in the persistence analysis of [9] to capture the access pattern information. In our new persistence analysis framework, we also fix an error in the original persistence analysis which may result in underestimation of the cache misses. Our contributions include the following.

First of all, given a data reference D and its access pattern, we derive not only the set of possible accessed memory blocks, but also their *temporal scopes*. The temporal scope of a memory block m captures the loop iterations in the program where m may get accessed. Our proposed data cache analysis decides whether a memory block is persistent within its temporal scope. In particular, two memory blocks accessed in mutually exclusive temporal scopes do not conflict with each other within their scopes, even though they are mapped to the same cache set.

Secondly, we also consider the static scopes in our analysis. Similar to the multi-level analysis proposed in [2] for instruction cache, we maintain a copy of abstract data cache states for each loop nesting level of the program execution. As a result, certain memory blocks can be classified as persistent within a local scope of program execution (though it can not be guaranteed to be persistent globally).

Thirdly, we utilize scope-aware persistence while computing the number of data cache misses. In original persistence analysis, a data reference is classified as globally persistent throughout the program execution. However, our persistence analysis framework can guarantee that a data reference is persistent within certain temporal and static scopes.

Last but not the least, we have integrated our proposed framework into the open-source Chronos WCET analyzer ([13]). The experimental results show that our proposed scope-aware persistence analysis produces up to 74% tighter WCET estimation comparing to multi-level persistence analysis framework without temporal scope information [2].

II. Related Work

Abstract interpretation methods have been successfully applied to instruction cache analysis for WCET estimation [18], [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 [16] 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, Sen and Srikant [16] proposes virtual loop unrolling, which makes the analysis computationally expensive. Moreover, in the presence of input-dependent branches, even with loop unrolling, no memory block can be guaranteed to be loaded to the cache for later reuse by must analysis. Lesage, Hardy and Puaut [12] applies persistence analysis to multi-level data caches.

In many real programs the access pattern of an array follows an uniform affine pattern. The cache miss equation (CME) framework [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. White et al. [19] proposes a framework to detect loop-affine array accesses at binary code level. Ramaprasad and Mueller [15] 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 [18] or unified cache analysis [5].

Staschulat and Ernst [17] 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.

III. Notations and Assumptions

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

- Capacity C: size of the cache in number of bytes
- Block (line) size B: number of contiguous bytes to be loaded from memory to cache on each memory access.
- Associativity A: A-way set associative cache means that information stored at some addresses in memory could be loaded into any of A locations in the cache (depends on the cache replacement policy).
- Cache set $F = \langle f_1, \ldots, f_{(C/B)/A} \rangle$: A cache set f_i is a sequence of cache blocks (lines) $CL = \langle l_1, \ldots, l_A \rangle$ which contains all the A ways that can be addressed with the same index. $set(m)$ returns the cache set memory block m maps to.

We assume LRU (Least Recently Used) replacement policy is used to determine relative age of a memory block in the A-way associative cache set. Among common cache replacement policies, LRU is the most predictable policy thus more suitable for safety critical real-time systems [6]. Given a concrete cache state c at a program point p, the concrete set state s_i describes the state of cache set $c[f_i]$ at p. If $s_i(l_x) = m$, memory block m has a relative age x in $c[f_i]$ ($1 \leq x \leq A$).

We assume write-through with no-write-allocate policy for a memory store instruction in our discussion of data cache analysis. However, our data cache analysis framework is applicable to different write policies with minor amendments in the analysis (discussed in Section VI-B). We consider the static and temporal scope information of data references at the assembly code level in our analysis. Finally, we would like to clarify that our proposed persistence analysis (Section VI) is "multi-level" in the sense that an independent analysis is performed at each loop nesting level (also referred as the

static scope), which should not be confused with analysis of the multi-level caches (e.g., the L2, L3 caches).

IV. Persistence Analysis in [8] and [9]

In this section, we briefly discuss the safety issue and pessimism of the the original persistence analysis in [8], [9]. The detailed description and proofs on how to fix the underestimation error in the original persistence analysis can be found in the technical report [11].

A. Overview

In persistence analysis, a memory block m is guaranteed to be persistent if no other memory references can evict m from the cache during program execution. Therefore, m incurs one cache miss when it is first accessed, and all future accesses to it are guaranteed cache hits. In comparison with reuse-based approaches such as CME [10] and must analysis [16], persistence analysis does not require first bringing memory blocks to the cache for subsequent reuse. Hence, it does not require a detailed access sequence analysis. Moreover, it can guarantee cache hit in the presence of input-dependent branches and unpredictable access addresses.

Persistence analysis is based on a fixed-point computation of the *abstract cache state (ACS)* $\hat{c} = \langle \hat{s}_1, ..., \hat{s}_{n/A} \rangle$ for each program execution point, where $\hat{s}_i$ is the *abstract set states* for cache set f_i. In the LRU replacement policy, the abstract set state captures the upper bound of the positions (the relative ages) of the memory blocks that can possibly reside in the corresponding concrete cache set. An *abstract line state* $\hat{s}_i(l_a)$ contains memory blocks that have maximal relative age of a in the abstract set state $\hat{s}_i$, where $1 \leq a \leq A$. For example, $\hat{s}_i(l_2) = \{m_a\}$ denotes that memory blocks m_a could reside in cache set f_i with maximal relative age of 2 at a program execution point. Furthermore, an additional abstract line state $\hat{s}_i(l_\top)$ is introduced to each abstract set state to keep track of memory blocks that have been referenced before but evicted out from the cache by other later memory references.

The analysis traverses the program's control flow graph (CFG) and manipulate the ACSs via *update* and *join* functions. The *update* function takes an input ACS $\hat{c}^{in}$ and a set of memory blocks M possibly accessed at the current program location, and produces an output ACS $\hat{c}^{out}$ which captures the worst-case cache behavior (maximal relative ages if LRU is used) due to the accesses in M. If a program point has more than one incoming edges in the CFG, the *join* function is applied to compute the input ACS of this point by combining the output ACSs of all its predecessors. In the original persistence analysis, the relative age of a memory block in the joined ACS is set to be the maximum relative age of all its occurrences in the predecessor's ACSs.

B. Safety Issue

We first discuss and fix the safety issue for the original persistence analysis in section IV-B and IV-C. Figure 1

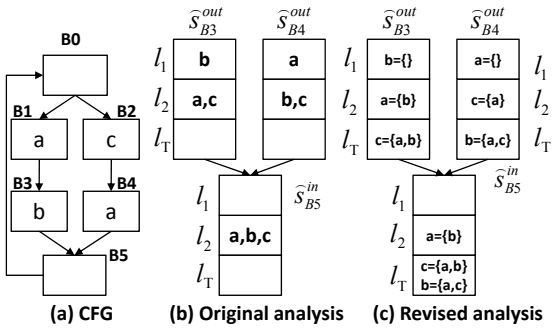


Figure 1. Underestimation in the original persistence analysis

illustrates an unsafe scenario of the original persistence analysis. The CFG in Figure 1(a) has six basic blocks $B0...B5$ within a loop, where accessed memory blocks mapped to the same cache set are shown. For example, memory block a is accessed in $B1$ and $B4$. Given cache associativity $A = 2$, Figure 1(b) gives the computed ACSs at fixed-point by original persistence analysis. Since relative ages of memory block c are both 2 in $\hat{s}^{out}_{B3}$ and $\hat{s}^{out}_{B4}$, relative age of c in $\hat{s}^{in}_{B5}$ is also 2 according to the original join function. The resulted ACSs indicates that memory block c is persistent in all possible program executions. However, in a concrete path $B0 \rightarrow B2 \rightarrow B4 \rightarrow B5 \rightarrow B0 \rightarrow B1 \rightarrow B3$, c is evicted by a and b. Hence, c is not persistent and the original analysis is unsafe.

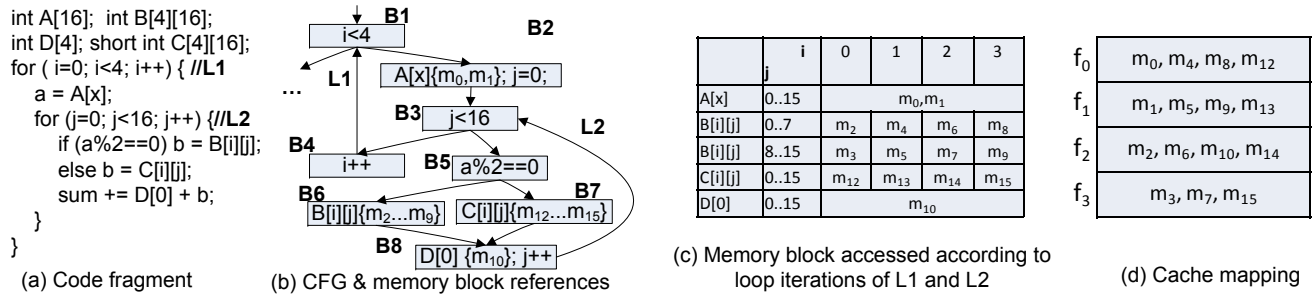
The possible underestimation of relative ages in the original analysis is due to an error in the update function. It wrongly assumes that if memory block b is in $\hat{s}^{in}_{B5}$, b is also in *all* concrete set states s^{in}_{B5}. Therefore, the update function will not age memory blocks with maximum relative age equal or older than b such as a and c in the ACS. However, when b is in $\hat{s}^{in}_{B5}$, there may exist concrete set states that do not contain b (e.g. only a and c are in the concrete cache state corresponds to the path $B0 \rightarrow B2 \rightarrow B4$). In these concrete set states, access of b will increase the relative ages of a and c. Therefore, the original persistence analysis may underestimate the relative ages of a and c. A more detailed discussion is presented in the technical report [11].

C. Fixing the Persistence Analysis

To fix the underestimation in the original persistence analysis, we propose to keep track of the memory blocks that *may* be younger (more recently accessed) than m for each memory block m during the analysis. We define the Younger Set ($\mathcal{YS}$) as follows.

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

In our revised persistence analysis, we maintain $\mathcal{YS}$ for all memory blocks at each program execution point. For an access of memory block m and corresponding abstract



	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}			

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

Figure 2. Motivating example

set state $\hat{s}$, the abstract update function computes both the relative ages and $\mathcal{YS}$. In particular, m should be included in the $\mathcal{YS}$ for each memory block m' in $\hat{s}$. Since m' can only be aged by memory blocks in $\mathcal{YS}(\hat{s}, m')$ in any possible execution, the new maximum relative age of m' due to access of m should be larger than the number of all possible younger memory blocks, i.e., $|\mathcal{YS}(\hat{s}, m')|+1$. Similarly, the revised join function also merges the $\mathcal{YS}$ for each memory block in the input ACSs, and computes its new relative age accordingly. Figure 1(c) shows the fixed-point ACSs (and the $\mathcal{YS}$ for each memory block) obtained by our revised persistence analysis. For example, $\mathcal{YS}(\hat{s}^{out}_{B3}, c) = \{a, b\}$, since both a and b are accessed (in basic block $B1$ and $B3$, respectively) after an access of c (in $B2$ of the previous iteration(s)). As a result, our revised persistence analysis correctly captures the scenario that memory block c could be possibly evicted during the program execution.

The details of our revised persistence analysis, and its safety proofs are presented in the technical report [11].

D. Pessimism in Data Cache Persistence Analysis

While our revised persistence analysis fixes the underestimation error, it still suffers from the overestimation issues as in the original persistence analysis in the context of data cache analysis. Figure 2(a) presents our motivating example which has four array references in two nested loops $L1$ (induction variable i) and $L2$ (induction variable j). We assume a data cache with block size is 32-Byte (contains 8 'int' elements or 16 'short int' elements), four cache line $f_0...f_3$, and associativity $A = 2$. Figure 2(b) gives the control flow graph (CFG) of the program fragment. For each array access, we also list the set of memory blocks possibly referenced. For example, $B[i][j]$ in basic block $B6$ may access memory blocks $\{m_2...m_9\}$. Figure 2(c) shows the relation between memory blocks accessed by each data reference, and the value of loop induction variables i and j. Given the assumed memory configuration, $A[x]$ may access m_0 or m_1 in any iteration of $L1$ since the value of input-dependent variable x is unknown at compile time. On the other hand, the access pattern of $B[i][j]$, $C[i][j]$, and $D[0]$ could be statically determined. For example, in first iteration of $L1$ ($i = 0$), $B[i][j]$ accesses m_2 in first 8 iterations of L2 ($0 \leq j \leq 7$), and m_3 in next 8 iterations of L2 ($8 \leq j \leq 15$). Finally, Figure 2(d) illustrates the cache set mapping.

Neither the CME frameworks [10], [15] nor must analysis [16] works well for this example, due to the input-dependent accesses (in basic block $B2$) and branches (in basic block $B5$). Furthermore, traditional abstract interpretation-based analysis techniques ([9], [16]) capture only global properties of memory accesses, which may lead to significant over-estimation. In the given example, since more than 2 (the associativity) memory blocks are mapped to each cache line within the outer loop $L1$ (Figure 2(d)), the original persistence analysis *cannot* guarantee any cache hits - leading to a large over-estimation.

V. Scope-aware 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 VI.

Definition 2: **(Temporal scope)** A temporal scope for memory block m which is possibly accessed by a data reference D is defined as

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

where $reside(D)$ is the set of loops where D resides in. For each of such loops L_i, temporal scope $\overline{m}^D[L_i]$ (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 $\overline{m}^D$ 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 [19] to identify address expression for each data reference at binary-code level. For each register used to specify address of load/store instruction, we recursively perform register expansion to trace the source registers and the computation performed, until it traces back to a defined constant c, an unpredictable value $\perp$, or a loop induction variable V. Readers are referred to [19] 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.

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

(a) Address expressions

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

(b) Temporal scopes

Figure 3. Address expressions and temporal scopes

- 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 3(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 3, $A[x]$ 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 3(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 $[128 + m_2, 128 + 28 + m_2]$, where m_2 is the base address of $B[i][j]$ ($BaseB$). Given our assumption that 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 scope overlapping between two temporal scopes over L is recursively defined as

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

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

In Figure 3, 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 another 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 $L1$.

VI. Proposed Data Cache Analysis

To reduce the pessimism of the original data cache persistence analysis as discussed in Section IV-D, 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 nesting levels of the program execution (the static scopes), and utilize the computed temporal scope information for a scope-aware analysis. Our framework is built on our correct version of persistence analysis as described in Section IV-C. The soundness proofs are presented in the technical report [11].

A. Scope-aware Persistence Analysis

The core idea of our scope-aware persistence analysis is to categorize the persistence of memory blocks in the calculated temporal scopes (Section V), instead of the globally defined persistence in [9]. For a data reference D, the temporal scope $\overline{m}^D$ 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 3: **(Scope persistence)** Let $Iter$ be the loop iterations bounded by $[\overline{m}^D[L].lw, \overline{m}^D[L].up]$ where data reference D may access memory block m during an execution of loop L (between L's entry and exit). The temporal scope $\overline{m}^D$ 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 $[\overline{m}^D[L].lw, \overline{m}^D[L].up]$, 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 4: **(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 evicted line $l_\top$) to set of all temporal scopes TS (refer to Figure 4(c) for an example). $\hat{S}$ denotes the set of all abstract set states. □

Definition 5: **(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-designed the persistence analysis framework to utilize the scope information. By capturing fine-grained persistence properties, our analysis can accurately model the local behavior of data cache for WCET estimation.

B. Overall Framework

As described in Section IV-D, a memory block m could be persistent in the inner loop but not in the outer loop (e.g., m_5 is persistent in L2 but non-persistent in L1, in the example given in Figure 2). 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 4(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 4 we show the estimation results of our analysis for the motivating example presented in Figure 2, and a detailed discussion will be given in Section VI-D.

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

```
1: ĉ^in_{L_entry}[L] = ⊥;
2: Queue.insert(L_entry);
3: while !Queue.empty() do
4:     n = Queue.remove();
5:     ĉ^in_n[L] = Ĵ_Ĉ({ĉ^out_n'[L] | ∀n' ∈ Pred(n) ∧ n' ∈ L});
6:     if reached_fixed_point( ĉ^in_n[L]) then continue;
7:     ĉ^out_n[L] = ĉ^in_n[L];
8:     for each data reference D in n do
9:         ĉ^out_n[L] = Û_Ĉ(ĉ^out_n[L], TS^D, L);
10:    end for
11:    Queue.insert({n' | ∀n' ∈ Succ(n) ∧ n' ∈ L});
12: end while
```

Algorithm 1 describes the multi-level persistence analysis framework which captures the static scope (loop nesting level) information. $\hat{c}^{in}_n[L]$ 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 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 (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).

C. 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 $\overline{m}^D_a$ (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). Therefore, to avoid overestimation, the scope-aware update function only adds m_a to the younger set of $\overline{m}$ (as in Definition 1) when their temporal scopes overlap.

At loop level L, the scope-aware update function for a given input ACS $\hat{c}$ and set of temporal scopes TS^D accessed by D can be defined as:

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

The scope-aware update function $\hat{\mathcal{U}}_{\hat{\mathcal{C}}}$ divides the accessed temporal scopes $\{\overline{m}_1...\overline{m}_k\}$ 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 $\hat{\mathcal{U}}_{\hat{\mathcal{S}}}$ computes the output abstract set state $\hat{s}^{out}$, via updating the younger set and the maximal relative age of each temporal scope $\overline{m} \in (\hat{s}^{in} \cup X_{f_i})$

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

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

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

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

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.
- Else if $\overline{m}$ is in $\hat{s}^{in}$ and it is also accessed in X_{f_i}. If there is no other temporal scope in TS^D overlaps

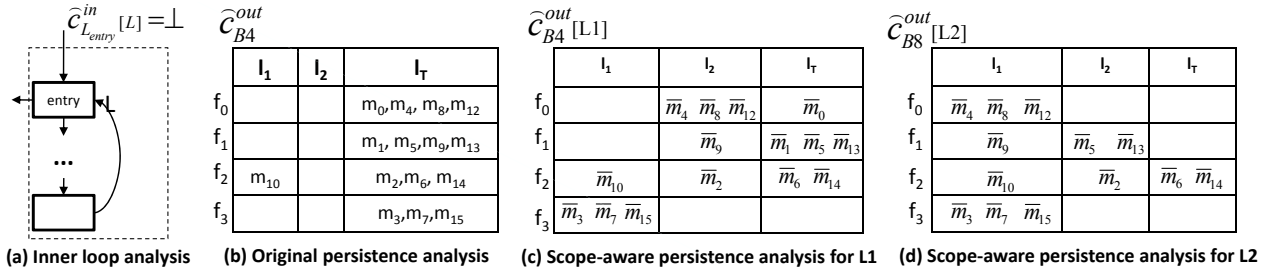


Figure 4. Multi-level analysis and results for the motivating example in Figure 2

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 must renew the relative age of $\overline{m}$ in $\hat{s}^{in}$, and we can set its younger set to be empty.

- Otherwise, relative age of a memory block m can be interfered by any memory block m_a accessed by D that maps to the same cache set, where the temporal scopes of $\overline{m}$ and $\overline{m}_a$ overlap at loop level L (according to Equation 1). We add all possible memory blocks m_a to the younger set $\mathcal{YS}(\hat{s}^{out}, \overline{m})$.

Figure 5(a) illustrates our scope-aware persistence analysis in loop $L2$ of the running example in Figure 2. 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. Hence, they do not age each other. On the other hand, in cache set f_1, as shown in Figure 2, $B[i][j]$ 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 $\overline{m}_5[L2]$, as shown in Figure 5(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. For each temporal scope $\overline{m}$ in $\hat{c}$, the scope-aware join function unionizes the younger set of $\overline{m}$ in both output ACSs from the control flow predecessors to form the younger set of $\overline{m}$ 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.

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

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

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

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

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

D. ACS Computation of the Motivating Example

Figure 4(b), (c) and (d) shows the fixed-point ACSs computed by the original persistence analysis (at basic block $B4$, exit of $L1$), the proposed scope-aware multi-level analysis for loop $L1$ (at basic block $B4$) and $L2$ (at basic block $B8$) for the motivating example in Figure 2, respectively. Given a 2-way associative cache with 4 cache sets, *no* memory block accessed by $B[i][j]$ and $C[i][j]$ can be categorized as persistence in the original persistence analysis. On the other hand, our proposed 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.

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 2, 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 4(c) and Figure 4(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 6. In Figure 4, 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

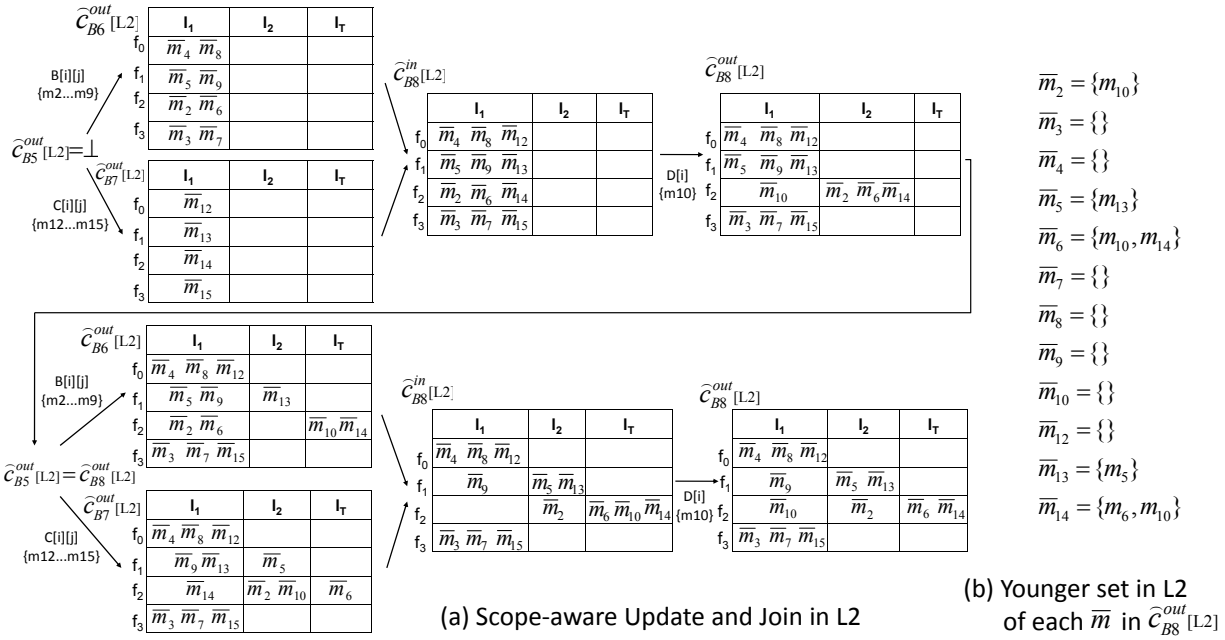


Figure 5. Scope-aware ACS computation for L2 of the motivating example in Figure 2

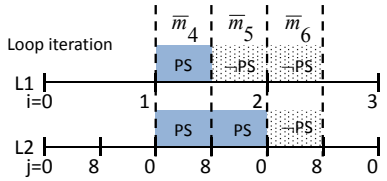


Figure 6. Temporal scopes and loop iterations

$\overline{m}_5[L2]$ 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 $\hat{c}^{in}_n[L]$ ($\hat{c}^{out}_n[L]$) for entry (exit) of each CFG node n in each loop level L. If m is persistent in scope $\overline{m}[L]$ (or $\overline{m}^D[L]$) of loop level L, accesses to m by data reference D incurs only one cold miss for each complete execution of L (between entry and exit). Let L_{ps} be the outer-most loop level where $\overline{m}$ is persistent. Hence, accesses to m incur 1 cold miss for each execution of L_{ps} (including all its inner loops). The following function $blockMiss(D, m)$ computes the maximum number of cache misses D may incur due to accesses of m during the entire program execution.

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

with $\overline{m} = \overline{m}^D$

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

As illustrated in Figure 6, $L1$ 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, $miss(D)$, is the summation of $blockMiss(D, m)$ over all memory blocks in $AddrSet(D)$ which D may access.

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

In our motivating example, $B[i][j]$ accesses 8 memory blocks ($\{m_2, \ldots, m_9\}$). According to our scope-aware analysis results shown in Figure 4, 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.

Table I
BENCHMARK DESCRIPTIONS AND WCET ESTIMATION RESULT

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

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 simulation to make it consistent with the assumptions made in our analysis. The cache analysis results on maximum number of data cache misses for each data reference are integrated as linear constraints into Chronos ([13]), an ILP-based WCET analysis tool for static WCET estimation. In our current implementation, we assume a processor architecture without timing anomalies [7]. However, the resulted cache modeling can be integrated with pipeline analysis as presented in [14] for architectures with timing anomalies.

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

We have implemented the revised persistence analysis (Section IV-C), multi-level persistence framework [2] (using the revised persistence analysis), and the must analysis with loop unrolling as proposed in [16] to compare with our proposed scope-aware analysis. Figure 7 shows the percentage of overestimation from various data cache analysis approaches, compared to the normalized observed WCET results from SimpleScalar simulation (shown in Table I). Given the array size in our experiment, since the entire array does not fit into the data cache for any of the benchmarks, no memory block can be categorized as persistent in the persistence analysis. Without the temporal scope information, multi-level persistence analysis [2] cannot give tighter estimation, except for the Lms benchmark, where only small arrays are accessed in different loop nesting levels. As a result, the estimated WCET results without temporal scope are up to 83% higher than the observed WCET (for *InsertSort*). We also compare the estimated WCET results using must analysis with 20% and 50% virtual unrolling of the loop nest ([16]), where the analysis is repeatedly performed for each unrolled loop iteration. As shown in Figure 7, even when 50% of the loop nest is unrolled, must analysis [16] still reports up to 65% higher WCET estimate compared to the observed simulation time (for *Adpcm*). Must analysis requires loop unrolling to bring memory blocks to the data cache and to capture subsequent reuse. Therefore, in the remaining portion of the loop nest where unrolling is not applied, they can not capture any cache reuse.

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

Matmult contains a column array access in addition to sequential array accesses. In our analysis, a temporal scope captures the lower and upper bound of loop iterations where a memory block may get accessed. For column array access, array elements contained in a single memory block are

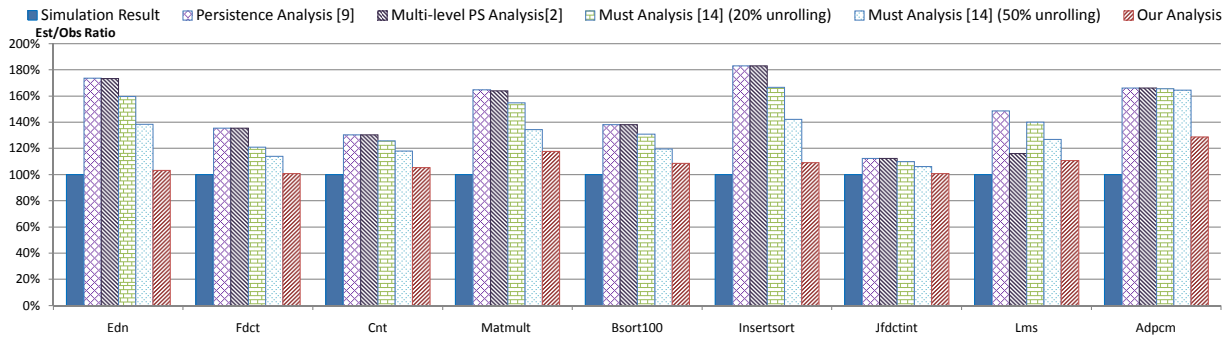


Figure 7. WCET estimation results from different analyses

usually accessed in non-contiguous loop iterations, which leads to over-estimation in the computed temporal scopes. However, as shown in Figure 7, our estimated WCET is only 25% higher than the observed WCET, and is 10% to 40% tighter than other approaches.

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

IX. Concluding Remarks

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

Our scope-aware multi-level data cache analysis extends the cache persistence analysis framework to compute fine-grained scope-based persistence information, which leads to substantially tighter worst-case cache miss estimation. While we have presented our analysis for LRU-based cache replacement policy, it can also be extended to handle other deterministic cache replacement policies like FIFO and MRU. In particular, the abstract cache update function has to be changed to cope with the chosen replacement policy. Finally, the proposed analysis has been integrated into the open-source Chronos WCET analyzer ([13] version 4.1).

X. Acknowledgements

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

References

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

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

[3] D. Burger and T.M. Austin. The SimpleScalar tool set, version 2.0. *ACM SIGARCH 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] Jan Reineke *et al.*, Timing predictability of cache replacement policies. Real-Time Systems 2007

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

[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. K. Huynh, L. Ju, and A. Roychoudhury. Scope-aware Data Cache Analysis for WCET Estimation. Technical Report TR20/10, National University of Singapore, 2010, http://www.comp.nus.edu.sg/~abhik/pdf/TR20-10.pdf.

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

[13] 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.

[14] X. Li, A. Roychoudhury, and T. Mitra. Modeling Out-of-Order Processors for Software Timing Analysis. In *RTSS*, 2004.

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

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

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

[18] 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.

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