Improving the Precision of Abstract Interpretation Based Cache Persistence Analysis

Zhenkai Zhang Xenofon Koutsoukos

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

Abstract

When designing hard real-time embedded systems, it is required to estimate the worst-case execution time (WCET) of each task for schedulability analysis. Precise cache persistence analysis can significantly tighten the WCET estimation, especially when the program has many loops. Methods for persistence analysis should safely and precisely classify memory references as persistent. Existing safe approaches suffer from multiple sources of pessimism and may not provide precise results. In this paper, we first identify some sources of pessimism that two recent approaches based on younger set and may analysis may encounter. Then, we propose two methods to eliminate these sources of pessimism. The first method improves the update function of the may analysis-based approach; and the second method integrates the younger set-based and may analysis-based approaches together to further reduce pessimism. We also prove the two proposed methods are still safe. We evaluate the approaches on a set of benchmarks and observe the number of memory references classified as persistent is increased by the proposed methods. Moreover, we empirically compare the storage space and analysis time used by different methods.

Categories and Subject Descriptors B.3.3 [*Performance Analysis and Design Aids*]: Worst-case analysis

General Terms Performance, Verification

Keywords Cache Analysis, WCET, Persistence Analysis

1. Introduction

When designing hard real-time embedded systems, we need to perform schedulability analysis to guarantee the stringent timing constraints will be met. Schedulability analysis needs the worst-case execution time (WCET) of each real-time task as input. Therefore, WCET analysis is one essential step in designing such systems, and has been studied extensively (see [17] for a survey). In general, the exact WCET of a task is impossible to derive. Thus, when estimating WCET, over-approximation is necessary to guarantee safety. However, in order to maximize the resource utilization, an WCET estimation should be as tight as possible.

Due to the big timing gap between a cache hit and a miss, cache behavior can affect the execution time significantly. In order to derive a tight WCET estimation, we want the cache behavior analysis to be as precise as possible. Although model checking-based cache analysis can yield precise results since all the possible executions are examined, potential state space explosion makes it hard to use in practice [16]. Compared to model checking, cache analysis methods based on abstract interpretation may lose some precision but can achieve much better scalability. In this paper, we focus on how to improve the precision of cache analysis that is based on abstract interpretation [4].

When predicting the cache behavior, a widely used method is to classify the memory references as – *AH* (it *always hits* the cache), *AM* (it *always misses* the cache), *PS* (it is *persistent* if the memory reference may result in a cache hit/miss for the first time but it hits the cache subsequently), and *NC* (it is *not classified* if the memory reference is not classified as *AH*, *AM*, or *PS*). These classifications are derived by performing three different analyses, **must**, **may**, and **persistence** analyses, on the control flow graph (CFG) [15]. While the must and may analyses are safe, it has been known that the original persistence analysis method proposed in [7] is unsafe. Several approaches have been proposed to ensure safe cache persistence analysis [5, 6, 11]. However, different approaches may suffer from certain pessimism under different scenarios (i.e. some references should have been classified as *PS* if the analysis were precise).

In this paper, we first analyze the sources of pessimism of safe cache persistence analysis of single-level loops. Then, we propose two methods to eliminate these sources of pessimism. We focus on persistence analysis of A-way set associative instruction caches which use LRU (least recently used) replacement policy. However, the methods can be easily extended to data/unified cache persistence analysis.

The main technical contributions of this paper include:

1. We identify the sources of pessimism that two recent state-of-the-art persistence analysis methods may encounter: the method proposed in [11] has a pessimistic join function, and the method proposed in [5] has a pessimistic update function.
2. We optimize the update function proposed in [5] by finding a safe limit that bounds the range of the blocks whose potential maximal ages should be increased in an updating process.
3. We integrate the improved method of [5] and the method proposed in [11] to further safely reduce pessimism but the integration may have a large storage overhead. By studying the rela-

LCTES'15, June 18–19, 2015, Portland, OR, USA.
Copyright © 2015 ACM 978-1-4503-3257-6//. . . $15.00.
http://dx.doi.org/10.1145/2670529.2754967

tions of these two approaches, we define two auxiliary functions to reduce this overhead in the integration.

4. We prove the proposed approaches are safe, namely if a memory reference at a program point is classified as *PS*, the memory block it accesses is not possibly evicted from the cache at this point after being loaded.
5. We demonstrate the number of memory references classified as *PS* can be increased by using the proposed methods from the experimental evaluations performed on a set of benchmarks. We also empirically compare the storage space and analysis time used by different methods.

The rest of the paper is organized as follows: Section 2 describes the related work; Section 3 briefly summarizes two recent state-of-the-art safe approaches for cache persistence analysis; Section 4 compares these two approaches in terms of their sources of pessimism; Section 5 improves the update function for the approach based on may analysis, and proves such an improvement is safe; Section 6 proposes an integration of the two existing approaches; Section 7 presents the evaluation and section 8 concludes this paper.

2. Related Work

Abstract interpretation based cache must and may analyses have been well-developed and widely used in WCET analysis to predict the behavior of caches [1, 8]. Since in a loop a reference to a memory block may be a cache miss in the first iteration but can be cache hits in all further iterations if the cache is large enough to hold all the memory blocks referenced within the loop, loop unrolling is used in [1, 8] to tighten the WCET estimates. The original unsafe cache persistence analysis based on abstract interpretation is proposed in [7], which aimed to derive a more precise cache behavior prediction than the previous work in [1, 8] in the presence of loops without using loop unrolling.

In order to perform safe cache persistence analysis, several approaches have been proposed. An approach based on an abstract domain called *younger set* is proposed in [11], and an approach based on *may analysis* is proposed in [5]. Both of the approaches realize that the update function of the original persistence analysis has a problematic aging strategy and they fix this problem based on different tracked information (*younger set* and *may analysis state* respectively). Later in [6], an approach based on a much simpler domain called *conflict counting* is proposed. Although the *conflict counting*-based method uses fewer iterations to converge, it yields less precise results. In [12], the authors also notice the original persistence analysis is unsafe and they adopt the safe one proposed in [5] to perform cache persistence analysis.

Data cache analysis is usually based on must analysis [14] or persistence analysis [9, 11]. Since persistence analysis is not sensitive to input-dependent branches and unpredictable access addresses [11], persistence analysis is more suitable for data cache analysis than must analysis. In [9], the first persistence analysis based data cache analysis is proposed, but it uses the original unsafe one. In [11], a scope-aware younger set based persistence analysis is proposed for its scope-aware data cache analysis, which tries to classify a data reference in a loop as *PS* within certain iterations rather than all of the iterations.

A multi-level persistence analysis method is proposed in [2] to cope with the presence of nested loops. Although it uses the original unsafe approach on each loop level, any safe approach can replace the unsafe one to yield precise analysis for nested loops. A similar cache behavior classification is called "first miss" (*FM*), and it can have varying meanings depending on what the "first" refers to [2]. In [1, 8], loop unrolling is used for distinguishing the first iteration context with others, and it classifies a reference as *FM* with respect to the first iteration of the unrolled loop. In [13], the static cache simulation method is summarized, and the method categorizes a memory reference as *FM* in terms of the first time it is accessed in a loop, which is very similar to the meaning of *PS* category.

It is always possible to improve the precision of cache analysis by ruling out the effects caused by infeasible paths. For example, in [3], the authors integrate an online SAT-based partial path infeasibility checking into cache analysis (and other processor behavior analyses). However, these methods are orthogonal to the focus of this paper. The methods we propose can be used as the basis to incorporate with these path infeasibility checking methods.

3. Background on Cache Persistence Analysis

We first present the objective of cache persistence analysis, and then briefly describe two recent state-of-the-art safe approaches, namely the approach based on younger set which is proposed in [11] and the approach based on may analysis which is proposed in [5].

We model an A-way set associative cache as a sequence of v cache sets $F = \langle f_1, f_2, \ldots, f_v \rangle$. Each cache set is an independent fully associative cache and is modeled by a sequence of A cache blocks $L = \langle l_1, l_2, \ldots, l_A \rangle$. Since the behaviors of the cache sets are independent of each other, we can focus on one cache set for the sake of readability. The memory consists of a set of w memory blocks $M = \{m_1, m_2, \ldots, m_w\}$, and the program has t program points $P = \{p_1, p_2, \ldots, p_t\}$.

3.1 Cache Persistence Analysis

Cache persistence analysis aims to categorize a memory reference that cannot be classified as *AH* at a program point in a loop as *PS*, if its accessed memory block stays in the cache after the first time this block is loaded. If a memory reference is categorized as *PS*, it can result in at most one cache miss. In the case of a loop bounded by n iterations, a reference classified as *PS* instead of *NC* can reduce the number of possible misses by $n - 1$. Thus, we want to safely classify as many references as possible as *PS* for a loop.

In order to guarantee safety, we need to over-approximate a memory block's maximal age at every program point. If a memory block is not among the set of possibly evicted memory blocks, any reference to it can be treated as *PS*. In order to keep track of possibly evicted memory blocks, an additional cache block l_{A+1} is appended to L. If a memory block's potential maximal age is greater than the cache's associativity A, it will be added into this additional cache block. Let $\top \equiv A + 1$, so we have $L' = \langle l_1, \ldots, l_A, l_\top \rangle$ model a cache set to capture persistent behavior. Therefore, in cache persistence analysis, an abstract set state $\hat{s}_{pers}$ is often modeled as $\hat{s}_{pers} \in D_{\mathcal{P}} = L' \to 2^M$, and $\hat{s}_{pers}(l_\top)$ gives the over-approximated set of memory blocks that are possibly evicted after being loaded into this cache set.

In order to improve the precision, we want to tighten the over-approximation of a memory block's maximal age. Therefore, we want to eliminate possible sources of pessimism in the analysis to keep $l_\top$ from containing too many persistent memory blocks.

3.2 Cache Persistence Analysis Based on Younger Set

The basic idea of the approach based on younger set (*YS-Pers*) is to keep track of all the memory blocks that may be younger than a memory block for that block. Thus, a memory reference can be categorized as *PS* if the cardinality of the accessed memory block's younger set is less than A.

Let $ys^p(m)$ denote the younger set of a memory block *m* at a program point *p*, and let $YS = M \to (2^M)_\perp$ denote the set of all the younger set mappings, i.e. we have $ys^p \in YS$. Since the ys^p may be a partial function, namely there may be no younger set for some memory block at some program point, we use the lifted co-domain $(2^M)_\perp = 2^M \cup \{\perp\}$, where $\perp$ means "no younger set at all". As defined in [11], $ys^p(m)$ is *a superset of all the memory blocks that may have smaller relative ages (younger) than m at p in some possible program execution that reaches p*. The potential maximal age of m can be calculated as $|ys^p(m)|+1$ which is in the range $[1 \ldots \top]$, assuming we stop tracking when $|ys(m)|$ reaches A.

Therefore, given the younger set mapping ys^p at a program point p, the i^{th} cache set's abstract set state $\hat{s}^{p,i}_{pers}$ is actually derived from ys^p by applying the function $G_{\mathcal{P}} : YS \times \{1, \cdots, v\} \to D_{\mathcal{P}}$ (i.e. $\hat{s}^{p,i}_{pers} = G_{\mathcal{P}}(ys^p, i)$), and the $G_{\mathcal{P}}$ function is defined as:

$$G_{\mathcal{P}}(ys, i) :=$$
$$[l_x \mapsto \{m | set(m) = i \wedge ys(m) \neq \perp \wedge x = |ys(m)| + 1\} \; ^1$$
$$\text{with } x = 1, \cdots, A]$$

where $[\delta \mapsto \theta]$ denotes a function that maps δ to θ and $set(m)$ gives the cache set number which m is mapped to.

If a memory block m' is going to be accessed at a program point p', which is immediately following a program point p, the younger set mapping $ys^{p'}$ can be calculated by performing the younger set mapping update function $\hat{U}_{\mathcal{YS}} : YS \times M \to YS$ on ys^p to take into account the effect of the reference to m' (i.e. $ys^{p'} = \hat{U}_{\mathcal{YS}}(ys^p, m')$), and the $\hat{U}_{\mathcal{YS}}$ is defined as:

$$\hat{U}_{\mathcal{YS}}(ys, m') :=$$
$$[m \mapsto \begin{cases} ys(m) & \text{if } set(m') \neq set(m) \\ ys(m) \cup \{m'\} & \text{else if } m' \neq m \\ \emptyset & \text{otherwise} \end{cases}]$$

If a program point p is a join point of two points $p1$ and $p2$ at which the younger set mappings are ys^{p1} and ys^{p2} respectively, the joined younger set mapping ys^p can be calculated by applying the younger set mapping join function $\hat{J}_{\mathcal{YS}} : YS \times YS \to YS$ (i.e. $ys^p = \hat{J}_{\mathcal{YS}}(ys^{p1}, ys^{p2})$), and the $\hat{J}_{\mathcal{YS}}$ is defined as:

$$\hat{J}_{\mathcal{YS}}(ys^{p1}, ys^{p2}) :=$$
$$[m \mapsto \begin{cases} ys^{p1}(m) \overline{\cup} ys^{p2}(m) & \text{if } ys^{p1}(m) \neq \perp \wedge ys^{p2}(m) \neq \perp \\ ys^{p1}(m) & \text{else if } ys^{p1}(m) \neq \perp \\ ys^{p2}(m) & \text{else if } ys^{p2}(m) \neq \perp \\ \perp & \text{otherwise} \end{cases}]$$

where $\overline{\cup}$ is a special set union operation which may truncate some memory blocks in the union at random to make the cardinality of the union at most A. For a persistent memory block m, the resulted younger set $ys^p(m)$ always contains all the potentially younger blocks of m (in the case that m is not persistent, namely it is possibly evicted, some of its younger blocks may be truncated, but it does not affect m will be placed in the corresponding $l_\top$).

3.3 Cache Persistence Analysis Based on May Analysis

The approach based on may analysis (*May-Pers*) utilizes the over-approximation of cache contents generated by a parallel running may analysis to guide the maximal age updating. Basically, *May-Pers* is a combination of two analyses: (1) the *may-part* analysis (whose join and update functions are $\hat{J}_{\mathcal{M}}$ and $\hat{U}_{\mathcal{M}}$ respectively) is the traditional may analysis and it is used to provide the other analysis with an over-approximation of cache contents; and (2) the *persistence-part* analysis (whose join and update functions are $\hat{J}_{\mathcal{Q}}$ and $\hat{U}_{\mathcal{Q}}$ respectively) is a modification of the traditional may analysis which tracks the maximal age of a memory block instead of the minimal age [5, 6]. The abstract set state domain used in this approach is

$$D^{\text{may-pers}}_{\mathcal{P}} = D_{\mathcal{M}} \times D_{\mathcal{P}} = (L \to 2^M) \times (L' \to 2^M)$$

where $D_{\mathcal{M}} = L \to 2^M$ is the abstract set state domain for the traditional may analysis and $D_{\mathcal{P}} = L' \to 2^M$ is the abstract set state domain for the original persistence analysis. Thus, an abstract set state is a 2-tuple $\langle \hat{s}_{may}, \hat{s}_{pers} \rangle$, a *may-part* $\hat{s}_{may}$ and a *persistence-part* $\hat{s}_{pers}$ respectively. While the parallel running *may-part* analysis is independent from the *persistence-part* analysis, when the *persistence-part* analysis updates $\hat{s}_{pers}$ it has to take into account $\hat{s}_{may}$.

The update function $\hat{U}_{\mathcal{P}} : D^{\text{may-pers}}_{\mathcal{P}} \times M \to D^{\text{may-pers}}_{\mathcal{P}}$ for the *May-Pers* is defined as:

$$\hat{U}_{\mathcal{P}}(\langle \hat{s}_{may}, \hat{s}_{pers} \rangle, m) :=$$
$$\langle \hat{U}_{\mathcal{M}}(\hat{s}_{may}, m), \hat{U}_{\mathcal{Q}}(\hat{s}_{may}, \hat{s}_{pers}, m) \rangle$$

where $\hat{U}_{\mathcal{M}}$ is the well-defined update function for the may analysis (whose definition can be found in [15]), and $\hat{U}_{\mathcal{Q}} : D_{\mathcal{M}} \times D_{\mathcal{P}} \times M \to D_{\mathcal{P}}$ is the update function for the *persistence-part* analysis, which is defined as:

$$\hat{U}_{\mathcal{Q}}(\hat{s}_{may}, \hat{s}_{pers}, m) :=$$
$$\begin{cases} [l_1 \mapsto \{m\}, & \\ l_i \mapsto \hat{s}_{pers}(l_{i-1}) \backslash \{m\} | i = 2 \ldots A, & \\ l_\top \mapsto (\hat{s}_{pers}(l_A) \cup \hat{s}_{pers}(l_\top)) \backslash \{m\}] & \text{if } \textit{mayevict}(\hat{s}_{may}, m) \\ & \\ [l_1 \mapsto \{m\}, & \\ l_i \mapsto \hat{s}_{pers}(l_{i-1}) \backslash \{m\} | i = 2 \ldots A-1, & \\ l_A \mapsto (\hat{s}_{pers}(l_A) \cup \hat{s}_{pers}(l_{A-1})) \backslash \{m\}, & \\ l_\top \mapsto \hat{s}_{pers}(l_\top) \backslash \{m\}] & \text{otherwise} \end{cases}$$

$$\textit{mayevict}(\hat{s}_{may}, m) := |\{m' | m' \neq m \wedge m' \in \hat{s}_{may}\}| \geq A$$

Basically, *mayevict*$(\hat{s}_{may}, m)$ checks whether the overestimated contents given by $\hat{s}_{may}$ have potentially filled the cache set or not. If the *mayevict* function returns true, the abstract set state $\hat{s}_{may}$ of the may analysis contains at least A many other memory blocks than m. In this case, the cache set may be completely filled already without counting m, so an access to m potentially increase the maximal ages of all the memory blocks in $\hat{s}_{may}$ and may cause some blocks evicted (as shown in the first case of the update function). On the contrary, if the *mayevict* function returns false, the cache set is definitely not full yet, so no eviction will happen due to loading m. In this case, the maximal ages of all memory blocks will not exceed A (as shown in the second case of the update function).

The join function $\hat{J}_{\mathcal{P}} : D^{\text{may-pers}}_{\mathcal{P}} \times D^{\text{may-pers}}_{\mathcal{P}} \to D^{\text{may-pers}}_{\mathcal{P}}$ for the *May-Pers* is defined as:

$$\hat{J}_{\mathcal{P}}(\langle \hat{s}^{p1}_{may}, \hat{s}^{p1}_{pers} \rangle, \langle \hat{s}^{p2}_{may}, \hat{s}^{p2}_{pers} \rangle) :=$$
$$\langle \hat{J}_{\mathcal{M}}(\hat{s}^{p1}_{may}, \hat{s}^{p2}_{may}), \hat{J}_{\mathcal{Q}}(\hat{s}^{p1}_{pers}, \hat{s}^{p2}_{pers}) \rangle$$

where the $\hat{J}_{\mathcal{M}}$ function is the well-defined join function for the may analysis (whose definition can be found in [15]), and $\hat{J}_{\mathcal{Q}}$:

[1] If we do not stop tracking new potentially younger blocks when $|ys(m)|$ reaches A, we would have $x = \min(|ys(m)| + 1, \top)$.

$D_{\mathcal{P}} \times D_{\mathcal{P}} \rightarrow D_{\mathcal{P}}$ is defined as:

$$\hat{J}_{\mathbb{Q}}(\hat{s}^{p1}_{pers}, \hat{s}^{p2}_{pers}) :=$$
$$[l_i \mapsto \begin{array}{l} \{m | m \in \hat{s}^{p1}_{pers}(l_i) \wedge \nexists b \in [1 \ldots \top] : m \in \hat{s}^{p2}_{pers}(l_b)\} \cup \\ \{m | m \in \hat{s}^{p2}_{pers}(l_i) \wedge \nexists a \in [1 \ldots \top] : m \in \hat{s}^{p1}_{pers}(l_a)\} \cup \\ \{m | \exists a, b \in [1 \ldots \top] : \\ \quad m \in \hat{s}^{p1}_{pers}(l_a) \wedge m \in \hat{s}^{p2}_{pers}(l_b) \wedge i = \max(a, b)\}] \end{array}$$

Basically, the $\hat{J}_{\mathbb{Q}}$ function is much similar to the join function of the original persistence analysis, which is similar to set union operation except that if a memory block has two different ages in the two joining set states then the function takes the oldest one.

4. Sources of Pessimism

There have been several approaches proposed to safely analyze cache persistence. However, there has been little work done to compare and find out whether these safe approaches are precise enough under different circumstances, and to improve their precision for a single-level loop. Although the advantages and disadvantages of the approaches based on may analysis and conflict counting are discussed in [6], that paper does not compare them with the approach based on younger set that is proposed in [11].

Since we know that the approach based on conflict counting is not as precise as the one based on may analysis (due to the loss of age information), we concentrate on the comparisons between the approaches based on younger set and may analysis – we discuss under what circumstances an approach may give pessimistic analysis results and show how the approaches can complement each other.

In order to enhance the readability of examples, we assume a 2-way set associative cache is used. Memory blocks m_a, m_b, and m_c are mapped into the same cache set that we focus on. In Fig. 1, a basic block with a memory block shown inside (e.g. BB_1 in the figure has m_a shown inside) denotes the basic block contains an instruction which references to the corresponding memory block; otherwise, the basic block (e.g. BB_2 in the figure has no relationship with the cache set we are examining.

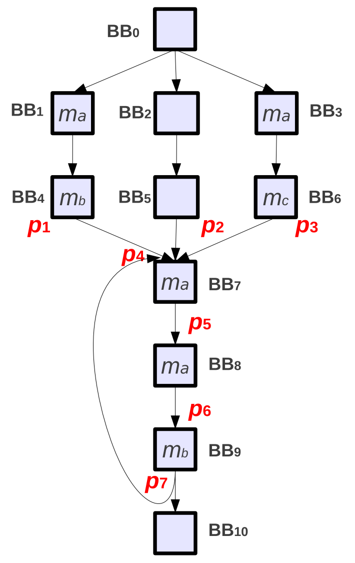

Figure 1. The CFG of a program: all of the references in the loop should be classified as *PS*

4.1 Pessimism in *YS-Pers*

The persistence analysis based on younger set is safe, but it may excessively overestimate the maximal age of a memory block m at a join point p_j, since the join function $\hat{J}_{\mathcal{YS}}$ uses the concept of set union (as mentioned in section 3.2, the $\overline{\cup}$ operation is used) to ensure all the possibly younger memory blocks on all the joined paths are captured for m, namely

$$ys^{p_j}(m) = \hat{J}_{\mathcal{YS}}(\ldots, \hat{J}_{\mathcal{YS}}(ys^{p_{i_1}}(m), ys^{p_{i_2}}(m)), \ldots, ys^{p_{i_n}}(m))$$

where $\{p_{i_1}, p_{i_2}, \ldots, p_{i_n}\}$ is the set of the exit points of p_j's n predecessors denoted as $pred(p_j)$. Therefore, this ***may*** introduce some pessimism if $\exists p_{i_x}, p_{i_y} \in pred(p_j) : ys^{p_{i_x}}(m) \neq ys^{p_{i_y}}(m)$, especially when disjoint sets of memory blocks are accessed in the disjoint parts of paths reaching p_{i_x} and p_{i_y}.

Consider the program point p_4 in Fig. 1 which is a join point with four predecessors (i.e. BB_4, BB_5, BB_6, and BB_9). Although the memory reference to m_a in BB_7 cannot be classified as *AH* due to the possible path $BB_0 \rightarrow BB_2 \rightarrow BB_5 \rightarrow BB_7$, we can easily observe the reference should be classified as *PS*, in which case, this memory reference contributes at most one cache miss to the loop independent of the number of its iterations.

However, when using *YS-Pers* to perform persistence analysis, we observe that at the exit point of BB_7's each predecessor (i.e. the program points p_1, p_2, p_3, and p_7) the m_a's younger set is as follows:

$$ys^{p_1}(m_a) = \{m_b\} \quad ys^{p_2}(m_a) = \emptyset$$
$$ys^{p_3}(m_a) = \{m_c\} \quad ys^{p_7}(m_a) = \{m_b\}$$

Since the join function of the younger set is based on the $\overline{\cup}$ operation, at p_4 the younger set of m_a is always:

$$ys^{p_4}(m_a) = \overline{\bigcup_{p \in \{p_1, p_2, p_3, p_7\}}} ys^{p}(m_a) = \{m_b, m_c\}$$

Given the cache associativity is 2, it means before the memory reference to m_a in BB_7, m_a always has the age $\top$, which prevents us from classifying the reference as *PS*.

4.2 Pessimism in *May-Pers*

Compared to *YS-Pers*, *May-Pers* can precisely classify the memory reference to m_a in BB_7 as *PS*. Although the approach does not suffer from the pessimism when joining the states, it does not mean the approach will always yield more precise analysis. Actually, one apparent source of pessimism in this approach comes from its pessimistic update function, which we will try to optimize in the next section. In order to ensure safety, when accessing a memory block m, the *persistence-part* update function $\hat{U}_{\mathbb{Q}}$ proposed in [5] increases the potential maximal ages of all memory blocks if the abstract set state $\hat{s}_{may}$ of the parallel running *may-part* analysis contains at least cache's associativity A many other elements than m, namely when the *mayevict*$(\hat{s}_{may}, m)$ is true, as described in section 3.3.

In order to help understand the source of pessimism in *May-Pers*, Fig. 2 shows the iterative process of deriving the fixed points of the abstract set states corresponding to the seven program points shown in Fig. 1. The dotted transition lines in Fig. 2 are related to the join function, and the solid transition lines are related to the update function. The initial set state at each program point is an empty $\langle \hat{s}_{may}, \hat{s}_{pers} \rangle$. The figure shows in the second iteration every abstract set state reaches its fixed point (and the third iteration is omitted, which only verifies each abstract set state has reached its fixed point).

Consider the program point p_5 in Fig. 1. Since there is a memory reference to m_a in BB_8, the execution of BB_8 needs to update the corresponding abstract set state. When performing the update function which is piecewise, we need to consider how many memory blocks other than m_a are in the *may-part* of the input abstract set state. In our case, as shown in the circled part of Fig. 2, the num-

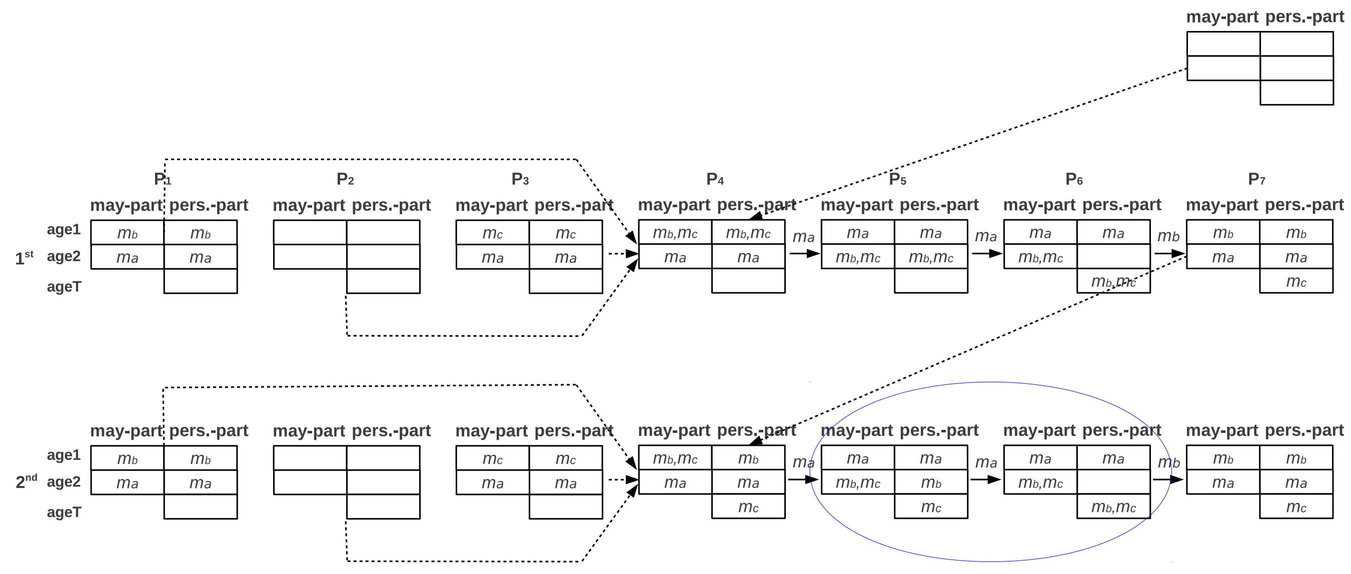
Figure 2. Abstract set states of Fig. 1 computed by the may analysis based approach

ber of other memory blocks is $|\{m_b, m_c\}| = 2$ equal to the cache's associativity, which means the cache set may be already full and the memory reference to m_a may cause an eviction. As a consequence, the update function will increase the potential maximal ages of all memory blocks in the *persistence-part* of the abstract set state and reset m_a to be the youngest, as shown in the circled abstract set state at p_6.

From the resultant abstract set state at p_6, we can find that the memory reference to m_b in BB_9 cannot be classified as *PS* (since its new age is $\top$). However, we can easily observe that it is indeed persistent in the loop independent of the number of its iterations. Again, the precision is reduced while the analysis is safe.

On the contrary, the approach based on younger set can give us the *PS* classification for the reference to m_b in BB_9, since at p_6 the younger set of m_b is $ys^{p_6} = \{m_a\}$ which contains one possibly younger memory block. Therefore, as a comparison, we can observe the *YS-Pers* approach may introduce some pessimism due to joining different younger sets while the *May-Pers* approach may introduce some pessimism due to updating the abstract set state.

4.3 Overview of the Proposed Approaches

Our problem is how to eliminate these sources of pessimism to improve the *PS* classification precision while keeping the analysis still safe. In order to solve this problem, we propose two approaches in the next two sections: the first one is based on the *May-Pers* approach but improves its updating strategy (see section 5); and the second one is an integration of the improved *May-Pers* and *YS-Pers* approaches (see section 6).

In Fig. 3, the relationship between the various approaches is illustrated by a Venn diagram. The whole area represents all of the memory references that are persistent in the program. As shown in the figure, the approaches based on may analysis (i.e. the original *May-Pers* and the improved *May-Pers*) can classify some references as *PS* while the approach based on younger set cannot; and vice versa (as we have discussed above). However, we can see that the amount of memory references classified as *PS* by the integration approach dominates that by the others. In section 7, this relationship is validated empirically by experiments.

5. More Precise Update Function for *May-Pers*

As discussed in 4.2, although the *persistence-part* update function $\hat{U}_{\mathcal{Q}}$ is safe, it is not precise enough – it makes m_b be treated as possibly evicted while in reality m_b definitely stays in the loop after the first time being loaded. We improve the $\hat{U}_{\mathcal{Q}}$ function by using a new strategy to decide where the process of maximal age updating should end, while keeping the join function of the analysis unchanged.

The new persistence analysis update function $\hat{U}^{\text{new}}_{\mathcal{P}} : D^{\text{may-pers}}_{\mathcal{P}} \times M \rightarrow D^{\text{may-pers}}_{\mathcal{P}}$ is based on the well-defined update function for the may analysis (i.e the $\hat{U}_{\mathcal{M}}$ function) and the improved $\hat{U}_{\mathcal{Q}}$ function (i.e. the $\hat{U}^{\text{new}}_{\mathcal{Q}}$ function), and the $\hat{U}^{\text{new}}_{\mathcal{P}}$ function is defined as:

$$\hat{U}^{\text{new}}_{\mathcal{P}}(\langle \hat{s}_{may}, \hat{s}_{pers}\rangle, m) :=$$
$$\langle \hat{U}_{\mathcal{M}}(\hat{s}_{may}, m), \hat{U}^{\text{new}}_{\mathcal{Q}}(\hat{s}_{pers}, m, H(\hat{s}_{may}, m))\rangle$$

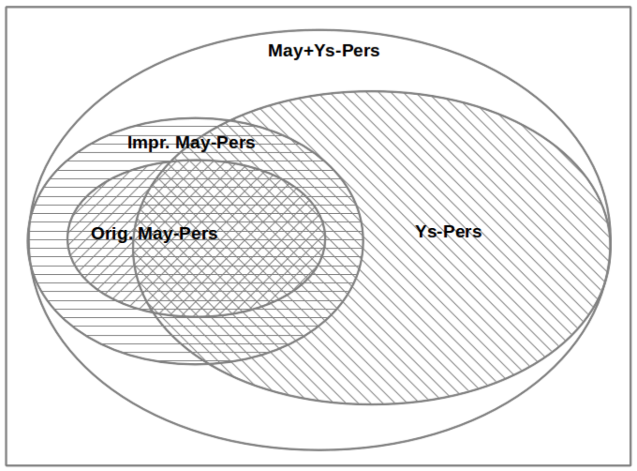

Figure 3. Venn diagram illustrating the relationship between different approaches: both existing and proposed

The $\hat{U}_{\mathcal{P}}^{\text{new}}$ function is much similar to the $\hat{U}_{\mathcal{P}}$ function described in section 3.3, except the update function for the *persistence-part* (i.e. the $\hat{U}_{Q}^{\text{new}}$ function instead of the $\hat{U}_{Q}$ function). In the $\hat{U}_{Q}^{\text{new}}$ function, we use a $H : D_{\mathcal{M}} \times M \rightarrow \{1, \cdots, A, \top\}$ function to select a relative age h which bounds the possibly affected age range due to the memory reference to m. The H function is defined as:

$$H(\hat{s}_{may}, m) :=$$
$$\min(\{y | 1 \leq y \leq A \wedge \sum_{1 \leq i \leq y} |\hat{s}_{may}(l_i) \backslash \{m\}| < y\} \cup \{\top\})$$

Basically, the H function uses the over-approximation of cache contents in $\hat{s}_{may}$ to try to find the youngest y which is smaller than $\top$ and strictly bigger than the total number of the memory blocks whose relative ages are possibly younger than this y; if it cannot find a single y, the age $\top$ is used. Note that if such a y exists, it means in any concrete cache set state, without considering m, the cache blocks $\langle l_1, \ldots, l_y \rangle$ are not full of memory blocks yet, since each age position of $\hat{s}_{may}$ contains all the memory blocks that are possibly in that position. Thus, a memory reference to m leads to m having the youngest relative age and there is no memory block being possibly evicted from the $\langle l_1, \ldots, l_y \rangle$ region. Therefore, any memory block's potential maximal age which is already beyond y will not be increased. In order to gain more precision, the H function returns the smallest y, since there may exist more than one y when the cache set is not full. If such a y does not exist, the cache set is possibly full, so every memory block's potential maximal age should be increased and the H function returns $\top$. By this rationale, we can have the safe but more precise updated function $\hat{U}_{Q}^{\text{new}}$ defined as:

$$\hat{U}_{Q}^{\text{new}}(\hat{s}_{pers}, m, h) :=$$
$$\begin{cases} [l_1 \mapsto \{m\}, \\ \ l_2 \mapsto (\hat{s}_{pers}(l_1) \cup \hat{s}_{pers}(l_2)) \backslash \{m\}, \\ \ l_i \mapsto \hat{s}_{pers}(l_i) \backslash \{m\} | i = 3 \ldots \top] & \text{if } \ h = 1 \\ \\ [l_1 \mapsto \{m\}, \\ \ l_i \mapsto \hat{s}_{pers}(l_{i-1}) \backslash \{m\} | i = 1 \ldots h-1, \\ \ l_h \mapsto (\hat{s}_{pers}(l_h) \cup \hat{s}_{pers}(l_{h-1})) \backslash \{m\}, \\ \ l_i \mapsto \hat{s}_{pers}(l_i) \backslash \{m\} | i = h+1 \ldots \top] & \text{else if } \ 1 < h \leq A \\ \\ [l_1 \mapsto \{m\}, \\ \ l_i \mapsto \hat{s}_{pers}(l_{i-1}) \backslash \{m\} | i = 2 \ldots A, \\ \ l_\top \mapsto (\hat{s}_{pers}(l_A) \cup \hat{s}_{pers}(l_\top)) \backslash \{m\}] & \text{otherwise} \end{cases}$$

From the $\hat{U}_{Q}^{\text{new}}$ function we can see if the value of $H(\hat{s}_{may}, m)$ (i.e. h) is not $\top$, we do not need to pessimistically increase the maximal ages of all memory blocks like the one does in the original approach (i.e. the $\hat{U}_{Q}$ function), even if $\hat{s}_{may}$ contains more than or equal to A many other elements than m.

For example, consider Fig. 1 again. When we need to update the corresponding abstract set state at p_5 as shown in Fig. 4 (which is the same as the circled one in Fig. 2) due to the memory reference to m_a in BB_8, $H(\hat{s}_{may}^{p5}, m_a)$ gives $h = 1$:

$$\text{when } y = 1, |\hat{s}_{may}^{p5}(l_1) \backslash \{m_a\}| = |\emptyset| = 0 < y$$

$$\text{when } y = 2, \sum_{i=1}^{2} |\hat{s}_{may}^{p5}(l_i) \backslash \{m_a\}| = |\emptyset| + |\{m_b, m_c\}| = 2 \geq y$$

$$\text{therefore } h = H(\hat{s}_{may}^{p5}, m_a) = \min(\{1\} \cup \{\top\}) = 1$$

which does not alter the state at all since $\hat{U}_{Q}^{\text{new}}(\hat{s}_{pers}^{p5}, m_a, 1)$ applies the first case of the $\hat{U}_{Q}^{\text{new}}$ function. Thus, the reference to m_b in BB_9 can be safely classified as *PS*.

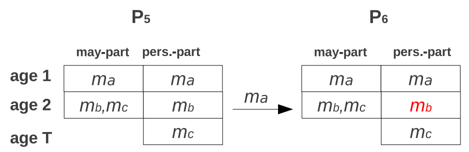

Figure 4. Updating abstract set state at p_5 more precisely

In the following, we use *Orig. May-Pers* to represent the *May-Pers* using the original update function $\hat{U}_{\mathcal{P}}$, and use *Impr. May-Pers* to represent the *May-Pers* using our improved update function $\hat{U}_{\mathcal{P}}^{\text{new}}$.

Theorem 5.1. *The Impr. May-Pers approach is safe, namely at a program point p, any memory block that is loaded into the cache is in an age position of $\hat{s}_{pers}^{p}$ and this age is greater than or equal to the possible maximal age of the block when the execution reaches p (which implies if this block is possibly evicted from the cache, it is in the $\top$ position of $\hat{s}_{pers}^{p}$).*

Proof. The well-developed cache may analysis is safe [1, 8]. Since in *Impr. May-Pers* the may analysis is parallel running independently, its soundness ensures that at a program point p, any memory block that is possibly in the cache is in an age position of $\hat{s}_{may}^{p}$ and this age is smaller than or equal to the possible minimal age of this block. We use this fact to prove this theorem holds at any program point by mathematical induction.

Base case: At the beginning of any execution, we have a cold start such that no memory block is loaded; and we also have an empty $\hat{s}_{pers}$. Therefore, this theorem holds at the beginning.

Inductive hypothesis: At any program point which is immediately before a program point p, this theorem holds.

Inductive step: The program point p can be either a point inside a basic block or a join point of different control flows. We need to prove in either case this theorem holds at p.

- Case 1: the program point p is a point inside a basic block. In this case, p only has one immediately previous program point, say p'. Let us assume a memory block m is accessed at p. Thus, $\langle \hat{s}_{may}^{p}, \hat{s}_{pers}^{p} \rangle$ is $\langle \hat{U}_{\mathcal{M}}(\hat{s}_{may}^{p'}, m), \hat{U}_{Q}^{\text{new}}(\hat{s}_{pers}^{p'}, m, H(\hat{s}_{may}^{p'}, m)) \rangle$. According to the discussion above, we know $\hat{s}_{may}^{p'}$ and $\hat{s}_{may}^{p}$ contains the over-approximated contents at the program point p' and p respectively. Therefore, with the over-approximated contents in $\hat{s}_{may}^{p'}$, according to the rationale of the H function, we know that $H(\hat{s}_{may}^{p'}, m)$ finds a position y which is the upper bound of a region $\langle l_1, \ldots, l_y \rangle$ such that no memory block will be evicted from this region due to m entering the region (note that if y is $\top$, this argument is still valid since no block will be removed from the region $\langle l_1, \ldots, l_\top \rangle$). According to the inductive hypothesis, we know the theorem holds at p', from which we can deduce any block in a position l_x has its possible maximal age at most x. Since $\hat{U}_{Q}^{\text{new}}$ increase the position of a memory block except for m (whose age becomes the youngest) in the region $\langle l_1, \ldots, l_y \rangle$, we can deduce any block is in a position of $\hat{s}_{pers}^{p}$ which is at least its possible maximal age, namely this theorem holds at p.

- Case 2: the program point p is a join point of the exit points of $i \geq 1$ basic blocks, say these exit points are $p'_1, \cdots, p'_i$. According to the inductive hypothesis, we know the theorem holds at $p'_1, \cdots, p'_i$, namely when the execution reaches either one of $p'_1, \cdots, p'_i$, say p', the possible maximal age of any loaded memory block m is at most x given $m \in \hat{s}_{pers}^{p'}(l_x)$. Since no memory block is accessed at a join point and the join function $\hat{J}_{\mathcal{P}}$ uses the maximum relative age of a memory block in $\hat{s}_{pers}^{p_1'}, \cdots, \hat{s}_{pers}^{p_i'}$, we can easily deduce this theorem holds at the join point p.

Combining Case 1 and Case 2, we can see this theorem holds at the program point p. Therefore, we conclude this theorem holds at any program point. □

6. Integration of the Two Approaches

Impr. May-Pers can precisely classify both the memory references to m_a in BB_7 and to m_b in BB_9 in Fig. 1 as *PS*, which cannot be achieved neither by *YS-Pers* nor by *Orig. May-Pers*. However, *Impr. May-Pers* may become imprecise when the overestimated cache contents becomes more conservative at a join point corresponding to a loop head (since may analysis does not distinguish different iterations in a loop).

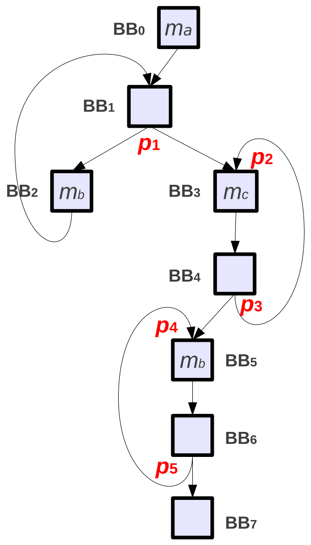

Figure 5. The CFG of another program: all of the references in the loops should be classified as *PS*

Consider the program whose CFG is shown in Fig. 5. We can easily see the memory reference to m_b in BB_5 should be classified as *PS*, since it is not possible to be evicted once it is loaded into the cache. However, as we can observe from Fig. 6 which shows the process of deriving the fixed points of the abstract set states at the five program points marked in Fig. 5, the reference to m_b in BB_5 cannot be classified as *PS*, since from the fixed point of the abstract set state at p_4 we can observe m_b is among the possibly evicted memory blocks before the reference.

The reason for this pessimism is that at the join point p_2 may analysis merges the information of different iterations to form an over-approximation of cache contents for each age position. If using the younger set generation technique as described in section 6.2, m_a can even be conservatively treated as younger than m_b, which is not possible in reality. Therefore, using this $\hat{s}_{may}$, it becomes harder for $H(\hat{s}_{may}, m_c)$ to find a position better than $\top$, which leads to the state with m_b being treated as possibly evicted.

Loop unrolling can eliminate this pessimism but with a very large overhead [2]. Fortunately, we can observe that *YS-Pers* is immune to this pessimism ($ys^{p_2}(m_b) = \{m_c\}$) since the younger block information is combined but cannot be collapsed at join point (m_a can never be younger than m_b and m_c). Thus, we want to integrate *YS-Pers* and *May-Pers* to take advantage of both approaches to further reduce the number of possibly evicted memory blocks.

6.1 Information Exchange between Abstract Domains

Intuitively, we can take advantage of *YS-Pers* and *May-Pers* by running the two methods separately and then classifying a memory reference as *PS* if at least one method can yield such a classification.

However, a more precise approach is to integrate *YS-Pers* and *May-Pers* (*May+YS-Pers*) to form an analysis that runs these two methods in parallel and increase a memory block's potential maximal age if both the methods find its current maximal age is not safe anymore. Thus, we have the abstract domain $E_{\mathcal{P}}$ for cache sets defined as $E_{\mathcal{P}} = D_{\mathcal{P}}^{\text{may-pers}} \times YS$. The join function $\overline{J}_{\mathcal{P}} : E_{\mathcal{P}} \times E_{\mathcal{P}} \rightarrow E_{\mathcal{P}}$ just simply joins corresponding components independently by using their own join functions, so it is defined as:

$$\overline{J}_{\mathcal{P}}(\langle \hat{s}_{may}^{p1}, \hat{s}_{pers}^{p1}, ys^{p1} \rangle, \langle \hat{s}_{may}^{p2}, \hat{s}_{pers}^{p2}, ys^{p2} \rangle) := \langle \hat{J}_{\mathcal{P}}(\langle \hat{s}_{may}^{p1}, \hat{s}_{pers}^{p1} \rangle, \langle \hat{s}_{may}^{p2}, \hat{s}_{pers}^{p2} \rangle), \hat{J}_{\mathcal{YS}}(ys^{p1}, ys^{p2}) \rangle$$

The update function $\overline{U}_{\mathcal{P}} : E_{\mathcal{P}} \times M \rightarrow E_{\mathcal{P}}$ uses our improved update function $\hat{U}_{\mathcal{P}}^{\text{new}}$ on $D_{\mathcal{P}}^{\text{may-pers}}$ and the update function $\hat{U}_{\mathcal{YS}}$ on YS, and is defined as:

$$\overline{U}_{\mathcal{P}}(\langle \hat{s}_{may}, \hat{s}_{pers}, ys \rangle, m) := \langle \hat{s}_{may}^{X}, \overline{XY}_{\mathcal{P}}(\hat{s}_{pers}^{X}, \hat{s}_{pers}^{Y}), ys^{Y} \rangle$$
$$\text{where } ys^{Y} = \hat{U}_{\mathcal{YS}}(ys, m)$$
$$\hat{s}_{pers}^{Y} = G_{\mathcal{P}}(ys^{Y}, set(m))$$
$$\langle \hat{s}_{may}^{X}, \hat{s}_{pers}^{X} \rangle = \hat{U}_{\mathcal{P}}^{\text{new}}(\langle \hat{s}_{may}, \hat{s}_{pers} \rangle, m)$$

and the $\overline{XY}_{\mathcal{P}} : D_{\mathcal{P}} \times D_{\mathcal{P}} \rightarrow D_{\mathcal{P}}$ function is similar to the join function $\hat{J}_{\mathcal{M}}$ for the traditional may analysis, which means it is to select the smaller one between two possible ages for a memory block. The $\overline{XY}_{\mathcal{P}}$ function is defined as:

$$\overline{XY}_{\mathcal{P}}(\hat{s}_{pers}^{X}, \hat{s}_{pers}^{Y}) :=$$
$$[l_i \mapsto \begin{array}{l} \{m | \exists a, b \in [1 \ldots \top] : \\ \quad m \in \hat{s}_{pers}^{X}(l_a) \wedge m \in \hat{s}_{pers}^{Y}(l_b) \wedge i = \min(a, b)\}] \end{array}$$

Since at a program point both $\hat{s}_{pers}^{X}$ and $\hat{s}_{pers}^{Y}$ would have the same set of memory blocks (that corresponds to the set of memory blocks having been referenced so far), we do not need to check if any block is absent from either $\hat{s}_{pers}^{X}$ or $\hat{s}_{pers}^{Y}$.

Basically, the strategy that the update function $\overline{U}_{\mathcal{P}}$ uses to increase the age of a memory block in $\hat{s}_{pers}$ can be described as follows: When a memory reference causes the corresponding abstract set state $\langle \hat{s}_{may}, \hat{s}_{pers}, ys \rangle$ to be updated, ys is updated first. When we need to increase a memory block m's potential maximal age, we compare its current age x in $\hat{s}_{pers}$ with the age y computed from its younger set, i.e. $y = |ys(m)| + 1$: if $x < y$, we increase m's potential maximal age as usual; otherwise, we do not increase its maximal age.

Theorem 6.1. *The May+YS-Pers approach is safe.*

Proof. Since we do not change the join function on each domain, after two safe abstract states are joined, the resultant state is still safe. To see why the new update strategy is also safe, we consider why a memory block m's current safe maximal age x in $\hat{s}_{pers}$ has to be increased when using the update functions $\hat{U}_{\mathcal{P}}^{\text{new}}$: the contents in $\hat{s}_{may}$ show before x the cache is *possibly* full, and a newly inserted younger block *might* make x no longer safe. When using the new update strategy, since we also track and update ys independently, the $y = |ys'(m)| + 1$ is the safe maximal age for m where ys' is the younger set mapping after the effect of the newly referenced memory block is taken into account. Thus, if $y \leq x$, we can guarantee x is still safe from independently updated ys and we do not need to increase x. □

For example, consider Fig. 5 again. As we have seen in Fig. 6, when the second iteration is finished, m_b is not as pessimistic as it

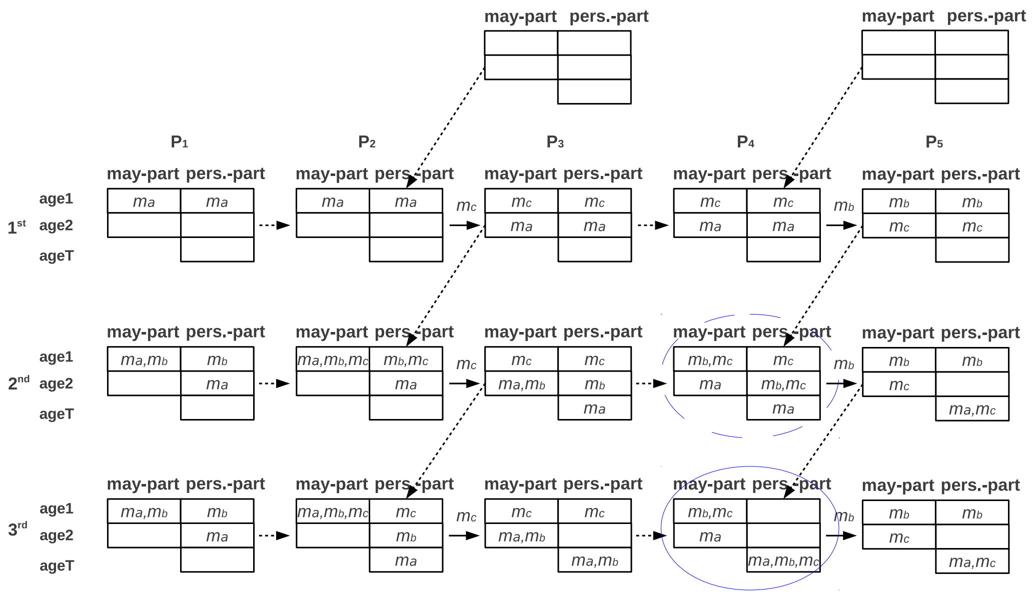

Figure 6. Abstract set states of Fig. 5 computed by the *Impr. May-Pers* approach

is when the fixed points are reached (as shown in the circled states in dashed line and in solid line). When considering the memory reference to m_c in the third iteration, we can observe ys yields $ys(m_a) = \{m_b, m_c\}$, $ys(m_b) = \{m_c\}$, and $ys(m_c) = \emptyset$. Although the $\hat{s}_{pers}$ updating still tries to increase m_b's maximal age to age $\top$, the age computed from $ys(m_b)$ prevents this from happening and cause m_b to stay in age 2. In the end, the fixed point of the abstract set state at p_4 is the same as the one at p_4 of its second iteration (i.e. the circled states will become identical). Thus, the reference to m_b in BB_5 can be classified as *PS*.

6.2 Younger Set Generation

For a memory block m, a less precise younger set (i.e. a bigger superset) can be derived from the abstract set state $\langle \hat{s}^p_{may}, \hat{s}^p_{pers} \rangle$ at a program point p: (1) if m can be found in $\hat{s}^p_{pers}$, i.e. $\exists x \in [1 \dots \top] : m \in \hat{s}^p_{pers}(l_x)$, the potential maximal age of m is x, and each memory block m_a, whose age y in $\hat{s}^p_{may}$ is strictly less than x, i.e. $\exists y \in [1 \dots A] : m_a \in \hat{s}^p_{may}(l_y) \wedge y < x$, is possibly younger than m (since the *may-part* gives the possible minimal age of a memory block); therefore, one of m's possible younger sets is the set of all the memory blocks whose age in $\hat{s}^p_{may}$ is less than m's age in $\hat{s}^p_{pers}$. (2) if m cannot be found in $\hat{s}^p_{pers}$, it means it has never been brought into the cache yet; thus, its younger set does not exist (i.e. $ys^p(m) = \perp$). Formally, we have a younger set generation function $G_\mathcal{Y} : D_\mathcal{P} \times M \to (2^M)_\perp$ which is defined as:

$$G_\mathcal{Y}(\langle \hat{s}_{may}, \hat{s}_{pers} \rangle, m) :=$$
$$\begin{cases} \bigcup\limits_{1 \le i < x} \hat{s}_{may}(l_i) \backslash \{m\} & \text{if } \exists x \in [2 \dots \top] : m \in \hat{s}_{pers}(l_x) \\ \emptyset & \text{if } m \in \hat{s}_{pers}(l_1) \\ \perp & \text{otherwise} \end{cases}$$

If there were no join points in the program, the generated younger sets could be as precise as the tracked ones, but they would use much less memory, since the tracked ones would have some identical younger blocks in more than one of them. However, when a join point is met, a generated younger set may become less precise since some younger information may be collapsed by may analysis.

The disadvantage of using the combination of *YS-Pers* and *May-Pers* is that: an abstract set state $\langle \hat{s}_{may}, \hat{s}_{pers}, ys \rangle$ may contain a lot of redundant information wasting a lot of storage space, since the same younger sets of some memory blocks can be derived from $\langle \hat{s}_{may}, \hat{s}_{pers} \rangle$ using the $G_\mathcal{Y}$ function. In order to decrease this storage overhead, we use two functions to help to compress the size of an abstract set state when saving it and to restore the precise information when using it. The compress function $\overline{C}_\mathcal{P} : E_\mathcal{P} \to E_\mathcal{P}$ is defined as:

$$\overline{C}_\mathcal{P}(\langle \hat{s}_{may}, \hat{s}_{pers}, ys \rangle) := \langle \hat{s}_{may}, \hat{s}_{pers}, \ddot{ys} \rangle \quad \text{where}$$
$$\ddot{ys}(m) := \begin{cases} ys(m) & \text{if } G_\mathcal{Y}(\langle \hat{s}_{may}, \hat{s}_{pers} \rangle, m) \neq ys(m) \\ \perp & \text{otherwise} \end{cases}$$

Thus, at a saving point (e.g. the entry and exit points of a basic block), if a memory block's younger set can be generated from the $\langle \hat{s}_{may}, \hat{s}_{pers} \rangle$ of that point, there is no need to keep it in the saved state. When a saved state needs to be used (e.g. joining several states at a join point), the precise abstract set state can be restored by a restore function $\overline{R}_\mathcal{P} : E_\mathcal{P} \to E_\mathcal{P}$, which is defined as:

$$\overline{R}_\mathcal{P}(\langle \hat{s}_{may}, \hat{s}_{pers}, \ddot{ys} \rangle) := \langle \hat{s}_{may}, \hat{s}_{pers}, ys \rangle \quad \text{where}$$
$$ys(m) := \begin{cases} \ddot{ys}(m) & \text{if } \ddot{ys}(m) \neq \perp \\ G_\mathcal{Y}(\langle \hat{s}_{may}, \hat{s}_{pers} \rangle, m) & \text{otherwise} \end{cases}$$

As mentioned in [11], we do not need to continue tracking a memory block m's younger set when it reaches $|ys(m)| = A$. In our case, if $|G_\mathcal{Y}(\langle \hat{s}_{may}, \hat{s}_{pers} \rangle, m)| \ge A$ and $|ys(m)| \ge A$ both hold, these two sets are considered as equal.

7. Evaluation

We carry out the evaluation on the Mälardalen benchmarks [10], which we compile for the MIPS R3000 architecture. The evaluation is performed by using our research prototype tool, in which we implement all the four mentioned safe methods for cache persistence analysis. At first, we compare the number of instruction memory references which are in the loops and cannot be classified as *AH* but can be classified as *PS*. In order to create enough con-

Table 1. The number of *PS* instructions under cache configurations: 128B/8B/2-way ╱ 256B/8B/4-way (capacity/block size/associativity)

Benchmark	*Orig. May-Pers*	*Impr. May-Pers*	*YS-Pers*	*May+YS-Pers*
adpcm	29/52	48/53	48/53	48/53
bs	6/16	6/18	0/27	6/27
compress	-/16	-/28	-/29	-/29
edn	28/49	44/98	42/118	44/121
expint	-/3	-/23	-/23	-/27
ludcmp	3/5	4/9	3/47	4/48
matmult	0/2	2/2	3/4	3/7
minver	23/41	29/41	25/67	29/73
ns	1/11	2/11	3/27	3/29
prime	-/0	-/2	-/0	-/2
statemate	-/-	-/-	-/-	-/-
ud	1/4	2/8	2/42	2/42

Note: we use "-" to denote every one is 0 to avoid cluttering.

flicts to observe differences between different methods, we utilize very small cache capacities (128B and 256B) in this experiment.

The experimental results are shown in Tab. 1, and the results validate the relationship between different approaches which is described in section 4.3 (see the Venn diagram in Fig. 3): (1) As shown in Tab. 1, the *May+YS-Pers* approach always gives the most number of *PS* references (either "more than" or "as many as"), which shows it dominates other approaches in terms of precision. (2) As we can observe from the results for *bs* under the 128B/8B/2-way configuration, the *Orig. May-Pers* approach can classify more references as *PS* than the *YS-Pers* approach (i.e. 6 references are classified as *PS* by the *Orig. May-Pers* approach but none are classified as *PS* by the *YS-Pers* approach); whereas, under other scenarios, the *YS-Pers* approach is not worse (sometimes much better) than the *Orig. May-Pers* approach. Thus, this shows they are not comparable but empirically in most cases the *YS-Pers* approach is better. (3) However, the *Orig. May-Pers* and the *Impr. May-Pers* approaches are comparable: the *Impr. May-Pers* approach can always classify more number of references as *PS* than (or at least as many as) the *Orig. May-Pers* classifies. (4) In some cases, the *Impr. May-Pers* approach can have more references classified as *PS* than the *YS-Pers* (e.g. *bs*, *edn*, *ludcmp*, and *minver* benchmarks under the 128B/8B/2-way configuration), but in some cases, the *YS-Pers* can give more *PS*. Thus, the *Impr. May-Pers* and the *YS-Pers* are not comparable. Therefore, we can see the relationship shown in Fig. 3 is empirically validated.

The ratio of cache size to loop body size has a direct effect on the usefulness of persistence analysis. From the results for *statemate* benchmark which has a relatively large loop body compared to the cache sizes, we can see that neither of the approaches can classify any reference as *PS*. This is expected, since too many capacity misses in each cache set will evict just referenced instructions soon before they are referenced again. Many of the benchmarks, such as *compress* and *edn*, also contain nested loops which have an effect on the precision of persistence analysis. In the experiments, we do not apply the multi-level method proposed in [2] to deal with the nested loops. Although using the multi-level method can improve the precision of any persistence analysis approach, the relationship between different approaches will still stay the same.

Next, we want to compare how much storage space and analysis time is used by each method. We save two abstract cache states for each basic block (the states of its entry and exit points). In this experiment, we use 512B, 1KB, 2KB, and 4KB capacities with 8B block size and 4-way associativity. Since some of the used benchmarks are relatively small compared to 2KB and 4KB cache capacities, we only show the results of *adpcm* and *statemate*. The relative memory usage is shown in Fig. 7, and the relative analysis time is shown in Fig. 8. The shown result is the ratio of memory (analysis time) used by a method to the corresponding value used by *Orig. May-Pers* under the same configuration.

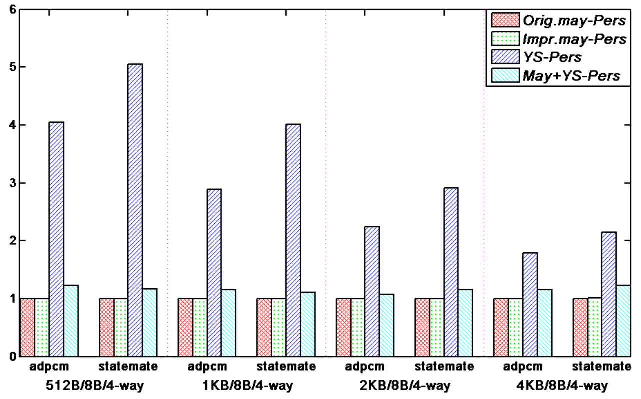

Figure 7. Relative storage space used by *adpcm* and *statemate*

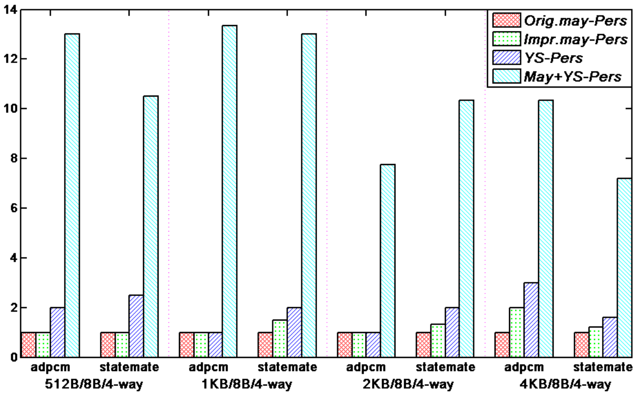

Figure 8. Relative analysis time of *adpcm* and *statemate*

Since there may exist redundant information in *YS-Pers*, we expect it requires more space. However, from Fig. 7, it is interesting to observe that: as the ratio of the total instruction size to the capacity decreases, the ratio of memory used by *YS-Pers* to that used by *Orig. May-Pers* decreases as well. When the cache capacity increases, there are fewer instructions mapped into the same cache set, so each memory block has fewer younger blocks, which means less redundant information for a single memory block. From Fig.

Table 2. memory usage ratio (compressed / uncompressed)

Benchmark	512B/8B/4way	1KB/8B/4way	2KB/8B/4way	4KB/8B/4way
adpcm	0.259	0.326	0.384	0.510
statemate	0.194	0.220	0.320	0.457

8, we can find *May+YS-Pers* requires more analysis time than the other approaches. One reason is the analysis iterates more times, and the other reason is it compresses/restores younger block information when processing a basic block in order to save more memory space. Tab. 2 shows the ratio of memory used in *May+YS-Pers* by using $\overline{C}_{\mathcal{P}}/\overline{R}_{\mathcal{P}}$ to that without using them. As we can see, more than a half memory space can be saved.

8. Conclusion

In this paper, we first analyze the sources of pessimism in two recent state-of-the-art safe persistence analysis methods. After identifying the update function of the may analysis-based approach is too pessimistic, we define a new safe update function for that approach but achieve more precision. We also try to integrate the approaches based on younger set and may analysis together to eliminate more pessimism. Through the evaluations, we can observe the proposed techniques can improve the precision, and there are trade-offs between precision, memory usage, and analysis time (i.e. the more precision, the more time and/or space spent).

Acknowledgment

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

References

[1] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In R. Cousot and D. Schmidt, editors, *Static Analysis*, volume 1145 of *Lecture Notes in Computer Science*, pages 52–66. Springer Berlin Heidelberg, 1996.

[2] C. Ballabriga and H. Casse. Improving the First-Miss Computation in Set-Associative Instruction Caches. In *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, ECRTS '08, pages 341–350, 2008.

[3] S. Chattopadhyay, A. Banerjee, and A. Roychoudhury. Precise Micro-architectural Modeling for WCET Analysis via AI+SAT. In *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, RTAS '13, pages 87–96, 2013.

[4] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, 1977.

[5] C. Cullmann. Cache Persistence Analysis: A Novel Approach Theory and Practice. In *Proceedings of the 2011 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '11, pages 121–130, 2011.

[6] C. Cullmann. Cache Persistence Analysis: Theory and Practice. *ACM Trans. Embed. Comput. Syst.*, 12(1s):40:1–40:25, Mar. 2013.

[7] C. Ferdinand. A fast and efficient cache persistence analysis. Technical report, Universitat des Saarlandes, 1997.

[8] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proceedings of the ACM SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 37–46, 1997.

[9] C. Ferdinand and R. Wilhelm. On Predicting Data Cache Behavior for Real-Time Systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '98, pages 16–30, 1998.

[10] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks – past, present and future. In *WCET 2010*, pages 137–147, 2010.

[11] B. K. Huynh, L. Ju, and A. Roychoudhury. Scope-Aware Data Cache Analysis for WCET Estimation. In *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '11, pages 203–212, 2011.

[12] M. Lv, W. Yi, N. Guan, and G. Yu. Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, RTSS '10, pages 339–349, 2010.

[13] F. Mueller. Timing Analysis for Instruction Caches. *Real-Time Syst.*, 18(2/3):217–247, May 2000.

[14] R. Sen and Y. N. Srikant. WCET Estimation for Executables in the Presence of Data Caches. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software*, EMSOFT '07, pages 203–212, 2007.

[15] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and Precise WCET Prediction by Separated Cache andPath Analyses. *Real-Time Syst.*, 18(2/3):157–179, May 2000.

[16] R. Wilhelm. Why AI+ ILP is good for WCET, but MC is not, nor ILP alone. In *Verification, Model Checking, and Abstract Interpretation*, pages 309–322. Springer, 2004.

[17] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-case Execution-time Problem – Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.