Cache Persistence Analysis for Embedded Real-Time Systems

Dissertation

Zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

von
Christoph Cullmann

Saarbrücken
Februar 2013

Dekan	Prof. Dr. Mark Groves

Prüfungsausschuss

Vorsitzender	Prof. Dr. Sebastian Hack
Berichterstatter	Prof. Dr. Dr. h.c. mult. Reinhard Wilhelm
	Prof. Dr. Jian-Jia Chen
Akademischer Beisitzer	Dr. Michael Feld

Tag des Kolloquiums	14.02.2013

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Saarbrücken, February 25, 2013

Zusammenfassung

Um eine obere Schranke für die Laufzeit eines Programms (WCET) auf einem sicherheitskritischen harten Echtzeit-System zu berechnen, müssen die Effekte der Architektur der zugrunde liegenden Hardware modelliert werden. Die klassische Cache-Analyse unterscheidet drei Kategorien für Speicherreferenzen: *always-hit*, *always-miss* und *not-classified*. Die *Cache-Persistenz-Analyse* versucht, die klassische Cache-Analyse zu verbessern, in dem sie *not-classified* Speicherreferenzen als *persistent* klassifiziert und damit die Zahl der möglichen Cache-Fehlzugriffe beschränkt.

Wir stellen mehrere neuartige auf abstrakter Interpretation basierende Cache-Persistenz-Analysen vor. Zwei basieren auf dem Konzept des Zählens von Konflikten, eine auf der May-Cache Analyse und die letzte kombiniert beide Ansätze miteinander. Alle Analysen korrigieren auch einen Fehler in der ursprünglichen Cache-Persistenz-Analyse von Ferdinand und Wilhelm.

Für *non-fully-timing-compositional Architekturen* ist die Persistenz nicht einfach zu benutzen. Eine neue Pfadanalyse erlaubt die Benutzung der Persistenz auch für aktuelle Architekturen, bei denen sowohl Timing-Anomalien als auch Domino-Effekte auftreten können.

Die vorgestellten Analysen werden innerhalb des industriell verwendeten WCET-Analysators aiT auf einer Reihe von Standard-Benchmark-Programmen und realen Avionic-Anwendungen evaluiert.

Abstract

To compute a worst-case execution time (WCET) estimate for a program running on a safety-critical hard real-time system, the effects of the architecture of the underlying hardware have to be modeled. The classical cache analysis distinguishes three categories for memory references to cached memory: *always-hit*, *always-miss* and *not-classified*. The *cache persistence analysis* tries to classify memory references as *persistent* thereby improving the classical cache analysis by limiting the number of misses for *not-classified* memory references.

We present several new abstract interpretation based cache persistence analyses. Two are based on the concept of conflict counting, one on the may cache analysis, and one combines both concepts. All analyses also fix a correctness issue of the original cache persistence analysis by Ferdinand and Wilhelm.

For *non-fully-timing-compositional architectures* using the persistence information is not straightforward. A novel path analysis enables the use of persistence information also for state-of-the-art architectures that exhibit timing anomalies / domino effects.

The new analyses are practically evaluated within the industrially used WCET analyzer aiT on a series of standard benchmark programs and a series of real avionic examples.

Extended Abstract

Embedded systems are widely used in the domains of avionics, automotive or space systems. Some are safety-critical. Beside functional requirements some also have to fulfill strict timing constraints. To compute a worst-case execution time (WCET) estimate for a program running on such systems, the effects of the hardware architecture have to be modeled. To obtain tight bounds for modern processors, cache and pipeline analyses are needed. The classical must and may cache analyses [Fer97] distinguish three categories for memory references to cached memory: *always-hit*, *always-miss* and *not-classified*. The *cache persistence analysis* tries to classify memory references as *persistent* thereby improving the classical cache analysis by limiting the number of misses for *not-classified* memory references.

We present a correctness issue with the abstract interpretation based cache persistence analysis by Ferdinand and Wilhelm and propose several new abstract interpretation based persistence analyses that fix this issue. Two are based on the concept of conflict counting (per cache set or per element), one on the may cache analysis, and one combines both concepts.

For *non-fully timing compositional architectures* [WGR+09] the persistence information is not straightforward to use. Typical state-of-the-art architectures exhibit both timing anomalies and domino effects. Such architectures do not allow to quantify the costs of a single cache hit or miss in isolation. Our novel path analysis enables the use of persistence information also for state-of-the-art architectures.

The new analyses are practically evaluated within the industrially used WCET analyzer aiT on a series of standard benchmark programs and a series of real avionic examples.

Acknowledgements

I want to thank Prof. Dr. Reinhard Wilhelm for the inspiration to work and research in the WCET analysis field and the fruitful discussions about the topic of this thesis.

I am grateful that AbsInt provided me with the framework I used to implement and evaluate the cache persistence analyses and let me use my time to design the individual analyses during my day-to-day work.

I also want to thank Prof. Dr. Jian-Jia Chen for reviewing this thesis.

Especially the feedback and discussions with Christian Ferdinand, Gernot Gebhard, Florian Martin, Reinhold Heckmann, Markus Pister, Marc Schlickling, Jan Reineke, Daniel Grund, Barbara Dörr and other colleagues at both AbsInt and the compiler construction chair made this thesis feasible at all. Many thanks for this, without this help, this thesis would not have been possible.

Last but not least, I thank my family and friends for their support.

Contents

CHAPTER 1

Introduction

1.1 Motivation

In our modern world, embedded systems become more and more wide-spread. They are also used for safety-critical tasks, like fly-by-wire or airbag controllers. In this safety-critical realm, both functional and timing correctness are required, to ensure correct behavior.

Scheduling analysis is used to ensure the timing constraints of the system are fulfilled. A crucial input for this analysis is the *WCET* (worst case execution time) for all tasks of the system.

During the last decades, static analysis techniques have been designed to compute safe and tight upper bounds of the *WCET*. On this field of *WCET* analysis especially analyses based on abstract interpretation are able to cope with current state-of-the-art hard- and software combinations. There exist tools based on this technology that are successfully used in the certification and verification of avionics software ([SLH+05]).

Modern hardware architectures feature complex pipelines with many performance features to increase the average case performance. One of the main performance limiting factors is the typically rather slow main memory. Therefore, caches are used to hide long memory access latencies. For computing precise WCET estimates of applications running on such architectures, the cache behavior must therefore be included into the analysis. The precision of the cache analysis has a major impact on the precision of the overall computed WCET bound.

A widely-used method is to assign each cache access a category that describes its cache behavior. Common categories are always-miss, always-hit and not-classified. A typical approach to calculate these categories for the cache accesses is to do either a standalone cache analysis or a combined cache and pipeline analysis based on abstract interpretation [FMWA96]. Analyses computing the always-hit and always-miss classifications are, for example, must and may analysis as introduced in [Fer97].

Especially for data-dependent programs not all accesses to the cache can be classified as always-hit or always-miss. Not-classified cases can lead to overestimations. The computed worst-case path might therefore consider more misses than actually possible for any real execution of the program.

An additional cache analysis, introduced as *cache persistence analysis* by Christian Ferdinand [Fer97], tries to limit the number of misses the analysis has to assume for such accesses. The persistence analysis classifies formerly not-classified accesses as persistent if the accessed element will still be in the cache in case it was loaded at a former point in time. Thus at most one miss can occur for accesses to this element in any access sequence. This classification is also known as *first-miss*, introduced by Frank Mueller [MWH94].

In this thesis we will show a correctness issue with the analysis by Ferdinand. We will then introduce several novel persistence analyses to overcome this issue. We will compare their precision and how they relate to other approaches. Then we will evaluate them inside a state-of-the-art static timing analysis framework on real-world applications.

1.2 Thesis Structure & Contributions

In this thesis we first introduce in Chapter 2 the theoretical background of abstract interpretation as used by the later proposed cache analyses.

In Chapter 3 we motivate why cache memories are used in current embedded systems and briefly introduce the concepts of cache analysis, including the well known must and may cache analyses.

In Chapter 4 we present the basic concepts and definitions of the cache persistence analysis and why it is interesting for real-world applications. We will highlight the different code structures that make cache persistence analysis interesting for the typical real-time software.

In Chapter 5 the existing persistence analysis designed by Christian Ferdinand is briefly described and applied to a simple example for better understanding of the basic ideas. Then a correctness problem of this analysis is shown in detail.

Then we present the main contributions of this thesis: several novel persistence analyses that overcome the problem and how they are integrated and evaluated inside the WCET analyzer aiT.

In Chapter 6 we introduce a novel persistence analysis based on counting the conflicting elements inside one cache set. We show its soundness and how it performs on our running examples and the problematic example for the persistence analysis by Ferdinand. A benchmarking framework is introduced to evaluate the precision of the analysis.

In Chapter 7 we improve the conflict counting by not only looking at conflicts per set but conflicts per element inside a set. We will show how this improves the precision for our examples and benchmarks.

In Chapter 8 we sketch a third novel persistence analysis, based on the may cache analysis. We compare it briefly with both conflict based analyses.

In Chapter 9 we design an analysis combining the concepts of all analyses introduced before. This is again evaluated in our benchmarking framework.

All novel persistence analyses will be evaluated in Chapter 10. The integration of the persistence analyses into the WCET analyzer aiT is presented with special account for architectures with timing anomalies. An evaluation demonstrates the applicability of our novel analyses to real industrial examples and architectures.

In Chapter 11 we discuss other state-of-the-art work.

Chapter 12 contains further extensions to the persistence analyses and an outlook to possible future work.

Chapter 13 concludes this thesis.

CHAPTER 2

Abstract Interpretation

The cache analyses presented in this thesis aim to determine properties about memory accesses inside a program: will an access be a cache hit or miss and how many misses can occur overall? Such properties are in general not decidable on the concrete semantics of a program.

A solution for this problem is to use an approximation of the concrete semantics for the computation, trading computability against precision. Given the approximation is sound, the results will hold for the concrete semantics, too.

P. Cousot & R. Cousot introduced in [CC76, CC77] *abstract interpretation* as a theoretical framework to compute such properties based on sound abstract semantics.

First we introduce the program representation and concrete semantics in Section 2.1. In Section 2.2 we briefly describe the abstract semantics. Finally we introduce the analysis framework we use to formulate our abstract interpretation based cache analyses in Section 2.3. We focus on the parts of this theory which we need for the later presented cache analyses. For more details see [CC76, CC77, Rei08, KU77, AM95a].

2.1 Concrete Semantics

2.1.1 Program Representation

In the following we represent a program as a *control-flow graph* and an execution as a *path*.

Definition 2.1 (Control-flow graph)
A control-flow graph $G = (V, E, i)$ *is the representation of a program, where*

- *each node $v \in V$ represents one program statement*
- *each edge $e \in E \subseteq V \times V$ represents a possible control-flow transition*
- *$i \in V$ is the unique start node*

Definition 2.2 (Path)
A sequence $(v_1, ..., v_n) \in V^*$ is a path π *through the control-flow graph $G = (V, E, i)$ iff π is empty or*

- $v_1 = i$
- $\forall k \in \{1, ..., n-1\} : (v_k, v_{k+1}) \in E$

Definition 2.3 (Path to a node v)
A path $\pi = (v_1, ..., v_n) \in V^*$ *through the control-flow graph $G = (V, E, i)$ is a path to $v \in V$ iff $(v_n, v) \in E$.*

2.1.2 Program Semantics

Be the concrete semantics of the program statements over a concrete domain $\mathcal{D}_{conc}$ given by a function $f : V \to \mathcal{D}_{conc} \to \mathcal{D}_{conc}$.

Based on this *concrete transformer* (or *concrete update function*) f we can formulate the *path semantics* for a program.

Definition 2.4 (Path semantics)
The path semantics *of a path $\pi = (v_1, ..., v_n)$ through the control-flow graph $G = (V, E, i)$ for a given concrete transformer $f : V \to \mathcal{D}_{conc} \to \mathcal{D}_{conc}$ is defined as:*

$$[\![\pi]\!]_{conc} := \begin{cases} id & \text{if } \pi \text{ is the empty path} \\ f(v_n) \circ [\![(v_1, ..., v_{(n-1)})]\!]_{conc} & \text{otherwise} \end{cases}$$

To be able to consider not only one individual start state but a set of possible start states for a program, we lift the *concrete semantics* to a *collecting semantics*.

Given a *concrete transformer* f working on single states in $\mathcal{D}_{conc}$ we construct a *collecting transformer* $f_{coll} : V \to \mathcal{D}_{coll} \to \mathcal{D}_{coll}$ working on sets of states $\mathcal{D}_{coll} = 2^{\mathcal{D}_{conc}}$ as follows:

$$\forall v \in V : \forall S \in D_{coll} : f_{coll}(v)(S) := \{f(v)(s) \mid s \in S\}$$

$(\mathcal{D}_{coll}, \subseteq, \bigcup, \bigcap, \emptyset, \mathcal{D}_{conc})$ is a complete lattice. The partial order $\subseteq$ orders states by their precision. If a state a is more precise than b, it contains a subset of the states of b.

The collecting transformer allows us the definition of the *collecting path semantics*.

Definition 2.5 (Collecting path semantics)
The collecting path semantics *of a path $\pi = (v_1, ..., v_n)$ through the control-flow graph $G = (V, E, i)$ for a given collecting transformer $f_{coll} : V \to \mathcal{D}_{coll} \to \mathcal{D}_{coll}$ is defined as:*

$$[\![\pi]\!]_{coll} := \begin{cases} id & \text{if } \pi \text{ is the empty path} \\ f_{coll}(v_n) \circ [\![(v_1, ..., v_{(n-1)})]\!]_{coll} & \text{otherwise} \end{cases}$$

For cache analyses we are interested in the properties of the memory accesses for individual control-flow graph locations. Therefore we define the *sticky collecting semantics* that maps the program points to possible concrete states for that location.

Definition 2.6 (Sticky collecting semantics)
The sticky collecting semantics $Coll : V \to \mathcal{D}_{coll}$ *of a control-flow graph $G = (V, E, i)$ for a set of initial states $Init \in \mathcal{D}_{coll}$ is defined as*

$$Coll(v) := \bigcup \{[\![\pi]\!]_{coll}(Init) \mid \pi \text{ is a path to } v\}$$

The sticky collecting semantics is in general not computable, as:

- The set of initial states $Init$ is often too large or possibly infinite.
- The definition uses the union over all possible path to any program point. This may result in infinitely many paths to follow because of loops or recursions in the program.

2.2 Abstract Semantics

2.2.1 Definition of Abstraction

We now apply the principles of abstract interpretation and transfer our problem from sets of concrete states $\mathcal{D}_{coll}$ to an abstract domain $\mathcal{D}_{abs} = (\mathcal{D}_{abs}, \sqsubseteq)$.

The basic idea is that states inside this abstract domain will represent possibly multiple concrete states and allow us therefore to compute on many states in parallel. Whereas this can lead to a loss of precision, it allows us to make the problems we are interested in computable.

The meaning of an element in the abstract domain is defined by a monotone *concretization function* $conc : \mathcal{D}_{abs} \rightarrow \mathcal{D}_{coll}$. The relation of a set of concrete elements to the abstract domain is defined by an *abstraction function* $abs : \mathcal{D}_{coll} \rightarrow \mathcal{D}_{abs}$.

Like in the concrete setting of the collecting semantics, the partial order $\sqsubseteq$ on $\mathcal{D}_{abs}$ can be used as an order by precision. If $a \sqsubseteq b$, a will comprise more precise information than b, as a will be the abstraction of fewer concrete states than b.

Instead of the collecting transformer, we now use a monotone *abstract transformer* $f_{abs} : V \rightarrow \mathcal{D}_{abs} \rightarrow \mathcal{D}_{abs}$ that works directly on the abstract domain to express the semantics of the program.

This leads to the definition of the *abstract collecting path semantics* and *abstract sticky collecting semantics* as follows:

Definition 2.7 (Abstract collecting path semantics)
The abstract collecting path semantics *of a path* $\pi = (v_1, ..., v_n)$ *through the control-flow graph* $G = (V, E, i)$ *for a given abstract transformer* $f_{abs} : V \rightarrow \mathcal{D}_{abs} \rightarrow \mathcal{D}_{abs}$ *is defined as:*

$$[\![\pi]\!]_{abs} := \begin{cases} id & \text{if } \pi \text{ is the empty path} \\ \\ f_{abs}(v_n) \circ [\![(v_1, ..., v_{(n-1)})]\!]_{abs} & \text{otherwise} \end{cases}$$

Definition 2.8 (Abstract sticky collecting semantics)
The abstract sticky collecting semantics $Abs : V \rightarrow \mathcal{D}_{abs}$ *of a control-flow graph* $G = (V, E, i)$ *for an abstract initial state* $Init_{abs} \in \mathcal{D}_{abs}$ *and an abstract transformer* $f_{abs} : V \rightarrow \mathcal{D}_{abs} \rightarrow \mathcal{D}_{abs}$ *is defined as*

$$Abs(v) := \bigsqcup \{[\![\pi]\!]_{abs}(Init_{abs}) \mid \pi \text{ is a path to } v\}$$

Compared with the concrete sticky collecting semantics, this definition has no issue with a potentially infinite large initial state space. Still the union over all paths may not be computable in practice. Because of this (possibly infinite) union, it is also known as the *meet over all paths solution* (*MOP*).

2.2.2 Soundness

To argue over the soundness of the previously defined abstract semantics, we need the concepts of *local consistency* and *strongly adjoint functions*.

Definition 2.9 (Local consistency)
f_{coll} and f_{abs} are locally consistent, *iff*

$$\forall s \in \mathcal{D}_{coll} : f_{coll}(s) \subseteq conc(f_{abs}(abs(s)))$$

Definition 2.10 (Strongly adjoint)
conc and abs are strongly adjoint, *iff*

$$\forall s \in \mathcal{D}_{coll} : s \subseteq conc(abs(s))$$
$$\forall s \in \mathcal{D}_{abs} : s = abs(conc(s))$$

Given these definitions, the soundness of the abstraction for the sticky collecting semantics follows from ([CC77]):

Theorem 2.1 (Soundness of abstract sticky collecting semantics)
If an abstract interpretation (see Figure 2.1) satisfies:

- *$(\mathcal{D}_{coll}, \subseteq, \bigcup, \bot_{coll})$ and $(\mathcal{D}_{abs}, \subseteq, \bigcup, \bot_{abs})$ are complete join semi lattices,*
- *abs and conc are monotone and strongly adjoint,*
- *the abstract transformer f_{abs} is locally consistent with the concrete transformer f_{coll}*

then $Coll \subseteq conc(Abs)$, i.e. the abstraction of the sticky collecting semantics is sound.

$$\begin{tikzcd}
s_{abs} \arrow[r, "f_{abs}"] & f_{abs}(s_{abs}) \arrow[d, "conc"] \\
 & \rotatebox{90}{$\subseteq$} \\
s \arrow[uu, "abs"] \arrow[r, "f_{coll}"] & f_{coll}(s_{abs})
\end{tikzcd}$$

Figure 2.1: Abstract Interpretation

2.3 Analysis Framework

In the scope of this thesis, we use the *program analyzer generator* (PAG) to specify our cache analyses [AM95b]. This framework allows to ignore the implementation details of the analysis and focus on the essential parts needed for the abstract interpretation:

- the abstract domain
- the abstract transformer
- the abstract join function

Given these elements, PAG will generate an analyzer that computes an approximation of the abstract sticky collecting semantics. This solution is known as the *maximal fixed point solution* (*MFP*). Such an approximation is necessary, as the *MOP* solution of the abstract sticky collecting semantics is in general not computable.

Definition 2.11 (MFP, maximal fixed point solution)
The maximal fixed point solution $MFP : V \to \mathcal{D}_{abs}$ *of a control-flow graph* $G = (V, E, i)$ *for an abstract initial state* $Init_{abs} \in \mathcal{D}_{abs}$ *and an abstract transformer* $f_{abs} : V \to \mathcal{D}_{abs} \to \mathcal{D}_{abs}$ *is the least fixed point of the functional* $F : (V \to \mathcal{D}_{abs}) \to (V \to \mathcal{D}_{abs})$

$$F(f)(v) := \begin{cases} Init_{abs} & \text{if } v = i \\ \bigsqcup\{f_{abs}(w)(f(w)) \mid (w, v) \in E\} & \text{otherwise} \end{cases}$$

The correctness of the MFP has been shown in [AM95a, KU77].

In addition, PAG provides the *VIVU* [MAWF98] (*virtual inlining & virtual unrolling*) extension to the well-known *callstring-approach* [SP81] for interprocedural data-flow analysis. Loops inside the control-flow are transformed to tail-recursive routines and the first k iterations of a loop can be distinguished from any further iteration.

2.4 Summary

In this chapter we introduced the theoretical foundation and the analysis framework for the later described cache analysis based on abstract interpretation. In the next chapter we will now introduce the concept of cache analysis and then focus on the persistence analysis.

CHAPTER 3

Cache Memories & Cache Analysis

In this chapter we show why caches are important today and introduce basic notions of cache analysis. For a more complete overview of cache memories and analysis refer to [Fer97, Rei08].

3.1 Why Use Caches?

There are today techniques to build efficient and cheap fast processors in the multi-gigahertz area, but we can't afford to do the same for their main memories. The gap between processor and memory speed is ever increasing, as shown in Figure 3.1 [MV99].

Therefore caches are used to hide the long access latencies of the available memory techniques (like the different instances of *DRAM* or slow *FLASH* memories) to still realize a reasonable performance level. Figure 3.2 shows a typical setup for current high-performance embedded systems with respect to their memory system, like e.g., found in the *MPC755*.

Caches are used to hide memory latencies by exploiting the fact that programs tend to reuse memory locations they have used recently. This principle of *locality* was described in [Hai86]. Hennessy and Patterson quantified this effect in [HP96]. They conclude that programs in general spend 90% of their execution time in only 10% of the code.

There are two main aspects of locality:

Spatial Locality: It is likely that memory locations whose addresses are close to each other will be used close together in time.

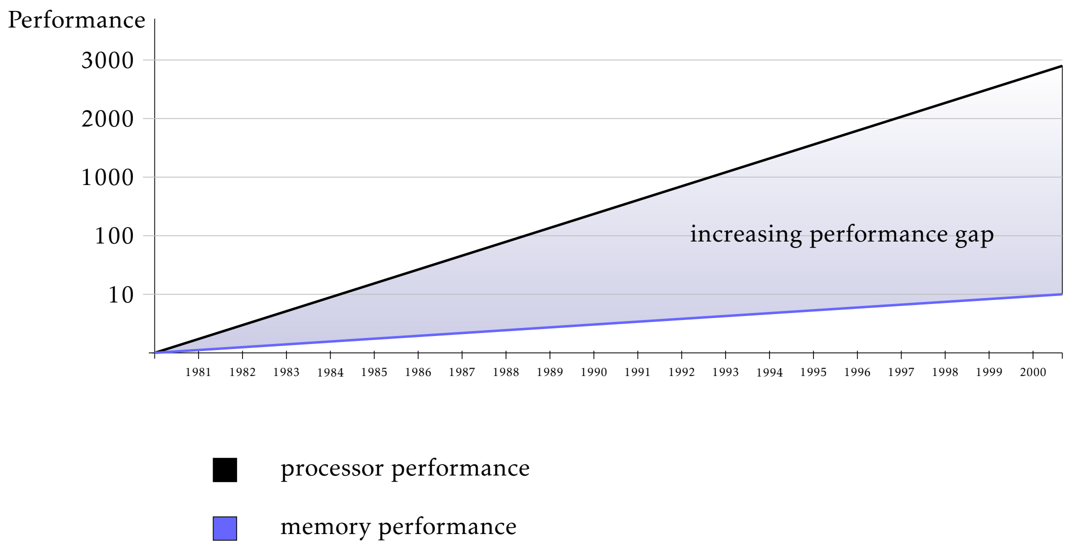


Figure 3.1: Simplified processor and memory performance evolution over 20 years, as presented by Mahapatra et al. in [MV99].

Temporal Locality: It is likely that memory locations that were used recently will soon be reused.

For a tight WCET estimation, we need to analyze the cache behavior. Alfred Roßkopf of EADS did a study in this field showing that a *PowerPC* 604 running at 300MHz with caches disabled delivers a similar performance on some benchmarks as a *Motorola* 68020 running at 20MHz, while the *PowerPC* outperforms the 68020 by a factor of 20 with caches [LTH02].

3.2 Cache Memory Parameters

A cache memory can be characterized by three parameters:

Capacity The *capacity* is the number of bytes the cache may contain.

Line Size The *line size* is the number of contiguous bytes that are transferred from memory on a cache miss. The cache can hold at most $n = capacity/line\ size$ blocks.

Associativity The *associativity* is the number of cache locations where a particular block may reside. *n/associativity* is the number of *sets* of a cache. A *set* can be considered as a fully associative sub-cache.

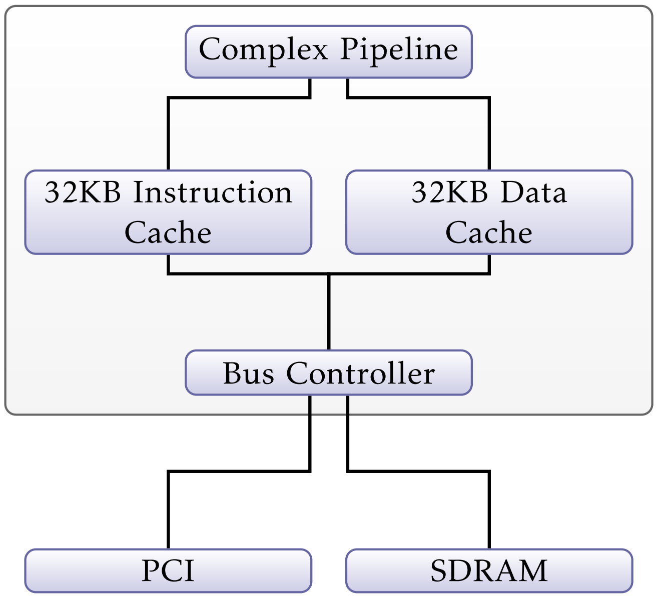


Figure 3.2: Memory hierarchy of the *MPC755*.

Depending on the associativity, three classifications of caches [Smi82] are commonly used:

Fully Associative Caches If a block can reside in any cache location, then the cache is called *fully associative*.

Direct Mapped Caches If a block can reside in exactly one location, then the cache is called *direct mapped*.

***A*-Way Set-Associative Caches** If a block can reside in exactly A locations, then the cache is called *A-way set-associative*.

A memory block has to be selected for replacement when the cache set is full and the processor requests further data. This is done according to a replacement strategy.

Common strategies are *LRU* (Least Recently Used), *PLRU* (Pseudo Least Recently Used), *FIFO* (First In First Out), and *random*. For more details about the individual strategies and how they compare with respect to static cache analysis see [RGBW07] and [Rei08].

3.3 Concrete Semantics

3.3.1 Program Representation

We represent programs by control-flow graphs consisting of nodes and edges as described in Definition 2.1. The nodes represent *basic blocks*. For each basic block, the sequence of references to memory is known, i.e., there exists a mapping from control flow nodes to sequences of references to memory blocks: $\mathcal{L} : V \to M^*$.

Definition 3.1 (Control-flow graph with memory references)
A control-flow graph with memory references *is the tuple $G_\mathcal{L} = (G, \mathcal{L})$ of a control-flow graph $G = (V, E, i)$ and the mapping of nodes to sequences of references to memory blocks $\mathcal{L} : V \to M^*$.*

To ease the description of semantics for individual memory references, we simplify this graph by transforming it into a graph with at most one memory reference per node.

Definition 3.2 (Control-flow graph with single memory references)
Given a control-flow graph with memory references $G_\mathcal{L} = ((V, E, i), \mathcal{L})$. We define the control-flow graph with single memory references *as the graph $G_\mathcal{L} = ((V', E', i), \mathcal{L}')$ constructed by splitting all nodes that map to multiple memory references to nodes with only one memory reference and introducing intermediate edges between them.*

For all later cache semantics and analysis descriptions, we will use this *control-flow graph with single memory references* as control-flow graph concept.

3.3.2 Cache Semantics

We restrict our description to the semantics of A-way set-associative caches with LRU replacement strategy. The fully associative and the direct mapped caches are special cases of the A-way set-associative cache where $A = n$ and $A = 1$, respectively.

In the following, we consider an A-way set-associative cache as a sequence of (fully associative) sets $F = (f_1, \ldots, f_{n/A})$, a set f_i as a sequence of set lines $L = (l_1, \ldots, l_A)$, and the store as a set of memory blocks $M = \{m_1, \ldots, m_s\}$. To indicate the absence of any memory block in a set line, we introduce a new element I and define $M' = M \cup \{I\}$.

Our cache semantics separates two key aspects:

- The set where a memory block may be stored:
  This can be determined statically as it depends only on the address of the memory block. The function mapping a block to its set is called $set : M \rightarrow F$. The dynamic allocation of memory blocks to the lines of a set is modeled by *cache states*.

- The aspect of associativity and the replacement strategy within one set of the cache:
  Here, the history of memory reference executions is relevant. This is modeled by *set states*.

Definition 3.3 (Concrete set state)
A (concrete) set state is a function $s : L \rightarrow M'$. S denotes the set of all concrete set states.

Definition 3.4 (Concrete cache state)
A (concrete) cache state is a function $c : F \rightarrow S$. C denotes the set of all concrete cache states.

If $s(l_x) = m$ for a concrete set state s, then x describes the relative age of the memory block according to the LRU replacement strategy and not the physical position in the cache hardware.

The most recently referenced memory block is put in the first position l_1 of the set. If the referenced memory block m was in the set before, then all memory blocks in the set that have been more recently used than m are shifted by one position to the next set line, i.e., they increase their relative age by one. If the memory block m is not yet in the set, then all memory blocks in the set are shifted and the 'oldest', i.e., least recently used memory block is removed from the set.

Definition 3.5 (Memory block in the cache state)
Given a concrete cache state $c \in C$. A memory block $m \in M$ is in the cache state *c iff $\exists l \in L : m = (c(set(m)))(l)$.*

The *update* function describes the side effects of accessing cached memory on the set/cache state:

- The set where a memory block may reside in the cache is uniquely determined by the address of the memory block. Therefore the behavior of the sets is independent of each other.

- The LRU replacement strategy is modeled by using the positions of memory blocks within a set to indicate their relative age. The order of the memory blocks reflects the *history* of memory references.

Definition 3.6 (Set update)
A set update function $\mathcal{U}_S : S \times M \to S$ describes the new set state for a given set state and a referenced memory block.

Definition 3.7 (Cache update)
A cache update function $\mathcal{U}_C : C \times M \to C$ describes the new cache state for a given cache state and a referenced memory block.

Updates of fully associative sets with LRU replacement strategy are modeled in the following way:

$$\mathcal{U}_S(s,m) = \begin{cases} [l_i \mapsto s(l_i) \mid i = 1 \mathinner{..} A]; & \text{if } m = s(l_1) \\ \\ [l_1 \mapsto m, & \\ l_i \mapsto s(l_{i-1}) \mid i = 2 \mathinner{..} h, & \text{if } \exists l_h : m = s(l_h) \wedge h > 1 \\ l_i \mapsto s(l_i) \mid i = h+1 \mathinner{..} A]; & \\ \\ [l_1 \mapsto m, & \text{otherwise} \\ l_i \mapsto s(l_{i-1}) \mid i = 2 \mathinner{..} A]; & \end{cases}$$

Notation:

- $[y \mapsto z]$ denotes a function that maps y to z.
- $f[y \mapsto z]$ denotes a function that maps y to z and all $x \neq y$ to $f(x)$.

Updates of A-way set-associative caches are modeled as follows:

$$\mathcal{U}_C(c,m) = c[\text{set}(m) \mapsto \mathcal{U}_S(\text{c}(set(m)), m)]$$

We describe the semantics of a concrete cache with the help of the above update function $\mathcal{U}_C$. We can extend $\mathcal{U}_C$ to the concrete cache semantics in a *control-flow graph with single memory references*.

Definition 3.8 (Concrete cache semantics)
Given a cache update function $\mathcal{U}_C : C \times M \to C$ and a control-flow graph with single memory references $G_{\mathcal{L}} = (G, \mathcal{L})$. We define the concrete cache semantics $\mathcal{U}_{C_N} : V \to C \to C$ *as follows:*

$$\mathcal{U}_{C_N}(n,c) := \begin{cases} c & \text{if } \mathcal{L}(n) \text{ is empty} \\ \mathcal{U}_C(c, \mathcal{L}(n)) & \text{otherwise} \end{cases}$$

Definition 3.9 (Cache path semantics)
The cache path semantics *of a path $\pi = (v_1, \ldots, v_n)$ through the control-flow graph with single memory references $G_{\mathcal{L}} = (G, \mathcal{L})$ for a given cache semantics $\mathcal{U}_{C_N} : V \to C \to C$ is defined as $[\![\pi]\!]_{\mathcal{U}_{C_N}}$.*

3.4 Abstract Semantics

The domain for our abstract interpretation consists of *abstract cache states* that are constructed from *abstract set states*:

Definition 3.10 (Abstract set state)
An abstract set state $\hat{s} : L \to 2^{M'}$ maps set lines to sets of memory blocks. $\hat{S}$ denotes the set of all abstract set states.

Definition 3.11 (Abstract cache state)
An abstract cache state $\hat{c} : F \to \hat{S}$ maps sets to abstract set states. $\hat{C}$ denotes the set of all abstract cache states.

The abstract semantics functions describe the effect of a memory reference on an element of the abstract domain. The *abstract set (cache) update* function $\hat{\mathcal{U}}$ for abstract set (cache) states is an extension of the set (cache) update function $\mathcal{U}$ to abstract set (cache) states.

Definition 3.12 (Abstract update function)
An abstract update function $\hat{\mathcal{U}} : \hat{C} \times M \to \hat{C}$ describes the new abstract cache state for a given abstract cache state and a referenced memory block.

Definition 3.13 (Abstract cache semantics)
Given an abstract cache update function $\hat{\mathcal{U}} : \hat{C} \times M \to \hat{C}$ and a control-flow graph with single memory references $G_{\mathcal{L}} = (G, \mathcal{L})$. We define the abstract cache semantics

$\mathcal{U}_{C_N} : V \to \hat{C} \to \hat{C}$ as follows:

$$\mathcal{U}_{C_N}(n, \hat{c}) := \begin{cases} \hat{c} & \text{if } \mathcal{L}(n) \text{ is empty} \\ \hat{\mathcal{U}}(\hat{c}, \mathcal{L}(n)) & \text{otherwise} \end{cases}$$

On control flow nodes with at least two predecessors, *join*-functions are used to combine the abstract cache states.

Definition 3.14 (Abstract join function)
A join function $\hat{\mathcal{J}} : \hat{C} \times \hat{C} \to \hat{C}$ combines two abstract cache states.

3.5 Must & May Cache Analyses

The must and may cache analyses as introduced in [Fer97] are used to classify individual memory accesses as cache hits or misses. We will now briefly introduce the individual update and join functions for both analyses.

For better intuition, we apply both analyses to the simple example from Figure 3.3. The example accesses possibly three different memory blocks *a*, *b* and *c* during one iteration which are mapped to the same cache set. We assume a 2-way associative cache for this example. In Figure 3.4 the control-flow graph with single memory references for this example is given.

```
void simpleLoop () {
  for (int i = 0; i < 100; ++i) {
    if (somethingUnknownForEachRound) accessA ();
    else accessB ();
    accessC ();
  }
}
```

Figure 3.3: Program containing one loop which accesses the memory blocks *a*, *b* and *c*.

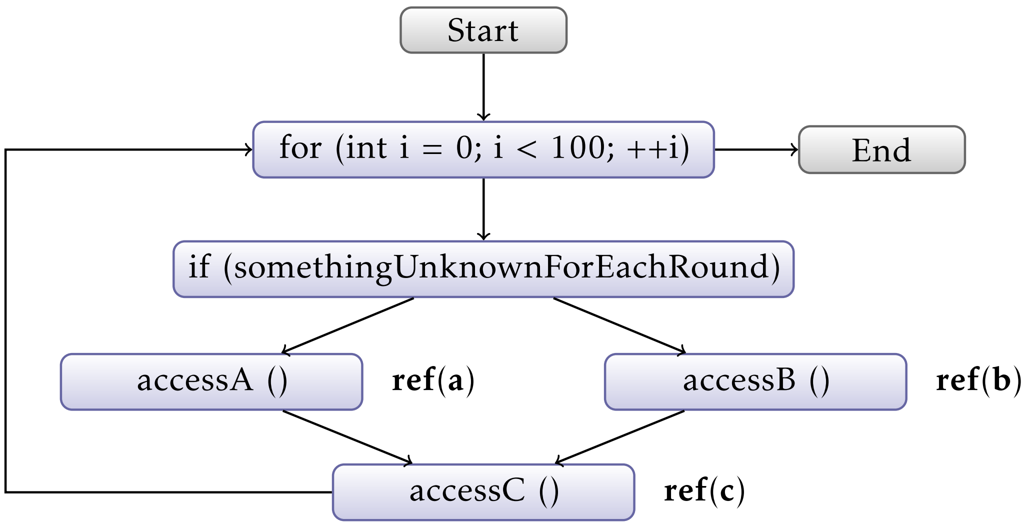

Start
for (int i = 0; i < 100; ++i)
End
if (somethingUnknownForEachRound)
accessA ()
ref(a)
accessB ()
ref(b)
accessC ()
ref(c)

Figure 3.4: Control-flow graph with single memory references for the program from Figure 3.3. Nodes with a memory reference to a memory block $x \in M$ are annotated with $\mathit{ref}(x)$.

3.5.1 Must Cache Analysis

The must cache analysis computes an under-approximation of the cache contents and maps cache lines to an upper bound of their ages in the cache. Given a cache line is in the must set, an access to it will result in a sure cache hit.

Definition 3.15 (Sure Hit Classification)
Given a control-flow graph with single memory references $((V, E, i), \mathcal{L})$, a concrete cache semantics $\mathcal{U}_{C_N} : V \to C \to C$ and a set of initial cache states $S \in 2^C$. A memory reference to memory block $m \in M$ in node $n \in V$ $(\mathcal{L}(n) = (m))$ is classified as a sure hit *iff for all paths π through the graph to n and for all initial cache states $c \in S$ it holds that m is in the cache state $[\![\pi]\!]_{\mathcal{U}_{C_N}}(c)$.*

Definition 3.16 (Set Update and Join – Must Cache Analysis)
Update and join function of the must cache analysis *for an abstract cache set are defined in Figure 3.5.*

Figure 3.6 shows the results for applying the must analysis to one round of the loop from Figure 3.3.

After one round, the must analysis can only classify hits for the cache line c, as the other cache lines are not accessed on all paths.

$$U_{must}(m,x) := \begin{cases} [l_i \mapsto m(l_i) \mid i = 1 \,..\, A]; & \text{if } x \in m(l_1) \\[2em] \begin{aligned}&[l_1 \mapsto \{x\}, \\ &l_i \mapsto m(l_{i-1}) \mid i = 2 \,..\, h-1, \\ &l_h \mapsto m(l_{h-1}) \cup (m(l_h) \setminus \{x\}), \\ &l_i \mapsto m(l_i) \mid i = h+1 \,..\, A];\end{aligned} & \text{if } \exists l_h : x \in m(l_h) \land h > 1 \\[2em] \begin{aligned}&[l_1 \mapsto \{x\}, \\ &l_i \mapsto m(l_{i-1}) \mid i = 2 \,..\, A];\end{aligned} & \text{otherwise} \end{cases}$$

$$J_{must}(m,m') := [l_i \mapsto \{x \mid \exists l_a, l_b : x \in m(l_a) \land x \in m'(l_b) \land i = \max(a,b)\}]$$

Figure 3.5: The update (U_{must}) and join (J_{must}) functions of the must cache analysis for an abstract cache set.

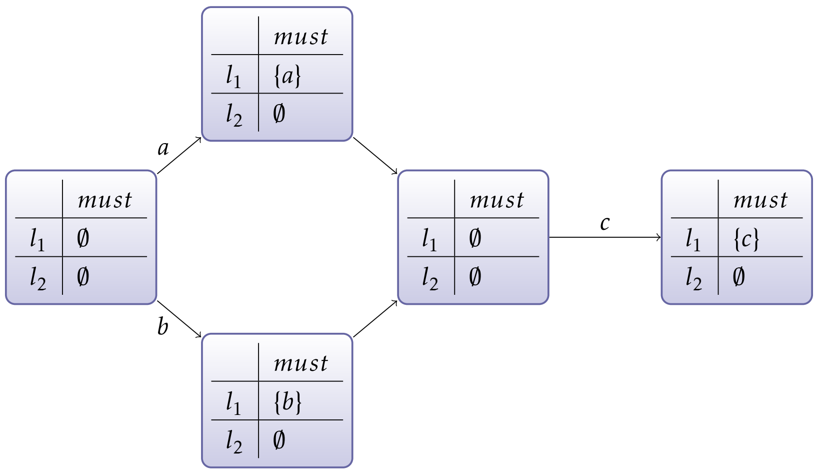


Figure 3.6: Must cache analysis for one round of the loop from Figure 3.3 starting with empty must cache.

If we would analyze the loop trivially without any unrolling, the join at the loop head will eliminate this information again and no hits will be classified in any iteration. If the loop is at least unrolled twice (see VIVU approach [MAWF98]), starting from the second round, one can predict the hits to *c*.

3.5.2 May Cache Analysis

The may cache analysis computes an over-approximation of the cache contents and maps cache lines to a lower bound of their ages in the cache. If a cache line is not in the may set, an access to it will result in a sure cache miss.

Definition 3.17 (Sure Miss Classification)
Given a control-flow graph with single memory references $((V, E, i), \mathcal{L})$, a concrete cache semantics $\mathcal{U}_{C_N} : V \to C \to C$ and a set of initial cache states $S \in 2^C$. A memory reference to memory block $m \in M$ in node $n \in V$ ($\mathcal{L}(n) = (m)$) is classified as a sure miss *iff for all paths π through the graph to n and for all initial cache states $c \in S$ it holds that m is not in the cache state $[\![\pi]\!]_{\mathcal{U}_{C_N}}(c)$.*

Definition 3.18 (Set Update and Join – May Cache Analysis)
Update and join function of the may cache analysis *for one cache set are defined in Figure 3.7.*

$$
U_{may}(m, x) := \begin{cases}
\begin{array}{l}
[l_1 \mapsto \{x\}, \\
l_i \mapsto m(l_{i-1}) \mid i = 2 \mathinner{..} h, \\
l_{h+1} \mapsto m(l_{h+1}) \cup (m(l_h) \setminus \{x\}), \\
l_i \mapsto m(l_i) \mid i = h+2 \mathinner{..} A];
\end{array} & \text{if } \exists l_h : x \in m(l_h) \land h < A \\[2em]
\begin{array}{l}
[l_1 \mapsto \{x\}, \\
l_i \mapsto m(l_{i-1}) \mid i = 2 \mathinner{..} A];
\end{array} & \text{otherwise}
\end{cases}
$$

$$
J_{may}(m, m') := [l_i \mapsto \begin{array}{ll}
 & \{x \mid \exists l_a, l_b : x \in m(l_a) \land x \in m'(l_b) \land i = \min(a, b)\} \\
\cup & \{x \mid x \in m(l_i) \land \nexists l_a : x \in m'(l_a)\} \\
\cup & \{x \mid x \in m'(l_i) \land \nexists l_a : x \in m(l_a)\}]
\end{array}
$$

Figure 3.7: The update (U_{may}) and join (J_{may}) functions of the may cache analysis for an abstract cache set.

Figure 3.8 shows the results for applying the may analysis to one round of the loop from Figure 3.3.

After only one round, the may cache analysis can no longer classify any sure misses as all accessed lines may be in the cache.

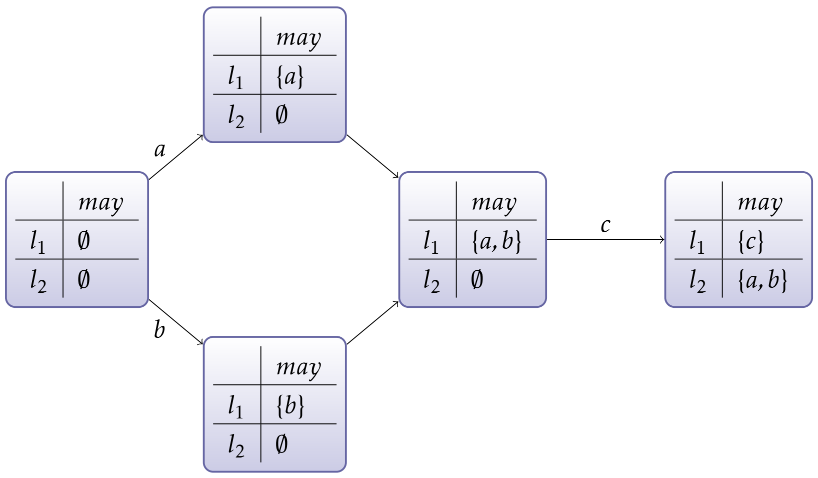


Figure 3.8: May cache analysis for one round of the loop from Figure 3.3 starting with empty may cache.

Any unrolling can't improve this situation, as the program might access any of these three cache lines per round and not more than two in one round. Therefore the may analysis can't prove any eviction in the selected analysis setting with two-way associativity.

3.6 Summary

In this chapter we explained why current real-time systems use cache memories and how they work. We briefly described the basic notions of concrete and abstract cache semantics as later used in this thesis. We have shown the must and may cache analysis.

CHAPTER 4

Cache Persistence Analysis

4.1 Motivation

We introduced in the previous Chapter 3 the concepts of cache analysis and the must and may cache analysis. Now let us look at the program from Figure 4.1.

```
void ifThenElseLoop () {
  for (int i = 0; i < 4; ++i) {
    if (somethingUnknownForEachRound) accessA ();
    else accessB ();
  }
}
```

Figure 4.1: Loop with if-then-else construct either loading memory block *a* or *b* depending on a condition not known statically.

In this code snippet, the loop is executed four times. Depending on some statically unknown condition *somethingUnknownForEachRound* either the function *accessA* or *accessB* is executed which will either load the memory block *a* or *b*. We assume we use a 2-way associative cache with LRU replacement policy and both accessed blocks map to the same set.

As the value of the condition is not statically known, a static analysis only knows that after any loop iteration both memory blocks *a* or *b* *may* be loaded but doesn't know which of them *must* be loaded. Therefore the static cache analysis has to assume that in each of the four rounds of the loop the accesses to *a* or *b* might miss the cache. This would result in four possible misses for this loop. Figure 4.2 shows the *control flow graph with single memory references* for this example and

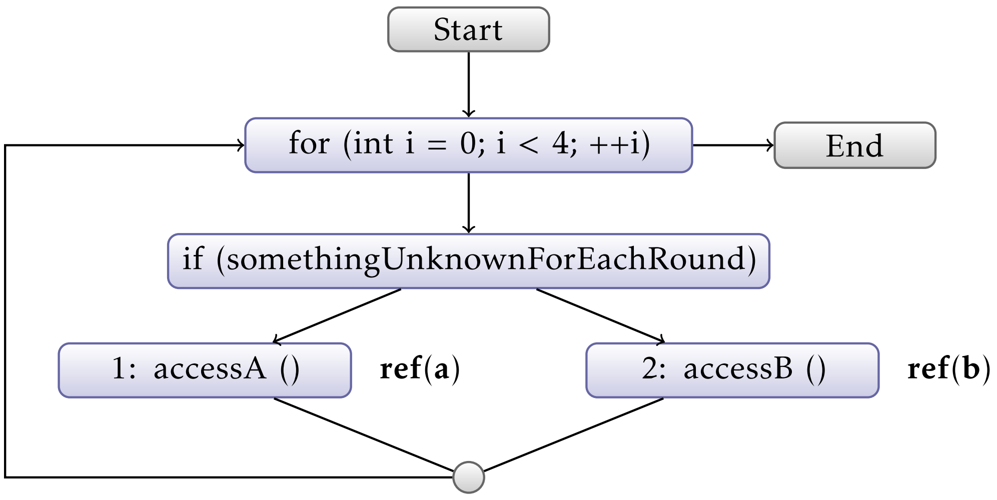

Start
for (int i = 0; i < 4; ++i)
End
if (somethingUnknownForEachRound)
1: accessA ()
ref(a)
2: accessB ()
ref(b)

Figure 4.2: Control-flow graph with single memory references for the program from Figure 4.1. Nodes with a memory reference to a memory block $x \in M$ are annotated with $ref(x)$.

Figure 4.3 and 4.4 show the must and may cache analysis applied to one round of the body. With only the must and may cache analysis a more precise result is not possible even if advanced techniques like virtual loop unrolling (see [MAWF98]) are used.

Intuitively it is clear that as the program accesses only two elements and we have a 2-way cache, after the first load of a and b all further executed memory references to a and b will be hits in the cache. The aim of cache persistence analysis is to compute exactly this *persistence* [Fer97] or *first-miss* [MWH94] classification to limit the number of possible misses to one per element in this example.

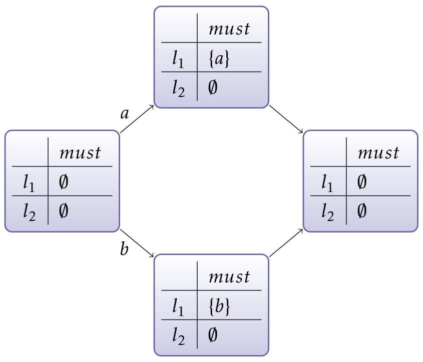


Figure 4.3: Must cache analysis for the loop inside the control-flow graph from Figure 4.2 starting with empty must cache until fixed point is reached.

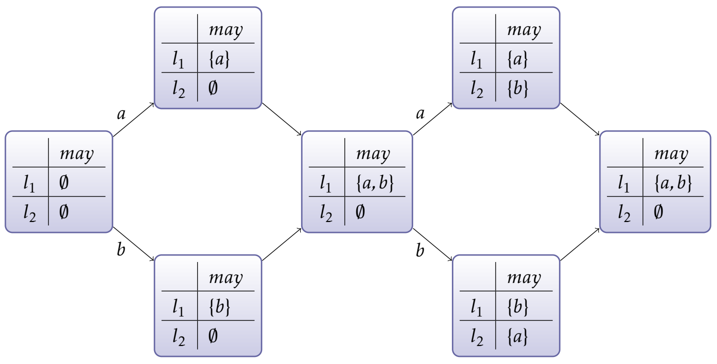


Figure 4.4: May cache analysis for the loop inside the control-flow graph Figure 4.2 starting with empty may cache until fixed point is reached.

4.2 Cache Persistence

In the previous section we presented an example that motivates the introduction of an additional analysis complementing the must and may cache analyses. Now the notion of the *persistence* or *first-miss* classification will be formalized.

Definition 4.1 (Persistence)
Given a control-flow graph with single memory references $((V,E,i),\mathcal{L})$, a concrete cache semantics $\mathcal{U}_{C_N} : V \to C \to C$ and a set of initial cache states $S \subseteq C$. A memory reference to memory block $m \in M$ in node $n \in V$ $(\mathcal{L}(n) = (m))$ is classified as persistent *iff for all paths $\pi = (v_1,\ldots,v_j)$ through the graph to n and for all initial cache states $c \in S$ one of these conditions holds:*

- *$\nexists v \in \{v_1,\ldots,v_j\} : \mathcal{L}(v) = (m)$ (it is the first reference to m) or*
- *m is contained in the cache state $[\![\pi]\!]_{\mathcal{U}_{C_N}}(c)$ (the reference will cause a cache hit)*

This implies that all memory references classified as *persistent* for the same block m can cause at most one cache miss, as for all beside the first memory reference to m it is guaranteed that m is already in the cache on all paths.

Theorem 4.1 (Persistence)
Given a control-flow graph with single memory references $((V,E,i),\mathcal{L})$, a concrete cache semantics $\mathcal{U}_{C_N} : V \to C \to C$, a set of initial cache states $S \subseteq C$ and a memory block $m \in M$. Let P be the set of all nodes $n \in V$ that contain a memory reference to m which are classified as persistent *for this concrete cache semantics and initial cache states. Then it holds that for any path through the graph the memory references of all nodes in P can cause at most one cache miss.*

Proof 4.1 (Persistence)
Given a control-flow graph with single memory references $G = ((V, E, i), \mathcal{L})$, a concrete cache semantics $\mathcal{U}_{C_N} : V \rightarrow C \rightarrow C$, a set of initial cache states $S \subseteq C$ and a memory block $m \in M$. Let P be the set of all nodes $n \in V$ that contain a memory reference to m which are classified as persistent *for this concrete cache semantics and initial cache states. Let us now regard an arbitrary path π through G. Per Definition 4.1, all instances of nodes from P beside the first instance on the path π must cause a hit. Therefore at most one miss can be caused for the whole path π by all instances of nodes from P.*

We now revisit our example from Figure 4.2. Let us look at the memory reference to a inside the graph. For all paths in the graph to the node containing this memory reference, the condition for the *persistence classification* holds: Either this memory reference is the first reference to memory block a or a is still in the cache. That a stays in the cache after the initial load is guaranteed for any initial concrete cache state at program start as we only access two elements and we use a 2-way LRU cache. For the memory reference to b the same argumentation is valid. Therefore both memory references can be classified as *persistent* and the *persistence constraint* can be applied for these references. This implies that at most one miss can happen for the memory reference to a and one for the memory reference to b on any path.

The above definitions always take into account the complete control-flow graph and all possible paths through it. Let us look at the modified version of our two accesses example from Figure 4.5 that contains two loops which access only two blocks and one third memory block is accessed in between. The control-flow graph with single memory references for this example is shown in Figure 4.6. We use the same cache setup as before and assume that the memory block c maps to the same set as a and b.

As this program potentially accesses three elements but we only have a 2-way cache, for some paths evictions from the cache can happen. Still, intuitively it is clear that inside each loop of this program, like for the example from Figure 4.1, we can limit the number of possible misses to one per memory block.

For the first loop in the program, the *persistence classification* and *constraint* from above can be used to limit the misses. For the second loop, this doesn't work, as there exist paths through the program for which neither the memory reference to a nor b can be classified as *persistent*. For example consider this possible path (only the memory references on the path are shown):

```
void main () {
  for (int i = 0; i < 4; ++i) {
    if (somethingUnknownForEachRound) accessA ();
    else accessB ();
  }
  accessC ();
  for (int i = 0; i < 4; ++i) {
    if (somethingUnknownForEachRound) accessA ();
    else accessB ();
  }
}
```

Figure 4.5: Program with two loops with if-then-else construct either loading memory block *a* or *b* depending on a condition not known statically and one load of a memory block *c* in between.

$$\begin{tikzcd}
\overset{1}{ref(a)} \arrow[r] & \overset{2}{ref(b)} \arrow[r] & \overset{1}{ref(a)} \arrow[r] & \overset{2}{ref(b)} \arrow[r] & \overset{3}{ref(c)} \arrow[r] & \overset{4}{ref(a)} \arrow[r] & \overset{5}{ref(b)} \arrow[r] & \overset{4}{ref(a)} \arrow[r] & \overset{5}{ref(b)}
\end{tikzcd}$$

For any initial concrete cache state at the beginning of the program execution, the memory reference to *a* at 4 will be a cache miss, as *a* got evicted because two other memory elements got loaded. The same holds for the following memory reference to *b*. Therefore these two references that belong to the nodes in the second loop of the control-flow graph can't be classified as persistent, as they are neither hits in the cache nor the first reference to their element on all paths.

Whereas no better global persistence classification is possible, as these memory references in the second loop really can lead to more misses, it makes sense to be able to argue about *persistence* for parts of the control-flow graph. We will call these parts *persistence scopes*.

Definition 4.2 (Persistence Scope)
A persistence scope *for a control-flow graph $G = (V, E, i)$ is a subgraph $S = (V', E', i')$, where*

- $V' \subseteq V$
- $E' = \{(x, x') \mid x \in V' \wedge x' \in V' \wedge (x, x') \in E\}$
- *$i' \in V'$ is the unique entry node for the subgraph*

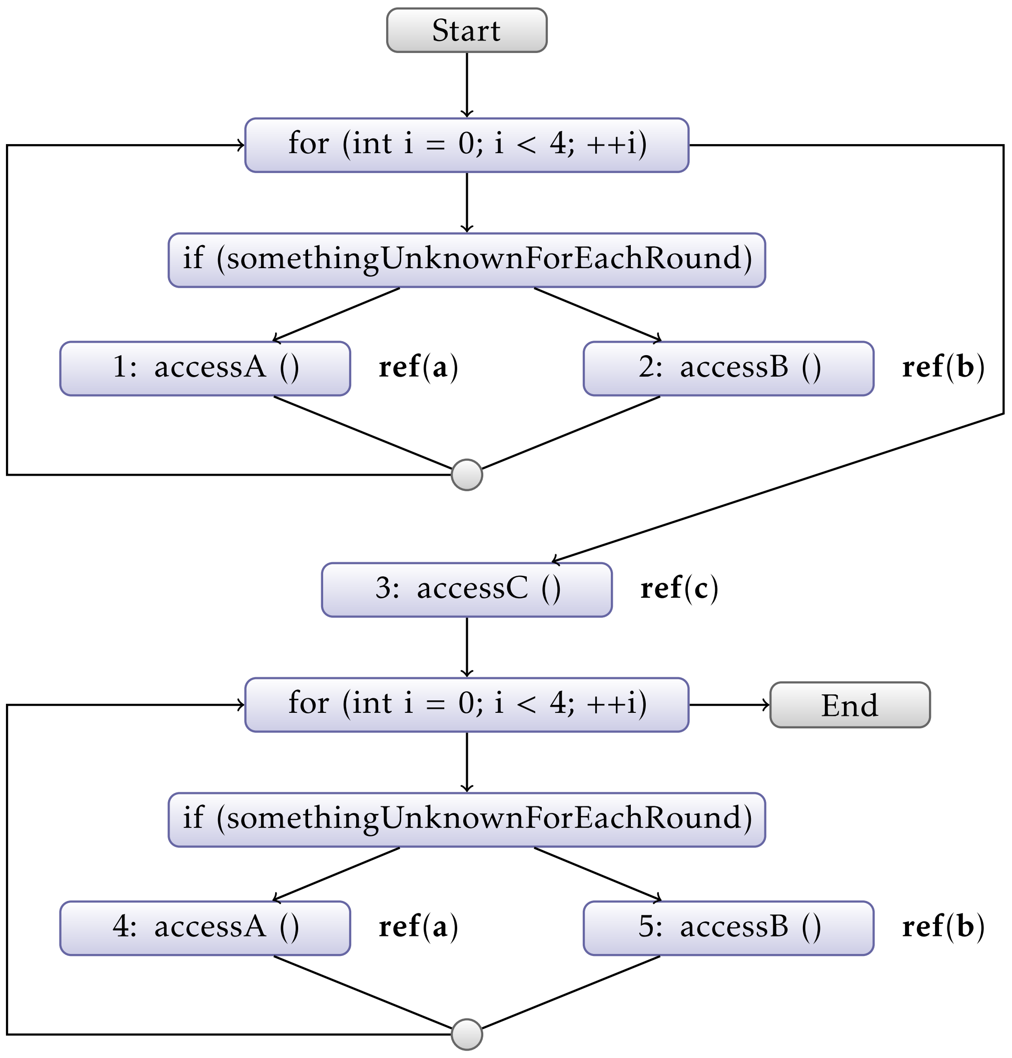

Start
for (int i = 0; i < 4; ++i)
if (somethingUnknownForEachRound)
1: accessA ()
ref(a)
2: accessB ()
ref(b)
3: accessC ()
ref(c)
for (int i = 0; i < 4; ++i)
End
if (somethingUnknownForEachRound)
4: accessA ()
ref(a)
5: accessB ()
ref(b)

Figure 4.6: Control-flow graph with single memory references for the program from Figure 4.5. Nodes with a memory reference to a memory block $x \in M$ are annotated with $ref(x)$.

A path $\pi = (v_1, ..., v_n)$ through G enters the scope *S at node v_j iff $v_j \in V' \land v_{(j-1)} \notin V'$.*

A path $\pi = (v_1, ..., v_n)$ through G leaves the scope *S at node v_j iff $v_j \in V' \land v_{(j+1)} \notin V'$.*

Given a path $\pi = (v_1, ..., v_n)$ through G and a scope S, $scopeSuffix(S, \pi)$ is the sequence of nodes $(v_k, ..., v_n)$ for $v_k \in \pi$ with the largest k such that the path π enters the scope S at v_k.

For the control-flow graph from Figure 4.6 the interesting *persistence scopes* would be the two loops inside the graph. This leads to a *scope-aware persistence classification.*

Definition 4.3 (Scope-Aware Persistence)
Given a control-flow graph with single memory references $((V,E,i),\mathcal{L})$, a concrete cache semantics $\mathcal{U}_{C_N} : V \to C \to C$, a set of initial cache states $I \subseteq C$ and a persistence scope $S = (V',E',i')$ inside this graph. A memory reference to memory block $m \in M$ in node $n \in V'$ $(\mathcal{L}(n) = (m))$ is classified as persistent for the scope S *iff for all paths $\pi = (v_1,...,v_j)$ through the graph to n and for all initial cache states $c \in I$ one of these conditions holds:*

- *$\nexists v \in scopeSuffix(S,\pi) : \mathcal{L}(v) = (m)$ (it is the first reference to m since entering the scope S) or*
- *m is contained in the cache state $[\![\pi]\!]_{\mathcal{U}_{C_N}}(c)$ (the reference will cause a cache hit)*

If we look again at the example with the two loops, all memory references can be classified as persistent for their scope according to this definition. Intuitively the scopes cut the control-flow into smaller graphs that can be considered in separation. For this example each of these subgraphs is exactly the control-flow graph we already regarded for the first example with the single loop from Figure 4.2. Therefore the same arguments for the persistence classification of the memory references hold.

As these new *scope-aware persistence classification* does only hold for a specific scope but not for the whole graph, it can't be used to formalize a *persistence constraint* that limits the number of misses for paths through the whole program, therefore we need a *scope-aware persistence constraint.*

Theorem 4.2 (Scope-Aware Persistence)
Given a control-flow graph with single memory references $((V,E,i),\mathcal{L})$, a concrete cache semantics $\mathcal{U}_{C_N} : V \to C \to C$, a set of initial cache states $I \subseteq C$, a memory block $m \in M$ and a persistence scope $S = (V',E',i')$ inside this graph. Let P_S be the set of all nodes $n \in V'$ in the scope S that contain a memory reference to m which are classified as persistent for the scope S *for this concrete cache semantics and initial cache states. It holds that for any path through the graph for each time the path enters the scope S the memory references of all nodes in P_S can cause at most one cache miss.*

Proof 4.2 (Scope-Aware Persistence)
Given a control-flow graph with single memory references $G = ((V,E,i),\mathcal{L})$, a concrete cache semantics $\mathcal{U}_{C_N} : V \to C \to C$, a set of initial cache states $I \subseteq C$, a memory block $m \in M$ and a persistence scope $S = (V',E',i')$ inside this graph. Let P_S be the set of all

nodes $n \in V'$ in the scope S that contain a memory reference to m which are classified as persistent for the scope *S for this concrete cache semantics and initial cache states. Let us now regard an arbitrary path π through G. Per Definition 4.3, for each entering of the scope S all instances of nodes from P_S beside the first instance on the path π must cause a hit. Therefore at most one miss can be caused for each entering of the scope S for the path π by all instances of nodes from P.*

4.3 Application to Real-World Software

The example given in Figure 4.1 shows the classic case when persistence analysis is useful: there exists conditional control flow inside loops with statically not known conditions. Whereas this is an easy to understand example useful for the description of the principles of persistence analysis, the interesting point is: where do such control flow constructs exist in practice and why isn't it possible to have a more precise value analysis to compute the value of the condition to avoid the need for persistence analysis? In this section we show typical code patterns that occur in real-world safety critical software and feature statically not decidable control-flow or data-access decision to make use of persistence analysis.

4.3.1 Input Handling - Interface to the Physical World

Typical control software needs to handle input data fed by physical sensors. For example a flight control software will need to handle the input of e.g., speedometers, altimeters and the actual cockpit control by the pilot. The values of these inputs are not available during analysis and one must assume that the complete input range is possible during execution. Therefore the actual code will likely contain checks as shown in Figure 4.7.

```
void flightControl () {
  for (int i = 0; i < 10; ++i) {
    if (currentAltitude() > 100.0) handleHigh();
    else handleLow();
  }
}
```

Figure 4.7: Loop containing an input-dependent if-condition.

4.3.2 Message Handling - Inter-System Communication

An other important aspect of embedded software is communication between the different systems. Inside a typical car for example, a multitude of embedded systems communicate with each other over different bus systems like *CAN* [THW94] or *FlexRay* [PPE+08]. Typical message handling code looks like 4.8 or 4.9. Either the possible message handlers are hard coded using switching over the message type or they are registered as callbacks and then handled in the driver layer for the bus system that is used. In any case, depending on the statically not know message type, a different code part is executed, either directly inside the switch or indirectly by invoking a function pointer out of a previously registered handler array.

```
void messageHandlingSwitch () {
  for (int i = 0; i < NUMBER_OF_MESSAGES; ++i) {
    switch (msg[i]->type) {
      case Type1:
        doStuffForThisMessageType1 (msg[i]);
        break;
      case Type2:
        doStuffForThisMessageType2 (msg[i]);
        break;
      ...
    }
  }
}
```

Figure 4.8: Loop handling messages. It loops over a buffer with messages and selects which kind of message handling is required by the type of the dynamic message.

4.3.3 Error Handling - Catching Runtime Errors

Even if runtime errors like out-of-bounds array accesses or arithmetic overflows can be ruled out by static software analysis [CCF+05, KWN+10] safety-critical software has to cope with e.g., hardware errors during runtime. Typically such error handling doesn't happen in the normal execution, but the constructs that

```
void messageHandlingCallbacks () {
  for (int i = 0; i < NUMBER_OF_MESSAGES; ++i) {
    registeredHandler [msg[i]->type] (msg[i]->content);
    ...
  }
}
```

Figure 4.9: Loop handling messages. It loops over a buffer with messages and calls the matching message handler callback routine.

check for errors will induce control flow changes depending on unknown values like shown in Figure 4.10.

```
void errorHandling () {
  for (int i = 0; i < NUMBER_OF_EVENTS; ++i) {
    if (didAnErrorOccur (event[i])) {
      errorHandling (event[i]);
      ...
    }
  }
}
```

Figure 4.10: Loop handling errors. It loops over a buffer with dynamic events and only executes the error handling if an error occurred.

4.3.4 Data Dependent Algorithms - State Machine Code

More cases of such uncertainty about the possible control-flow transitions are introduced by the imprecision of the value analysis for complex algorithms or state machines. Such data-dependent algorithms and state machines are widely used in the embedded domain [GCH11]. An example for a state machine update function can be found in Figure 4.11.

Many safety-critical control-software applications, like fly-by-wire or engine control software contain state machine code, which is mostly generated from *SCADE*

```
void stateMachineUpdate (State *state) {
    switch (state->id) {
      case InitState:
        doInit (state);
        state->id = Read;
        break;
      case Read:
        doRead (state)
        state->id = Waiting;
        break;
      ...
    }
  }
}
```

Figure 4.11: State machine update function. Typically called inside a loop, will do state transition.

or *MATLAB* models. Whereas there is ongoing research to improve the precision [WLP+10] by making model level information available for the value analysis, current state-of-the-art value analysis can't eliminate all transitions that are impossible in reality.

4.4 Summary

In this chapter, we provided the basic idea and motivation behind cache persistence analysis. Beside giving a classic example, we showed typical code structures and techniques in real-time software that will benefit from the persistence analysis. The next chapters will focus on the actual analyses.

CHAPTER 5

Cache Persistence Analysis by Ferdinand

5.1 Introduction of the Analysis

Ferdinand's analysis is based on the idea to compute the maximal age for all memory blocks that may be in the cache. This means the age of a memory block in the abstract set state m is an upper bound of the age of the memory block in the concrete set states that are represented by m. To keep track of memory blocks that already were evicted from the cache, and therefore are older than all blocks in the cache, an additional age larger than the cache associativity A is introduced. For a point in the control flow, a block is persistent if it has not been assigned the additional age $A + 1$.

The analysis is based on abstract interpretation and uses as abstract set update function a modification of the *must cache analysis* update function with the additional age $A + 1$. The join function is an altered version of the *must cache analysis* join, where set intersection is replaced with set union. The update and join function definitions are presented in Figure 5.1.

5.2 Application to an Example

For a better intuition, the analysis will be applied to the example code snippet shown in Figure 4.1.

For this and following examples we assume the following analysis settings (if not specified otherwise):

$$U_{pers}(m,x) = \begin{cases} l_1 \mapsto \{x\} & \\ l_2 \mapsto (m(l_1) \cup m(l_2)) \setminus \{x\} & \text{if } x \in m(l_1) \\ l_i \mapsto m(l_i) \mid i = 3 \mathrel{..} A+1 & \\ & \\ l_1 \mapsto \{x\} & \\ l_i \mapsto m(l_{i-1}) \mid i = 2 \mathrel{..} h-1 & \text{if } \exists h \in \{2,\ldots,A\} : x \in m(l_h) \\ l_h \mapsto m(l_{h-1}) \cup (m(l_h) \setminus \{x\}) & \\ l_i \mapsto m(l_i) \mid i = h+1 \mathrel{..} A+1 & \\ & \\ l_1 \mapsto \{x\} & \\ l_i \mapsto m(l_{i-1}) \mid i = 2 \mathrel{..} A & \text{otherwise} \\ l_{A+1} \mapsto m(l_A) \cup (m(l_{A+1}) \setminus \{x\}) & \end{cases}$$

$$J_{pers}(m,m') = l_i \mapsto \begin{array}{cl} & \{x \mid \exists l_a, l_b : x \in m(l_a) \wedge x \in m'(l_b) \wedge i = \max(a,b)\} \\ \cup & \{x \mid x \in m(l_i) \wedge \nexists l_a : x \in m'(l_a)\} \\ \cup & \{x \mid x \in m'(l_i) \wedge \nexists l_a : x \in m(l_a)\} \end{array}$$

Figure 5.1: The update (U_{pers}) and join (J_{pers}) functions of the persistence analysis of Ferdinand.

- A 2-way set associative cache with LRU replacement policy.
- The analyses will only cover one cache set, as the sets are independent.

The semantics of the program snippet are:

- accessA will read memory block *a*, accessB memory block *b*, *a* and *b* map to the analyzed set.
- No other memory blocks that are mapped to the analyzed set are accessed inside the loop.
- The condition of the if-statement is not known by the static analyses for any loop iteration.

Figure 5.2 shows the fixed point iteration of the persistence analysis for the loop body of the example given above. The persistence analysis starts with an empty persistence set at the entry of the loop. Then the set is updated for either a and b in the two parallel if-then-else parts. The two resulting sets are joined afterwards, as the control flow joins again after the if-then-else construct. After three rounds of this computation, the fixed point is reached. Both memory references, to the memory blocks a and b, are classified as *persistent* as their age is two in the fixed point persistence set, which means they will not be evicted, as the age is not higher than the associativity. Given this classification, the WCET analysis can safely assume that at most one miss can occur for an access to a and b over the whole execution of the loop, as they will never be evicted again inside the loop.

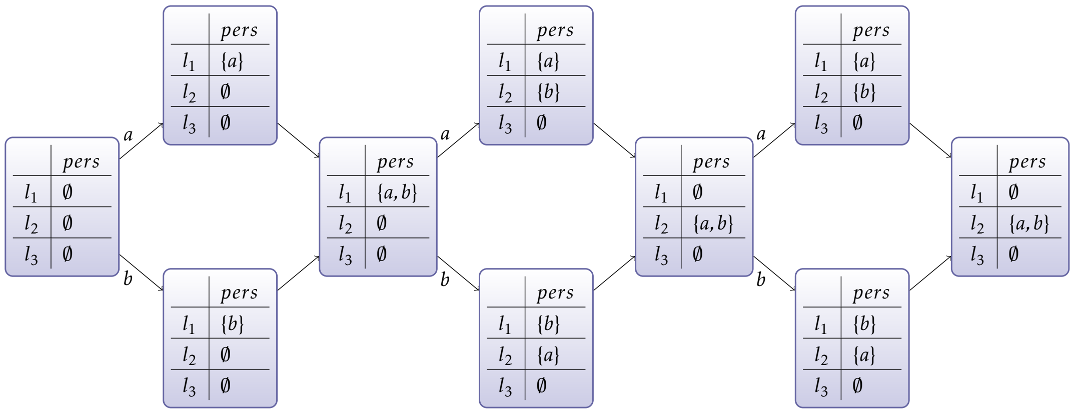


Figure 5.2: Fixed point iteration for the persistence analysis by Ferdinand for the if-then-else loop example (Figure 4.1): The figure shows the fixed point iteration without joins at the loop head like if the loop would be infinitely unrolled. After three rounds, the fixed point is reached. The memory references to the memory blocks a and b are classified as *persistent*.

5.3 Persistence Analysis Bug

5.3.1 Counter-Example

In the last sections the persistence analysis by Ferdinand was introduced and its applicability was demonstrated on a small example. In contrast to the must and may cache analyses, there exists no correctness proof for the cache persistence analysis by Ferdinand. If the analysis is applied to a slightly modified example, as shown in Figure 5.3, a bug of the analysis is revealed. It differs from the example of the last section by accessing three and not only two memory blocks inside the loop body.

```
void switchLoop () {
  for (int i = 0; i < 100; ++i) {
    switch (somethingUnknown()) {
      case  0: accessA (); break;
      case  1: accessB (); break;
      default: accessC (); break;
    }
  }
}
```

Figure 5.3: Loop with switch construct accessing memory blocks *a*, *b* or *c* depending on a condition not known statically.

This problem was discovered by C. Ballabriga and H. Cassé during their use of the persistence analysis for their own research and remedies were discussed with AbsInt. Recently other research groups like Mingsong Lv et al. [LYGY10] and Huynh et al. [HJR11] published similar problems and counter examples.

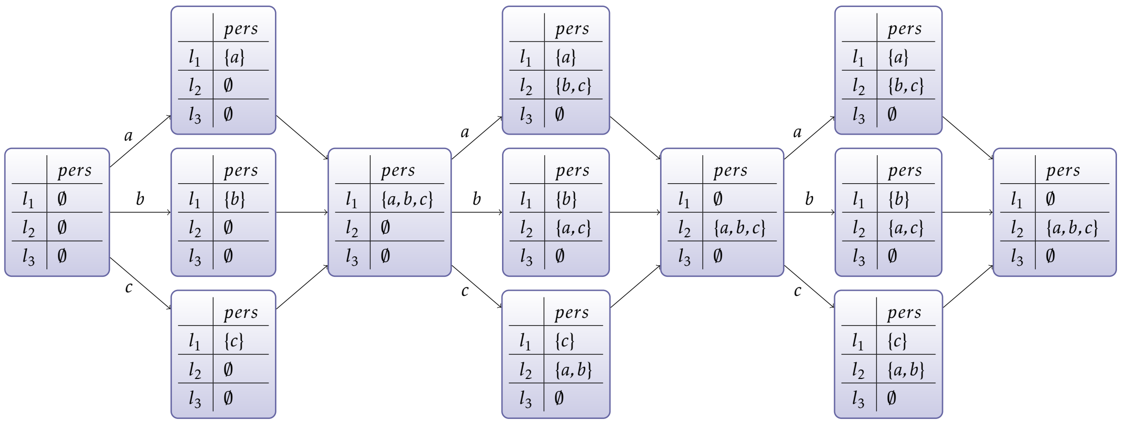

pers: l_1 ∅, l_2 ∅, l_3 ∅
a
pers: l_1 {a}, l_2 ∅, l_3 ∅
b
pers: l_1 {b}, l_2 ∅, l_3 ∅
c
pers: l_1 {c}, l_2 ∅, l_3 ∅
pers: l_1 {a,b,c}, l_2 ∅, l_3 ∅
a
pers: l_1 {a}, l_2 {b,c}, l_3 ∅
b
pers: l_1 {b}, l_2 {a,c}, l_3 ∅
c
pers: l_1 {c}, l_2 {a,b}, l_3 ∅
pers: l_1 ∅, l_2 {a,b,c}, l_3 ∅
a
pers: l_1 {a}, l_2 {b,c}, l_3 ∅
b
pers: l_1 {b}, l_2 {a,c}, l_3 ∅
c
pers: l_1 {c}, l_2 {a,b}, l_3 ∅
pers: l_1 ∅, l_2 {a,b,c}, l_3 ∅

Figure 5.4: Fixed point iteration for the persistence analysis by Ferdinand for the switch loop example (Figure 5.3): After three rounds, the fixed point is reached. The memory references to the three memory blocks *a*, *b* and *c* are classified as *persistent*.

Figure 5.4 shows the fixed point iteration for the problematic example. The persistence analysis starts with an empty persistence set at the entry of the loop. Then the set is updated for either a, b or c in the three parallel switch cases. Afterwards all three resulting sets are joined, as the control flow joins again after the switch construct. After three rounds of this computation, the fixed point is reached. All memory references are classified as persistent, as the ages of a, b and c in the persistence set after reaching the fixed point are still not higher than the associativity. But intuitively it is clear that with a cache set of two entries, but possibly three accessed memory blocks mapped to it, at least one of them might be evicted. A possible access sequence could be: $(a, b, c)^*$. The access to c will already evict a, which then needs to be reloaded on the next access to a.

5.3.2 Evaluation

The problem is the combination of the abstract *must cache analysis* update function, which only gives correct results if the abstract cache sets under-approximate the concrete cache contents, with the modified *must cache analysis* join function. This join function uses set union and therefore violates this invariant. Even if the maximal ages are chosen in the join, the join result will potentially contain elements that might already have been evicted from the cache and therefore result in an over-approximation of the concrete cache contents.

5.4 Summary

In this chapter, we introduced the well-known cache persistence analysis by Ferdinand. It is an abstract interpretation based analysis using a modified *must cache analysis*. As this analysis has a correctness issue, in the next chapters we will introduce different novel persistence analyses which are sound. We will experiment with different approaches and compare their precision.

CHAPTER 6

Set-Wise Conflict Counting Persistence Analysis

6.1 Introduction of the Analysis

The basic idea of this analysis is to count the possible number of *conflicts* per cache set. We will formulate an analysis based on abstract interpretation that keeps track of possible conflicting memory blocks per cache set by computing an over-approximation of all accessed memory blocks.

Frank Mueller introduced a similar analysis together with the notion of *first-miss* restricted to direct-mapped instruction caches in [MWH94]. Later the approach was extended to handle set-associative instruction caches with *LRU* replacement strategy in [Mue00]. These analyses are based on a proprietary static cache simulation framework [Mue95].

Definition 6.1 (Abstract Set Domain – Set-Wise Conflict Counting)
The abstract domain $\mathcal{D}_{pers_{cs}} := 2^M$ *of the* set-wise conflict counting cache persistence analysis *for one cache set consists of sets of memory blocks ordered by inclusion.*

Definition 6.2 (Set Update and Join – Set-Wise Conflict Counting)
The abstract update function $U_{pers_{cs}} : \mathcal{D}_{pers_{cs}} \times M \rightarrow \mathcal{D}_{pers_{cs}}$ *and join function* $J_{pers_{cs}} : \mathcal{D}_{pers_{cs}} \times \mathcal{D}_{pers_{cs}} \rightarrow \mathcal{D}_{pers_{cs}}$ *of the* set-wise conflict counting cache persistence analysis *are defined as:*

$$U_{pers_{cs}}(m, x) := m \cup \{x\}$$

$$J_{pers_{cs}}(m, m') := m \cup m'$$

The basic idea behind this analysis is to compute the set of all memory blocks possibly accessed so far inside the given persistence scope. The analysis starts with the empty set at the entry of the persistence scope. Memory blocks are never removed but only added to the set and on control flow graph joins set union is used. As long as the set doesn't contain more memory blocks than the associativity of the cache, any access to any memory blocks of the set within the persistence scope is classified as persistent.

Definition 6.3 (Persistence – Set-Wise Conflict Counting)
For a given incoming abstract set state m of a cache set with associativity A, a memory reference to $x \in M$ is persistent iff $|m| \leq A$.

6.2 Application to the Examples

6.2.1 if-then-else Loop

Figure 6.1 shows the analysis on the if-then-else example, for which it computes the same results as the original analysis by Ferdinand. Like for the other analysis, this analysis starts with an empty persistence set at the loop entry. Then the set is updated for either a or b in parallel and the resulting sets are joined afterwards. As this analysis doesn't keep track of any abstract ages, the fixed point is already reached after two rounds. As the computed persistence sets contain not more entries than the associativity of the cache, the two memory references are classified as persistent and therefore each of them can only produce at most one miss.

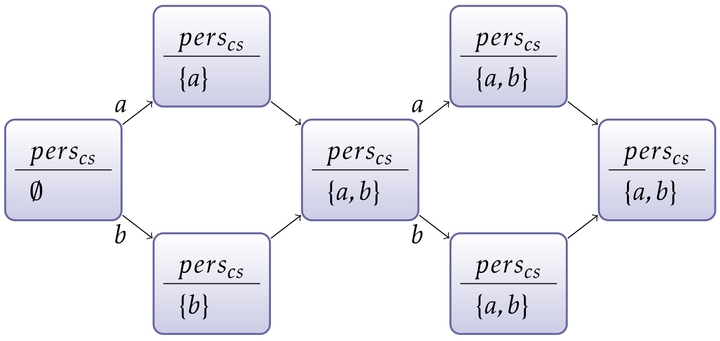


Figure 6.1: Fixed point iteration for the conflict counting persistence analysis for the if-then-else loop (Figure 4.1): The fixed point is reached after two rounds. The memory references to a and b are classified as *persistent*.

6.2.2 switch Loop

The analysis of the more complex switch loop example (Figure 5.3) with three possible accesses inside the loop is presented in Figure 6.2. The conflict based analysis computes a fixed point result that allows none of the three memory references to be classified as persistent.

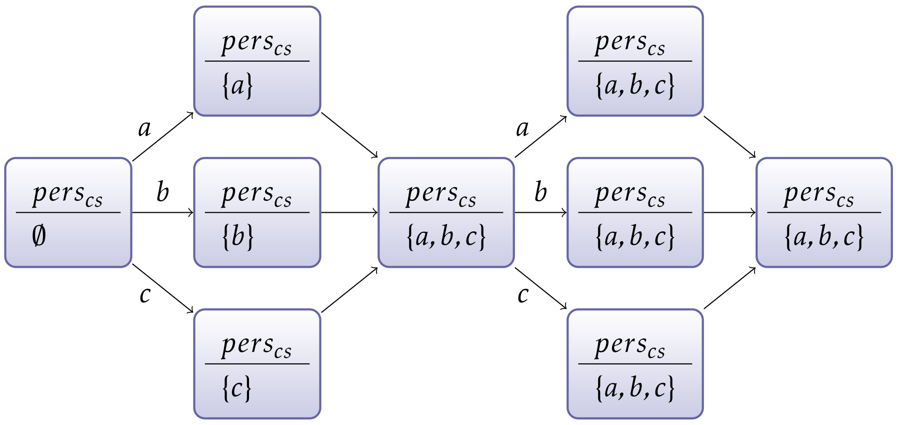


Figure 6.2: Fixed point iteration for the conflict counting persistence analysis for the switch loop (Figure 5.3): After two rounds, the fixed point is reached. None of the memory references to a, b and c is classified as *persistent*.

6.3 Discussion of Analysis Properties

6.3.1 Soundness

Overview

For our persistence analysis we regard caches using the *LRU* replacement policy. Given the concrete semantics for this policy as defined in Section 3.3.2 the following holds: If only k different memory blocks mapped to one cache set are accessed for a k-way *LRU* cache, none of these k memory blocks will be evicted.

Using this fact, the persistence classification as proposed in Definition 6.3 is sound if the abstract set state m at a program point p contains all the memory blocks possibly loaded into the concrete cache on any path from the entry of the persistence scope to p.

We will prove this by induction in three steps:

- Soundness of the initial abstract cache set state at the persistence scope entry;
- Soundness of the abstract update function $U_{pers_{cs}}$;
- Soundness of the abstract join function $J_{pers_{cs}}$.

Initial State

At the entry of the persistence scope we start with an empty abstract set state $m = \emptyset$. As at the entry of the scope still no accesses have been done for any concrete execution, m contains all the memory blocks loaded so far.

Update Function

Given the input abstract set state m for the update function $U_{pers_{cs}}$ for a memory reference to the memory block x at program point p. The set m contains all the memory blocks loaded into the cache set before program point p.

The update function inserts the referenced element x into m. Therefore the result $m \cup \{x\}$ of the update function contains all loaded memory blocks after program point p.

Join Function

Given the two abstract set states m and m' as input for the join function $J_{pers_{cs}}$ for some control-flow join of two paths at program point p. The sets m and m' contain the memory blocks loaded on the individual paths.

As the join function is set union, the result $m \cup m'$ of the join will contain all the possibly loaded memory blocks of both paths.

6.3.2 Termination

As the number of different referenced memory blocks inside a program is finite, the abstract domain $\mathcal{D}_{pers_{cs}} = 2^M$ is finite, too. Therefore the termination of the analysis is guaranteed by the monotonicity of the abstract update $U_{pers_{cs}}$ and join function $J_{pers_{cs}}$ for the persistence sets.

6.4 Analysis Space-Optimization

As this analysis will classify any memory reference as not persistent as soon as the persistence set contains more than associativity A memory blocks, an optimized version is used for the implementation. It will only store up to A memory blocks in the set and if the set becomes overfull only remember it as a special $\top$ value.

Definition 6.4 (Optimized Set Update and Join – Set-Wise Conflict Counting)
The abstract update function $U_{pers_{cs\_optimized}}$ and join function $J_{pers_{cs\_optimized}}$ for the space-optimized variant are defined as:

$$U_{pers_{cs\_optimized}}(m,x) := \begin{cases} \top & \text{if } m = \top \vee |m \cup \{x\}| > A \\ m \cup \{x\} & \text{otherwise} \end{cases}$$

$$J_{pers_{cs\_optimized}}(m,m') := \begin{cases} \top & \text{if } m = \top \vee m' = \top \vee |m \cup m'| > A \\ m \cup m' & \text{otherwise} \end{cases}$$

As the cache associativity is constant for the analysis, the space complexity per set is reduced from $O(n)$ to $O(1)$ if the analysis accesses n different memory blocks inside the persistence scope.

Beside saving space, the analysis reaches faster the fixed point, as the conflict set isn't changing once the overfull set has been reached.

6.5 Imprecision Scenarios

Given that this cache persistence analysis seems to perform well for our examples of Figure 4.1 and Figure 5.3 the question remains: in which areas can we improve precision further with more advanced persistence analyses?

The key idea is to look at situations where overfull conflict sets can occur but still, memory references may be persistent. We will look at two different cases to illustrate such situations: unrolled loops and persistent memory references inside one loop iteration.

6.5.1 Imprecision for unrolled loops

For unrolled loops, two kinds of problems show up even for small examples:

- *too fast possible eviction*: inside the unrolled iterations, memory references may be classified as not persistent, even if the referenced memory blocks will stay in the cache several more rounds;
- *no resilience against overfull abstract cache sets*: if the analysis once computes an overfull abstract cache set, all further memory references will be classified as non-persistent.

For the first problem, we can use our switch example from Figure 5.3. The fixed point iteration in Figure 6.2 results in no memory references classified as persistent after the first loop iteration. In reality, all memory references inside the first three rounds are persistent because with a two-way LRU cache, there can't be two misses to the same element during only three rounds as each round accesses only one element. Our analysis result allows all memory references to be classified as misses.

For the second problem, we can construct an example like in Figure 6.3.

```
void prefixLoop () {
  for (int i = 0; i < NUMBER_OF_EVENTS; ++i) {
    if (i == 0) // only done in first round
      accessA ();
    if (somethingUnknownForEachCall()) accessB ();
    else accessC ();
  }
}
```

Figure 6.3: Loop which accesses three memory blocks a, b and c mapping to the same cache set. a is only accessed once in the first iteration.

If we assume that the accessed memory blocks a, b and c in the example map to the same set and we have a two-way associative cache, in reality, b and c will never be evicted after the first access.

Given that we use our persistence analysis in an analysis framework that allows loop unrolling [MAWF98], we would assume the analysis yields the same result if we unroll this loop at least twice. But as the fixed point iteration for this example in Figure 6.4 shows, neither the memory references to b nor c will be classified as persistent by the analysis.

The conflict counting analysis will compute the over-approximation of all accessed memory blocks a, b, c after the first round and then classify all further memory references in any iteration as not persistent. The analysis will never recover.

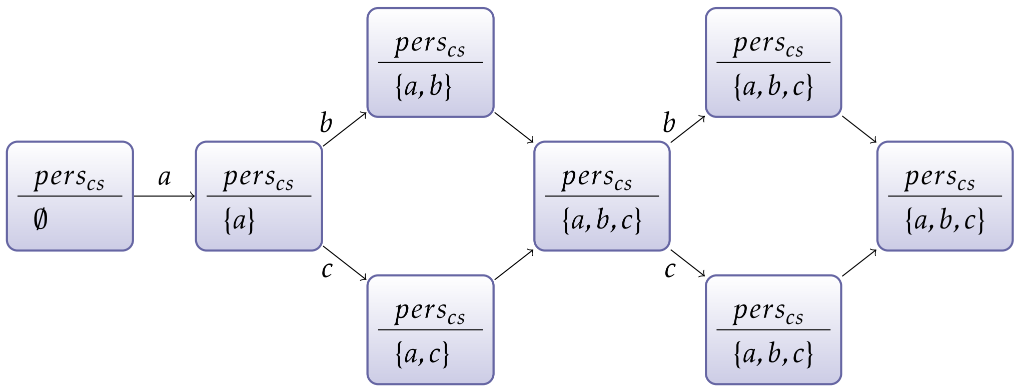


Figure 6.4: Fixed point iteration for the conflict counting persistence analysis for the example in Figure 6.3: After two rounds, the fixed point is reached. The memory references to b and c are classified as *not persistent*.

6.5.2 Imprecision for inner-iteration persistence

Let's consider a loop as shown in Figure 6.5.

```
void innerPersistenceLoop () {
  for (int i = 0; i < NUMBER_OF_EVENTS; ++i) {
    if (somethingUnknown1()) accessA ();
    else accessB ();
    accessC ();
    if (somethingUnknown2()) accessA ();
    else accessB ();
  }
}
```

Figure 6.5: Loop which accesses three memory blocks a, b and c mapping to the same cache set. a and b are possibly accessed twice.

If we assume a two-way associative cache and all memory blocks are mapped to the same set, it is clear that we will have potential evictions inside the loop, as the body accesses three different memory blocks. Whereas inside on loop iteration, only the first access to a or b can be a miss, as we only access one other element c in between. Therefore it might be a good idea to use the loop body as a persistence scope, to capture this behavior.

If we apply the *set-wise conflict counting cache persistence analysis* to the loop body as scope, this result is not computed (see Figure 6.6). The analysis only classifies the first memory references to either a or b as possibly persistent, but not the second.

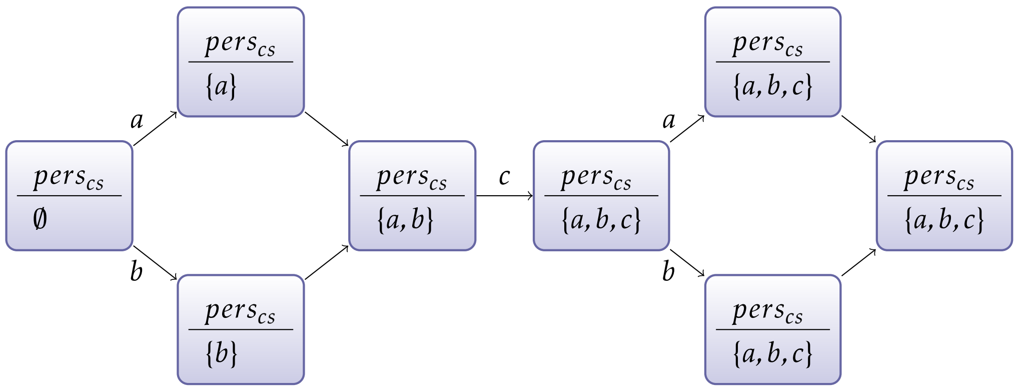


Figure 6.6: Analysis for one round of the innerPersistenceLoop loop example (Figure 6.5). The second memory references to neither a nor b are classified as persistent.

6.5.3 Proposed improvements

The reason for the imprecision shown above is the simplistic conflict counting for the whole set. If the set becomes overfull at one point, no more memory references can be classified as persistent by the analysis. In addition, the overfull conflict set is reached too early during the fixed point computation.

To overcome these limitations, we will introduce more precise analyses in the next chapters and compare their results.

To have a better way to quantify the precision of our analyses and to compare them, we introduce a framework to generate and run synthetic tests for our persistence analyses.

6.6 Benchmarking

Until now, we only looked at small examples to understand the principles of persistence analysis. Whereas this allows a good understanding of how the analysis works and to show special cases which lead to imprecision, it provides no good overview about the general analysis precision.

To allow a better comparison of the different analyses introduced in this thesis, we will use a framework for automated test case generation and execution of the analyses. The framework will generate a control-flow graph for a set of parameters and execute the analysis for one cache set on this graph.

We will choose eleven different test situations, and then generate 10.000 control-flow graphs for each of the situations, and run the individual analyses on them. The generator is deterministic, therefore the results will be comparable for the different analyses. This larger set of analysis results will allow for a more realistic analysis comparison.

The C++ implementation of the benchmarking framework is shown in Section A.1. The framework first constructs the control-flow graphs for the different scenarios and then executes our persistence analyses on them. It computes the *MFP* solution as introduced in Section 2.3 and provides statistics about the number of persistent-classified memory references.

6.6.1 Benchmark Scenarios

We consider the following eleven scenarios:

scenario1 2-way associative cache, loop unrolled twice, two different memory blocks accessed on two control-flow paths inside the loop. Our example in Figure 4.1 is one instance of this scenario.

scenario2 2-way associative cache, loop unrolled eight times, three different memory blocks accessed on three control-flow paths inside the loop. Our example in Figure 5.3 is one instance of this scenario.

scenario3 2-way associative cache, loop unrolled twice, two different memory blocks accessed on two control-flow paths inside the loop, potentially one extra element accessed in the unrolled iterations. Our example in Figure 6.3 is one instance of this scenario.

scenario4 4-way associative cache, loop unrolled eight times, four different memory blocks accessed on four control-flow paths inside the loop, potentially two extra memory blocks accessed in the unrolled iterations. Similar to *scenario3*, but with more possible control-flow paths and memory blocks.

scenario5 8-way associative cache, loop unrolled eight times, eight different memory blocks accessed on four control-flow paths inside the loop, potentially four extra memory blocks accessed in the unrolled iterations, but only four of the other memory blocks accessed there. Similar to *scenario3* or *scenario4*, but inside the unrolled prefix, it accesses still less than associativity many memory blocks, but different ones than in the not-unrolled last iteration. The unrolled iterations form control-flows similar to the one seen inside the loop in Figure 6.5.

scenario6 8-way associative cache, loop unrolled sixteen times, eight different memory blocks accessed on eight control-flow paths inside the loop, potentially sixteen extra memory blocks accessed in the unrolled iterations, but none of the other memory blocks accessed there. Similar to *scenario5*, but inside the unrolled prefix, it accesses only different memory blocks compared with the last not-unrolled iteration.

scenario7 8-way associative cache, loop unrolled eight times, eight different memory blocks accessed on four control-flow paths inside the loop, potentially eight extra memory blocks accessed in the unrolled iterations, but only six of the other memory blocks accessed there. Another variation of *scenario5*, more different memory blocks accessed in the unrolled iterations.

scenario8 variation of *scenario4*, but with reuse of one random element per iteration on random paths.

scenario9 variation of *scenario5*, but with reuse of two random memory blocks per iteration on random paths.

scenario10 variation of *scenario6*, but with reuse of four random memory blocks per iteration on random paths.

scenario11 variation of *scenario8*, but with reuse of six random memory blocks per iteration on random paths.

As benchmark results we get:

- number of memory references during the unrolled iterations
- number of memory references that are classified as persistent during unrolled iterations

- number of memory references in the not-unrolled iterations
- number of memory references that are classified as persistence after the fixed point is reached for the remaining iterations

We will use these results to quantitatively compare our analyses.

6.6.2 Benchmark Results

Overview In Table 6.1 the results for the set-wise conflict counting analysis are shown. The percentage of persistent-classified memory references for the different scenarios is illustrated in Figure 6.7.

Review of the Scenarios Only for the first scenario *scenario1*, which never accesses more than associativity different memory blocks on any path, the analysis can classify all memory references as persistent. For the other scenarios, only for some of the 10.000 generated control-flow graphs of *scenario3* and *scenario4*, for which no overfull conflict sets occur, persistent classifications appear. For all other scenarios and runs, the analysis eventually reaches an overfull conflict set. Only some memory references during the first unrolled iterations are therefore classified as persistent in all other scenarios. During the unrolled iterations, we can classify persistent memory references until an overfull conflict set is reached. As shown above in Figure 6.2 for the switch example from Figure 5.3, this can be relatively early compared to the real execution.

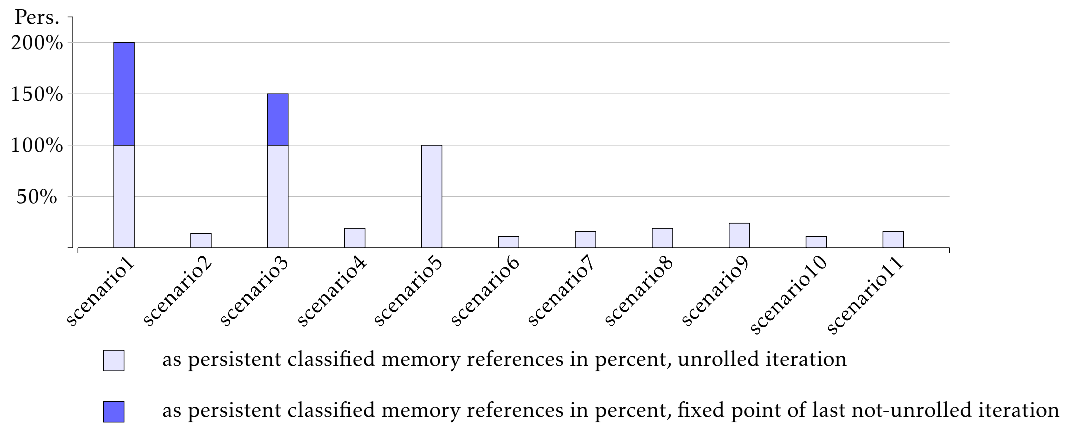


Figure 6.7: Comparison of analysis precision for different synthetic benchmark scenarios for the set-wise conflict counting analysis.

	Unrolled Iterations			Last Not-Unrolled Iteration		
	References	Pers.	Pers. %	References	Pers.	Pers. %
scenario1	19998	19998	100.00%	19998	19998	100.00%
scenario2	209979	29997	14.29%	29997	0	0.00%
scenario3	25004	25004	100.00%	19998	9986	49.93%
scenario4	349706	66540	19.03%	39996	4	0.01%
scenario5	420025	420025	100.00%	79992	0	0.00%
scenario6	1199585	126241	10.52%	79992	0	0.00%
scenario7	699808	113661	16.24%	79992	0	0.00%
scenario8	419714	79402	18.92%	49995	0	0.00%
scenario9	699967	168674	24.10%	99990	0	0.00%
scenario10	1199472	126138	10.52%	119988	0	0.00%
scenario11	1120730	177753	15.86%	139986	0	0.00%

Table 6.1: Benchmark results for the set-wise conflict counting analysis.

6.7 Summary

In this chapter we introduced a persistence analysis based on counting the possible conflicts inside one cache set. This approach is similar to the concept of the *first-miss* analysis introduced in [MWH94]. We applied the analysis to our running examples to get a better intuition and argued about its correctness.

We have shown possible areas of analysis imprecision and introduced a benchmarking framework to get quantitative results for later analyses comparison. The set-wise conflict counting persistence analysis performed only well for benchmarks never accessing more memory blocks than fitting into the cache sets. In the following chapters, we will try to improve on these results by designing more precise persistence analyses.

CHAPTER 7

Element-Wise Conflict Counting Persistence Analysis

7.1 Introduction of the Analysis

In Chapter 6 on page 43 we have shown the basic idea of set-wise conflict counting and a cache persistence analysis based on this. Whereas this analysis yields sound results, it has some imprecisions as shown in Section 6.5.

The reason for this imprecision is that our analysis only considers the conflicts inside the whole set, but doesn't differentiate between conflicts for the individual elements that enter the set inside the persistence scope. Therefore we now improve the previous persistence analysis by tracking the conflicting elements per cache set entry.

As we now track the conflicts per element, the abstract domain $\mathcal{D}_{pers_{ce}}$ for our analysis is no longer a set of memory blocks as for the *set-wise conflict-counting* persistence analysis. Instead we use a mapping of memory blocks to such sets of blocks $M \to 2^M$.

Definition 7.1 (Abstract Set Domain – Element-Wise Conflict Counting)
The abstract domain $\mathcal{D}_{pers_{ce}}$ of the element-wise conflict counting cache persistence analysis *for one cache set is a mapping from memory blocks to potential conflict sets $M \to 2^M$.*

Definition 7.2 (Set Update and Join – Element-Wise Conflict Counting)
The abstract update function $U_{pers_{ce}} : \mathcal{D}_{pers_{ce}} \times M \to \mathcal{D}_{pers_{ce}}$ and join function $J_{pers_{ce}}$: $\mathcal{D}_{pers_{ce}} \times \mathcal{D}_{pers_{ce}} \to \mathcal{D}_{pers_{ce}}$ of the element-wise conflict counting cache persistence

analysis *are defined as:*

$$\begin{aligned} U_{pers_{ce}}(m,x) :=\ & [\, x \mapsto U_{pers_{cs}}(\emptyset, x), \\ & y \mapsto U_{pers_{cs}}(m(y), x) \mid y \neq x \wedge m(y) \neq \emptyset \\ & y \mapsto \emptyset \mid y \neq x \wedge m(y) = \emptyset \,]; \\ J_{pers_{ce}}(m,m') :=\ & [\, x \mapsto J_{pers_{cs}}(m(x), m'(x)) \,]; \end{aligned}$$

They reuse the abstract update function $U_{pers_{cs}}$ and join function $J_{pers_{cs}}$ of the set-wise conflict counting cache persistence analysis *as defined in Definition 6.2. If we substitute these functions with their definitions, the update and join functions are:*

$$\begin{aligned} U_{pers_{ce}}(m,x) :=\ & [\, x \mapsto \{x\}, \\ & y \mapsto m(y) \cup \{x\} \mid y \neq x \wedge m(y) \neq \emptyset \\ & y \mapsto \emptyset \mid y \neq x \wedge m(y) = \emptyset \,]; \\ J_{pers_{ce}}(m,m') :=\ & [\, x \mapsto m(x) \cup m'(x) \,]; \end{aligned}$$

If the program references a memory block $x \in M$ we ensure that we update its conflict counting set. Given the definition of the $U_{pers_{cs}}$ function, this ensures we have at least x itself inside this set afterwards. For all other elements $y \in M \wedge y \neq x$, we only update the individual conflicting sets if y was already referenced, e.g., the set for y is not empty.

Definition 7.3 (Persistence – Element-Wise Conflict Counting)
For a given abstract set state m of a cache set with associativity A, a memory reference to $x \in M$ is persistent iff $|m(x)| \leq A$.

Huynh et al. present a similar approach in [HJR11]. Instead of arguing about the number of possible conflicts, they argue about the number of *younger elements* per element. Their analysis assigns, like the original persistence analysis by Ferdinand, abstract ages to the possible elements inside the cache that are based on the size of these *younger elements* sets. Only if associativity many different *younger elements* are collected, an element may be evicted.

7.2 Application to the Examples

7.2.1 if-then-else Loop

Figure 7.1 shows the analysis on the if-then-else example, for which it computes the same results as both the analysis by Ferdinand and the set-wise conflict counting cache persistence analysis. The analysis starts with an empty conflict set for every memory block at the loop entry. Then the set is updated for either *a* or *b* in parallel and the resulting sets are joined afterwards. As this analysis keeps track of conflicts per element, it needs one round more to stabilize than the set-wise conflict counting. Since the computed conflict sets for all elements contain not more entries than the associativity of the cache, the two memory references are classified as persistent and therefore each of them can only cause at most one miss.

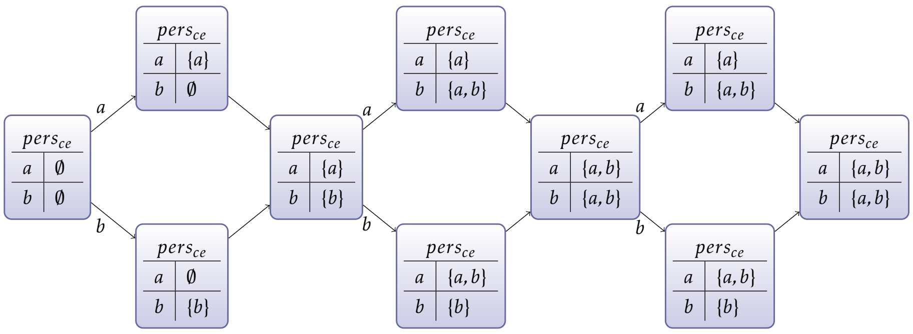


Figure 7.1: Fixed point iteration for the element-wise conflict counting persistence analysis for the if-then-else loop example (Figure 4.1): After three rounds, the fixed point is reached. The memory references to *a* and *b* are classified as *persistent*.

7.2.2 switch Loop

The more complex example with three memory references is presented in Figure 7.2. Like the set-wise conflict counting, this analysis computes a fixed point result that allows none of the three memory references to be classified as persistent. But in contrast to the simpler set-wise analysis results shown in Figure 6.2, this analysis only classifies all memory references as non-persistent after the second fixed point iteration round. This will allow for a more precise analysis if we unroll the loop, as only one miss per element is allowed in the first two iterations.

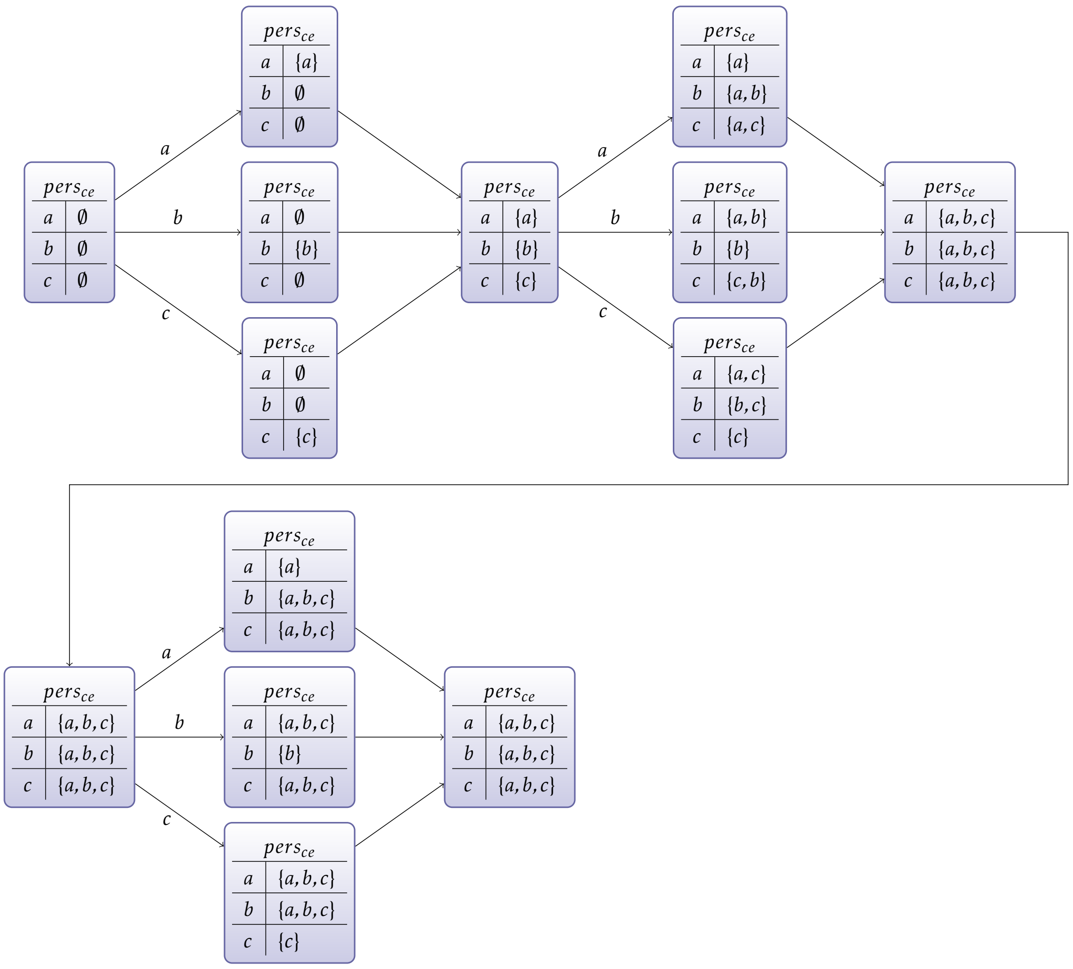

$pers_{ce}$: a $\emptyset$, b $\emptyset$, c $\emptyset$
a
$pers_{ce}$: a $\{a\}$, b $\emptyset$, c $\emptyset$
b
$pers_{ce}$: a $\emptyset$, b $\{b\}$, c $\emptyset$
c
$pers_{ce}$: a $\emptyset$, b $\emptyset$, c $\{c\}$
$pers_{ce}$: a $\{a\}$, b $\{b\}$, c $\{c\}$
a
$pers_{ce}$: a $\{a\}$, b $\{a,b\}$, c $\{a,c\}$
b
$pers_{ce}$: a $\{a,b\}$, b $\{b\}$, c $\{c,b\}$
c
$pers_{ce}$: a $\{a,c\}$, b $\{b,c\}$, c $\{c\}$
$pers_{ce}$: a $\{a,b,c\}$, b $\{a,b,c\}$, c $\{a,b,c\}$
$pers_{ce}$: a $\{a,b,c\}$, b $\{a,b,c\}$, c $\{a,b,c\}$
a
$pers_{ce}$: a $\{a\}$, b $\{a,b,c\}$, c $\{a,b,c\}$
b
$pers_{ce}$: a $\{a,b,c\}$, b $\{b\}$, c $\{a,b,c\}$
c
$pers_{ce}$: a $\{a,b,c\}$, b $\{a,b,c\}$, c $\{c\}$
$pers_{ce}$: a $\{a,b,c\}$, b $\{a,b,c\}$, c $\{a,b,c\}$

Figure 7.2: Fixed point iteration for the element-wise conflict counting persistence analysis for the switch loop example (Figure 5.3): After three rounds, the fixed point is reached. None of the memory references to *a*, *b* and *c* is classified as *persistent*.

7.3 Discussion of Analysis Properties

7.3.1 Soundness

Overview

For our persistence analysis we consider caches using the *LRU* replacement policy. Like in Section 6.3 we use a characteristic of the *LRU* replacement semantics as defined in Section 3.3.2: A memory block x is earliest evicted from a k-way *LRU* cache after k accesses to different memory blocks.

Using this fact, the persistence classification as proposed in Definition 7.3 is sound, if for the abstract set state m holds: $m(x)$ contains the memory blocks loaded into the concrete cache inside the persistence scope since the last reference to x on any path including the memory block x itself (an over-approximation of all conflicting memory blocks).

We will prove this by induction in three steps:

- Soundness of the initial abstract cache set state at the persistence scope entry;
- Soundness of the abstract update function $U_{pers_{ce}}$;
- Soundness of the abstract join function $J_{pers_{ce}}$.

Initial State

At the entry of the persistence scope we start with an abstract set state $m = [\, x \rightarrow \emptyset \mid \forall x \in M \,]$. As at the entry of the scope still no accesses have been performed for any concrete execution, the property holds for $m(x)$ for all $x \in M$.

Update Function

Given the input abstract set state m for the update function $U_{pers_{ce}}$ for a memory reference to the memory block x at program point p.

The update function will modify m in the following ways:

- It will reset $m(x)$ to only contain x itself. As we just have referenced x, this is the correct approximation.

- For all other $m(y) \mid y \in M$ with $y \neq x$ there are two cases:
  - $m(y) = \emptyset$: As $m(y)$ is a safe approximation, this means, y has not been loaded in the cache since the start of the scope. Therefore no memory block is inserted into $m(y)$, the approximation is still safe.
  - $m(y) \neq \emptyset$: The memory block x will be inserted into the set $m(y)$ to ensure the safety of the approximation.

Therefore the update function results in a safe approximation after program point p.

Join Function

Given the two abstract set states m and m' as input for the join function $J_{pers_{ce}}$ for some control-flow join of two paths at program point p. For all $x \in M$, $m(x)$ and $m'(x)$ are safe approximations of the loaded memory blocks that conflict with x on the individual paths.

As the join function is the set union for the individual $m(x)$ and $m'(x)$ for all $x \in M$, the result of the join will be an safe approximation of the loaded memory blocks that conflict with x for both paths.

7.3.2 Termination

As the number of different referenced memory blocks inside a program is finite, the abstract domain $\mathcal{D}_{pers_{ce}} = M \rightarrow 2^M$ is finite, too. Therefore the termination of the analysis is guaranteed by the monotonicity of the abstract update $U_{pers_{ce}}$ and join function $J_{pers_{ce}}$ for the persistence sets.

7.4 Analysis Space-Optimization

We can apply the same optimization as described in Section 6.4. As the cache associativity is constant for the analysis, the space complexity per set is reduced from $O(n^2)$ to $O(n)$ if the program references n different memory blocks inside the persistence scope.

Definition 7.4 (Optimized Set Update and Join – Element-Wise Conflict Counting)
The abstract update function $U_{pers_{ce\_optimized}}$ and join function $J_{pers_{ce\_optimized}}$ for the space-optimized variant are defined as:

$$
\begin{aligned}
U_{pers_{ce\_optimized}}(m,x) := \; & [\, x \mapsto U_{pers_{cs\_optimized}}(\emptyset, x), \\
& \; y \mapsto U_{pers_{cs\_optimized}}(m(y), x) \mid y \neq x \wedge m(y) \neq \emptyset \\
& \; y \mapsto \emptyset \mid y \neq x \wedge m(y) = \emptyset \,]; \\
J_{pers_{ce\_optimized}}(m,m') := \; & [\, x \mapsto J_{pers_{cs\_optimized}}(m(x), m'(x)) \,];
\end{aligned}
$$

7.5 Precision Improvements

In Section 6.5 two cases of imprecision in the set-wise conflict counting cache persistence analysis are demonstrated. They both stem from the fact, that once the conflict set is overfull, the set-wise conflict counting cache persistence analysis can't classify any memory reference as persistent. We now review these cases for the improved element-wise conflict counting cache persistence analysis.

7.5.1 Precision for unrolled loops

For the set-wise conflict counting cache persistence analysis it was not possible to classify any memory references inside the example from Figure 6.3 as persistent after one iteration. In practice neither b nor c will be evicted in any iteration. In Figure 7.3 we show the fixed point iteration for this analysis. As we now track the conflicts per element, the analysis is able to classify the memory references to b and c as persistent for all iterations. Only a will be possibly evicted after the first iteration.

7.5.2 Precision for inner-iteration persistence

The other precision problem for the set-wise conflict counting was the *innerPersistenceLoop* as shown in Figure 6.5. Figure 7.4 shows the fixed point iteration for the element-wise conflict counting analysis. All memory references can be classified as persistent by the improved analysis in contrast to the set-wise conflict counting cache persistence analysis.

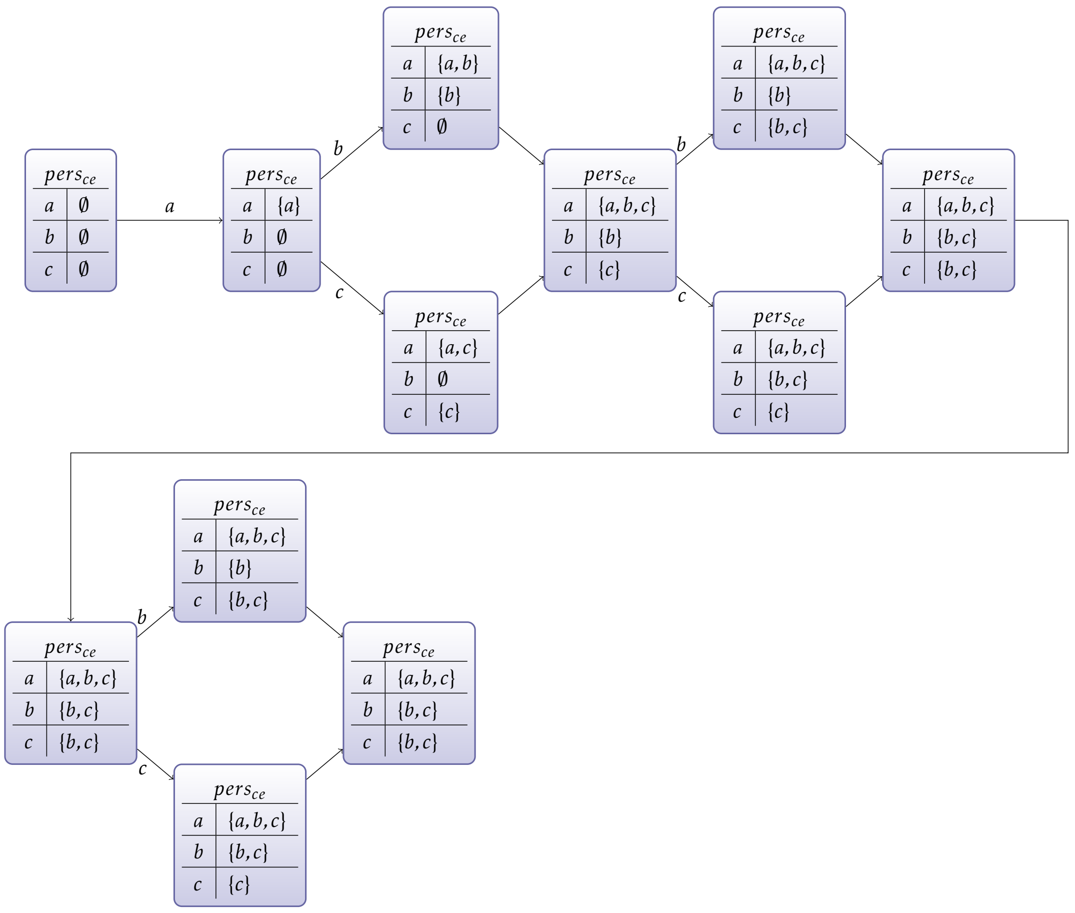


Figure 7.3: Fixed point iteration for the element-wise conflict counting persistence analysis for the example in Figure 6.3: After two rounds, the fixed point is reached. The memory references to b and c are classified as *persistent*.

7.6 Benchmarking

Overview In Table 7.1 the results for the element-wise conflict counting persistence analysis for our benchmark scenarios from Section 6.6 are shown. The percentage of persistent classified memory references for the different scenarios are illustrated in Figure 7.5. The results of the set-wise conflict counting analysis from Chapter 6 are included for comparison.

Review of the Scenarios By construction, the element-wise conflict counting analysis is at least as precise as the set-wise conflict counting variant. For the

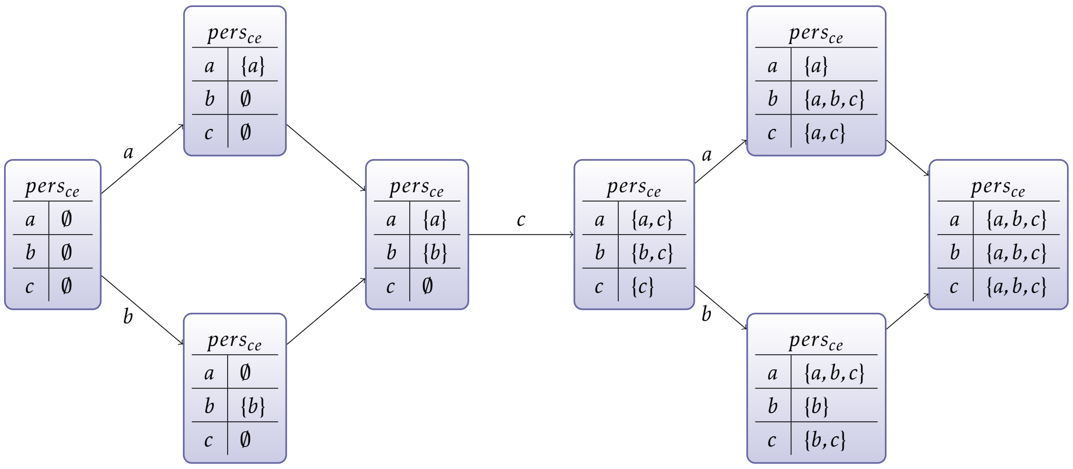


Figure 7.4: Analysis for one round of the innerPersistenceLoop loop example (Figure 6.5). The memory references to a and b are classified as persistent.

simple *scenario1* both analysis are as precise as possible, all memory references are persistent, as not more than associativity different elements are accessed. For the *scenario2*, which accesses more than associativity many elements in the not unrolled last iteration, the precision gain is the slower eviction during the unrolled iterations. The same effect occurs for *scenario4*. Whereas this scenario accesses only associativity different elements in the not unrolled iterations, there exists no analysis run for which the conflict sets per element don't become overfull. This happens because of the random accessed two additional elements during the eight unrolled iterations. The tests *scenario6* and *scenario10* accesses none of the blocks out of the last iteration during the unrolled iterations. Therefor the conflict sets for the elements accessed in the not unrolled iterations become never overfull, as they only get instantiated and filled during the last iteration. This allows persistent classifications of all memory references in the not unrolled iterations. For all other scenarios, there exists at least some analysis runs for which the fixed point result still allows persistent classifications. For these analysis runs, similar effects as in the Figures 7.3 and 7.4 happen.

Conclusion The results show, that this analysis is even able to classify memory references as persistent, if in the unrolled iterations more elements than the size of the set get accessed. This is by construction, as we reset the conflict set for an element if it is referenced. Still, during the unrolled iterations, we evict the elements faster than in the real execution, as shown in the previous section for the switch example from Figure 5.3. The analysis is by design in all scenarios at least as precise as the set-wise conflict counting.

	Unrolled Iterations			Last Not-Unrolled Iteration		
	References	Pers.	Pers. %	References	Pers.	Pers. %
scenario1	19998	19998	100.00%	19998	19998	100.00%
scenario2	209979	66000	31.43%	29997	0	0.00%
scenario3	25004	25004	100.00%	19998	15030	75.16%
scenario4	349706	128185	36.66%	39996	15	0.04%
scenario5	420025	420025	100.00%	79992	39996	50.00%
scenario6	1199585	277212	23.11%	79992	79992	100.00%
scenario7	699808	254571	36.38%	79992	19998	25.00%
scenario8	419714	165179	39.36%	49995	2489	4.98%
scenario9	699967	332990	47.57%	99990	52510	52.52%
scenario10	1199472	277340	23.12%	119988	119988	100.00%
scenario11	1120730	494098	44.09%	139986	49268	35.19%

Table 7.1: Benchmark results for the element-wise conflict counting analysis.

7.7 Summary

In this chapter we introduce a refined version of the conflict counting. Instead of tracking conflicts per whole cache set, we track them per element. Huynh et al. introduced a similar approach in [HJR11] by tracking the *younger elements* of all accessed elements.

This analysis performs well both on our running examples and in the benchmarking framework. It is more precise than the previous set-wise conflict counting analysis from Chapter 6. For eight of eleven of our testing scenarios, the set-wise conflict counting analysis is able to classify more than one quarter of the memory references as persistent.

Still the analysis has room for improvements for the unrolled loop iterations, as it potentially evicts elements too early, as shown by the example in Figure 5.3. In the next chapter, we introduce a novel persistence analysis based on the *may cache analysis* instead conflict counting to fix this imprecision.

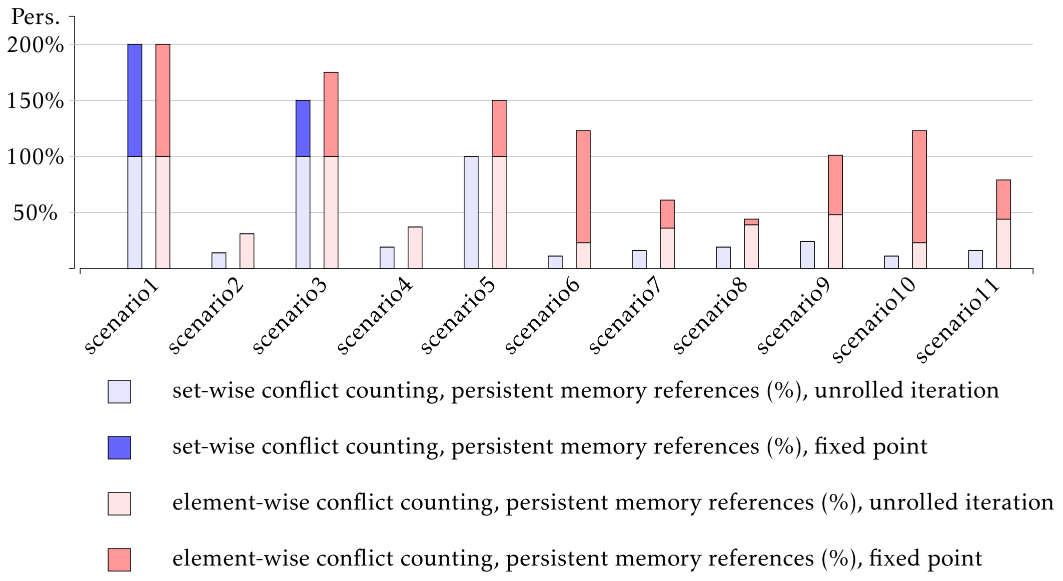


Figure 7.5: Comparison of analyses precision for different synthetic benchmark scenarios and persistence analyses.

CHAPTER 8

May Analysis Based Cache Persistence Analysis

8.1 Introduction of the Analysis

In the previous Chapters 6 and 7 we introduced two novel persistence analyses based on the principles of conflict counting. Both are able to derive sound and precise results for the persistence classification of memory references. Still, the *switch loop* example from Figure 8.2 shows a little imprecision if we unroll the loop. Instead of classifying the memory references inside the first three iterations as persistent, the set-wise conflict counting persistence analysis already classifies the memory references as not persistent from the second round on, the element-wise analysis from the third. We now will sketch an analysis that improves this.

The new analysis is a combination of two analyses that keep track of the possible ages of the cache set entries:

- The *may cache analysis* that computes an over-approximation of the cache set contents, see Section 3.5.
- A $\widehat{may}$*-cache analysis* that similarly to the persistence analysis by Ferdinand keeps track of the maximal ages of the cache contents and whether an memory block was evicted. The $\widehat{may}$*-cache analysis* is a modification of the *may cache analysis*. Instead of the minimal age, it tracks the maximal age.

Like for the persistence analysis by Ferdinand, an additional age $A+1$ is introduced to keep track of all memory blocks that might be evicted. The $\widehat{may}$-analysis uses the knowledge about the possible number of cache set entries provided by the *may-analysis* running in parallel to prevent eviction of memory blocks if the cache set is not yet completely filled.

As the analysis only keeps track of the memory blocks being accessed since the start of the persistence analysis scope – in the examples this would be the entry of the loop – the assumption that no memory blocks will be evicted until full associativity of the cache set is reached is only sound if *LRU* policy is used. Other policies are less predictable [Rei08] and might evict memory blocks earlier depending on the initial cache state at the start of the persistence analysis.

Definition 8.1 (Abstract Set Domain – May-Based Persistence)
The abstract domain $\mathcal{D}_{pers_{may}}$ *of the* may-based cache persistence analysis *for one cache set is defined as* $(\{1,..,A\} \to 2^M) \times (\{1,..,A+1\} \to 2^M)$.

Definition 8.2 (Set Update and Join – May-Based Persistence)
The abstract update and join functions of the may-based cache persistence analysis *for one cache set are defined as:*

$$U_{pers_{may}}((m,\hat{m}),x) := (U_{may}(m,x),\ U_{\widehat{may}}(m,\hat{m},x))$$

$$J_{pers_{may}}((m,\hat{m}),(m',\hat{m}')) := (J_{may}(m,m'),\ J_{\widehat{may}}(\hat{m},\hat{m}'))$$

$$U_{\widehat{may}}(m,\hat{m},x) := \begin{cases} [l_1 \mapsto \{x\}, & \\ l_i \mapsto \hat{m}(l_{i-1}) \setminus \{x\} \mid i = 2\,..\,A, & \textit{if } mayevict(m,x) \\ l_{A+1} \mapsto (\hat{m}(l_{A+1}) \cup \hat{m}(l_A)) \setminus \{x\}]; & \\ & \\ [l_1 \mapsto \{x\}, & \\ l_i \mapsto \hat{m}(l_{i-1}) \setminus \{x\} \mid i = 2\,..\,A-1, & \textit{otherwise} \\ l_A \mapsto (\hat{m}(l_A) \cup \hat{m}(l_{A-1})) \setminus \{x\}, & \\ l_{A+1} \mapsto \hat{m}(l_{A+1}) \setminus \{x\}]; & \end{cases}$$

$$mayevict(m,x) := (|\{y \mid y \neq x \land \exists l_i : y \in m(l_i)\}| \geq A)$$

$$J_{\widehat{may}}(\hat{m},\hat{m}') := [l_i \mapsto \begin{array}{ll} & \{x \mid \exists l_a, l_b : x \in \hat{m}(l_a) \land x \in \hat{m}'(l_b) \land i = max(a,b)\} \\ \cup & \{x \mid x \in \hat{m}(l_i) \land \nexists l_a : x \in \hat{m}'(l_a)\} \\ \cup & \{x \mid x \in \hat{m}'(l_i) \land \nexists l_a : x \in \hat{m}(l_a)\}] \end{array}$$

U_{may} *and* J_{may} *are the original* may cache analysis *update and join functions for one cache set as shown in Definition 3.18.*

Definition 8.3 (Persistence – May-Based Persistence)
For a given abstract set state $(m, \hat{m})$ *of a cache set with associativity* A*, a memory reference to* $x \in M$ *is persistent iff* $x \notin \hat{m}(l_{A+1})$.

8.2 Application to the Examples

8.2.1 if-then-else Loop

Figure 8.1 shows the may-based analysis on the if-then-else example, for which it computes the same results as the other analyses. The may-based analysis starts with an empty abstract persistence state at the loop entry (empty *may cache* and $\widehat{may}$ *cache*). Then the state is updated for either a or b in parallel and the resulting states are joined afterwards. The fixed point is reached after three rounds. Both memory references are classified as persistent and therefore each of them can only cause at most one miss.

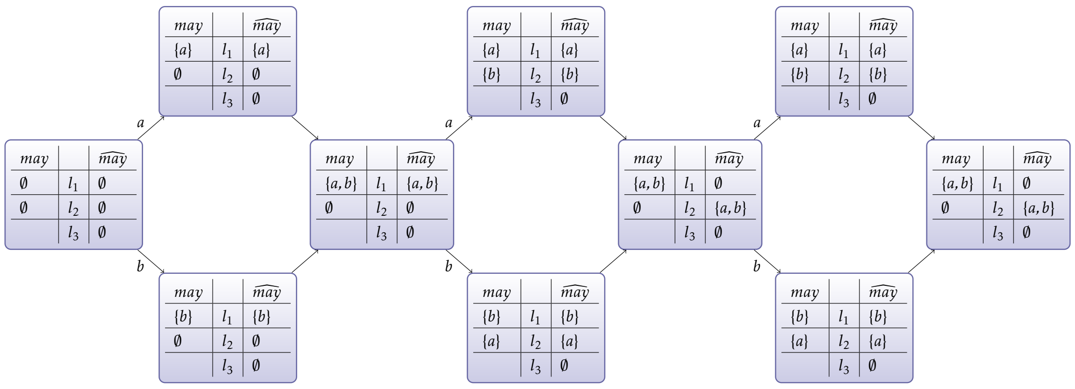


Figure 8.1: Fixed point iteration for the may-based persistence analysis for the if-then-else loop example (Figure 4.1): After three rounds, the fixed point is reached. The memory references to a and b are classified as *persistent*.

8.2.2 switch Loop

In contrast to the similar persistence analysis by Ferdinand, the may-based analysis handles the problematic switch example correctly, as illustrated in Figure 8.2. Unlike the analysis by Ferdinand, which uses the *must-analysis* aging, the novel analysis based on *may-analysis* aging computes in the third round of the fixed-point iteration that all of the accessed memory blocks a, b and c might be evicted. This is both a result of the used *may-analysis* aging for the entries and the over-approximation of the cache contents, which ensures that never more entries will be persistent than the cache can contain. Therefore none of the three memory references is classified as persistent after the fixed-point is reached. Any access to a, b or c might therefore be a miss, unlike the incorrect result that any of them can only cause at most one miss by Ferdinand.

8.3 Discussion of Analysis Properties

8.3.1 Soundness

Overview

For our persistence analysis we consider caches using the *LRU* replacement policy. We use the same characteristic of the *LRU* replacement semantics as in Section 7.3: An memory block x is earliest evicted from a k-way *LRU* cache after k accesses to different memory blocks.

Using this fact, the persistence classification as proposed in Definition 8.3 is sound, if for the abstract set state $s = (m, \hat{m})$ holds: If $x \in \hat{m}(l_i)$ at most $i-1$ further memory blocks got loaded after the last reference to x on any path. In other words: if memory block x is in the cache, its maximal age is i.

We will prove this by induction in three steps:

- Soundness of the initial abstract cache set state at the persistence scope entry;
- Soundness of the abstract update function $U_{pers_{may}}$;
- Soundness of the abstract join function $J_{pers_{may}}$.

Initial State

At the entry of the persistence scope we start with an empty abstract set state. As at the entry of the scope still no accesses have been done for any concrete execution, this is safe.

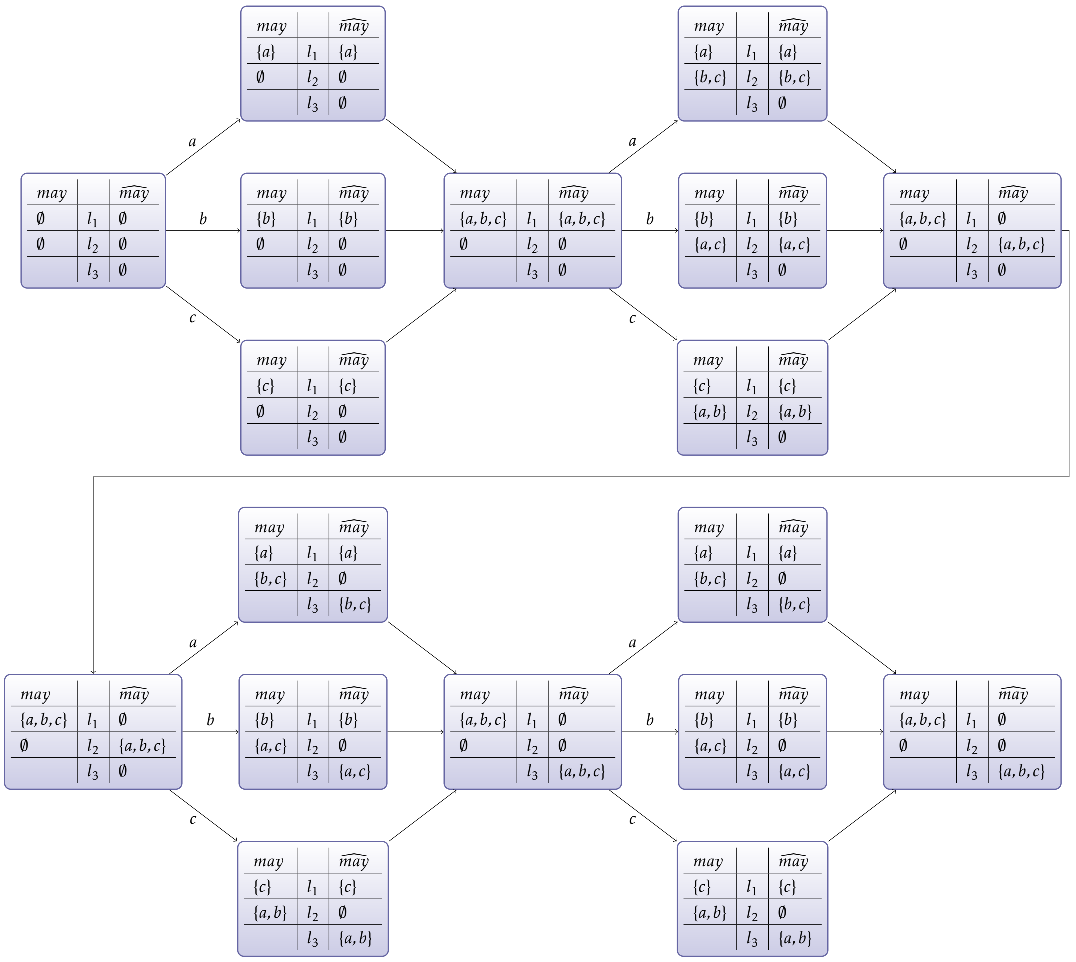

may | | $\widehat{may}$
a
b
c
∅ | l_1 | ∅; ∅ | l_2 | ∅; | l_3 | ∅
{a} | l_1 | {a}; ∅ | l_2 | ∅; | l_3 | ∅
{b} | l_1 | {b}; ∅ | l_2 | ∅; | l_3 | ∅
{c} | l_1 | {c}; ∅ | l_2 | ∅; | l_3 | ∅
{a,b,c} | l_1 | {a,b,c}; ∅ | l_2 | ∅; | l_3 | ∅
{a} | l_1 | {a}; {b,c} | l_2 | {b,c}; | l_3 | ∅
{b} | l_1 | {b}; {a,c} | l_2 | {a,c}; | l_3 | ∅
{c} | l_1 | {c}; {a,b} | l_2 | {a,b}; | l_3 | ∅
{a,b,c} | l_1 | ∅; ∅ | l_2 | {a,b,c}; | l_3 | ∅
{a,b,c} | l_1 | ∅; ∅ | l_2 | {a,b,c}; | l_3 | ∅
{a} | l_1 | {a}; {b,c} | l_2 | ∅; | l_3 | {b,c}
{b} | l_1 | {b}; {a,c} | l_2 | ∅; | l_3 | {a,c}
{c} | l_1 | {c}; {a,b} | l_2 | ∅; | l_3 | {a,b}
{a,b,c} | l_1 | ∅; ∅ | l_2 | ∅; | l_3 | {a,b,c}
{a} | l_1 | {a}; {b,c} | l_2 | ∅; | l_3 | {b,c}
{b} | l_1 | {b}; {a,c} | l_2 | ∅; | l_3 | {a,c}
{c} | l_1 | {c}; {a,b} | l_2 | ∅; | l_3 | {a,b}
{a,b,c} | l_1 | ∅; ∅ | l_2 | ∅; | l_3 | {a,b,c}

Figure 8.2: Fixed point iteration for the may-based persistence analysis for the switch loop example (Figure 5.3): After four rounds, the fixed point is reached. The memory references to a, b and c are not classified as *persistent*. All three memory blocks might be evicted during the loop.

Update Function

Given the input abstract set state $s = (m, \hat{m})$ for the update function $U_{pers_{may}}$ for a memory reference to the memory block x at program point p.

For $s = (m, \hat{m})$ holds:

- For any memory block y, $\exists l_i : y \in m(l_i)$ means if y is in the cache, then its age is i or larger, and $\nexists l_i : y \in m(l_i)$ means y is not in the cache.
- For any memory block y, $\exists l_i : i \leq A \wedge y \in \hat{m}(l_i)$ means if y is in the cache, then its age is i or less, and $\nexists l_i : y \in \hat{m}(l_i)$ means y has not been referenced so far.

The update function $U_{pers_{may}}$ now considers two cases for updates to $\hat{m}$:

- $mayevict(m, x)$ is true: This is the case if the *may analysis* abstract cache set contains at least associativity A many other memory blocks than x. In this situation, an reference to x may lead to an eviction in an *LRU* cache set, as the set may be already completely filled. Therefore the update function will increase the potential maximal ages of all memory blocks in $\hat{m}$. x itself will be set to age l_1.
- $mayevict(m, x)$ is false: This is the case if the *may analysis* abstract cache set contains less than associativity A many other memory blocks than x. The cache set is therefore still not full, no evictions can happen by loading x. Therefore the maximal ages of all memory blocks in $\hat{m}$ are only increased up to the age l_A, the highest possible age that may be in the cache. x itself will be set to age l_1.

Therefore $\hat{m}$ will still map all memory blocks to their maximal ages after the update.

In parallel the safe update function for the *may analysis* as shown in Definition 3.18 will be applied to m for an reference to x.

Join Function

Given the two abstract set states $s = (m, \hat{m})$ and $s = (m', \hat{m}')$ as input for the join function $J_{pers_{may}}$ for some control-flow join of two paths at program point p. s and s' are safe for the individual paths.

The join function will use the safe *may analysis* join function as shown in Definition 3.18 for the join of m and m'. The join for the $\hat{m}$ and $\hat{m}'$ is equivalently defined, but will maximize the ages. By construction the result is then safe for both paths.

8.3.2 Termination

As the number of different referenced memory blocks inside a program is finite, the abstract domain is finite, too. Therefore the termination of the analysis is guaranteed by the monotonicity of the abstract update $U_{pers_{may}}$ and join function $J_{pers_{may}}$ for the persistence sets.

8.4 Precision Improvements

In Section 7.5 two cases of imprecision in the set-wise conflict counting cache persistence analysis and the improvements by the element-wise conflict counting cache persistence analysis are demonstrated. We now review these cases for the may-based analysis.

8.4.1 Precision for unrolled loops

For the set-wise conflict counting cache persistence analysis it was not possible to classify any memory references inside the example from Figure 6.3 as persistent after one iteration. The element-wise conflict counting cache persistence analysis improved this and allowed to classify all memory references as persistent, which is the optimal case. In Figure 8.3 we show the fixed point iteration for this analysis. As a never leaves the may analysis set, the may-based persistence analysis allows possible evictions of b and c. Unlike the conflict counting per set analysis, this analysis will find the first evictions in the fourth round instead in the second round. Still, the fixed point result will not allow any memory references to be classified as persistent, which is a step back compared to the element-wise conflict counting analysis, which allows no evictions at all.

8.4.2 Precision for inner-iteration persistence

The other precision problem for the set-wise conflict counting was the *innerPersistenceLoop* as shown in Figure 6.5. Figure 8.4 shows the fixed point iteration for the

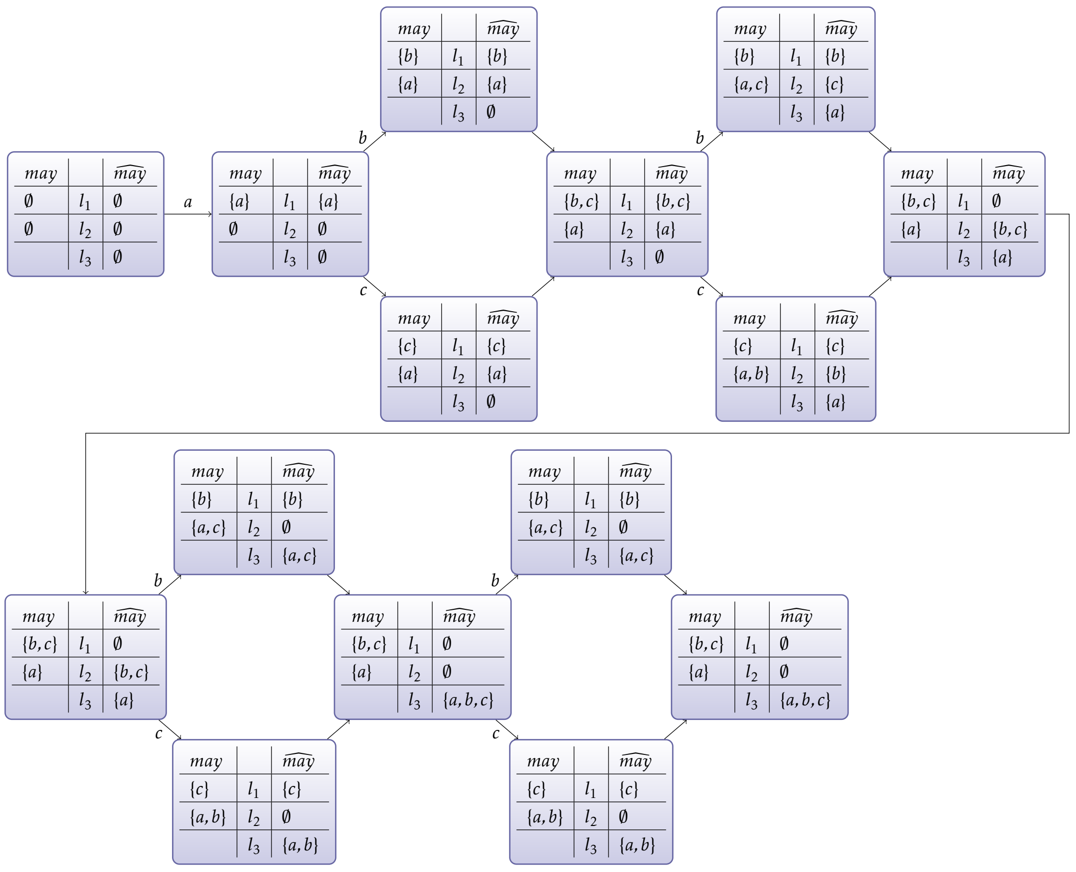


Figure 8.3: Fixed point iteration for the may-based persistence analysis for the example in Figure 6.3: After four rounds, the fixed point is reached. The memory references to *b* and *c* are classified as *not persistent*.

may-based analysis. Like the element-wise conflict counting analysis all memory references can be classified as persistent.

8.5 Benchmarking

Overview In Table 8.1 the results for the may-based persistence analysis for our benchmark scenarios from Section 6.6 are shown. The percentage of persistent classified memory references for the different scenarios are illustrated in Figure 8.5. For comparison the results for the previous two conflict counting analyses are included.

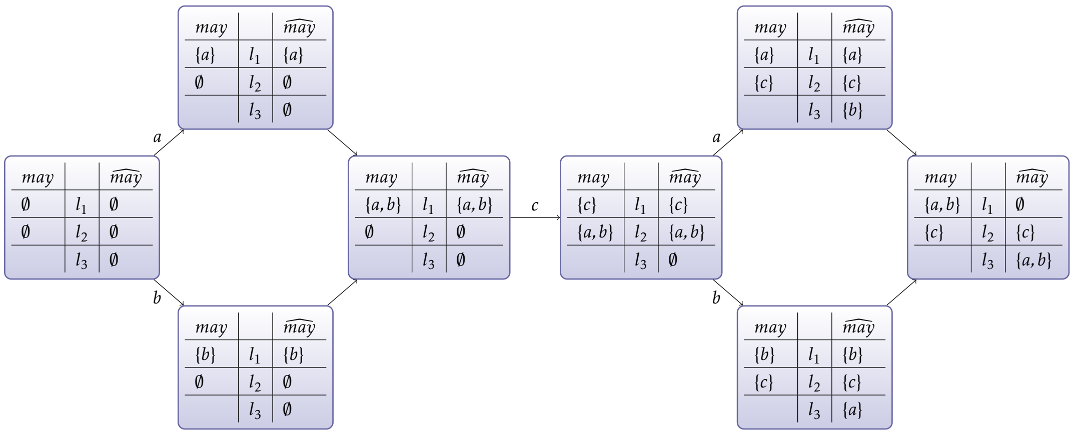


Figure 8.4: Analysis for one round of the innerPersistenceLoop loop example (Figure 6.5). The memory references to *a* and *b* are classified as persistent.

Review of the Scenarios *Scenario1* only accesses associativity different memory blocks. The may-based analysis classifies all memory references as persistent, like the two previous analyses. For all other scenarios, it shows the same weakness as the conflict counting per set analysis. Like the conflict counting per set, the *may analysis* will not recover from overfull sets and this leads to less persistence classifications. Only for the analysis runs, for which the *may analysis* set contains only the memory blocks accessed at the end, any persistent classifications are possible there. Unlike both previous analyses, the may-based analysis starts to evict memory blocks later during the unrolled iterations. This precision gain is visible in all testing scenarios.

Conclusion The results show, that the may-based persistence analysis performs well during the unrolled loop iterations. Tracking the maximal ages of the memory blocks in the set leads to less early evictions. But, unlike the conflict counting per set analysis, the may-based analysis doesn't perform well if the set was ever overfull.

For our benchmarking scenarios, the may-based persistence is always at least as precise as the simpler conflict counting per set. In comparison with the conflict counting per element, more precision is achieved in the unrolled iterations. The conflict counting per element is more precise, as it recovers better from overfull set situations.

	Unrolled Iterations			Last Not-Unrolled Iteration		
	References	Pers.	Pers. %	References	Pers.	Pers. %
scenario1	19998	19998	100.00%	19998	19998	100.00%
scenario2	209979	68821	32.78%	29997	0	0.00%
scenario3	25004	25004	100.00%	19998	13125	65.63%
scenario4	349706	152581	43.63%	39996	4	0.01%
scenario5	420025	420025	100.00%	79992	0	0.00%
scenario6	1199585	419424	34.96%	79992	0	0.00%
scenario7	699808	332928	47.57%	79992	2	0.00%
scenario8	419714	178644	42.56%	49995	2457	4.91%
scenario9	699967	363162	51.88%	99990	5361	5.36%
scenario10	1199472	419669	34.99%	119988	5825	4.85%
scenario11	1120730	495222	44.19%	139986	18821	13.44%

Table 8.1: Benchmark results for the may-based persistence analysis.

8.6 Summary

In this chapter, we introduced a *may cache analysis* based persistence analysis. The goal was to not evict memory blocks too early, even if the cache set becomes overfull.

Whereas this goal was reached, the may-based analysis doesn't always have as precise results as the element-wise conflict counting persistence analysis. It suffers from the same problem as the set-wise conflict counting persistence analysis. If the may cache set is once overfull during analysis, it doesn't recover well.

In the next chapter we will introduce a combination of this analysis and the element-wise conflict counting to overcome this limitation.

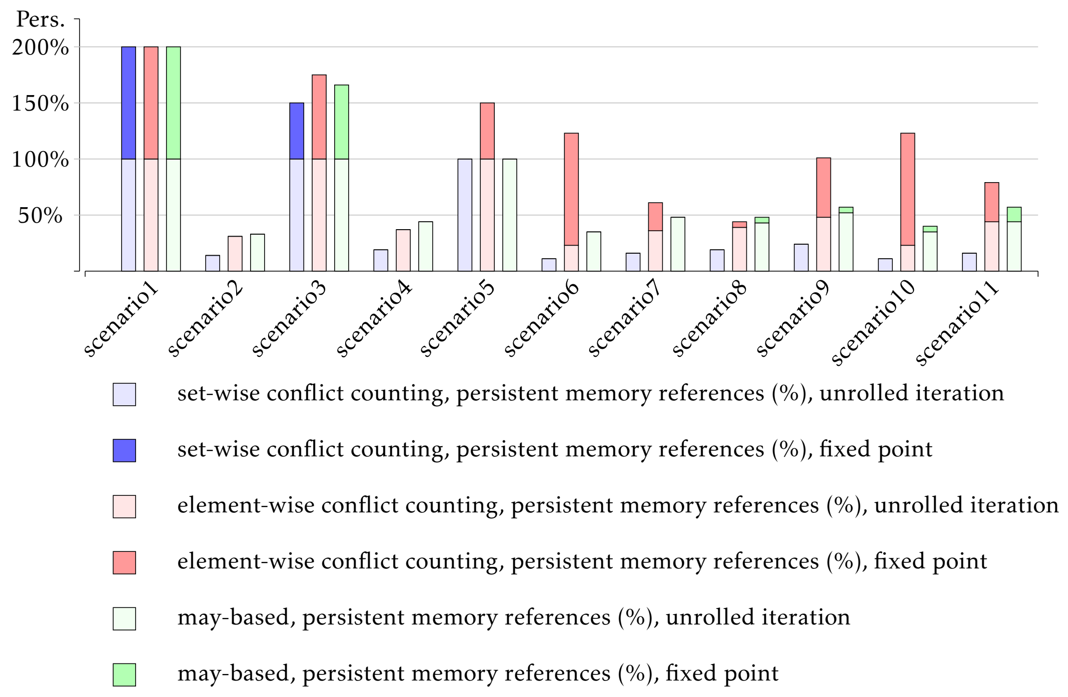


Figure 8.5: Comparison of analyses precision for different synthetic benchmark scenarios and persistence analyses.

CHAPTER 9

Age-Tracking Conflict Counting Persistence Analysis

9.1 Introduction of the Analysis

In Chapter 6 we have shown the basic idea of the set-wise conflict counting cache persistence analysis. Whereas this analysis yields sound results, it has some imprecisions as shown in Section 6.5.

We then introduced two more precise analyses to overcome this imprecision:

- A element-wise conflict counting persistence analysis in Chapter 7, that performed much better for our benchmarks during the fixed point.
- A may-based persistence analysis in Chapter 8, that performed better for our benchmarks during the unrolled iterations.

The precision improvements for both of these analyses stem from:

- The conflict per element analysis keeps track of a conflict set per accessed element. This allows the analysis to recover, if during the unrolled iterations an overfull cache set occurs, but the inside the remaining iterations that are not unrolled all elements fit into the cache set again.
- The may-based analysis keeps track of the maximal ages of the accessed elements. This avoids early evictions, if the set becomes overfull but there are not enough accesses to actually trigger an eviction during concrete execution.

These behaviors are confirmed by the results of the benchmarking scenarios in Section 7.6 and Section 8.5.

We now design an analysis that combines both precision increasing components: conflict counting and maximal age tracking per element.

We reuse the abstract domain of the conflict counting per element analysis and extend it with an upper bound of the element age.

Definition 9.1 (Abstract Set Domain – Age-Tracking Conflict Counting)
The abstract domain $\mathcal{D}_{pers_{ca}}$ of the age-tracking conflict counting persistence analysis *for one cache set with associativity A is a mapping from memory blocks to maximal ages $\mathcal{A} = \{0..A\}$ and potential conflict sets $M \to 2^M$. It is defined as $\mathcal{D}_{pers_{ca}} := M \to \mathcal{A} \times 2^M$.*

Definition 9.2 (Set Update and Join – Age-Tracking Conflict Counting)
The abstract update function $U_{pers_{ca}} : \mathcal{D}_{pers_{ca}} \times M \to \mathcal{D}_{pers_{ca}}$ and join function $J_{pers_{ca}} : \mathcal{D}_{pers_{ca}} \times \mathcal{D}_{pers_{ca}} \to \mathcal{D}_{pers_{ca}}$ of the age-tracking conflict counting persistence analysis *are defined as:*

$$U_{pers_{ca}}(m,x) = \begin{array}{l} [x \mapsto (0, U_{pers_{cs}}(\emptyset, x)), \\ y \mapsto (min(age+1, A), U_{pers_{cs}}(m(y), x)) \\ \quad with\ (age, set) = m(x) \mid y \neq x \wedge m(y) \neq (0, \emptyset) \\ y \mapsto (0, \emptyset) \mid y \neq x \wedge m(y) = (0, \emptyset)]; \end{array}$$

$$\begin{aligned} J_{pers_{ca}}(m, m') = [x \mapsto (&max(age, age'), J_{pers_{cs}}(set, set') \\ &with\ (age, set) = m(x) \wedge (age', set') = m'(x))]; \end{aligned}$$

They reuse the abstract update function $U_{pers_{cs}}$ and join function $J_{pers_{cs}}$ of the set-wise conflict counting persistence analysis *as shown in Definition 6.2.*

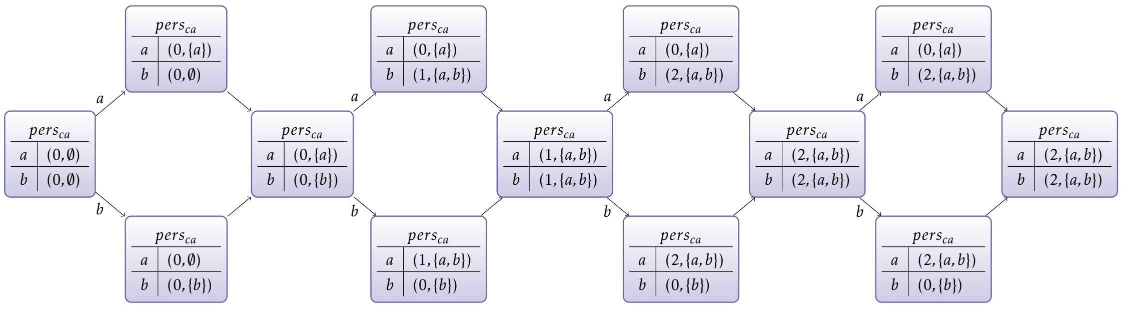
If the program references a memory block $x \in M$ we ensure that we update its conflict counting set and reset its upper age bound to 0, as it is now the latest memory block in the cache set. Given the definition of the $U_{pers_{cs}}$ function, this ensures we have at least x itself inside this set afterwards. For all other memory blocks $y \in M \wedge y \neq x$, we only update the individual conflicting sets if y was already accessed, i.e. the set for y is not empty. For these memory blocks, we increment the age bound up to the maximum of A, which means potentially evicted.

Definition 9.3 (Persistence – Conflict Counting with Aging)
For a given abstract set state m of a cache set with associativity A, a memory reference to $x \in M$ is persistent iff $age < A \vee |set| \leq A$ for $(age, set) = m(x)$.

9.2 Application to the Examples

9.2.1 if-then-else Loop

Figure 9.1 shows the analysis on the if-then-else example, for which it computes the same results as both element-wise conflict counting and may-based analyses. As for the other analyses, this analysis start with an empty persistence set at the loop entry. Then the set is updated for either *a* or *b* in parallel and the resulting sets are joined afterwards. As the computed conflict sets for all memory blocks contain not more entries than the associativity of the cache, both memory references are classified as persistent and therefore each of them can only cause at most one miss.



$pers_{ca}$ | a $(0,\emptyset)$ | b $(0,\emptyset)$
a
$pers_{ca}$ | a $(0,\{a\})$ | b $(0,\emptyset)$
b
$pers_{ca}$ | a $(0,\emptyset)$ | b $(0,\{b\})$
$pers_{ca}$ | a $(0,\{a\})$ | b $(0,\{b\})$
a
$pers_{ca}$ | a $(0,\{a\})$ | b $(1,\{a,b\})$
b
$pers_{ca}$ | a $(1,\{a,b\})$ | b $(0,\{b\})$
$pers_{ca}$ | a $(1,\{a,b\})$ | b $(1,\{a,b\})$
a
$pers_{ca}$ | a $(0,\{a\})$ | b $(2,\{a,b\})$
b
$pers_{ca}$ | a $(2,\{a,b\})$ | b $(0,\{b\})$
$pers_{ca}$ | a $(2,\{a,b\})$ | b $(2,\{a,b\})$
a
$pers_{ca}$ | a $(0,\{a\})$ | b $(2,\{a,b\})$
b
$pers_{ca}$ | a $(2,\{a,b\})$ | b $(0,\{b\})$
$pers_{ca}$ | a $(2,\{a,b\})$ | b $(2,\{a,b\})$

Figure 9.1: Fixed point iteration for the age-tracking conflict counting persistence analysis for the if-then-else loop example (Figure 4.1): After four rounds, the fixed point is reached. The memory references to *a* and *b* are classified as *persistent*.

9.2.2 switch Loop

The more complex example with three memory references is presented in Figure 9.2. Like the element-wise conflict counting, this analysis computes a fixed point result that allows none of the three memory references to be classified as persistent. But with the age tracking, the analysis can classify all memory references in the first three iterations as persistent, unlike simple conflict counting. This will allow for a more precise analysis if we unroll the loop, as only one miss per memory block is allowed in the first three iterations, which is the optimal result. Only beginning with the fourth round more misses are possible. This is as precise as the result of the may-based analysis.

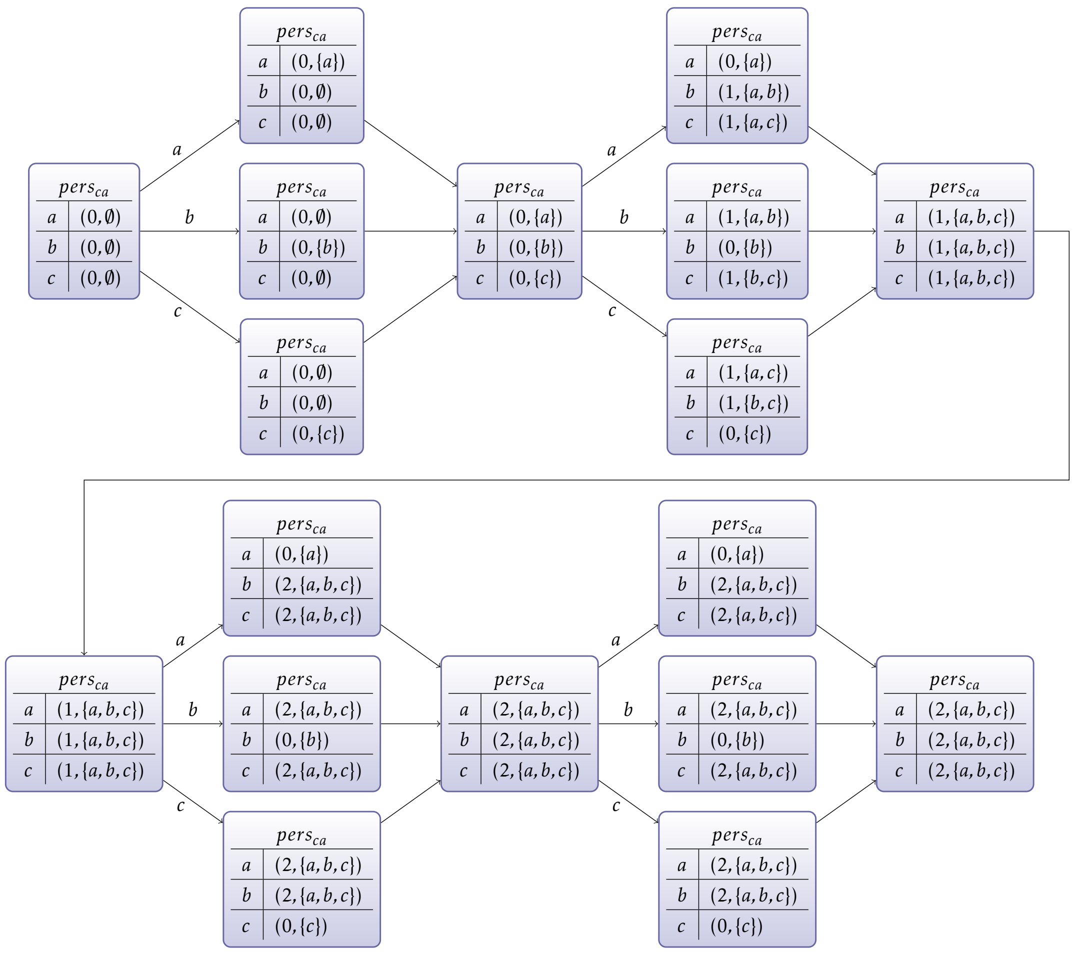

$pers_{ca}$: a $(0,\emptyset)$, b $(0,\emptyset)$, c $(0,\emptyset)$
a
b
c
$pers_{ca}$: a $(0,\{a\})$, b $(0,\emptyset)$, c $(0,\emptyset)$
$pers_{ca}$: a $(0,\emptyset)$, b $(0,\{b\})$, c $(0,\emptyset)$
$pers_{ca}$: a $(0,\emptyset)$, b $(0,\emptyset)$, c $(0,\{c\})$
$pers_{ca}$: a $(0,\{a\})$, b $(0,\{b\})$, c $(0,\{c\})$
a
b
c
$pers_{ca}$: a $(0,\{a\})$, b $(1,\{a,b\})$, c $(1,\{a,c\})$
$pers_{ca}$: a $(1,\{a,b\})$, b $(0,\{b\})$, c $(1,\{b,c\})$
$pers_{ca}$: a $(1,\{a,c\})$, b $(1,\{b,c\})$, c $(0,\{c\})$
$pers_{ca}$: a $(1,\{a,b,c\})$, b $(1,\{a,b,c\})$, c $(1,\{a,b,c\})$
$pers_{ca}$: a $(1,\{a,b,c\})$, b $(1,\{a,b,c\})$, c $(1,\{a,b,c\})$
a
b
c
$pers_{ca}$: a $(0,\{a\})$, b $(2,\{a,b,c\})$, c $(2,\{a,b,c\})$
$pers_{ca}$: a $(2,\{a,b,c\})$, b $(0,\{b\})$, c $(2,\{a,b,c\})$
$pers_{ca}$: a $(2,\{a,b,c\})$, b $(2,\{a,b,c\})$, c $(0,\{c\})$
$pers_{ca}$: a $(2,\{a,b,c\})$, b $(2,\{a,b,c\})$, c $(2,\{a,b,c\})$
a
b
c
$pers_{ca}$: a $(0,\{a\})$, b $(2,\{a,b,c\})$, c $(2,\{a,b,c\})$
$pers_{ca}$: a $(2,\{a,b,c\})$, b $(0,\{b\})$, c $(2,\{a,b,c\})$
$pers_{ca}$: a $(2,\{a,b,c\})$, b $(2,\{a,b,c\})$, c $(0,\{c\})$
$pers_{ca}$: a $(2,\{a,b,c\})$, b $(2,\{a,b,c\})$, c $(2,\{a,b,c\})$

Figure 9.2: Fixed point iteration for the age-tracking conflict counting persistence analysis for the switch loop example (Figure 5.3): After four rounds, the fixed point is reached. None of the memory references to *a*, *b* and *c* are classified as *persistent*.

9.3 Discussion of Analysis Properties

9.3.1 Soundness

Overview

The persistence classification as proposed in Definition 9.3 contains two cases.

An memory references to the memory block x is persistence, for the abstract set state m of an k-way associative cache, iff:

- $age < k$ or
- $|set| \leq k$

with $(age, set) = m(x)$.

The conflicts sets for each memory block inside the abstract set state is updated like for the conflict counting per set analysis introduced in Chapter 7. The same soundness proof as shown in Section 7.3 holds for this part of the persistence definition of this analysis.

To ensure the soundness of the above definition, we must ensure that the $age < k$ case allows for a sound persistence classification.

For our persistence analysis we consider caches using the *LRU* replacement policy. Like in Section 6.3 we use a characteristic of the *LRU* replacement semantics as defined in Section 3.3.2: A memory block x is earliest evicted from a k-way *LRU* cache after k accesses to different memory blocks.

Using this fact, the persistence classification is sound, if for the abstract set state m holds: age is an upper bound for the age of a memory block x potentially in the cache, with $(age, set) = m(x)$.

We will prove this by induction in three steps:

- Soundness of the initial abstract cache set state at the persistence scope entry
- Soundness of the abstract update function $U_{pers_{ca}}$
- Soundness of the abstract join function $J_{pers_{ca}}$

Initial State

At the entry of the persistence scope the analysis starts with an abstract set state $m = [x\ to(0, \emptyset) \mid \forall x \in M];$. As at the entry of the scope still no accesses are done for any concrete execution, the age 0 is a valid upper bound of the age of the memory blocks loaded since start of the persistence scope in the cache.

Update Function

Given the input abstract set state m for the update function $U_{pers_{ca}}$ for a memory reference to the memory block x at program point p.

The update function will modify m in the following ways:

- It will reset the age of x itself to 0. As we just have accessed x, this is a correct upper bound of the age of x.
- For all other $m(y) \mid y \in M$ with $y \neq x$ there are two cases:
  - $m(y) = (0, \emptyset)$: As $m(y)$ is an over-approximation, this means, y is not loaded in the cache since the start of the scope, as the conflicting set is still empty. Therefore we don't need to increase the age 0, as y isn't in the cache. This holds because $\emptyset$ is only possible, if y was not loaded already, see proof in Section 7.3.
  - $m(y) \neq (0, \emptyset)$: The age of y will be increased. Given LRU replacement, this is a safe upper bound of the age of y after the access to x.

Therefore the update function results in a sound over-approximation after program point p with safe upper age bounds.

Join Function

Given the two abstract set states m and m' as input for the join function $J_{pers_{ca}}$ for some control-flow join of two paths at program point p. For all $x \in M$ $(age, set) = m(x)$ and $(age', set') = m'(x)$ are upper bounds of the ages and over-approximations of the loaded memory blocks that conflict with x on the individual paths.

As the join function is the set union for the individual set and set' for all $x \in M$, the result of the join will be an over-approximation of the loaded memory blocks that conflict with x for both paths, as proven in Section 7.3. As the join function will maximize *age* and *age′* of the element x, the same holds for the resulting age, it still is an upper bound for both paths.

9.3.2 Termination

As the number of different referenced memory blocks inside a program is finite, the abstract domain $\mathcal{D}_{pers_{ca}} = M \rightarrow \mathcal{A} \times 2^M$ is finite, too. Therefore the termination of the analysis is guaranteed by the monotonicity of the abstract update $U_{pers_{ca}}$ and join function $J_{pers_{ca}}$ for the persistence sets.

9.4 Analysis Space-Optimization

We can apply the same optimization as described in Section 6.4. As the cache associativity is constant for the analysis, the space complexity per set is reduced from $O(n^2)$ to $O(n)$ if the program references n different elements inside the persistence scope.

Definition 9.4 (Optimized Set Update and Join – Age-Tracking Conflict Counting) *The abstract update function $U_{pers_{ca\_optimized}}$ and join function $J_{pers_{ca\_optimized}}$ for the space-optimized variant are defined as:*

$$U_{pers_{ca_{optimized}}}(m,x) = \begin{array}{l} [x \mapsto (0, U_{pers_{cs_{optimized}}}(\emptyset, x)), \\ y \mapsto (min(age+1, A), U_{pers_{cs_{optimized}}}(m(y), x)) \\ \quad with\ (age, set) = m(x) \mid y \neq x \wedge m(y) \neq (0, \emptyset) \\ y \mapsto (0, \emptyset) \mid y \neq x \wedge m(y) = (0, \emptyset)]; \end{array}$$

$$\begin{aligned} J_{pers_{ca_{optimized}}}(m,m') = {} & [x \mapsto (max(age, age'), J_{pers_{cs_{optimized}}}(set, set') \\ & with\ (age, set) = m(x) \wedge (age', set') = m'(x))]; \end{aligned}$$

9.5 Precision Improvements

In Section 6.5 two cases of imprecision in the conflict counting per set analysis are demonstrated. We now review these cases for the improved conflict counting with aging analysis.

9.5.1 Precision for unrolled loops

For the set-wise conflict counting analysis it was not feasible to classify any memory references inside the example from Figure 6.3 as persistent after one iteration, whereas in practice neither b nor c will never be evicted. In Figure 9.3 we show the fixed point iteration for this analysis. Like the element-wise conflict counting analysis, this analysis is able to classify the memory references to b and c as persistent for all iterations. Only a will be possibly evicted after the first iteration.

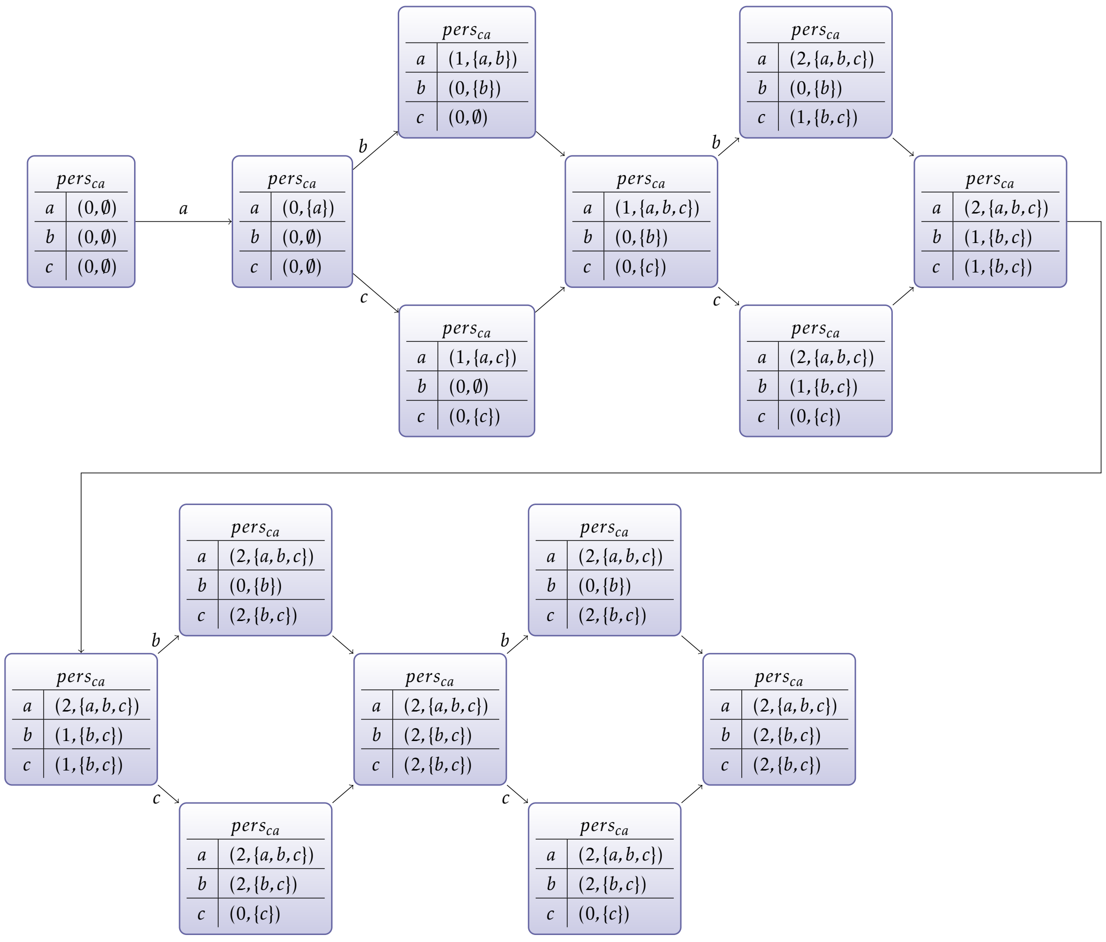


Figure 9.3: Fixed point iteration for the age-tracking conflict counting analysis for the example in Figure 6.3: After two rounds, the fixed point is reached. The memory references to b are classified as *persistent*.

9.5.2 Precision for inner-iteration persistence

The other precision problem for the set-wise conflict counting was the *innerPersistenceLoop* as shown in Figure 6.5. Figure 9.4 shows the fixed point iteration for this analysis. Like both the may-based and element-wise conflict counting analyses, all memory references are classified as persistent.

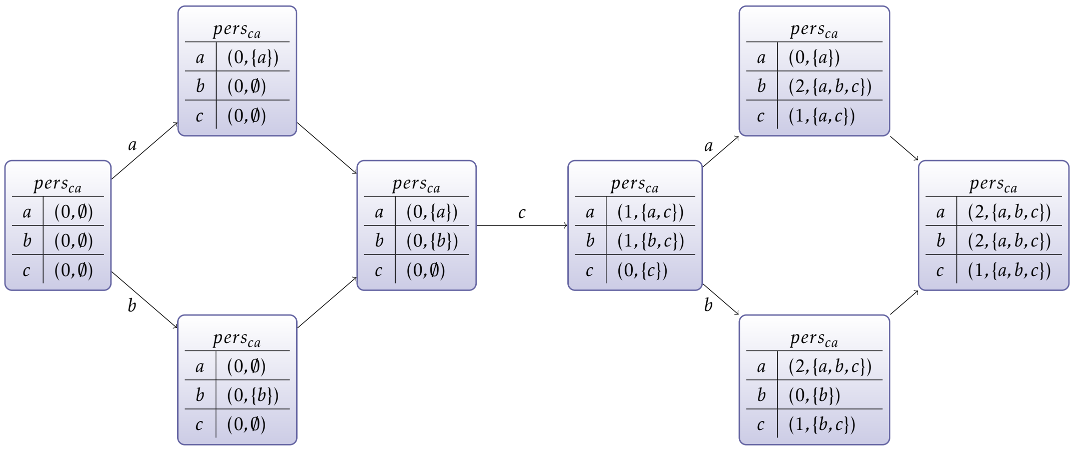

$pers_{ca}$: a (0,∅), b (0,∅), c (0,∅)
a
$pers_{ca}$: a (0,{a}), b (0,∅), c (0,∅)
b
$pers_{ca}$: a (0,∅), b (0,{b}), c (0,∅)
$pers_{ca}$: a (0,{a}), b (0,{b}), c (0,∅)
c
$pers_{ca}$: a (1,{a,c}), b (1,{b,c}), c (0,{c})
a
$pers_{ca}$: a (0,{a}), b (2,{a,b,c}), c (1,{a,c})
b
$pers_{ca}$: a (2,{a,b,c}), b (0,{b}), c (1,{b,c})
$pers_{ca}$: a (2,{a,b,c}), b (2,{a,b,c}), c (1,{a,b,c})

Figure 9.4: Age-tracking conflict counting analysis for one round of the innerPersistenceLoop loop example (Figure 6.5). The memory references to *a* and *b* are classified as persistent.

9.6 Benchmarking

Overview In Table 9.1 the results for the age-tracking conflict counting persistence analysis for our benchmark scenarios from Section 6.6 are shown. The percentage of persistent classified memory references for the different scenarios are illustrated in Figure 9.5. For comparison the results for the previous two conflict counting analyses and the may-based analysis are included.

Review of the Scenarios By construction, the age-tracking conflict counting persistence analysis is at least as precise as the element-wise conflict counting and the may-based analyses. For the simple *scenario1* all analyses are precise. All memory references are persistent, as not more than associativity different memory blocks are accessed. For all other scenarios, the age-tracking conflict counting persistence analysis is the most precise one. It combines the better handling of the unrolled iterations by the age tracking of the may-based analysis with the better recovery for overfull sets for unrolled iterations of the conflict counting per element analysis.

Conclusion The results show that the novel age-tracking conflict counting combines the precision of the two different approaches presented before:

- good handling of overfull sets during the unrolled iterations, by counting the conflicts per element
- only allow evictions after the elements are old enough, by tracking upper bounds for the element ages

	Unrolled Iterations			Last Not-Unrolled Iteration		
	References	Pers.	Pers. %	References	Pers.	Pers. %
scenario1	19998	19998	100.00%	19998	19998	100.00%
scenario2	209979	72443	34.50%	29997	0	0.00%
scenario3	25004	25004	100.00%	19998	15633	78.17%
scenario4	349706	155188	44.38%	39996	15	0.04%
scenario5	420025	420025	100.00%	79992	39996	50.00%
scenario6	1199585	419428	34.96%	79992	79992	100.00%
scenario7	699808	333391	47.64%	79992	19998	25.00%
scenario8	419714	184908	44.06%	49995	2489	4.98%
scenario9	699967	373711	53.39%	99990	52510	52.52%
scenario10	1199472	419677	34.99%	119988	119988	100.00%
scenario11	1120730	516721	46.11%	139986	49276	35.20%

Table 9.1: Benchmark results for the age-tracking conflict counting persistence analysis.

9.7 Summary

In this chapter we introduced a novel persistence analysis combining the benefits of both the conflict counting and the may-based approach. The analysis not only

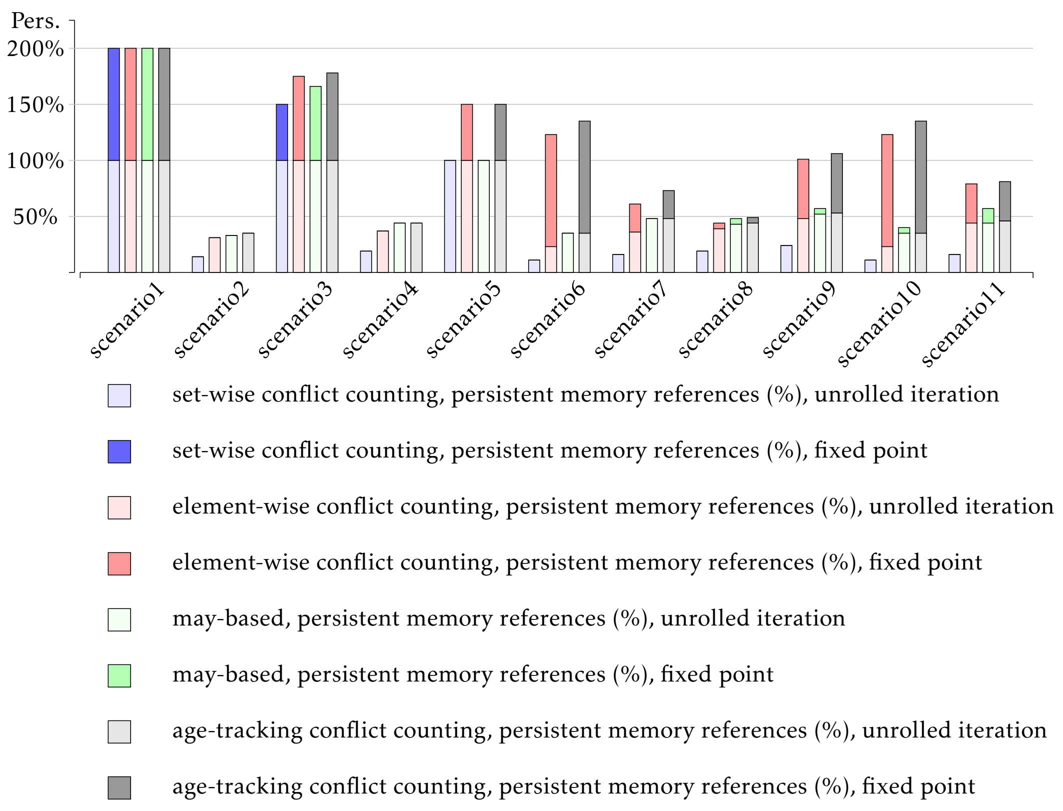


Figure 9.5: Comparison of analyses precision for different synthetic benchmark scenarios and persistence analyses.

collects all possible conflicts per element but also tracks the possible maximal age of each element.

The application to our imprecision examples and the evaluation in our benchmarking framework show, that these combined techniques compute in the most precise results.

In the next chapter, we will evaluate our new analysis inside a state-of-the-art *WCET* analysis framework. We will implement the analysis for different target architectures in the commercial *aiT WCET* analyzer and evaluate it on both academic benchmarks and real-word software.

CHAPTER 10

Practical Evaluation

In this chapter, we evaluate the novel persistence analyses on different hardware architectures within the analysis framework of the aiT WCET analyzer. First we briefly introduce the overall architecture of aiT and focus on the parts which need adjustment for the cache persistence analysis. Then we introduce the different architectures we have selected for the evaluation and why these architectures are of general interest. Afterwards we describe the test setup and the different programs which are used for the evaluation. This chapter is concluded with the presentation and analysis of the results.

10.1 The aiT WCET Analyzer Framework

aiT is a commercial WCET analysis tool developed by AbsInt GmbH. The architecture of the aiT analyzer is shown in Figure 10.1. It follows a more or less standard architecture for timing-analysis tools [HWH95, TFW00, Erm03, FMC+07, CFG+10]. aiT has been successfully used to determine precise bounds on execution times of real-time software [FW99, FHL+01, TSH+03, HLTW03, SLH+05]. We will now briefly introduce the major parts of this framework and then focus on the components which needs adjustments for the persistence analysis evaluation.

10.1.1 Control-flow Reconstruction

This phase identifies the instructions and their operands, and reconstructs the program control-flow. The reconstructed control-flow is annotated with the information needed by subsequent analysis phases and then translated into a control-flow graph [The00, The02].

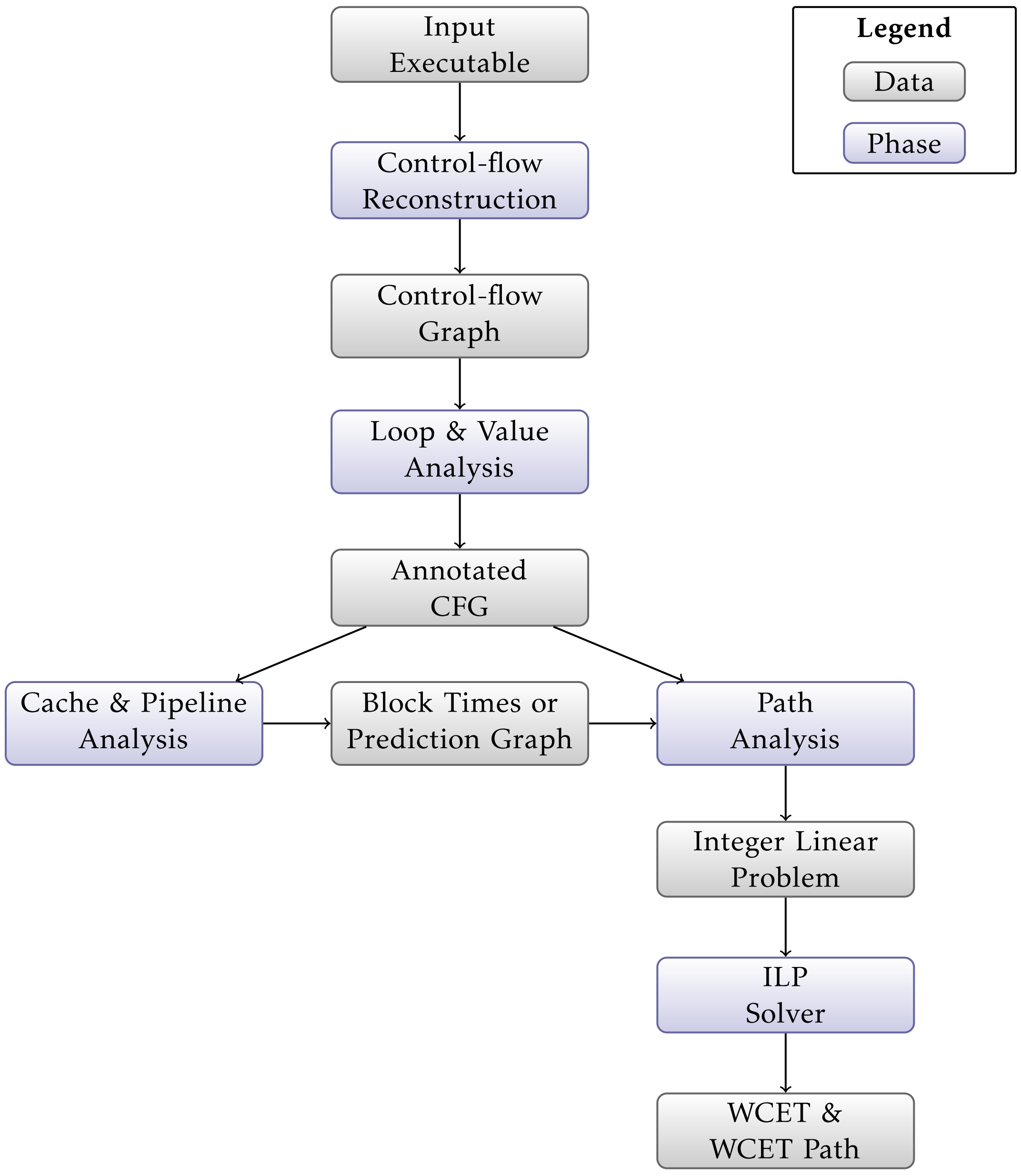


Figure 10.1: Components of the aiT framework and their interaction.

10.1.2 Loop & Value Analysis

This phase tries to determine the values in the processor registers and memory cells for every program point and execution context [Sic97]. Often it cannot determine these values exactly, but only finds safe lower and upper bounds, i.e. intervals that are guaranteed to contain the exact values. In addition this phase computes lower and upper bounds for loops inside the control-flow graph [CM07, HSR+00] and additional flow-facts like information about never executed program paths.

10.1.3 Cache & Pipeline Analysis

The integrated cache and pipeline analysis classifies the memory references to main memory and models the pipeline behavior to determine execution times for all basic blocks in the individual analysis contexts. The analysis takes the current pipeline state(s) into account, in particular resource occupancies, contents of processor internal queues (prefetch, load and store), grouping of instructions, and classification of memory references as cache hits or misses [Eng02, The04, FW99].

10.1.4 Path Analysis

This last phase models the program's control flow by an ILP [TFW00] so that the solution to the objective function is the predicted worst-case execution time bound for the input program. This ILP is then solved using one of the common optimized MIP solvers, like the open source solver *CBC* from the *COIN-OR* project [LH03] or *CPLEX* from *ILOG*.

10.1.5 Relevant Parts for Persistence Analysis

The interesting parts of the framework for the application of the cache persistence analysis are the combined cache and pipeline analysis and the path analysis. The cache and pipeline analysis incorporates the persistence cache analysis and the path analysis is modified to add additional constraints into the generated ILP.

10.2 Cache & Pipeline Analysis

The pipeline analysis determines upper bounds of the execution times of basic blocks performing an abstract interpretation of the program execution on the particular architecture, taking into account its pipeline, caches, memory buses, and attached peripheral devices. By means of an abstract model of the hardware architecture, the pipeline analysis statically simulates the execution of each instruction.

The integrated cache analysis provides safe approximations of the contents of the caches at each program point using the must and may cache analyses.

Abstract states may lack information about the state of some processor components, e.g., caches, queues, or predictors. Transitions of the pipeline may depend on such missing information. This causes the abstract pipeline model to become non-deterministic although the concrete pipeline is deterministic. When dealing with this non-determinism, one could be tempted to design the WCET analysis such that only the locally most expensive pipeline transition is chosen. However, in the presence of *timing anomalies* [LS99, RWT+06, Geb10] this approach is unsound. Thus, in general, the analysis has to follow all possible successor states, it will *split* for any possible successor [The04].

As a result of this *splitting* the pipeline analysis produces no linear execution trace but a graph for all possible executions, the so-called *prediction graph*. Figure 10.2 shows a simple prediction graph part for two consecutive basic blocks *a* and *b*. The pipeline analysis gets for block *a* one incoming state. Due to unknown cache behavior it produces two outgoing states. The miss case takes 10 cycles, the hit case two. The next block *b* then has these two states as input and computes two output states, this time for the resulting state of the miss the processing time is one and for the hit it is 10. Actually this is a *timing anomaly*, a case in which a locally worst decision (assuming the miss with 10 cycles) will lead to an overall shorter execution time of 11 cycles, shorter than the locally best decision, the hit case, with overall 12 cycles.

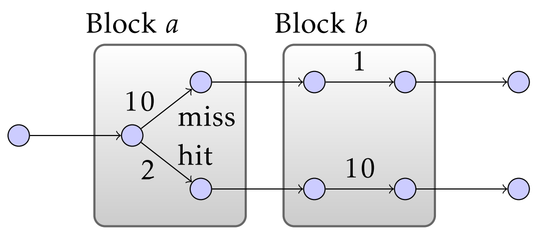


Figure 10.2: Small snippet of a prediction graph: Two subsequent basic blocks *a* and *b* and their internal state graph created by the pipeline analysis.

Two different outputs for the subsequent path analysis are available:

The Basic Block Times The pipeline analysis performs abstract simulation internally and follows all possible paths in the prediction graph for an individual basic block, but only outputs the upper bound of the execution time of the block to the path analysis. For the example in Figure 10.2 this would be ten cycles for each of the two basic blocks.

The Prediction Graph The pipeline analysis outputs the whole computed prediction graph to allow the path analysis to incorporate it into the overall control flow graph for the ILP. For the example in Figure 10.2 this means that the relation between the different states is kept and that therefore a global bound of 12 can be determined.

Only the prediction graph allows the path analysis to build additional constraints for the number of cache misses as required by the persistence analysis. The pipeline analysis annotates the prediction graph not only if it assumed a miss or hit, but also if the corresponding cache line was classified as persistent. This information will be used by the path analysis to build an optimized ILP.

10.3 Path Analysis

There are different kinds of path analyses available in aiT, the normal ILP based path analysis on basic block level and the ILP based path analysis on the prediction graph. Both construct an ILP to represent the control flow and use the output of the pipeline analysis as coefficients.

The normal ILP based path analysis works on the granularity of basic blocks and uses the maximal basic block timings computed by the pipeline analysis. It was introduced by Theiling [The02] and is used successfully in practice with quite precise results, for example for avionics software [TSH+03].

Unfortunately its level of granularity does not allow to express additional constraints like the number of allowed misses on the WCET path, as only constraints on block level are possible and both hits and misses can happen inside one block (see Figure 10.2).

Therefore the novel ILP based path analysis on the prediction graph is used to integrate the persistence analysis. This path analysis is based on the master thesis of Matthies [Mat06] that introduced a path analysis on the prediction graph for the special case of an acyclic graph. It was designed to increase the precision by honoring inter-leavings over basic block boundaries in the path analysis.

For the example snippet in Figure 10.2 the prediction graph based path analysis would not assume an execution time of 10 cycles per block which results in 20 cycles for both blocks. Instead it constructs an ILP that honors the possible state transitions and allows only a maximal execution time of overall 12 cycles for the complete snippet. This concept was extended during development of the novel persistence analysis to handle cyclic graphs by using the concepts of

Theiling [The02] to add additional constraints to the ILP to bound loops and recursions.

For each *persistence scope* the path analysis will create additional constraints in the ILP to restrict the amount of misses for all *persistent memory references* to the same memory block to at most one miss per entering of the scope. This is the usage of the *persistence constraint* as formulated in Theorem 4.2.

For the example of Figure 4.1 this would mean that the following two constraints for *persistent memory references* to memory blocks a and b will be added:

$$\sum \text{misses}(\text{persistentReferences}(a)) \leq 1$$
$$\sum \text{misses}(\text{persistentReferences}(b)) \leq 1$$

These additional constraints will restrict the WCET path computed by the ILP solver to only allow paths which have at most one miss for these two cache lines and to cut away all other paths that would lead to over-estimations because of unnecessary memory accesses to reload the cache lines.

10.4 Selection of Architectures

The chosen WCET analysis framework aiT supports a wide range of very different embedded hardware architectures. For the selection of interesting architectures we both consider a state-of-the-art classification of architectures with respect to timing analysis and their use in the current academic and industrial community. Based on this, we select the aiT WCET analyzer for the *ARM7*, *MPC5554* and *MPC755* architectures.

10.4.1 Classification of Architectures

In respect to timing analysis there exists a widely used classification of architectures by Wilhelm et al. [WGR$^+$09]:

Fully timing compositional architectures The (abstract model of an) architecture does not exhibit timing anomalies. Hence, the analysis can safely follow local worst-case paths only.

Compositional architectures with constant-bounded effects The architecture exhibit timing anomalies but no domino effects. In general, an analysis has to consider all paths. To trade precision with efficiency, it would be possible to safely discard local non-worst-case paths by adding a constant number of cycles to the local worst-case path [RS09].

Non-compositional architectures The architecture exhibit domino effects and timing anomalies. For such architectures timing analyses always have to follow all paths since a local effect may influence the future execution arbitrarily.

Our three selected architecture spread over all three classifications:

Fully timing compositional architectures The *ARM7* is the classical example for such an architecture.

Compositional architectures with constant-bounded effects The *MPC5554* is assumed to belong to this class, if certain hardware features are configured correctly, to avoid known domino effects because of the used cache replacement strategies. The instruction and data cache with *Pseudo-RoundRobin* replacement must be locked down to one way and the flash buffer and *BTB* with *FIFO* replacement must be deactivated.

Non-compositional architectures The *MPC755* is such an architecture, as it exibits at least one known domino effect because of its imbalanced integer units [Sch03].

10.4.2 Relevance in Practice

ARM7 TDMI The *ARM7* is widely-used in the academic world, as *ARM7* evaluation hardware is cheap and there exist cycle accurate software simulators like the *MPARM* simulator [BBB+05].

Freescale MPC5554 The *MPC5554* is a modern state-of-the-art architecture designed for the automotive & avionics industry. Is is widely used in both fields, even in the hard-realtime avionics domain.

Freescale MPC755 The *MPC755* is a high-performance architecture used in safety-critical avionics systems, like the *Airbus A380* [SLH+05]. Whereas the *MPC755* was not designed to be used for hard-realtime systems, the performance demands of current control-software required an architecture with its performance.

10.5 Test Selection & Hardware Setup

10.5.1 ARM7 TDMI Tests

The *ARM7* is configured with separated 4 KB 4-way associative instruction and data caches with *LRU* replacement policy.

As evaluation tests we will use:

Our Running Examples We will use the small examples from the previous chapters. The persistent if-then-else loop construct *ifelse* (Figure 4.1), the switch statement that leads to evictions *switch* (Figure 5.3) and the loop that has a different behavior in the first iteration *prefix* (Figure 6.3). They are analyzed with two settings, without unrolling and with loop unrolling (the unrolled examples have the *_u* suffix).

WCET Benchmarks We analyze a subset of the well-known *WCET Benchmarks* from Mälardalen university [GBEL10].

10.5.2 Freescale MPC5554 Tests

The *MPC5554* features 32 KB instruction cache with *Pseudo-RoundRobin* replacement policy. To make it analyzable it is locked down to one way, resulting in an 4 KB direct mapped cache.

We use 20 code examples that are derived from real avionic applications. For example they feature error and message handling, complex state machines and include both hand written and generated code.

10.5.3 Freescale MPC755 Tests

The *MPC755* features seperated 32 KB instruction and data caches with *PLRU* replacement policy. To make it analyzable, the instruction cache is locked down to two ways, resulting in an effective instruction cache size of 8 KB with *LRU* replacement. The data cache is locked down to 16 KB as one half is used as stack space.

Like for the *MPC5554*, our code examples are derived from real avionic applications. We will analyze two different categories of tasks:

755_avionics1 .. 755_avionics10 Smaller tasks with below 16 KB code size. Most of them can fully fit into the 8 KB instruction cache.

755_avionics11 .. 755_avionics20 Larger tasks with over 32 KB code size. As here overful cache sets occur more often, the precision of the different persistence analyses can be evaluated.

10.6 Test Setup

The following different analysis settings are used for the tests:

S1: ILP Based Normal aiT analysis, no persistence, normal ILP based path analysis on basic block level, using only the basic block timings.

S2: Prediction Graph Normal aiT analysis, no persistence, ILP based path analysis on the prediction graph, using the more granular prediction graph.

S3: Persistence aiT analysis with persistence, ILP based path analysis on the prediction graph and automatic additional persistence constraints. The persistence scopes for the most examples are automatically generated by an analysis that determines if a loop or recursion may contain control flow which is interesting for persistence analysis (e.g., if-then-else or switch like structures). This analysis is integrated into the aiT framework and is performed before the cache and pipeline analysis which then uses the generated persistence scopes. Therefore, there is normally no need for manual annotations. Additional manual scope selection can increase the precision further. For the real-world tasks, manually persistence scopes were selected to include the outermost loops.

There are no comparisons with the original persistence analysis by Ferdinand, as its results are not conservative (see Section 5.3) and no implementation for a comparable aiT version is available.

The tests are performed on a Linux workstation featuring an *Intel Core i5* processor and 8 GB of RAM.

10.7 Test Results

We present the test results for the different architectures in two variants:

- As tables containing the computed WCET estimate and analysis runtime for the different analysis settings and improvement of the different settings over setting *S1*.
- As figures highlighting the precision differences for the computed WCET estimate between the different analysis settings.

The results for the *ARM7* architecture can be found in Tables 10.1 & 10.2 and Figure 10.3. As for this *fully timing-compositional* architecture there is no difference in the analysis precision and runtime between *S1* and *S2*, we only provide results for *S1*.

The *MPC5554* results are presented in Tables 10.3 & 10.4 and Figure 10.4.

The *MPC755* results are shown in Tables 10.5 & 10.6 and Figure 10.5.

10.8 Discussion of the Results

10.8.1 ARM7 TDMI - Synthetic and WCET Benchmarks

The synthetic benchmarks for the *ARM7* show that the persistence analyses behave as described in the previous sections. The if-then-else loop construct (Figure 4.1) is classified as persistent, independent of the chosen loop unrolling. The switch loop (Figure 5.3) causes evictions for the accessed data elements but still benefits from instruction cache persistence. The conflict couting with aging analysis here has a benefit if we unroll the loop, as it doesn't evict elements in the first iterations. The loop which has different behavior inside the first iteration (Figure 6.3) benefits from the age tracking, too, as described in Section 9.5.

Similar results are obtained for the *WCET Benchmarks*. If the instruction or data sets fit into the cache, both persistence analyses perform equally well. If there can be evictions, the conflict couting with aging analysis performs better, as evictions are delayed for unrolled loops and the analysis tracks the conflicts per element. The improvements vary for the benchmarks because for many memory references the may and must analyses already provide exact classifications.

In average, the *age-tracking conflict counting persistence analysis* allows a precision gain of around 38% over the original ILP method for the chosen synthetic and WCET benchmarks.

The analysis run times for the *ARM7* tests are all very low, as the tests are rather small, and the analysis for this architecture is not complex. For most tests the difference in runtime between the different analyses is not measurable.

10.8.2 Freescale MPC5554 & MPC755 - Avionics Benchmarks

For the *MPC5554* there is already a gap of several percent between basic-block and prediction-graph based path analysis. This is the result of the more complex interleaving of memory accesses caused by performance features like branch prediction and store buffers.

For the very complex *MPC755* there is a relatively big gap between both path analyses of up to 20% of the computed WCET bound. This is a result of the *non-compositionality* of this architecture which leads to effects spanning multiple basic blocks. The *MPC755* features speculative execution and many parallel execution units and internal queues that lead to this high amount of interleaving.

Whereas the normal ILP based path analysis on basic block level only takes into account the upper bounds of the execution time per block, the analysis based on the prediction graph has the full pipeline state graph to exclude paths not possible between different basic blocks.

The persistence analysis for the selected examples further reduces the calculated WCET bound.

For the avionics test cases, the code includes similar constructs but not as prominently on the WCET path as in the synthetic examples, therefore the gain in precision is less visible. Still, the analysis produces a clear improvement on top of the gain by the novel prediction based path analysis. The typical code patterns that appear in the used fly-by-wire and avionics control software and benefit by the persistence analyses are instances of the fault handling and data dependent algorithms as presented in Section 4.3. In addition some analyzed tasks contain state machine code generated by SCADE or hand written state machines, with a similar structure as described in [SLH+05]. The variance in the percentage of improvements results in the different distribution of such constructs in the program and respectively their weight on the worst-case path.

For the *MPC5554* benchmarks, beside some tight loops, there is mostly linear code with little locality [SLH+05]. Therefore, only inside the tight loops, there is reuse and a gain in precision by persistence analysis for the code structures shown above. As the cache is relatively large, even locked down to one way, these small loops and the functions called inside will be persistent for both persistence analyses in most cases. The imprecision of the simple conflict counting variant does only show up in some benchmarks like *5554_avionics3* and *5554_avionics8*. There a gain of 1-2% in precision can be observed for the *age-tracking conflict counting persistence analysis*. In average, the *age-tracking conflict counting persistence analysis* combined with the prediction graph base path analysis allow a precision gain of 7% over the original ILP method for the chosen benchmarks.

The *MPC755* benchmarks have similar characteristics. As a cache miss is more expensive for this hardware architecture, as main memory is quiet slow, the improvements by the persistence analysis are higher, thought. And as we have here benchmarks available which are much larger than the instruction cache (*755_avionics11* - *744_avionics20*) the imprecision of the simple conflict counting analysis can be shown better. For the large benchmarks, the more complex persistence analysis in in average 4% more precise than the simple conflict counting. Overall, the precision gain by prediction graph bases analysis combined with persistence analysis is in average 16% for all *MPC755* benchmarks.

For these complex architectures the resulting ILP of the prediction graph based path analysis is harder to solve, and the analysis runtime increases in some cases up to a factor of 7 for large avionics tasks on the *MPC5554*. For the more complex *MPC755* the differences in analysis runtime vary between no difference and a factor of 6.

10.9 Summary

In this chapter we presented the integration of the novel persistence analyses combined with a novel prediction graph based path analysis in the WCET analyzer *aiT*. This integration was used to evaluate the novel persistence analyses on several different benchmarks for hardware architectures falling into the three different classification of architectures by Wilhelm et al. [WGR+09]. The evaluation shows, that the persistence analyses allow for more precise WCET estimation both for synthetic and real-world derived applications. The results obtained by the synthetic benchmarking framework are confirmed.

	S1: ILP Based			S1: ILP Based	
	WCET	Time		WCET	Time
ifelse	140625	0:01	adpcm	27767709	0:01
ifelse_u	27468	0:01	bsort100	11102831	0:01
switch	257102	0:01	bs	5417	0:01
switch_u	176203	0:01	crc	1448259	0:01
prefix	350832	0:01	expint	812057	0:01
prefix_u	28633	0:01	fir	144132	0:01
			lcdnum	5898	0:01
			matmult	4143296	0:01
			ndes	1337681	0:08
			ns	189640	0:01
			qsort-exam	178369	0:01

Table 10.1: WCET in cycles and analysis runtime in minutes for the *ARM7* with setting S1. The left column contains our synthetic examples, the right column the tests from the WCET benchmarks suite.

	S2: Conflicts			S3: Conflicts & Aging		
	WCET	Time	Impr. (S1)	WCET	Time	Impr. (S1)
ifelse	11459	0:01	91.9%	11459	0:01	91.9%
ifelse_u	11459	0:01	58.3%	11459	0:01	58.3%
switch	47395	0:01	81.6%	47395	0:01	81.6%
switch_u	47395	0:03	73.1%	47237	0:03	73.2%
prefix	173413	0:01	50.6%	141028	0:01	59.8%
prefix_u	28633	0:01	0.0%	12504	0:01	56.3%
adpcm	19278648	0:03	30.6%	19270647	0:03	30.6%
bsort100	7406369	0:02	33.3%	7406369	0:02	33.3%
bs	5029	0:01	7.2%	5029	0:01	7.2%
crc	1178511	0:01	18.6%	1168083	0:01	19.3%
expint	443630	0:01	45.4%	443630	0:01	45.4%
fir	105651	0:01	26.7%	105651	0:01	26.7%
lcdnum	5263	0:01	10.8%	4755	0:01	19.4%
matmult	3945557	0:01	4.8%	3945557	0:01	4.8%
ndes	1171045	0:09	12.5%	1161520	0:09	13.2%
ns	163689	0:01	13.7%	163689	0:01	13.7%
qsort-exam	144918	0:01	18.8%	143394	0:01	19.6%

Table 10.2: WCET in cycles and analysis runtime in minutes for the *ARM7* with setting S3 and the set-wise or age-tracking conflict counting persistence analysis.

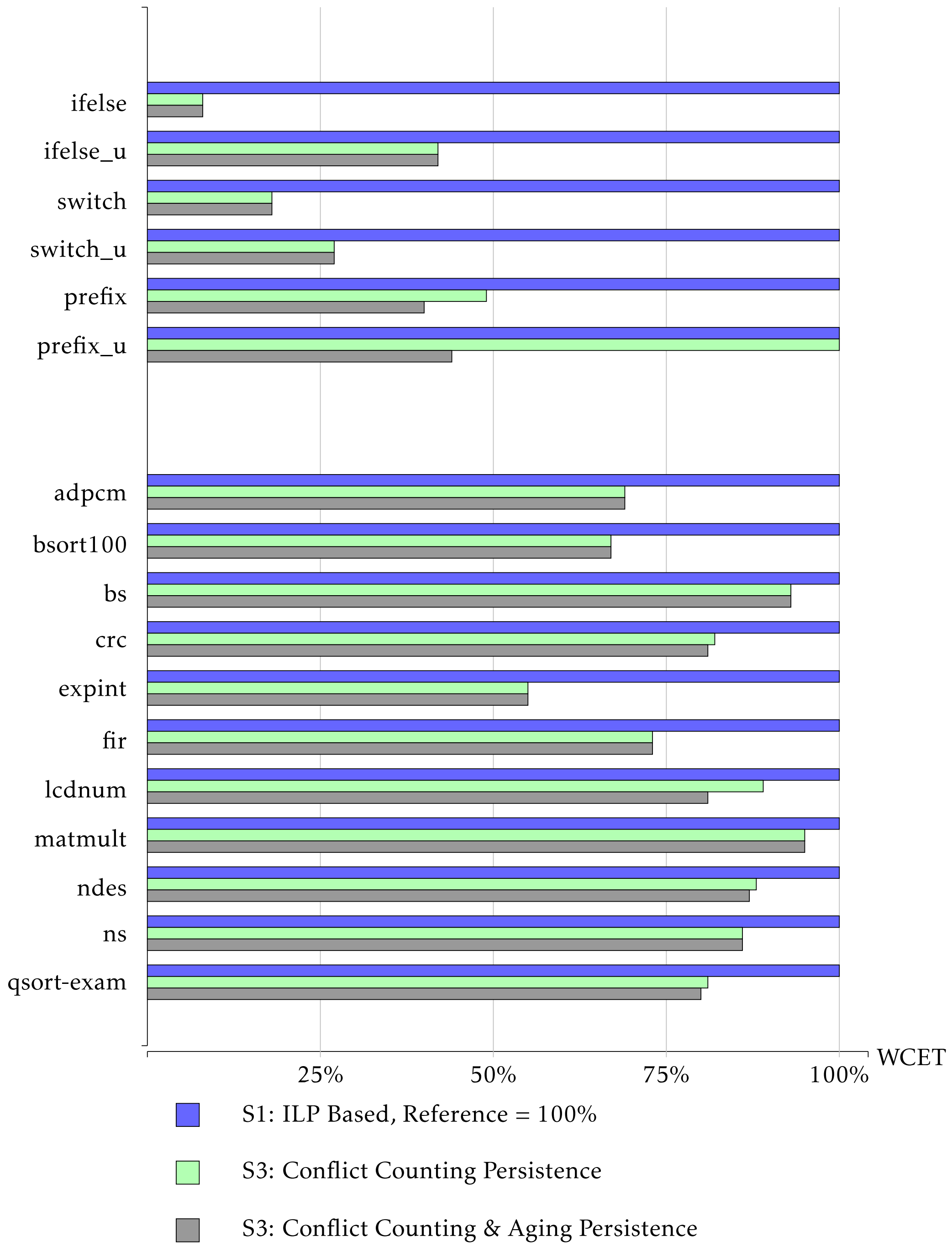


Figure 10.3: Comparison of WCET estimates for different *ARM7* tests and analysis settings, S1 is the reference with 100%.

	S1: ILP Based		S2: Prediction Graph		
	WCET	Time	WCET	Time	Impr. (S1)
5554_avionics1	239898	0:40	231598	3:20	3.46%
5554_avionics2	235171	0:47	226943	3:16	3.50%
5554_avionics3	761312	4:52	737843	8:39	3.08%
5554_avionics4	430123	2:17	414856	6:06	3.55%
5554_avionics5	238661	1:09	230578	3:54	3.39%
5554_avionics6	179727	0:37	172999	3:06	3.74%
5554_avionics7	257479	1:07	249293	3:34	3.18%
5554_avionics8	688575	4:46	667620	8:05	3.04%
5554_avionics9	170545	0:36	164916	0:41	3.30%
5554_avionics10	163180	0:33	157236	0:37	3.64%
5554_avionics11	186955	0:40	181093	0:46	3.14%
5554_avionics12	224494	0:41	217553	0:47	3.09%
5554_avionics13	95882	0:32	92732	0:35	3.29%
5554_avionics14	147674	0:35	142844	0:39	3.27%
5554_avionics15	260862	0:48	257545	0:51	1.27%
5554_avionics16	101756	0:27	99765	0:29	1.96%

Table 10.3: WCET in cycles and analysis runtime in minutes for the *MPC5554* with settings S1 and S2.

	S2: Conflicts			S3: Conflicts & Aging		
	WCET	Time	Impr. (S1)	WCET	Time	Impr. (S1)
5554_avionics1	218595	2:45	8.88%	218595	2:46	8.88%
5554_avionics2	217206	3:02	7.64%	217206	3:02	7.64%
5554_avionics3	684059	15:09	10.15%	670031	28:07	11.99%
5554_avionics4	387235	6:05	9.97%	387235	6:06	9.97%
5554_avionics5	221441	3:03	7.22%	221441	3:03	7.22%
5554_avionics6	164873	2:29	8.26%	164873	2:29	8.26%
5554_avionics7	234884	3:06	8.78%	234884	3:06	8.78%
5554_avionics8	617706	14:59	10.29%	603678	25:40	12.33%
5554_avionics9	158301	1:10	7.18%	158301	1:11	7.18%
5554_avionics10	152592	0:49	6.49%	152592	0:50	6.49%
5554_avionics11	169642	1:16	9.26%	169642	1:20	9.26%
5554_avionics12	212629	1:01	5.29%	212629	1:01	5.29%
5554_avionics13	91619	0:41	4.45%	91619	0:41	4.45%
5554_avionics14	138659	0:52	6.10%	138659	0:52	6.10%
5554_avionics15	247326	0:59	5.19%	247326	1:02	5.19%
5554_avionics16	99000	0:32	2.71%	98973	0:33	2.73%

Table 10.4: WCET in cycles and analysis runtime in minutes for the *MPC5554* with setting S3 and the set-wise or age-tracking conflict counting persistence analysis.

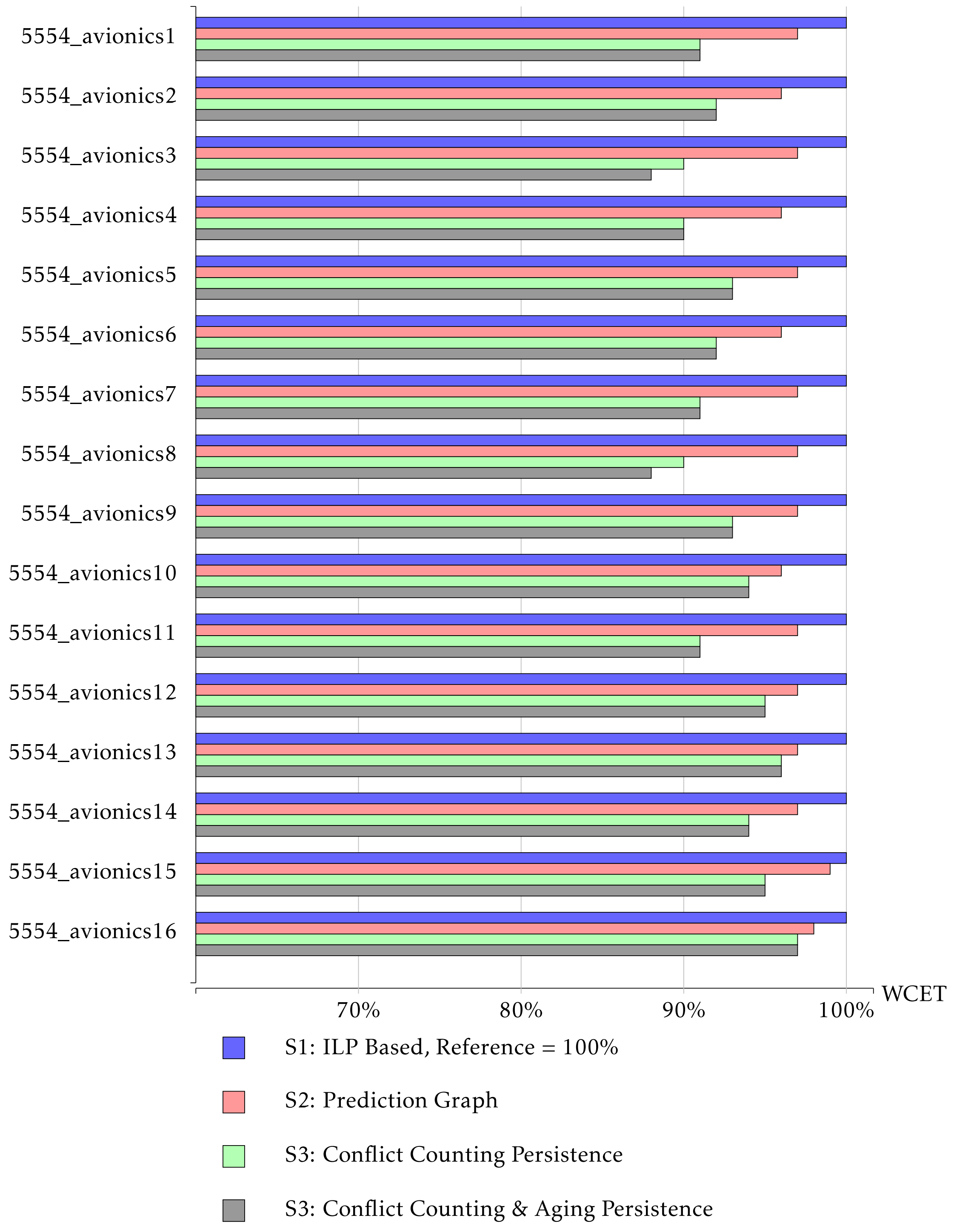


Figure 10.4: Comparison of WCET estimates for different *MPC5554* tests and analysis settings, S1 is the reference with 100%.

	S1: ILP Based		S2: Prediction Graph		
	WCET	Time	WCET	Time	Impr. (S1)
755_avionics1	30184	0:21	28212	0:25	6.53%
755_avionics2	82139	0:29	78754	0:36	4.12%
755_avionics3	74724	0:31	72202	0:35	3.38%
755_avionics4	69129	0:24	66788	0:27	3.39%
755_avionics5	80940	0:26	77784	0:31	3.90%
755_avionics6	103588	0:22	100604	0:24	2.88%
755_avionics7	30386	0:21	28346	0:25	6.71%
755_avionics8	31117	0:23	29006	0:27	6.78%
755_avionics9	69078	0:25	66651	0:27	3.51%
755_avionics10	84967	0:31	81894	0:36	3.62%
755_avionics11	895184	1:21	808165	1:32	9.72%
755_avionics12	1042489	1:30	946987	1:50	9.16%
755_avionics13	1188760	1:45	1079622	2:02	9.18%
755_avionics14	595568	1:03	538369	1:05	9.60%
755_avionics15	1074711	1:25	954226	1:34	11.21%
755_avionics16	923855	1:46	830002	1:55	10.16%
755_avionics17	842246	1:46	751052	1:52	10.83%
755_avionics18	991719	1:57	893113	2:01	9.94%
755_avionics19	1093188	1:26	979889	1:41	10.36%
755_avionics20	1108734	1:27	994473	1:42	10.31%

Table 10.5: WCET in cycles and analysis runtime in minutes for the *MPC755* with settings S1 and S2.

	S2: Conflicts			S3: Conflicts & Aging		
	WCET	Time	Impr. (S1)	WCET	Time	Impr. (S1)
755_avionics1	26216	0:28	13.15%	26216	0:26	13.15%
755_avionics2	74758	0:37	8.99%	74758	0:37	8.99%
755_avionics3	70206	0:37	6.05%	70206	0:37	6.05%
755_avionics4	64792	0:29	6.27%	64792	0:29	6.27%
755_avionics5	73788	0:40	8.84%	73788	0:40	8.84%
755_avionics6	98608	0:26	4.81%	98608	0:26	4.81%
755_avionics7	26350	0:28	13.28%	26350	0:26	13.28%
755_avionics8	27010	0:29	13.20%	27010	0:29	13.20%
755_avionics9	64655	0:29	6.40%	64655	0:28	6.40%
755_avionics10	79898	0:36	5.97%	79898	0:37	5.97%
755_avionics11	716200	2:30	19.99%	690200	3:41	22.90%
755_avionics12	845059	3:01	18.94%	803926	5:00	22.88%
755_avionics13	976392	3:42	17.86%	931647	6:21	21.63%
755_avionics14	456859	1:48	23.29%	443370	2:37	25.56%
755_avionics15	855662	3:30	20.38%	771011	7:00	28.26%
755_avionics16	725449	3:13	21.48%	692313	5:52	25.06%
755_avionics17	636029	3:13	24.48%	608555	4:37	27.75%
755_avionics18	778054	3:47	21.54%	724181	4:54	26.98%
755_avionics19	940102	3:40	14.00%	874127	7:45	20.04%
755_avionics20	954686	3:26	13.89%	887758	7:53	19.93%

Table 10.6: WCET in cycles and analysis runtime in minutes for the *MPC755* with setting S3 and the set-wise or age-tracking conflict counting persistence analysis.

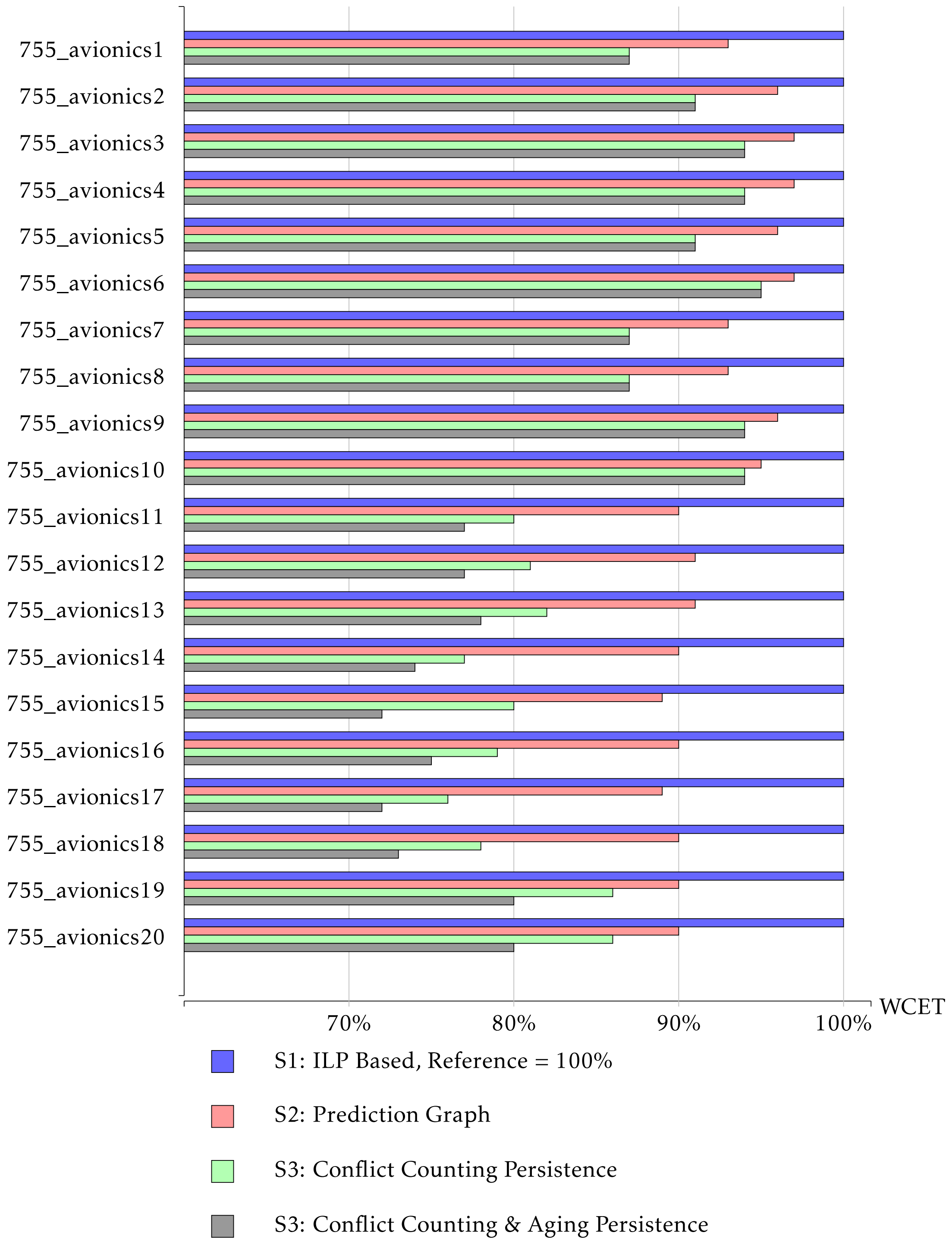


Figure 10.5: Comparison of WCET estimates for different *MPC755* tests and analysis settings, S1 is the reference with 100%.

CHAPTER 11

Related Work

There exist two areas of related work:

- the research on different persistence analysis techniques
- the application of existing persistence analyses to different problems and improvements to their precision

In this chapter, we will show, how these relate to the analyses proposed in this thesis. First, we will take a look at other existing persistence analyses, then we will cover the different improvements and applications of them.

11.1 Other Cache Persistence Analyses

There exist two major other different persistence analyses, beside the one by Ferdinand (see Chapter 5) and the ones proposed in this thesis: first-miss analysis by Mueller and a novel persistence analysis by Huynh et al.. In addition, Kartik Nagar proposes a similar may-based cache persistence analysis as introduced in [Cul11].

11.1.1 First-Miss Analysis by Mueller

Frank Mueller introduced his *first-miss* analysis in [MWH94] and applied it to direct-mapped instruction caches in [HAM+99]. Later this approach was extended to handle set-associative instruction caches with LRU replacement strategy in [Mue00]. The method is not based on abstract interpretation but integrated into a proprietary static cache simulation framework [Mue95]. Unlike the analyses introduced in this thesis, it was never applied to complex architectures featuring timing anomalies. In principle, it is the same technique as the conflict counting persistence per set analysis as described in Chapter 6. Given our benchmarking results, this technique has the least precision compared to our other analyses.

11.1.2 Persistence Analysis by Huynh et al.

Huynh et al. present in [HJR11] a similar counter example for the persistence of Ferdinand as shown in Section 5.3. They introduce a novel persistence analysis that solves this issue. Their analysis keeps for any accessed element inside the persistence scope track of which other elements might be accessed before the next use of this element. This approach is equivalent to the element-wise conflict counting persistence analysis introduced in Chapter 7 which keeps track of possible conflicting elements for each element. They perform their evaluation within an analysis framework incompatible to the one chosen by us and for a different hardware architecture (an in-order simulated processor) the direct evaluation results of their paper are not comparable to the results in this thesis. Still, as the conflict counting per set analysis is equivalent, our benchmarking results will hold for this analysis, too. It is more precise than the analysis by Mueller, still our age-tracking conflict counting analysis from Chapter 9 allows more precise results.

11.1.3 Persistence Analysis by Kartik Nagar

Inspired by Huynh et al. in [HJR11] and our publications in [Cul11] Kartik Nagar designed a persistence analysis based on may cache analysis, too. The analysis as introduced in [Nag12] is equivalent to the may-based cache persistence analysis as described in Chapter 8 but gains more precision on corner cases by using must cache analysis information in addition. In Section 12.1 we will show how this analysis cooperation can be done and which precision gains can be achieved for analyses that keep track of the ages of cache entries, like the may-based or age-tracking conflict counting persistence analysis.

11.2 Application of Cache Persistence Analyses

Besides research on the actual design of persistence analyses, there are other groups working on different application fields for existing persistence analyses and how to optimize the analysis results by improving the scope computation, etc..

11.2.1 Persistence Scope Optimizations

Ballabriga and Cassé [BC08] propose the use of nested persistence scopes to improve the analysis precision. Whereas they base their work on the buggy persistence analysis by Ferdinand, their scope computation and usage is independent of the concrete analysis. Therefore it will be applicable to the analyses proposed in this thesis.

Huynh et al. present in [HJR11] in addition to their novel persistence analysis an intelligent selection of data cache persistence scopes, e.g., to differentiate multiple scopes per loop. This technique can be used for our analyses, too, as it doesn't depend on the actual analysis design, as long as it is an abstract interpretation based one.

11.2.2 Application to Muli-Level Caches

In [LHP09] Lesage et al. show how to apply the persistence analysis to architectures with multi-level caches. They reuse an existing analysis for their study, likely based on Ferdinand's abstract interpretation, as it fits their analysis framework.

11.2.3 Application to Multi-Core Architectures

Mingsong Lv et al. used in [LYGY10] a preliminary version of our may-based analysis as introduced in Chapter 8 for their research on timing analysis for multi-core architectures.

CHAPTER 12

Extensions & Future Work

We introduced in this thesis several novel cache persistence analyses and evaluated them on synthetic benchmarks and real-world applications. Whereas the proposed analyses perform well on both, there is still room for extensions of the analyses. Additionally there are similar problem areas where adapted variants of the persistence analysis might be applicable. We will now show such possible extensions and future research areas.

12.1 Using Must & May Analyses Information

In this thesis we concentrated on the design of cache persistence analyses and their precision. Still, during normal timing analysis runs, global must and may cache analysis information is often available. This global knowledge can be used to enhance the analysis results of the persistence analyses inside their scopes.

12.1.1 Useful Must Cache Analysis Information

The may-based and age-tracking conflict counting persistence analyses keep track of upper age bounds of possible cache set entries. Given the example from Figure 12.1, let us assume that the referenced memory blocks a, b and c map to the same 2-way associative cache set with LRU replacement.

As the number of accessed memory blocks in the loop is larger than the associativity, evictions for some blocks will occur during the loop execution. Still the memory block a won't be evicted after the initial loading as only one other block will be accessed until the next access to a happens.

```
void reusedMemoryBlocks () {
  for (int i = 0; i < NUMBER_OF_EVENTS; ++i) {
    accessA ();
    if (somethingUnknownForEachCall()) {
        accessB ();
        accessB ();
    } else {
        accessC ();
        accessC ();
    }
  }
}
```

Figure 12.1: Loop which accesses three memory blocks a, b and c mapping to the same cache set. b and c are always accessed twice inside their if-then-else branch.

This won't be detected by the may-based or age-tracking persistence analysis, though, as these analyses don't know that the second access to b or c in the individual if-then-else branch won't age any element that is older than one.

Such information is not computed by these analyses, as they only keep track of upper ages of memory blocks possibly in the cache, but they have no information if a block is definitively in the cache.

This knowledge is available in the global must cache analysis. The must cache analysis will compute that the second memory reference to b and c in the if-then-else branch will be a *sure-hit* and that b and c will have age one.

By using this information, the persistence analyses can avoid to age memory blocks that are older than b or c for the respective second reference and will be able to classify the memory reference to a as persistent for all loop iterations.

Similar examples for the precision gain through this analysis cooperation are given by Kartik Nagar in [Nag12].

12.1.2 Useful May Cache Analysis Information

If we analyze caches with *write-through* policy, only write accesses that hit the cache will trigger a cache update on the real hardware. If we use the persistence analyses as introduced in this thesis, on a write access, always both possibilities must be handled to be sound:

access hits the cache persistence analysis update for the memory reference must be done

access misses the cache no persistence analysis update is allowed

Afterwards the results of both possibilities must be joined.

Let us look at the example from Figure 12.2 and assume a direct mapped cache with *write-through* policy.

```
void writeToCache () {
  for (int i = 0; i < NUMBER_OF_EVENTS; ++i) {
    if (somethingUnknownForEachCall())
        accessA ();
    writeOnlyB ();
  }
}
```

Figure 12.2: Loop which accesses two memory blocks a and b mapping to the same cache set. b is only written, never read.

Inside the loop only the memory block a is possibly loaded, b is only written. Without any additional information, the persistence analyses will need to handle the references to b. This will lead to false possible evictions of a.

If global may analysis information is available and the reference to b can be classified as a *sure-miss*, the persistence analysis can ignore the reference to b completely and the memory reference to a will be classified as persistent.

12.2 Better Handling of Replacement Policies

The cache persistence analyses introduced in this thesis and the ones by Mueller, Huynh or Nagar all assume either the *LRU* replacement policy or a direct mapped cache. We will introduce two methods to cope with other replacement policies: handle them like smaller *LRU* caches if possible or argue about cumulative limits for misses.

12.2.1 Exploiting Relations between Non-LRU and LRU Policies

Using the concept of *relative miss-competitiveness* as introduced by Jan Reineke in [Rei08, RG08], we can apply the conflict counting analyses to the *PLRU* replacement policy. But as a k-way associative *PLRU* cache is *(1, 0)-miss-competitive* relative to a $(1 + log_2(k))$-way *LRU* cache, the persistence analysis must assume a cache associativity of only $(1 + log_2(k))$. This allows only a part of the cache sizes occurring in real systems to be exploited by the analyses.

For other replacement policies like *FIFO* [GR10a, GR10b], *MRU* [GLYY12] or *Pseudo-RoundRobin* the situation is even worse. The persistence analysis must assume that only one way of the cache is used if we want to handle them with the *LRU* semantics based analyses.

12.2.2 Cumulative Arguments for Non-LRU Policies

For policies for which no good relations to *LRU* exist, like *FIFO* or *MRU*, the reduction to smaller direct mapped caches enables the use of the *LRU* based analyses, yet it leads to a large loss in precision as only one way is considered.

For these policies it makes sense to argue not about the *persistence classification* of individual memory references but compute cumulative bounds for possible misses over the whole *persistence scope*. For most of these policies this is possible if only a limited set of different memory blocks is referenced that is at most as large as the associativity. The set-wise conflict counting persistence analysis introduced in Chapter 6 can be used to compute for which references inside a scope this constraint is met.

These cumulative limits can be used in the ILP to limit the number of misses for the individual memory blocks potentially accessed in the scope.

FIFO Replacement

For a k-way cache with *FIFO* replacement it holds, that if inside a scope the program accesses at most k different memory blocks mapping to the same set, at most one miss can happen per memory block [Gru12]. Unlike the *first-miss* characteristic of the persistence as introduced in Chapter 4 this cumulative argument doesn't require the first access to be the possible miss. Instead, it is possible that the first access might be a hit and a later access the single possible miss.

MRU Replacement

For a k-way cache with *MRU* replacement it holds, that if inside a scope the program accesses at most k different memory blocks mapping to the same set, at most k misses can happen per block [GLYY12]. Like for the *FIFO* replacement no guarantees can be made, that the first accesses must be the misses.

12.3 Enhanced Persistence Scope Selection

We used for our evaluation in Chapter 10 an automatic persistence scope selection that is implemented as a separate analysis and is performed before the cache and pipeline analysis.

This scope selection may be guided by the user who can specify interesting routines in the control-flow graph. Independent of this, the scope selection analysis will try to detect which loops might benefit from persistence analysis because they e.g., contain conditional control-flow.

As modern processor architectures employ techniques like prefetching, branch-prediction or speculative execution, actually only the pipeline analysis knows, which memory blocks might be really accessed. For example an if-then-else construct might look interesting for the persistence analysis in the scope heuristics as the condition is unknown, but the pipeline might prefetch both paths anyway.

Therefore one might enhance the persistence scope selection by first running a cache and pipeline analysis without persistence analysis to compute which memory addresses are really accessed. This information can then be used to compute more precise cache persistence scopes that can be used in a second cache and pipeline analysis run.

12.4 Write-Back Cache Analysis

The analysis of caches that employ the write-back strategy instead of write-through is still an open issue. There exists no precise analysis to compute if a load into the cache will cause an eviction and write-back of a dirty cache line.

The conservative assumption that any load of a new cache line into the cache might cause such a write-back activity doesn't allow for precise *WCET* estimates.

The cache persistence analysis can help to improve the precision by proving the absence of evictions for elements in the cache inside the persistence scope.

Besides, the may-based persistence as introduced in Chapter 8 uses a modified *may cache* analysis to track the maximal ages of all possible cache entries. This information might be reused to find the earliest point in time at which an element can be evicted. Combined with the traditional *may cache* analysis, that detects the latest point in time a element can leave the cache, one might be able to compute tight time windows for possible evictions of individual dirty cache lines.

CHAPTER 13

Conclusion

Information on the *persistence* of memory blocks in cache memories can considerably help to more precisely bound the worst-case execution times of real-time tasks.

Over the last years the topic of *cache persistence analysis* has received considerable attention in academia.

Unlike other approaches we did not only focus on the construction of more (or less) precise persistence analyses. We also provided a concise formalization of cache persistence enabling better problem understanding and concise correctness proofs.

This thesis compares the two existing state-of-the-art analyses and two completely novel analyses using a framework of synthetic benchmarking. Our novel *age-tracking conflict counting persistence analysis* combines the ideas of all other analyses and is the most precise.

In addition we showed a correctness issue with the original persistence analysis by Ferdinand.

Beside our work, there are no results of practical evaluations of cache persistence analyses on real architectures available, as for *non-fully-timing-compositional* processors persistence information cannot be used as easily as the results of the classical must or may cache analysis. The cache hits and misses have to be taken into account precisely.

We have integrated all four persistence analyses into the industrial-strength WCET analyzer aiT based on a refined path analysis on so-called prediction graphs. The analyses were evaluated on well known WCET benchmarks and real programs of the avionics domain. The evaluation showed considerable precision improvements of the computed WCET varying between 7% on average for the moderately complex *MPC5554* architecture, and 16% for the quite complex *MPC755* architecture.

APPENDIX A

Source Code

In this chapter we provide source code snippets as used for the analyses implementations and evaluations. All shown sources are *C++* and compilable with a normal compiler like the *GNU G++* compiler.

A.1 Synthetic Benchmarking

```
/**
 * standard includes + data structures
 */
#include <cstdio>
#include <cstdlib>
#include <map>
#include <set>
#include <vector>

/**
 * deterministic random generator
 * http://en.wikipedia.org/wiki/Xorshift
 */
static unsigned int x;
static unsigned int y;
static unsigned int z;
static unsigned int w;

void initRandomSeed ()
{
  x = 123456789;
  y = 362436069;
  z = 521288629;
  w = 88675123;
}

unsigned int randomNumber ()
{
  unsigned int t = x ^ (x << 11);
  x = y; y = z; z = w;
  return w = w ^ (w >> 19) ^ (t ^ (t >> 8));
}

/**
 * Address == unsigned integers for us
 * we use place holders for the different blocks, e.g. 0, 1, 2, ...
 */
typedef unsigned int Address;

/**
 * Persistence Analysis Set, Generic Implemetation => does nothing, no access persistent
 */
class PersistenceSet {
    public:
        /**
         * Construct new empty persistence set
         */
        PersistenceSet ()
        {
        }

        /**
         * Destruct it
         */
        virtual ~PersistenceSet ()
        {
        }

        /**
         * Create a new empty set for given associativity
         */
        virtual PersistenceSet *create (unsigned int associativity) const = 0;

        /**
         * Clone a set, needed for implementation
         */
        virtual PersistenceSet *clone () const = 0;

        /**
         * update function, update the set for given access, non-destructive
         */
        virtual PersistenceSet *update (Address memoryBlock) const = 0;

        /**
         * join function, join this set with an other set, non-destructive
         */
        virtual PersistenceSet *join (const PersistenceSet *set) const = 0;

        /**
         * compare function, needed for fixed point iteration
         */
        virtual bool operator== (const PersistenceSet &set) const = 0;

        /**
         * classification of a access: is the access persistent?
         */
        virtual bool isPersistent (Address memoryBlock) const = 0;

        /**
         * print function
         */
        virtual void print () const = 0;
};

/**
 * Persistence Analysis Set, Conflict Counting Per Set
 */
class PersistenceSetConflictSet : public PersistenceSet {
    public:
        /**
         * Construct new empty persistence set
         */
        PersistenceSetConflictSet (unsigned int associativity = 0)
         : m_associativity (associativity)
         , m_isOverFull (false)
        {
        }

        /**
         * Destruct it
         */
        virtual ~PersistenceSetConflictSet ()
        {
        }

        /**
         * Create a new empty set for given associativity
         */
        virtual PersistenceSet *create (unsigned int associativity) const
        {
            return new PersistenceSetConflictSet (associativity);
        }

        /**
 * Clone a set, needed for implementation
 */
virtual PersistenceSet *clone () const
{
    /**
     * create exact copy
     */
    PersistenceSetConflictSet *newSet = new PersistenceSetConflictSet ();
    *newSet = *this;
    return newSet;
}

/**
 * update function, update the set for given access, non-destructive
 */
virtual PersistenceSet *update (Address memoryBlock) const
{
    /**
     * insert element into conflicts set
     */
    PersistenceSetConflictSet *newSet = static_cast<PersistenceSetConflictSet *> (clone ());
    newSet->m_conflicts.insert (memoryBlock);

    if (m_isOverFull || newSet->m_conflicts.size() > m_associativity) {
        newSet->m_isOverFull = true;
        newSet->m_conflicts.clear ();
    }

    return newSet;
}

/**
 * join function, join this set with an other set, non-destructive
 */
virtual PersistenceSet *join (const PersistenceSet *set) const
{
    /**
     * union of elements in conflicts sets
     */
    PersistenceSetConflictSet *newSet = static_cast<PersistenceSetConflictSet *> (clone ());
    const PersistenceSetConflictSet *_set = static_cast<const PersistenceSetConflictSet *>(set);
    for (std::set<Address>::const_iterator it = _set->m_conflicts.begin();
         it != _set->m_conflicts.end(); ++it)
        newSet->m_conflicts.insert (*it);

    if (m_isOverFull || _set->m_isOverFull || newSet->m_conflicts.size() > m_associativity) {
        newSet->m_isOverFull = true;
        newSet->m_conflicts.clear ();
    }

    return newSet;
}

/**
 * compare function, needed for fixed point iteration
 */
virtual bool operator== (const PersistenceSet &set) const
{
    return m_isOverFull == static_cast<const PersistenceSetConflictSet *>(&set)->m_isOverFull
        && m_conflicts == static_cast<const PersistenceSetConflictSet *>(&set)->m_conflicts;
}

/**
 * classification of a access: is the access persistent?
 */
virtual bool isPersistent (Address memoryBlock) const
{
    return !m_isOverFull && (m_conflicts.size() <= m_associativity);
}

/**
 * print function
 */
virtual void print () const
{
    if (m_isOverFull) {
        printf ("{ TOP }");
        return;
    }

    printf ("{");
```

```
            for (std::set<Address>::const_iterator it = m_conflicts.begin(); it != m_conflicts.end(); ++it)
                printf (" %u", *it);
            printf ("}\n");
        }

    private:
        /**
         * cache associativity
         */
        unsigned int m_associativity;

        /**
         * optmization: full?
         */
        bool m_isOverFull;

        /**
         * data structure: set of conflicts
         */
        std::set<Address> m_conflicts;
};

/**
 * Persistence Analysis Set, Conflict Counting Per Element
 */
class PersistenceSetConflictElement : public PersistenceSet {
    public:
        /**
         * Construct new empty persistence set
         */
        PersistenceSetConflictElement (unsigned int associativity = 0)
         : m_associativity (associativity)
        {
        }

        /**
         * Destruct it
         */
        virtual ~PersistenceSetConflictElement ()
        {
        }

        /**
         * Create a new empty set for given associativity
         */
        virtual PersistenceSet *create (unsigned int associativity) const
        {
            return new PersistenceSetConflictElement (associativity);
        }

        /**
         * Clone a set, needed for implementation
         */
        virtual PersistenceSet *clone () const
        {
            /**
             * create exact copy
             */
            PersistenceSetConflictElement *newSet = new PersistenceSetConflictElement ();
            *newSet = *this;
            return newSet;
        }

        /**
         * update function, update the set for given access, non-destructive
         */
        virtual PersistenceSet *update (Address memoryBlock) const
        {
            /**
             * insert element into conflicts set
             */
            PersistenceSetConflictElement *newSet = static_cast<PersistenceSetConflictElement *> (clone ());
            for (std::map<Address, PersistenceSetConflictSet >::iterator it = newSet->m_conflicts.begin();
                 it != newSet->m_conflicts.end(); ++it) {
                /**
                 * update all sets not for this element
                 */
                if (it->first != memoryBlock) {
                    PersistenceSetConflictSet *newConflictSet
                      = static_cast<PersistenceSetConflictSet *>(it->second.update (memoryBlock));
                    it->second = *newConflictSet;
                delete newConflictSet;
            }
        }

        /**
         * init set with only this element for this element
         */
        PersistenceSetConflictSet newConf (m_associativity);
        PersistenceSetConflictSet *filledNewConf
          = static_cast<PersistenceSetConflictSet *>(newConf.update (memoryBlock));
        newSet->m_conflicts.erase (memoryBlock);
        newSet->m_conflicts.insert (std::make_pair (memoryBlock, *filledNewConf));
        delete filledNewConf;

        /**
         * be done, return new set
         */
        return newSet;
    }

    /**
     * join function, join this set with an other set, non-destructive
     */
    virtual PersistenceSet *join (const PersistenceSet *set) const
    {
        /**
         * union of elements in conflicts sets
         */
        PersistenceSetConflictElement *newSet = static_cast<PersistenceSetConflictElement *> (clone ());
        const PersistenceSetConflictElement * _set = static_cast<const PersistenceSetConflictElement *>(set);
        for (std::map<Address, PersistenceSetConflictSet >::const_iterator it = _set->m_conflicts.begin();
             it != _set->m_conflicts.end(); ++it) {
            /**
             * join individual sets
             */
            std::map<Address, PersistenceSetConflictSet >::iterator itNew
              = newSet->m_conflicts.find (it->first);
            if (itNew == newSet->m_conflicts.end())
                newSet->m_conflicts.insert (std::make_pair (it->first, it->second));
            else {
                PersistenceSetConflictSet *newConflictSet
                  = static_cast<PersistenceSetConflictSet *>(itNew->second.join (&it->second));
                itNew->second = *newConflictSet;
                delete newConflictSet;
            }
        }
        return newSet;
    }

    /**
     * compare function, needed for fixed point iteration
     */
    virtual bool operator== (const PersistenceSet &set) const
    {
        return m_conflicts == static_cast<const PersistenceSetConflictElement *>(&set)->m_conflicts;
    }

    /**
     * classification of a access: is the access persistent?
     */
    virtual bool isPersistent (Address memoryBlock) const
    {
        std::map<Address, PersistenceSetConflictSet >::const_iterator it = m_conflicts.find (memoryBlock);
        if (it == m_conflicts.end())
            return true;
        return it->second.isPersistent (memoryBlock);
    }

    /**
     * print function
     */
    virtual void print () const
    {
        for (std::map<Address, PersistenceSetConflictSet >::const_iterator it = m_conflicts.begin();
             it != m_conflicts.end(); ++it) {
            printf ("%u => ", it->first);
            it->second.print ();
        }
    }

private:
        /**
         * cache associativity
         */
        unsigned int m_associativity;

        /**
         * data structure: set of sets of conflicts
         */
        std::map<Address, PersistenceSetConflictSet> m_conflicts;
};

/**
 * Persistence Analysis Set, May Based
 */
class PersistenceSetMayBased : public PersistenceSet {
    public:
        /**
         * Construct new empty persistence set
         */
        PersistenceSetMayBased (unsigned int associativity = 0)
         : m_associativity (associativity)
        {
        }

        /**
         * Destruct it
         */
        virtual ~PersistenceSetMayBased ()
        {
        }

        /**
         * Create a new empty set for given associativity
         */
        virtual PersistenceSet *create (unsigned int associativity) const
        {
            return new PersistenceSetMayBased (associativity);
        }

        /**
         * Clone a set, needed for implementation
         */
        virtual PersistenceSet *clone () const
        {
            /**
             * create exact copy
             */
            PersistenceSetMayBased *newSet = new PersistenceSetMayBased ();
            *newSet = *this;
            return newSet;
        }

        /**
         * update function, update the set for given access, non-destructive
         */
        virtual PersistenceSet *update (Address memoryBlock) const
        {
            PersistenceSetMayBased *newSet = static_cast<PersistenceSetMayBased *> (clone ());

            /**
             * first: calculate may evict, regard the may set
             */
            unsigned int otherElementsInCache = 0;
            unsigned int ageOfMemoryBlock = m_associativity;
            for (std::map<Address, unsigned int>::const_iterator it = m_maySet.begin();
                 it != m_maySet.end(); ++it) {
                if (it->first == memoryBlock)
                    ageOfMemoryBlock = it->second;
                else
                    ++otherElementsInCache;
            }
            bool mayEvict = (otherElementsInCache >= m_associativity);

            /**
             * update may, age all elements, new element in first age
             */
            newSet->m_maySet.clear ();
            for (std::map<Address, unsigned int>::const_iterator it = m_maySet.begin();
                 it != m_maySet.end(); ++it) {
                if (it->second <= ageOfMemoryBlock) {
                    unsigned int age = it->second + 1;
                if (age < m_associativity)
                    newSet->m_maySet.insert (std::make_pair (it->first, age));
            } else
                newSet->m_maySet.insert (std::make_pair (it->first, it->second));
        }
        newSet->m_maySet[memoryBlock] = 0;

        /**
         * update may may, age all elements, if evictions possible => age into the bottom line
         */
        for (std::map<Address, unsigned int>::iterator it = newSet->m_mayMaxSet.begin();
             it != newSet->m_mayMaxSet.end(); ++it) {
            if (it->second == m_associativity)
                continue;

            if ((it->second + 1) == m_associativity) {
                if (mayEvict)
                    it->second = m_associativity;
            } else
                ++(it->second);
        }
        newSet->m_mayMaxSet[memoryBlock] = 0;

        return newSet;
    }

    /**
     * join function, join this set with an other set, non-destructive
     */
    virtual PersistenceSet *join (const PersistenceSet *set) const
    {
        PersistenceSetMayBased *newSet = static_cast<PersistenceSetMayBased *> (clone ());
        const PersistenceSetMayBased * _set = static_cast<const PersistenceSetMayBased *>(set);

        /**
         * may analysis join
         */
        for (std::map<Address, unsigned int>::const_iterator it = _set->m_maySet.begin();
             it != _set->m_maySet.end(); ++it) {
            /**
             * element needs to be inserted or minimal age computed?
             */
            std::map<Address, unsigned int>::iterator itNew = newSet->m_maySet.find (it->first);
            if (itNew == newSet->m_maySet.end())
                newSet->m_maySet.insert (std::make_pair (it->first, it->second));
            else {
                if (it->second < itNew->second)
                    itNew->second = it->second;
            }
        }

        /**
         * may max analysis join
         */
        for (std::map<Address, unsigned int>::const_iterator it = _set->m_mayMaxSet.begin();
             it != _set->m_mayMaxSet.end(); ++it) {
            /**
             * element needs to be inserted or minimal age computed?
             */
            std::map<Address, unsigned int>::iterator itNew = newSet->m_mayMaxSet.find (it->first);
            if (itNew == newSet->m_mayMaxSet.end())
                newSet->m_mayMaxSet.insert (std::make_pair (it->first, it->second));
            else {
                if (it->second > itNew->second)
                    itNew->second = it->second;
            }
        }

        return newSet;
    }

    /**
     * compare function, needed for fixed point iteration
     */
    virtual bool operator== (const PersistenceSet &set) const
    {
        return m_maySet == static_cast<const PersistenceSetMayBased *>(&set)->m_maySet
            && m_mayMaxSet == static_cast<const PersistenceSetMayBased *>(&set)->m_mayMaxSet;
    }

    /**
```

```
         * classification of a access: is the access persistent?
         */
        virtual bool isPersistent (Address memoryBlock) const
        {
            std::map<Address, unsigned int>::const_iterator it = m_mayMaxSet.find (memoryBlock);
            if (it == m_mayMaxSet.end())
                return true;
            return it->second < m_associativity;
        }

        /**
         * print function
         */
        virtual void print () const
        {
        }

    private:
        /**
         * cache associativity
         */
        unsigned int m_associativity;

        /**
         * data structures: may and may max sets
         */
        std::map<Address, unsigned int> m_maySet;
        std::map<Address, unsigned int> m_mayMaxSet;
};

/**
 * Persistence Analysis Set, Conflict Counting Per Element
 */
class PersistenceSetConflictWithAge : public PersistenceSet {
    public:
        /**
         * Construct new empty persistence set
         */
        PersistenceSetConflictWithAge (unsigned int associativity = 0)
         : m_associativity (associativity)
        {
        }

        /**
         * Destruct it
         */
        virtual ~PersistenceSetConflictWithAge ()
        {
        }

        /**
         * Create a new empty set for given associativity
         */
        virtual PersistenceSet *create (unsigned int associativity) const
        {
            return new PersistenceSetConflictWithAge (associativity);
        }

        /**
         * Clone a set, needed for implementation
         */
        virtual PersistenceSet *clone () const
        {
            /**
             * create exact copy
             */
            PersistenceSetConflictWithAge *newSet = new PersistenceSetConflictWithAge ();
            *newSet = *this;
            return newSet;
        }

        /**
         * update function, update the set for given access, non-destructive
         */
        virtual PersistenceSet *update (Address memoryBlock) const
        {
            /**
             * insert element into conflicts set
             */
            PersistenceSetConflictWithAge *newSet = static_cast<PersistenceSetConflictWithAge *> (clone ());
            for (std::map<Address, std::pair<unsigned int, PersistenceSetConflictSet> >::iterator
             it = newSet->m_conflicts.begin();
             it != newSet->m_conflicts.end(); ++it) {
            /**
             * update all sets not for this element
             */
            if (it->first != memoryBlock) {
                PersistenceSetConflictSet *newConflictSet
                  = static_cast<PersistenceSetConflictSet *>(it->second.second.update (memoryBlock));
                if (it->second.first < m_associativity)
                    ++(it->second.first);
                it->second.second = *newConflictSet;
                delete newConflictSet;
            }
        }

        /**
         * init set with only this element for this element
         */
        PersistenceSetConflictSet newConf (m_associativity);
        PersistenceSetConflictSet *filledNewConf
          = static_cast<PersistenceSetConflictSet *>(newConf.update (memoryBlock));
        newSet->m_conflicts.erase (memoryBlock);
        newSet->m_conflicts.insert (std::make_pair (memoryBlock, std::make_pair (0U, *filledNewConf)));
        delete filledNewConf;

        /**
         * be done, return new set
         */
        return newSet;
    }

    /**
     * join function, join this set with an other set, non-destructive
     */
    virtual PersistenceSet *join (const PersistenceSet *set) const
    {
        /**
         * union of elements in conflicts sets
         */
        PersistenceSetConflictWithAge *newSet = static_cast<PersistenceSetConflictWithAge *> (clone ());
        const PersistenceSetConflictWithAge *_set = static_cast<const PersistenceSetConflictWithAge *>(set);

        for (std::map<Address, std::pair<unsigned int, PersistenceSetConflictSet> >::const_iterator
             it = _set->m_conflicts.begin();
             it != _set->m_conflicts.end(); ++it) {
            /**
             * join individual sets
             */
            std::map<Address, std::pair<unsigned int, PersistenceSetConflictSet> >::iterator itNew
              = newSet->m_conflicts.find (it->first);
            if (itNew == newSet->m_conflicts.end())
                newSet->m_conflicts.insert (std::make_pair (it->first, it->second));
            else {
                PersistenceSetConflictSet *newConflictSet
                  = static_cast<PersistenceSetConflictSet *>(itNew->second.second.join (&it->second.second));
                if (it->second.first > itNew->second.first)
                    itNew->second.first = it->second.first;
                itNew->second.second = *newConflictSet;
                delete newConflictSet;
            }
        }
        return newSet;
    }

    /**
     * compare function, needed for fixed point iteration
     */
    virtual bool operator== (const PersistenceSet &set) const
    {
        return m_conflicts == static_cast<const PersistenceSetConflictWithAge *>(&set)->m_conflicts;
    }

    /**
     * classification of a access: is the access persistent?
     */
    virtual bool isPersistent (Address memoryBlock) const
    {
        std::map<Address, std::pair<unsigned int, PersistenceSetConflictSet> >::const_iterator it
          = m_conflicts.find (memoryBlock);
        if (it == m_conflicts.end())
            return true;
            return (it->second.first < m_associativity) || it->second.second.isPersistent (memoryBlock);
        }

        /**
         * print function
         */
        virtual void print () const
        {
            for (std::map<Address, std::pair<unsigned int, PersistenceSetConflictSet> >::const_iterator
                 it = m_conflicts.begin(); it != m_conflicts.end(); ++it) {
                printf ("%u => max age %u ", it->second.first, it->first);
                it->second.second.print ();
            }
        }

    private:
        /**
         * cache associativity
         */
        unsigned int m_associativity;

        /**
         * data structure: set of sets of conflicts
         */
        std::map<Address, std::pair<unsigned int, PersistenceSetConflictSet> > m_conflicts;
};

/**
 * Structure to describe one test setup
 */
class TestSetup {
    public:
        /**
         * cache associativity
         */
        unsigned int associativity;

        /**
         * loop unrolling
         */
        unsigned int unrolling;

        /**
         * number of different accessed memory blocks inside the loop
         * (both for the complete loop and one iteration itself)
         */
        unsigned int differentAccessedBlocks;

        /**
         * subset of above memory blocks accessed before the lubbed
         * context is reached, e.g. in the unrolled-1 iterations
         */
        unsigned int commonBlocksInPrefix;

        /**
         * number of additional different memory blocks accessed before the lubbed
         * context is reached, e.g. in the unrolled-1 iterations
         * will only be added randomly to some iterations
         */
        unsigned int extraBlocksInPrefix;

        /**
         * reuse: number of blocks that are used twice inside on loop
         */
        unsigned int reusedBlocks;

        /**
         * number of parallel possible control flow paths inside the loop
         * if differentAccessedBlocksPrefix > 0, we add one path in the unrolled iterations
         * on which these blocks are accessed
         */
        unsigned int numberOfControlFlowPaths;
};

/**
 * number of random variants we try per test setup
 */
const unsigned int randomVariants = 10000;

/**
 * analyze one iteration of the given test setup.
 * we pass which iteration and the initial persistence set to use for the analysis
 * fills two counters: counter of accessed done + counter of as persistent classified accesses
 */
PersistenceSet *analyzeIteration (const TestSetup &test
                                 , unsigned int iteration, PersistenceSet *startPersistenceSet,
    unsigned int &accessesDone, unsigned int &accessesPersistent)
{
    /**
     * vector of accesses per path
     */
    static std::vector<std::vector<Address> > accessesPerPath;
    static unsigned int lastIteration = 0;

    /**
     * reuse the same sequence if lubbed context
     */
    if (lastIteration == iteration && iteration == test.unrolling) {
        /**
         * keep results
         */
    } else {
       /**
        * compute new results
        */
       lastIteration = iteration;
       accessesPerPath.clear ();
    }

    if (accessesPerPath.empty()) {
        /**
         * resize vector
         */
        accessesPerPath.resize (test.numberOfControlFlowPaths);

        /**
         * two cases: for the unrolled first iterations and for the lubbed last one
         */
        unsigned int maxUsed = 0;
        bool anyUsed = false;
        if (iteration < test.unrolling) {
            /**
             * distribute accesses over wanted number of control-flow paths
             */
            for (unsigned int access = 0; access < test.commonBlocksInPrefix; ++access) {
                accessesPerPath [randomNumber() % test.numberOfControlFlowPaths].push_back (access);
                maxUsed = access;
                anyUsed = true;
            }

            /**
             * add some blocks to some iterations
             */
            for (unsigned int access = test.differentAccessedBlocks;
                             access < test.differentAccessedBlocks + test.extraBlocksInPrefix; ++access) {
                unsigned int randomValue1 = randomNumber();
                unsigned int randomValue2 = randomNumber();
                if ((randomValue1 % 2) == 0)
                    accessesPerPath [randomValue2 % test.numberOfControlFlowPaths].push_back (access);
            }
        } else {
            /**
             * distribute accesses over wanted number of control-flow paths
             */
            for (unsigned int access = 0; access < test.differentAccessedBlocks; ++access) {
                accessesPerPath [randomNumber() % test.numberOfControlFlowPaths].push_back (access);
                maxUsed = access;
                anyUsed = true;
            }
        }

        /**
         * add reuse, if any common elements used
         */
        if (anyUsed)
            for (unsigned int reuse = 0; reuse < test.reusedBlocks; ++reuse)
                accessesPerPath [randomNumber() % test.numberOfControlFlowPaths].push_back (randomNumber()
                                                                                         % (maxUsed + 1));
    }

    /**
     * simulate the paths and join the results
     */
    PersistenceSet *resultPersistenceSet = 0;
    for (unsigned int path = 0; path < test.numberOfControlFlowPaths; ++path) {
        /**
         * start with initial set
         */
        PersistenceSet *set = startPersistenceSet->clone ();

        /**
         * simulate accesses
         */
        const std::vector<Address> &accesses = accessesPerPath[path];
        for (size_t i = 0; i < accesses.size(); ++i) {
            /**
             * remember access
             */
            ++accessesDone;

            /**
             * classify access, if persistent remember
             */
            if (set->isPersistent (accesses[i]))
                ++accessesPersistent;

            /**
             * update persistence set
             */
            PersistenceSet *newSet = set->update (accesses[i]);
            delete set;
            set = newSet;
        }

        /**
         * compute join
         */
        if (resultPersistenceSet) {
            PersistenceSet *newSet = resultPersistenceSet->join (set);
            delete set;
            delete resultPersistenceSet;
            resultPersistenceSet = newSet;
        } else {
            resultPersistenceSet = set;
        }
    }

    /**
     * return new result set
     */
    return resultPersistenceSet;
}

/**
 * run all test setups for chosen persistence analysis
 */
void runTests (const std::vector<TestSetup> &testSetups, const PersistenceSet *initialSet)
{
    /**
     * run all test setups
     */
    for (unsigned int i = 0; i < testSetups.size(); ++i) {
        /**
         * init random generator
         * we want random sequence, but for any scenario / analysis the same one
         */
        initRandomSeed ();

        /**
         * chose one setup
         */
        const TestSetup &test = testSetups[i];

        /**
         * accesses pre fixed point reached in all analyzed rounds
         */
        unsigned int totalAccessesPreFixedPoint = 0, totalAccessesPreFixedPointPersistent = 0;

        /**
         * accesses in one iteration after fixed point reached in one round
         */
        unsigned int totalAccessesInFixedPoint = 0, totalAccessesInFixedPointPersistent = 0;
```

```
/**
 * we do multiple runs for one setup, with different random distributions
 */
for (unsigned int variant = 1; variant < randomVariants; ++variant) {
    /**
     * accesses pre fixed point reached in all analyzed rounds
     */
    unsigned int accessesPreFixedPoint = 0, accessesPreFixedPointPersistent = 0;

    /**
     * accesses in one iteration after fixed point reached in one round
     */
    unsigned int accessesInFixedPoint = 0, accessesInFixedPointPersistent = 0;

    /**
     * start with clone of initial persistence set
     */
    PersistenceSet *persistenceSet = initialSet->create (test.associativity);

    /**
     * run the analysis for the unrolled iterations
     */
    unsigned int iteration = 1;
    for (; iteration < test.unrolling; ++iteration) {
        /**
         * analyzed one iteration, remember the analysis result
         */
        PersistenceSet *setAfterIteration = analyzeIteration (test, iteration
                                            , persistenceSet, accessesPreFixedPoint
                                            , accessesPreFixedPointPersistent);
        delete persistenceSet;
        persistenceSet = setAfterIteration;
    }

    /**
     * run fixed point iteration on lubbed context
     */
    while (true) {
        /**
         * analyze on iteration
         */
        unsigned int accesses = 0, accessesPersistent = 0;
        PersistenceSet *setAfterIteration = analyzeIteration (test, iteration
                                            , persistenceSet, accesses, accessesPersistent);

        /**
         * join with start
         */
        PersistenceSet *finalSet = setAfterIteration->join (persistenceSet);
        delete setAfterIteration;

        /**
         * fixed point reached?
         */
        if (*finalSet == *persistenceSet) {
            accessesInFixedPoint = accesses;
            accessesInFixedPointPersistent = accessesPersistent;
            delete finalSet;
            delete persistenceSet;
            break;
        }

        /**
         * else: reiterate
         */
        delete persistenceSet;
        persistenceSet = finalSet;
    }

    /**
     * global stats
     */
    totalAccessesPreFixedPoint += accessesPreFixedPoint;
    totalAccessesPreFixedPointPersistent += accessesPreFixedPointPersistent;
    totalAccessesInFixedPoint += accessesInFixedPoint;
    totalAccessesInFixedPointPersistent += accessesInFixedPointPersistent;
}

/**
 * output stats
 */
```

```
        printf ("scenario%u & %u & %u & %0.2f\\%%"
                , i + 1, totalAccessesPreFixedPoint
                , totalAccessesPreFixedPointPersistent
                , 100 * ((double)totalAccessesPreFixedPointPersistent / (double)totalAccessesPreFixedPoint));

        printf (" && %u & %u & %0.2f\\%% \\\\\n"
                , totalAccessesInFixedPoint
                , totalAccessesInFixedPointPersistent
                , 100 * ((double)totalAccessesInFixedPointPersistent / (double)totalAccessesInFixedPoint));
    }

    printf ("\n");
}

int main ()
{
    /**
     * first: construct different test settings
     */
    TestSetup testSetup;
    std::vector<TestSetup> testSetups;

    /**
     * unrolling == 2, 2 ways, 2 different elements accessed, 2 possible control-flow paths
     */
    testSetup.associativity = 2;
    testSetup.unrolling = 2;
    testSetup.differentAccessedBlocks = 2;
    testSetup.commonBlocksInPrefix = 2;
    testSetup.extraBlocksInPrefix = 0;
    testSetup.numberOfControlFlowPaths = 2;
    testSetup.reusedBlocks = 0;
    testSetups.push_back (testSetup);

    /**
     * unrolling == 8, 2 ways, 3 different elements accessed, 3 possible control-flow paths
     */
    testSetup.associativity = 2;
    testSetup.unrolling = 8;
    testSetup.differentAccessedBlocks = 3;
    testSetup.commonBlocksInPrefix = 3;
    testSetup.extraBlocksInPrefix = 0;
    testSetup.numberOfControlFlowPaths = 3;
    testSetup.reusedBlocks = 0;
    testSetups.push_back (testSetup);

    /**
     * unrolling == 2, 2 ways, 2 different elements accessed, 2 possible control-flow paths
     * 1 extra element for the unrolled first iteration
     */
    testSetup.associativity = 2;
    testSetup.unrolling = 2;
    testSetup.differentAccessedBlocks = 2;
    testSetup.commonBlocksInPrefix = 2;
    testSetup.extraBlocksInPrefix = 1;
    testSetup.numberOfControlFlowPaths = 2;
    testSetup.reusedBlocks = 0;
    testSetups.push_back (testSetup);

    /**
     * unrolling == 8, 4 ways, 4 different elements accessed, 4 possible control-flow paths
     * 2 extra element for the unrolled first iteration
     */
    testSetup.associativity = 4;
    testSetup.unrolling = 8;
    testSetup.differentAccessedBlocks = 4;
    testSetup.commonBlocksInPrefix = 4;
    testSetup.extraBlocksInPrefix = 2;
    testSetup.numberOfControlFlowPaths = 4;
    testSetup.reusedBlocks = 0;
    testSetups.push_back (testSetup);

    /**
     * unrolling == 8, 8 ways, 8 different elements accessed, 4 possible control-flow paths
     * 8 extra element for the unrolled first iteration, only 4 of 8 accesses from lubed context
     */
    testSetup.associativity = 8;
    testSetup.unrolling = 8;
    testSetup.differentAccessedBlocks = 8;
    testSetup.commonBlocksInPrefix = 4;
    testSetup.extraBlocksInPrefix = 4;
testSetup.numberOfControlFlowPaths = 4;
testSetup.reusedBlocks = 0;
testSetups.push_back (testSetup);

/**
 * unrolling == 16, 8 ways, 8 different elements accessed, 8 possible control-flow paths
 * 16 extra element for the unrolled first iteration, no elements of fixed point accessed during unrolling
 */
testSetup.associativity = 8;
testSetup.unrolling = 16;
testSetup.differentAccessedBlocks = 8;
testSetup.commonBlocksInPrefix = 0;
testSetup.extraBlocksInPrefix = 16;
testSetup.numberOfControlFlowPaths = 8;
testSetup.reusedBlocks = 0;
testSetups.push_back (testSetup);

/**
 * unrolling == 8, 8 ways, 8 different elements accessed, 4 possible control-flow paths
 * 8 extra element for the unrolled first iteration, only 4 of 8 accesses from lubed context
 */
testSetup.associativity = 8;
testSetup.unrolling = 8;
testSetup.differentAccessedBlocks = 8;
testSetup.commonBlocksInPrefix = 6;
testSetup.extraBlocksInPrefix = 8;
testSetup.numberOfControlFlowPaths = 4;
testSetup.reusedBlocks = 0;
testSetups.push_back (testSetup);

/**
 * unrolling == 8, 4 ways, 4 different elements accessed, 4 possible control-flow paths
 * 2 extra element for the unrolled first iteration
 */
testSetup.associativity = 4;
testSetup.unrolling = 8;
testSetup.differentAccessedBlocks = 4;
testSetup.commonBlocksInPrefix = 4;
testSetup.extraBlocksInPrefix = 2;
testSetup.numberOfControlFlowPaths = 4;
testSetup.reusedBlocks = 1;
testSetups.push_back (testSetup);

/**
 * unrolling == 8, 8 ways, 8 different elements accessed, 4 possible control-flow paths
 * 8 extra element for the unrolled first iteration, only 4 of 8 accesses from lubed context
 */
testSetup.associativity = 8;
testSetup.unrolling = 8;
testSetup.differentAccessedBlocks = 8;
testSetup.commonBlocksInPrefix = 4;
testSetup.extraBlocksInPrefix = 8;
testSetup.numberOfControlFlowPaths = 4;
testSetup.reusedBlocks = 2;
testSetups.push_back (testSetup);

/**
 * unrolling == 16, 8 ways, 8 different elements accessed, 8 possible control-flow paths
 * 16 extra element for the unrolled first iteration, no elements of fixed point accessed during unrolling
 */
testSetup.associativity = 8;
testSetup.unrolling = 16;
testSetup.differentAccessedBlocks = 8;
testSetup.commonBlocksInPrefix = 0;
testSetup.extraBlocksInPrefix = 16;
testSetup.numberOfControlFlowPaths = 8;
testSetup.reusedBlocks = 4;
testSetups.push_back (testSetup);

/**
 * unrolling == 8, 8 ways, 8 different elements accessed, 4 possible control-flow paths
 * 8 extra element for the unrolled first iteration, only 4 of 8 accesses from lubed context
 */
testSetup.associativity = 8;
testSetup.unrolling = 8;
testSetup.differentAccessedBlocks = 8;
testSetup.commonBlocksInPrefix = 6;
testSetup.extraBlocksInPrefix = 8;
testSetup.numberOfControlFlowPaths = 4;
testSetup.reusedBlocks = 6;
testSetups.push_back (testSetup);
```

```
    /**
     * run tests for the conflict counting persistence per set
     */
    printf ("Conflict Counting Per Set\n");
    PersistenceSetConflictSet conflictSetPersistence;
    runTests (testSetups, &conflictSetPersistence);

    /**
     * run tests for the conflict counting persistence per element
     */
    printf ("Conflict Counting Per Element\n");
    PersistenceSetConflictElement conflictElementPersistence;
    runTests (testSetups, &conflictElementPersistence);

    /**
     * run tests for the may based persistence
     */
    printf ("May Based\n");
    PersistenceSetMayBased mayBasedPersistence;
    runTests (testSetups, &mayBasedPersistence);

    /**
     * run tests for the conflicts per element with aging
     */
    printf ("Conflict Counting Per Element With Aging\n");
    PersistenceSetConflictWithAge conflictWithAgingPersistence;
    runTests (testSetups, &conflictWithAgingPersistence);

    /**
     * be done
     */
    return 0;
}
```

A.2 Evaluation Examples

Synthetic Examples for ARM7

Loop with if-then-else construct depending on a condition not known statically that will be persistent:

```
// not known condition
volatile int x;

// test program, loop with statically
// not determined control flow
int main(void)
{
    volatile int t;

    while (t) {
        if (x % 2 == 1)
            t += x * 2;
        else
            x += x * 3;
    }

    return t;
}
```

Loop with switch construct depending on a condition not known statically that leads to evictions:

```
// array to access, to touch the same
// cache set multiple times
volatile int x[4096];

// test program, loop with statically
// not determined control flow
int main(void)
{
    volatile int t;

    while (t) {
        // not decidable switch
        // not all paths same costs
        switch (++t) {
            case 2:
                t = x[0];
                break;

            case 5:
                t = x[4*64];
                break;

            case 12:
                t = x[4*128];
                break;

            case 42:
                t = x[4*192];
                break;

            case 56:
                t = x[4*512];
                ++t;
                ++t;
                break;

            default:
                t = x[4*256];
                break;
        }
    }

    return t;
}
```

Loop with if-then-else construct and a prefix only executed in first iteration that leads to evictions:

```
// array to access, to touch the same
// cache set multiple times
volatile int x[4096];

// test program, loop with prefix
// that differs from normal iterations
int main(void)
{
    volatile int t;
    int first = 1;

    while (t) {
        // do something only in first round
        // may evict stuff
        if (first) {
            t += x[4*512];
            t += x[4*128];
            t += x[4*192];
            t += x[4*256];
            t += x[4*768];
            first = 0;
        }

        // normal iteration
        // like if-then-else example
        if (t % 2 == 1)
            t += x[4*64] * 2;
        else
            x[0] += x[0] * 3;
    }

    return t;
}
```

List of Theorems

List of Figures

List of Tables

Bibliography

[AM95a] Martin Alt and Florian Martin. Generating Analyzers with PAG. Technical report, 1995.

[AM95b] Martin Alt and Florian Martin. Generation of Efficient Interprocedural Analyzers with PAG. In *Proceedings of SAS'95, Static Analysis Symposium*, volume 983, 1995.

[BBB+05] Luca Benini, Davide Bertozzi, Alessandro Bogliolo, Francesco Menichelli, and Mauro Olivieri. Mparm: Exploring the multi-processor soc design space with systemc. *J. VLSI Signal Process. Syst.*, 41:169–182, September 2005.

[BC08] Clément Ballabriga and Hugues 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, Washington, DC, USA, 2008. IEEE Computer Society.

[CC76] Patrick Cousot and Radhia Cousot. Static Determination of Dynamic Properties of Programs. In *Proceedings of the Second International Symposium on Programming*, Dunod, Paris, France, 1976.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[CCF+05] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astrÉe analyzer. In *Programming Languages and Systems, Proceedings of the 14th European Symposium on Programming, volume 3444 of Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[CFG+10] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, and Reinhard Wilhelm. Predictability considerations in the design of multi-core embedded systems. In *Proceedings of Embedded Real Time Software and Systems*, May 2010.

[CM07] Christoph Cullmann and Florian Martin. Data-Flow Based Detection of Loop Bounds. In *Workshop on Worst-Case Execution-Time Analysis (WCET)*, July 2007.

[Cul11] Christoph Cullmann. Cache Persistence Analysis: A Novel Approach - Theory and practice. In *Proceedings of the ACM SIGPLAN/SIGBED 2011 conference on Languages, compilers, and tools for embedded systems, LCTES 2011, Chicago, IL, USA, April 11-14, 2011*, pages 121–130, 2011.

[Eng02] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution-Time Analysis*. PhD thesis, Uppsala University, 2002.

[Erm03] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution-Time Analysis*. PhD thesis, Uppsala University, 2003.

[Fer97] Christian Ferdinand. Cache Behavior Prediction for Real-Time Systems. PhD Thesis, Universität des Saarlandes, 1997.

[FHL$^+$01] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Conference on Embedded Software (EMSOFT)*, volume 2211 of *LNCS*, 2001.

[FMC$^+$07] Christian Ferdinand, Florian Martin, Christoph Cullmann, Marc Schlickling, Ingmar Stein, Stephan Thesing, and Reinhold Heckmann. New Developments in WCET Analysis. In T. Reps, M. Sagiv, and J. Bauer, editors, *Program Analysis and Compilation, Theory and Practice: Essays dedicated to Reinhard Wilhelm*, volume 4444 of *LNCS*, pages 12–52. Springer Verlag, 2007.

[FMWA96] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. In *Science of Computer Programming*, volume 1145, pages 52–66. Springer, 1996.

[FW99] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999.

[GBEL10] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. pages 137–147, Brussels, Belgium, July 2010. OCG.

[GCH11] Gernot Gebhard, Christoph Cullmann, and Reinhold Heckmann. Software structure and WCET predictability. In Philipp Lucas, Lothar Thiele, Benoit Triquet, Theo Ungerer, and Reinhard Wilhelm, editors, *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–10, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Geb10] Gernot Gebhard. Timing anomalies reloaded. In Björn Lisper, editor, *Proceedings of 10th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 5–15. Austrian Computer Society, July 2010.

[GLYY12] Nan Guan, Mingsong Lv, Wang Yi, and Ge Yu. WCET analysis with MRU caches: Challenging LRU for predictability. In *Proceedings of the 2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, RTAS '12, pages 55–64, Washington, DC, USA, 2012. IEEE Computer Society.

[GR10a] Daniel Grund and Jan Reineke. Precise and efficient FIFO-replacement analysis based on static phase detection. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS '10)*, pages 155–164, July 2010.

[GR10b] Daniel Grund and Jan Reineke. Toward precise PLRU cache analysis. In Björn Lisper, editor, *Proceedings of 10th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 28–39. Austrian Computer Society, July 2010.

[Gru12] Daniel Grund. *Static Cache Analysis for Real-Time Systems – LRU, FIFO, PLRU*. PhD thesis, Saarland University, 2012.

[Hai86] I. J. Haikala. Program Behavior in Memory Hierarchies. PhD Thesis, Technical Report A-1986-2, CS Department, University of Helsinki, 1986.

[HAM$^+$99] Christopher A. Healy, Robert D. Arnold, Frank Mueller, Marion G. Harmon, and David B. Whalley. Bounding pipeline and instruction cache performance. *IEEE Trans. Comput.*, 48:53–70, January 1999.

[HJR11] Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. Scope-Aware Data Cache Analysis for WCET Estimation. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, *RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011*, pages 203–212, 2011.

[HLTW03] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Real-Time Systems*, 91(7):1038–1054, 2003.

[HP96] John L. Hennessy and David A. Patterson. *Computer architecture (2nd ed.): a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[HSR$^+$00] C. Healy, M. Sjodin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, pages 121–148, May 2000.

[HWH95] Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Real-Time Systems Symposium (RTSS)*, 1995.

[KU77] J.B. Kam and Jeffrey D. Ullman. Monotone Data Flow Analysis Frameworks. *Acta Informatica*, 7(3), 1977.

[KWN$^+$10] Daniel Kästner, Stephan Wilhelm, Stefana Nenova, Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné, Laurent Mauborgne, and Xavier Rival. Astrée: Proving the absence of runtime errors. In *Embedded Real Time Software and Systems - ERTSS 2010*, 2010.

[LH03] R. Lougee-Heimer. The common optimization interface for operations research: Promoting open-source software in the operations research community. *IBM J. Res. Dev.*, 47(1):57–66, 2003.

[LHP09] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. WCET analysis of multi-level set-associative data caches. In Niklas Holsti, editor, *9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[LS99] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Real-Time Systems Symposium (RTSS)*, December 1999.

[LTH02] Marc Langenbach, Stephan Thesing, and Reinhold Heckmann. Pipeline modeling for timing analysis. In *Proceedings of the 9th International Symposium on Static Analysis*, SAS '02, pages 294–309, London, UK, 2002. Springer-Verlag.

[LYGY10] Mingsong Lv, Wang Yi, Nan Guan, and Ge Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *Real-Time Systems Symposium (RTSS)*, pages 339–349, 2010.

[Mat06] Niklas Matthies. Präzise Bestimmung längster Programmpfade anhand von Zustandsgraphen unter Berücksichtigung von Schleifen-Nebenbedingungen. Master's thesis, Universität des Saarlandes, February 2006.

[MAWF98] Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of loops. In Kai Koskimies, editor, *Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 80–94. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0026424.

[Mue95] Frank Mueller. *Static cache simulation and its applications*. PhD thesis, Tallahassee, FL, USA, 1995. UMI Order No. GAX95-02820.

[Mue00] Frank Mueller. Timing analysis for instruction caches. *Real-Time Syst.*, 18:217–247, May 2000.

[MV99] Nihar R. Mahapatra and Balakrishna Venkatrao. The processor-memory bottleneck: Problems and solutions. *Crossroads - Computer architecture Crossroads Homepage archive*, 5(3es), Spring 1999.

[MWH94] Frank Mueller, David B. Whalley, and Marion Harmon. Predicting instruction cache behavior. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, 1994.

[Nag12] Kartik Nagar. Cache Analysis for Multi-level Data Caches. Master's thesis, Indian Institute of Science, June 2012.

[PPE+08] Traian Pop, Paul Pop, Petru Eles, Zebo Peng, and Alexandru Andrei. Timing analysis of the flexray communication protocol. *Real-Time Systems*, 39:205–235, 2008. 10.1007/s11241-007-9040-3.

[Rei08] Jan Reineke. *Caches in WCET Analysis*. PhD thesis, Universität des Saarlandes, November 2008.

[RG08] Jan Reineke and Daniel Grund. Relative competitive analysis of cache replacement policies. In *LCTES ’08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 51–60, New York, NY, USA, June 2008. ACM.

[RGBW07] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, November 2007.

[RS09] Jan Reineke and Rathijit Sen. Sound and efficient WCET analysis in presence of timing anomalies. In *Workshop on Worst-Case Execution-Time Analysis (WCET)*, 2009.

[RWT$^+$06] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *Workshop on Worst-Case Execution-Time Analysis (WCET)*, July 2006.

[Sch03] Jörn Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Universität des Saarlandes, 2003.

[Sic97] Martin Sicks. Adreßbestimmung zur Vorhersage des Verhaltens von Daten-Caches. Diplomarbeit, Universität des Saarlandes, Fachbereich 14, 1997.

[SLH$^+$05] Jean Souyris, Erwan Le Pavec, Guillaume Himbert, Victor Jégu, Guillaume Borios, and Reinhold Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. In *Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 21–24, 2005.

[Smi82] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[SP81] Micha Sharir and Amir Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7. Prentice-Hall, 1981.

[TFW00] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):157–179, May 2000.

[The00] Henrik Theiling. Extracting safe and precise control flow from binaries. In *Proceedings of the 7th Conference on Real-Time Computing Systems and Applications*, Cheju Island, South Korea, 2000.

[The02] Henrik Theiling. *Control Flow Graphs For Real-Time Systems Analysis*. PhD thesis, Universität des Saarlandes, 2002.

[The04] Stephan Thesing. *Safe and Precise WCET Determinations by Abstract Interpretation of Pipeline Models*. PhD thesis, Universität des Saarlandes, 2004.

[THW94] Ken Tindell, H. Hanssmon, and Andy J. Wellings. Analysing real-time communications: Controller area network (can). In *IEEE Real-Time Systems Symposium*, pages 259–263, 1994.

[TSH+03] Stephan Thesing, Jean Souyris, Reinhold Heckmann, Famantanantsoa Randimbivololona, Marc Langenbach, Reinhard Wilhelm, and Christian Ferdinand. An abstract-interpretation-based timing validation of hard real-time avionics software systems. In *Dependable Systems and Networks (DSN)*, June 2003.

[WGR+09] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7):966–978, July 2009.

[WLP+10] Reinhard Wilhelm, Philipp Lucas, Oleg Parshin, Lili Tan, and Björn Wachter. Improving the precision of WCET analysis by input constraints and model-derived flow constraints. In Samarjit Chakraborty and Jörg Eberspächer, editors, *Advances in Real-Time Systems*, LNCS. Springer-Verlag, 2010.

Index