Noname manuscript No.
(will be inserted by the editor)

Precise, Efficient, and Context-Sensitive Cache Analysis

Florian Brandner · Camille Noûs

Received: date / Accepted: date

Abstract Bounding the *Worst-Case Execution Time* (WCET) of real-time software requires precise knowledge about the reachable program and hardware states that might be observed at runtime. The analysis of precise cache states is particularly important and challenging. Due to the high cost of cache misses the analysis precision may have an important impact on the obtainable WCET bounds, while the large state space of the cache's history leads to high analysis complexity.

This work explores the use of *cache summaries* in order to optimize the computation of *precise* cache states. These cache summaries allow us to pre-compute the impact of executing a portion of a program, typically a function, on the cache state. This allows us, for instance, to *skip* the analysis of entire functions (including nested function calls) when the cache states within these functions are not relevant for the classification of memory accesses into hits/misses. Furthermore, the summaries can be extended to efficiently compute fully context-sensitive cache states. The summaries then not only allow to derive typical cache hit/miss classifications, but also provide fully context-sensitive cache persistence information.

Keywords Cache Analysis, Cache Persistence, Conflict Sets, Cache Summaries, LRU Cache Replacement, Worst-Case Execution Time

Extended Version

This paper is an extension of a paper published at the International Conference on Real-Time Networks and Systems 2020 (RTNS '20) entitled 'Precise and Efficient Analysis of Context-Sensitive Cache Conflict Sets' (Brandner and Noûs, 2020). The

Florian Brandner
LTCI, Télécom Paris, Institut Polytechnique de Paris, France
E-mail: florian.brandner@telecom-paris.fr
ORCID: 0000-0002-2493-7864

Camille Noûs
Laboratoire Cogitamus
E-mail: camille.nous@cogitamus.fr
ORCID: 0000-0002-0778-8115

extensions cover additional background information and formal definitions of the least-recently used cache replacement policy as well as cache persistence analysis (Section 2). Based on this formalization a generalized LRU-based replacement policy, used in the method cache, and formal correctness proofs of the various analyses presented in the work are developed (Section 7). Minor additions include clarifications and corrections, new or improved examples, as well as additional results and figures in the experimental evaluation.

1 Introduction

The computation of tight *Worst-Case Execution Time* (WCET) bounds is challenging due to the increasing size of real-time software (Dvorak, 2009) as well as the increasing complexity of the underlying computer platforms. In hard real-time systems, the WCET analysis needs to consider all reachable program and hardware states that might be observable at runtime. Static program analysis has been applied successfully (Theiling and Ferdinand, 1998) to model both hardware and software states. The information on these states can then be represented as a weighted graph, which is used by the *Implicit Path Enumeration Technique* (IPET) (Li and Malik, 1995; Puschner and Schedl, 1997) to compute the final WCET bound.

A crucial problem is to model the timing-relevant impact of all hardware components in the underlying hardware, including, for instance, the processor pipeline (Stein, 2010). Caches have received considerable attention in the last 20 years, due to the large state space of the cache with regard to the program's execution history. This work focuses on instruction/code caches with a *least-recently used* replacement policy (LRU). Such caches associate an *age* counter with each cache block loaded into the cache. The age of a given cache block m is reset to 0 whenever m is accessed and incremented whenever another *conflicting* cache block (mapped to the same cache set) is accessed, that was *older* than m or not cached (*miss*). On a miss, the LRU policy evicts the *oldest* block of a set if the set is full.

Traditionally, memory accesses of a program (e.g., load, store, instr. fetch) are classified (Alt et al., 1996; Mueller, 1994) as either *always hit* (`AH`), *always miss* (`AM`), or *not classified* (`NC`). One might also consider *cache persistence.* A cache block is persistent with regard to a specific *scope*, i.e., portion of a program, when it stays in the cache once loaded. Persistence gives rise to a fourth classification, with respect to a scope, that is often referred to as *first miss* (`FM`) (Ferdinand and Wilhelm, 1999; Mueller, 1994).

The classifications for the memory accesses of a program can be derived from *conflict sets* (Mueller, 2000; Cullmann, 2013; Huber et al., 2014). A conflict set is usually defined with regard to a *memory block* m (Theiling and Ferdinand, 1998), i.e., an address range in memory that is potentially loaded into the cache as a *cache block*, and denotes the set of *conflicting* memory blocks that map to the same cache set as m and that are loaded into the cache after m. From the size of m's conflict set it is then possible to judge whether m might still be in the cache or not. If the conflict set is sufficiently small, i.e., its cardinality is smaller than or equal to the cache's associativity, then m is known to be in the cache. As analyses compute an over-approximation of conflict sets, the inverse does not necessarily mean that m actually has been evicted.

Conflict sets for LRU caches denote, in fact, the memory blocks that are *younger* than the analyzed memory block m (Touzeau et al., 2017, 2019). Precisely computing conflict sets consequently provides a precise *abstraction* of the concrete cache states with regard to m (modulo the order of blocks w.r.t. their ages). Using an efficient representation of sets of conflict sets (aka. families), Touzeau et al. (2019) proposed to compute precise upper and lower bounds (on the cardinality) of conflict sets to derive cache hit/miss classifications in two passes.

The starting point of this work is *essentially* the same representation of conflict sets, which was developed independently at the same time. We also rely on *Zero-Suppressed Decision Diagrams* (ZDDs) (Minato, 1993) in order to efficiently represent families of conflict sets. In addition we make the following main contributions with respect to the state of the art:

– We propose a *precise* analysis that retains *all* possible conflict sets instead of computing lower/upper bounds on conflict sets.
– Retaining all states inevitably leads to a larger state space and longer analysis times. We thus present novel techniques to drastically reduce the analysis overhead using *cache summaries*. Cache summaries represent the impact on conflict sets when a given portion of a program, such as a function, is executed.
– *Outer cache summaries* allow us to efficiently obtain the conflict sets at the exit points of the considered portion of the program. This allows us, for instance, to quickly derive the cache state after a function call independent from the call context and without reanalyzing the function itself.
– *Inner cache summaries* on the other hand can be used to efficiently obtain the conflict sets right before a memory access within the consider portion of the program. For instance, from the cache state before a function call it is possible to quickly derive the cache state within the called function – again without the need to reanalyze the function.
– We then show how to implement a *context-sensitive hit/miss classification* by combining both kinds of summaries. Even more, we show that *persistence analysis* can be performed virtually for free along with the hit/miss classification.
– Finally, we provide complete proofs demonstrating that the proposed analysis is *correct* and delivers *precise* results, i.e., the analysis computes precisely those conflict sets that can be observed on structurally feasible execution paths.
– Due to the use of cache summaries large parts of programs, that only produce intermediate conflict sets that are irrelevant for the final cache hit/miss classification, are *skipped*, which leads to considerable improvements in analysis time and memory consumption by up to a factor of 200 and 42 respectively.

The paper is organized as follows. We first provide background on the *LRU cache replacement policy*, the *method cache* – a cache variant used in the time-predictable processor architecture Patmos – *inter-procedural control-flow graphs*, as well as *cache* and *persistence analysis* in Section 2. We then provide a motivating example to illustrate shortcomings in the current state-of-the-art in Section 3, before providing a high-level overview of the proposed approach. Section 5 and 6 present a detailed description of the proposed analysis based on outer and inner cache summaries. The correctness of all the analyses presented in this work is discussed in Section 7. The approaches were evaluated and compared against the state-of-the-art in Section 8 using the TACLe benchmark suite (Falk et al., 2016). Finally, related work is discussed in Section 9 before concluding in Section 10.

2 Background

This section provides a brief discussion of the *least-recently used* (LRU) replacement policy, the method cache (Degasperi et al., 2014) of the *time-predictable* Patmos processor (Schoeberl et al., 2011), and finally introduces a precise analysis over families of conflict sets, similar to Touzeau et al. (2019). The analysis relies on a single analysis pass and is extended to support the method cache. Finally, a brief formal definition of persistence is provided. We refer interested readers to the review of Lv et al. (Lv et al., 2016) for an introduction to cache analysis.

2.1 Least-Recently Used Replacement Policy

The LRU replacement policy is a demand-based replacement policy, that determines which *cache blocks* needs to be replaced on a cache miss. As the name suggests, the policy selects the least-recently used cache block for eviction. In order to determine this block the policy needs to keep track of a relative order between cache blocks. This is frequently explained through a stack-like data structure, where the cache blocks are stored. Once a cache block is accessed it is placed on the top of the stack, pushing other blocks down by one position. The least-recently used block is then always located at the bottom of the stack.

This stack representation is not ideal for an actual hardware implementation of an LRU cache. Instead various dedicated data structures have been proposed (Smith, 1982; Kadota et al., 1987). The approach of Kadota et al. (1987), for instance, associates an age-counter with each cache block present in the cache. The behavior of an LRU cache thus can be described by a function that updates the values of the age-counters as blocks are accessed.

Note that here we also track the age of blocks that are currently not present in the cache. We thus introduce the notion of memory blocks, i.e., "cache blocks" in the cache and/or memory:

Definition 1. A **memory block** $m \in MB$ specifies an address range in main memory that is potentially loaded into the (method) cache. The set of memory blocks accessed by the program is given by MB. Each memory block is associated with a non-zero size in bytes via the function $\mathit{size}\colon MB \to \mathbb{N}^+$.

For standard caches the size of the memory blocks is uniform, i.e., all blocks have the same size. However, as explained in the next subsection, alternative designs are possible. The state of a standard LRU cache can be described by associating an age with each memory block. Blocks that are not present in the cache are assigned the maximum age a, as stated by the following definition:

Definition 2. The set of **LRU cache states** of a cache with associativity a is given by $LCS^a \subseteq \mathcal{P}(MB \times \{0, \ldots, a\})$. The function $\mathit{age}\colon LCS^a \times MB \to \{0, \ldots, a\}$ provides the age of a given cache block.

The hardware then operates on such cache states, which evolve according to the performed memory accesses. The following update function – similar to the one by Kadota et al. (1987) – specifies the cache states after an access:

Definition 3. The **LRU update** function takes a cache state $s \in LCS^a$ and a cache block $m \in MB$ as argument and returns a new cache state:

$$update\_LRU^a(s, m) = \{(o, i) \mid \forall o \in MB\colon i = update\_age^a(s, m, o)\}$$

$$update\_age^a(s, m, o) = \begin{cases} 0 & \text{if } m = o \\ age(s, o) + 1 & \text{if } age(s, o) < age(s, m) \\ age(s, o) & \text{otherwise} \end{cases}$$

Example 1. Assume an empty LRU cache with associativity 4 and a sequence of accesses to memory blocks $(\mathtt{m_1}, \mathtt{m_2}, \mathtt{m_3}, \mathtt{m_4}, \mathtt{m_1}, \mathtt{m_5})$. Initially the cache is empty, the hardware cache state is thus given by $\{(\mathtt{m_i}, 4) \mid i \in \{1, \ldots, 5\}\}$. The cache state after the accesses to $\mathtt{m_1}$ are given by $\{(\mathtt{m_1}, 0)\} \cup \{(\mathtt{m_i}, 4) \mid i \in \{2, \ldots, 5\}\}$ and $\{(\mathtt{m_i}, 4 - i) \mid i \in \{1, \ldots, 4\}\} \cup \{(\mathtt{m_5}, 4)\}$ respectively. The final state after accessing $\mathtt{m_5}$ is given by $\{(\mathtt{m_5}, 0), (\mathtt{m_1}, 1), (\mathtt{m_4}, 2), (\mathtt{m_3}, 3), (\mathtt{m_2}, 4)\}$.

It is easy to see that, starting from an empty cache state where all memory blocks are associated with the maximum age a, the update function ensures that each memory block is either assigned a unique age smaller then a or the maximum age a.

2.2 Patmos' Method Cache

The method cache deals with executable code, similar to traditional instruction caches. The main difference is that the cache blocks are formed by the compiler (Hepp and Brandner, 2014) and may exhibit variable sizes. The size of a cache block is prepended to the block's code, along with complementary meta-information.

Like traditional associative caches the method cache (Degasperi et al., 2014) consists of a *cache controller*, a *tag memory*, and a *cache memory*. In traditional caches the number of tag memory entries and the number of cache blocks in the cache memory match. Consequently, the cardinality of the conflict set is sufficient for traditional conflict-set-based cache analyses to obtain a hit/miss classification. However, this is no longer possible for the method cache, due to the variable-sized cache blocks. Both, the number of occupied tag entries (limited by the size of the tag memory) and the space occupied in the cache memory (limited by the size of the cache memory) have to be considered. These limits are specified by cache configurations:

Definition 4. A (method) **cache configuration** is specified by a pair $\langle a, s \rangle$, where a indicates the number of entries in the tag memory and s the size of the cache memory (in bytes).

Note that the method cache typically consists of a single cache set and thus behaves like a fully-associative cache with *least-recently used* (LRU) replacement or a single cache set of a traditional LRU-based (instruction) cache. The subsequent analysis is thus more generic than traditional cache analyses, i.e., standard instruction (and data) caches are a special case of the method cache in terms of the analysis where cache block sizes are fixed.

In addition, cache misses may only occur at specific control-flow instructions: function calls and returns as well as dedicated branches *with cache fill*. This simplifies the processor's pipeline, as misses are handled in the same stage as data cache misses (Schoeberl et al., 2011; Degasperi et al., 2014) – which eliminates timing anomalies known from traditional instruction caches (Hahn and Reineke, 2018). This also benefits cache analysis, since the cache's state may only change when a control-flow instruction is executed. This can explicitly be represented by edges in the control-flow graph, defined next.

2.3 Inter-Procedural Control-Flow Graphs

We rely on a special kind of *Inter-Procedural Control-Flow Graph* (ICFG), which not only captures the calling relations between functions but also explicitly represents the method cache's branch instructions (with/without cache fill) (Jordan et al., 2013; Naji and Brandner, 2015):

Definition 5. An **Inter-Procedural Control-Flow Graph** is given by a tuple $G = (V, E, MB)$ consisting of control-flow nodes in V and control-flow edges in $E \subseteq V \times V$. Each node is associated with a memory block in MB via a function $mb\colon V \to MB$, while edges may represent different kinds of control flow via the function $kind\colon E \to \{\texttt{FLOW}, \texttt{FILL}, \texttt{CALL}, \texttt{RET}, \texttt{LINK}\}$.

Definition 6. A **path** in an ICFG $G = (V, E, MB)$ is a sequence of ICFG nodes $p = (\texttt{n}_1, \ldots, \texttt{n}_\texttt{k})$ such that any consecutive two nodes $\texttt{n}_\texttt{i}, \texttt{n}_{\texttt{i}+1}$, $1 \leq i < k$, in the sequence are connected by an ICFG edge, i.e., $(\texttt{n}_\texttt{i}, \texttt{n}_{\texttt{i}+1}) \in E$.

Definition 7. We call a path an **execution path** if it starts at an ICFG node that has no predecessor and terminates at an ICFG node that has no successor.

Definition 8. A path $p = (\texttt{n}_1, \ldots, \texttt{n}_\texttt{k})$ is **well-formed** either if it does not contain any `LINK` edges, the first `CALL` edge on the path and the last `RET` edge on the path correspond to the same call site, and the sub-path between these `CALL` and `RET` edges is itself well-formed, or otherwise if it is a concatenation of well-formed paths.

For the purpose of this work the code inside the ICFG nodes is actually not relevant, only the memory block of an ICFG node is considered. Apart from `LINK` edges, the various edge kinds actually correspond to different classes of Patmos' control-flow instructions (Schoeberl et al., 2011, 2013). More specifically, `FLOW` edges represent branches *without cache fill*, which do not impact the cache's state, and `FILL` edges correspond to branches *with cache fill*. Function calls and returns are represented by `CALL`/`RET` edges. Edges thus explicitly represent program points where method cache misses may occur. `LINK` edges designate the control-flow successor within a function when *by-passing* a call, i.e., `LINK` edges represent function-local control flow. Such ICFGs can also be defined for standard instruction caches, e.g., by splitting nodes at cache block boundaries.

2.4 Analysis via Families of Conflict Sets

Based on these definitions we can formalize a cache analysis using abstract interpretation (Cousot and Cousot, 1977). We will call this analysis the *baseline analysis* in later sections. This requires the formal definition of an abstract domain, a transfer function, and a meet/join operator. We refer interested readers to the book of Khedker et al. (2009), which gives an excellent introduction.

Abstract Domain: Before defining the abstract domain itself, we first provide a definition of conflict sets and a test that indicates whether a conflict set fits into a given cache:

Definition 9. A **conflict set** $C \subseteq MB$ is a subset of the memory blocks of a program represented by an ICFG $G = (V, E, MB)$.

Definition 10. The conflict set **fits** into a cache with a given cache configuration $\langle a, s\rangle$, if the set's cardinality $|C|$ (tag memory) and size (cache memory) are smaller than or equal to the associativity and size of the cache respectively:

$$\mathit{fits}^{\langle a,s\rangle}(C) = |C| \le a \ \wedge \sum_{m\in C} \mathit{size}(m) \le s. \tag{1}$$

We define the abstract domain using families over power sets of the program's memory blocks ($\mathcal{P}(MB)$). These families (indicated by double stroke letters, e.g., $\mathbb{A}$) represent an over-approximation of the concrete conflict sets on sub-paths starting at an access to a given memory block $m \in MB$. However, one notices that conflict sets may only grow larger as sub-paths get longer. We thus only need to track conflict sets that are small enough to fit into the cache and replace conflict sets, that do not fit, by the special symbol Aleph ($\aleph$):

Definition 11. The **abstract domain** of the static analysis is given by $\mathcal{D} = \mathcal{P}(\{\aleph\} \cup \mathcal{P}(MB))$. The special symbols $\bot = \emptyset \in \mathcal{D}$ indicates the absence of analysis information, while $\aleph$ indicates the presence of conflict sets that do not fit into the cache (c.f. Definition 10).

Definition 12. From a family $\mathbb{I} \in \mathcal{D}$ the **cache hit/miss classification** is derived as follows: *always hit* (`AH`) if $\aleph \notin \mathbb{I}$, *always miss* (`AM`) if $\mathbb{I} = \{\aleph\}$, or *not classified* (`NC`) otherwise.

Transfer Function: Reusing the notation for the *dot product* of two families from previous work (Mishchenko, 2001), given by $\mathbb{A} \cdot \mathbb{B} = \{S \mid \exists A \in \mathbb{A}, \exists B \in \mathbb{B} \colon S = A \cup B\}$, we define the dot product for values from the abstract domain. It replaces conflict sets that do not fit into the cache by $\aleph$ (2nd line), which is needed in the transfer function defined below:

Definition 13. The **dot product with cardinality and size constraints** for a given cache configuration $\langle a, s\rangle$ is given by:

$$\begin{aligned}\mathbb{A} \overset{\langle a,s\rangle}{\cdot} \mathbb{B} = &\{S \in \mathbb{A}\cdot\mathbb{B} \mid \aleph \notin S \wedge \mathit{fits}^{\langle a,s\rangle}(S)\} \cup \\ &\begin{cases} \{\aleph\} & \text{if } \exists R \in \mathbb{A}\cdot\mathbb{B} \colon \aleph \in R \vee \neg \mathit{fits}^{\langle a,s\rangle}(R) \\ \emptyset & \text{otherwise}\end{cases}\end{aligned}$$

The transfer function models the evolution of the conflict sets along sub-paths with respect to a memory block m, considering a cache configuration $\langle a, s\rangle$.

Definition 14. The **transfer function** takes two arguments, an ICFG node n and a family of conflict sets $\mathbb{I} \in \mathcal{D}$, representing *all* sub-paths, starting at another access to m or the program entry and ending right before n:

$$T_m^{\langle a,s\rangle}(\mathbb{I}, n) = \begin{cases} \{\{mb(n)\}\} & \text{if } mb(n) = m \\ \mathbb{I} \overset{\langle a,s\rangle}{\cdot} \{\{mb(n)\}\} & \text{otherwise.}\end{cases}$$

The transfer function produces a new family in $\mathcal{D}$ that either represents extensions of the various sub-paths by appending the memory block accessed by n, or a new sub-path starting at n, i.e., after accessing m.

Meet Operator: The meet operator merges the analysis information along disjoint sets of paths at confluence points, i.e., control-flow nodes with multiple predecessors.

Definition 15. The **meet operator** takes k families of conflict sets $\mathbb{A}_i$ from disjoint sets of sub-paths as input and produces their union:

$$M(\mathbb{A}_1, \ldots, \mathbb{A}_\mathbb{k}) = \{S \mid \exists i, 1 \leq i \leq k \colon S \in \mathbb{A}_i\}.$$

Overall Analysis Flow: The analysis determines the family of conflict sets at every program point one by one for each memory block m potentially accessed by the program. In the case of standard caches the analysis also proceeds per cache set, i.e., the transfer function and meet operator only consider conflicting memory blocks that map to the same cache set. The final hit/miss classification is derived according to Definition 12 on the control flow edges right before accesses to the analyzed memory block m.

The resulting data-flow equations can be solved using the usual fixed-point algorithm (Khedker et al., 2009), while ignoring `LINK` edges in the ICFG, initializing the equations at the program entry to $\{\aleph\}$ (compulsory misses for an empty cache), and initializing the equations to $\perp$ everywhere else. We refer interested readers to Touzeau et al. (2017, 2019) for additional discussion.

Example 2. Assume memory block $\mathtt{m_1}$ is analyzed for a 4-way set-associative cache configuration $\langle 4, 4\rangle$ and an initial family $\mathbb{I} = \{\{\mathtt{m_1}, \mathtt{m_2}, \mathtt{m_3}, \mathtt{m_4}\}, \{\mathtt{m_1}, \mathtt{m_2}, \mathtt{m_4}, \mathtt{m_5}\}\}$. $\mathbb{I}$ at this point contains two conflict sets that both fit into the cache, which represents an *always hit* classification (`AH`). Applying the transfer function $T^{\langle 4,4\rangle}_{\mathtt{m_1}}$ on $\mathbb{I}$ for ICFG nodes $\mathtt{n_3}$ and $\mathtt{n_6}$, accessing memory blocks $\mathtt{m_3}$ and $\mathtt{m_6}$ respectively, yields: $T^{\langle 4,4\rangle}_{\mathtt{m_1}}(\mathbb{I}, \mathtt{n_3}) = \{\{\mathtt{m_1}, \mathtt{m_2}, \mathtt{m_3}, \mathtt{m_4}\}, \aleph\}$ and $T^{\langle 4,4\rangle}_{\mathtt{m_1}}(\mathbb{I}, \mathtt{n_6}) = \{\aleph\}$. The results thus represent a *not classified* (`NC`) and an *always miss* (`AM`) classification respectively.

2.5 Persistence

Persistence has been studied heavily over the years (Ferdinand and Wilhelm, 1999; Huynh et al., 2011; Cullmann, 2013; Stock et al., 2019). The main insight is that code (as well as data) is often accessed repeatedly. It is thus beneficial to specifically analyze whether a given memory block stays in the cache once it was loaded. A typical example concerns memory blocks that are accessed in a loop, e.g., the code of the loop itself. If the code of the entire loop is small enough to fit into the cache, the corresponding memory blocks will not be evicted as long as the loop executes. The memory blocks are thus *persistent* with regard to that loop. Note that the classical hit/miss classification is of no use here: all of the loop's memory blocks would be classified as not-classified (`NC`). This is due to the fact that the memory blocks need to be loaded once, which generally prevents the accesses within the loop to be classified as always-hit (`AH`).

We can illustrate persistence using the specification of the LRU replacement policy using a simple example:

Example 3. Consider a sequence of accesses to memory blocks $(\mathtt{m_1}, \mathtt{m_2}, \mathtt{m_3}, \mathtt{m_4}, \mathtt{m_2}, \mathtt{m_5}, \mathtt{m_1})$ starting from an empty LRU cache with associativity 4. The cache state before the second access to $\mathtt{m_2}$ then evaluates to $\{(\mathtt{m_4}, 0), (\mathtt{m_3}, 1), (\mathtt{m_2}, 2), (\mathtt{m_1}, 3), (\mathtt{m_5}, 4)\}$, which results in a cache hit – the memory block is *persistent*. The second access to $\mathtt{m_1}$, on the other hand, will result in a cache miss, since it was evicted after the access to $\mathtt{m_5}$ resulting in the cache state $\{(\mathtt{m_5}, 0), (\mathtt{m_2}, 1), (\mathtt{m_4}, 2), (\mathtt{m_3}, 3), (\mathtt{m_1}, 4)\}$. Thus, $\mathtt{m_1}$ is not persistent for this example.

From this example it is clear that persistence is a property relative to pairs of memory accesses where the same memory block $m \in MB$ is reused within a specific scope:

Definition 16. A memory block $m \in MB$ is **reused** on an execution path p of an ICFG $G = (V, E, MB)$ with regard to a sub-graph $G' = (V', E', MB)$, representing a scope with $V' \subset V$ and $E' = E \cap V' \times V'$, if p has the following properties:

- p is of the form $(\mathtt{n_1}, \ldots, \mathtt{n_i}, \ldots, \mathtt{n_j}, \ldots \mathtt{n_k})$,
- $mb(\mathtt{n_i}) = mb(\mathtt{n_j})$,
- $\forall l, i < l < j \colon mb(\mathtt{n_l}) \neq mb(\mathtt{n_i})$, and
- $\forall l, i \leq l < j \colon (\mathtt{n_l}, \mathtt{n_{l+1}}) \in E'$.

Based on this notion of reuse we can directly define persistence:

Definition 17. Given the empty cache state S of an LRU cache with associativity a, an ICFG G, a sub-graph G', and a memory block $m \in MB$, persistence is defined by considering *all* execution paths p as well as positions i, and j, where p contains a reuse at nodes $\mathtt{n_i}$ and $\mathtt{n_j} \in p$. Memory block m is **persistent**, with regard to G', if m is still present in the cache for any such path p at the reuse $\mathtt{n_j}$:

$$age(update\_LRU^a(update\_LRU^a(S, \mathtt{n_1}) \ldots, \mathtt{n_j}), m) < a.$$

Note that in this definition, the entire history of cache states from the beginning of the execution paths is considered. However, it is clear from the definition of the LRU update function that the states before reaching $\mathtt{n_i}$ are in fact irrelevant since any access to a memory block other than m will have the same effect on the final age of m.

3 Motivating Example

This section illustrates the baseline analysis from the last section on a small example and highlights two shortcomings.

Example 4. Figure 1 shows the ICFG of a program's `main` function, calling another function `F` several times in a `switch` statement. The internal control flow of the called function is not relevant for this example and thus only illustrated by a simple cloud shape. However, the program's memory blocks, ICFG nodes ($\mathtt{n_i}$) and edges for the `main` function are depicted. We assume that the analysis does not distinguish calling contexts, i.e., the calls to `F` are represented by the same sub-graph of the ICFG (i.e., the cloud shape).

Let's assume that the called function is large and contains highly complex control flow, (conditionally) accessing many different memory blocks. Furthermore, assume that `F` does not access any of the memory blocks of `main`. The cache states

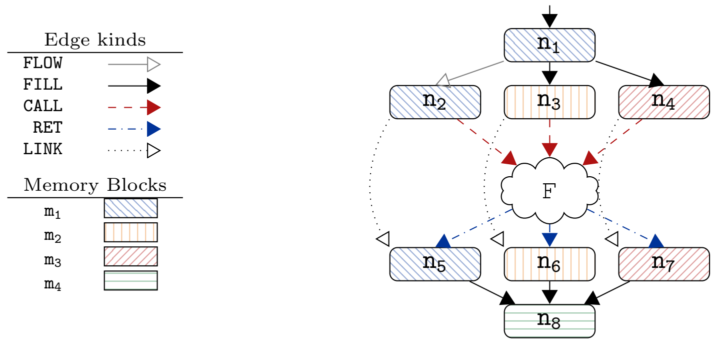


Fig. 1 ICFG of a program (see Example 4).

within the function F are thus irrelevant for the hit/miss classification of main's memory blocks, only the cache states at the accesses in main are actually relevant.

Assume, for instance that memory block m_1, accessed by ICFG nodes n_1, n_2, and n_5, is analyzed. This means that the cache state at the out-going edges of n_1 is represented by the family $\{\{m_1\}\}$. For the execution path on the left-hand side, passing through n_2 and n_5, the same cache state is propagated into F – potentially triggering the computation of a large number of cache states. For the execution path in the middle (n_3, ..., n_6) and on the right (n_4, ..., n_7) different cache states are propagated to the entry of F: $\{\{m_1, m_2\}\}$ and $\{\{m_1, m_3\}\}$ respectively. The function F is reanalyzed every time a new cache state is propagated to its function entry – adding F's memory blocks and merging the conflict sets along the various paths in F. The intermediate cache states for all program points have to be retained in order to obtain the cache state at the RET edges of F, i.e., leading back to the ICFG nodes n_5, n_6, and n_7.

Function F is in some sense analyzed 3 *times* in this example – once for every call context. However, none of the intermediate states concerns an access to memory block m_1 and are thus, by themselves, not relevant for the hit/miss classification of that memory block.

Another issue caused by the call-context insensitivity also becomes apparent. The analysis has no means to differentiate the cache states originating from the calls at n_2, n_3, and n_4. Consequently, all the cache states are propagated along the RET edges. Notably, bogus states containing m_2 or m_3 may reach the node n_5.

The previous example illustrates the high sensitivity of the precise conflict set analysis of Touzeau et al. (2019) with regard to calling contexts: different cache states at different contexts may frequently trigger the computation of a large number of *irrelevant* cache states. The second issue, related to the propagation of bogus states, is circumvented in most WCET analysis tools by completely unrolling all loops (whose iteration bounds have to be known in real-time software anyways) and by (virtually) inlining all functions (recursion is typically discouraged in real-time software). However, this aggressive duplication of code only exacerbates the complexity issue.

The next section introduces cache summaries to avoid both of these problems, with the final goal of obtaining an efficient and fully context-sensitive analysis.

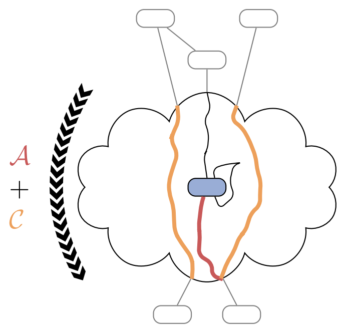
4 Analysis Overview

The analysis proposed in this work proceeds in a similar fashion as the baseline analysis from Section 2. Abstract interpretation is performed in order to compute an over-approximation of the cache states that might appear during any program execution. The analysis is performed independently for each memory block. As illustrated by the motivating example, considerable analysis overhead is caused by re-analyzing sub-graphs of the ICFG representing the program.

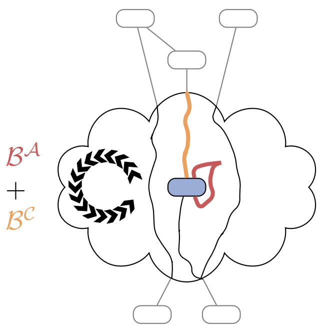
To avoid this issue, this work proposes so-called *cache summaries*. Cache summaries allow us to reason about the evolution of cache states with regard to a sub-graph of the ICFG – for instance functions or loops. The summaries can be *reused* and thus considerably reduce analysis time. However, as illustrated in the following section in more detail, these cache summaries have to cover different execution scenarios in order to capture *all* possible cache states. We thus distinguish two forms of summaries: *outer* and *inner cache summaries*.

Subfigure 2a illustrates the use of outer cache summaries during the analysis, which allow us to capture the evolution of cache states along execution paths passing through a sub-graph. For this, the analysis tracks two kinds of execution paths through the sub-graph along with their respective cache states. Paths that access the analyzed memory block are described by the $\mathcal{A}$ summaries (red), while paths that do not access the analyzed memory block are captured by $\mathcal{C}$ summaries (orange). The $\mathcal{A}$ and $\mathcal{C}$ summaries allow us to efficiently compute the cache states when *leaving* the sub-graph (at the bottom) from the cache states before *entering* the sub-graph (top) – as indicated by the black arrow. The analyses to obtain outer summaries and their application are described in Section 5.

Subfigure 2b illustrates inner cache summaries, which allow us to efficiently derive the cache states that occur before *accessing* the analyzed memory block within the given sub-graph. For this, inner cache summaries have to track the potential cache states along all execution paths that lead to an access of the analyzed memory block. Again two classes of paths are considered. Firstly, the $\mathcal{B}^{\mathcal{C}}$ summaries capture paths that enter the sub-graph from the outside (orange) and lead to the *first* access to the analyzed memory block within the sub-graph, while $\mathcal{B}^{\mathcal{A}}$ summaries capture execution paths that lead from one access to the



(a) Outer Summaries



(b) Inner Summaries

Fig. 2 Cache summaries for the analyzed memory block (blue) with regard to a sub-graph, i.e., the cloud shape.

Notation	Description	Def./Sec.
$\langle a, s\rangle$	Cache configuration (associativity and size)	Definition 4
$\mathbb{I}$, $\mathbb{A}$, $\mathbb{B}$	Families of conflict sets	Definition 11
$\mathbb{A} \overset{\langle a,s\rangle}{\cdot} \mathbb{B}$	Dot product with cardinality and size constraints	Definition 13
$m \in MB$	Memory block accessed/analyzed	Definition 1
$size(m)$	Size (bytes) of a memory block	Definition 1
$G = (V, E, MB)$	Complete ICFG of program	Definition 5
$n \in V$	ICFG node representing code	Definition 5
$mb(n) \in MB$	Memory block of ICFG node	Definition 5
$(u, v) \in E$	ICFG edge representing possible executions	Definition 5
$kind(e)$	Kind of an ICFG edge $e = (u, v)$	Definition 5
G'	Sub-graph of ICFG (e.g., a function)	Definition 18
$entry(G')$	ICFG edges entering a sub-graph	Definition 19
$exit(G')$	ICFG edges exiting a sub-graph	Definition 20
$\overline{G}'$	Collapsed sub-graph for summary application	Definition 21
$\overline{n}_{G''}$	Node representing collapsed sub-graph G'' within G'	Definition 21
$\mathcal{A}_m^{G'\langle a,s\rangle}$	Summary over paths in sub-graph *accessing* m	Section 5.2
$\mathcal{C}_m^{G'\langle a,s\rangle}$	Summary over paths in sub-graph *not accessing* m	Section 5.1
$\mathcal{B}_m^{\mathcal{A}(G')\langle a,s\rangle}$	Summary over paths from access to *access* of m	Section 6
$\mathcal{B}_m^{\mathcal{C}(G')\langle a,s\rangle}$	Summary over paths from sub-graph entry to *access* of m	Section 6
$T_m^{\mathcal{X}(G')\langle a,s\rangle}(\mathbb{I}, n)$	Transfer function of analysis $\mathcal{X}$ for node n in a sub-graph	Sections 5, 6
$M(\mathbb{A}_1, \ldots, \mathbb{A}_k)$	Meet operator (common to all analyses)	Definition 15
$\perp$	Symbol representing *invalid* analysis information	Definition 11
$\aleph$	Symbol (Aleph) representing conflict sets that are too large	Definition 11

Table 1 Summary of notations used for the formal definition of the analysis.

analyzed memory block to another access (red). Combining the information from these two summaries can be used to compute *persistence* information with regard to the scope of that sub-graph (Ferdinand and Wilhelm, 1999; Mueller, 1994). Section 6 provides a detailed description of the analyses required to obtain inner cache summaries.

Inner and outer cache summaries are computed via abstract interpretation on the respective sub-graph only – ignoring other parts of the program. In addition, summaries of nested sub-graphs, i.e., sub-graphs appearing within each other, can be efficiently reused to compute the cache summaries of surrounding sub-graphs (Subsections 5.3 and 6.2). Summaries thus represent *partial* analysis information that can be efficiently combined and reused during the analysis of a given memory block, but also for other memory blocks – resulting in a considerable reduction of analysis complexity. An overview over the various symbols and notations used in the subsequent sections to define the respective analyses is given in Table 1.

5 Outer Cache Summaries

The baseline analysis, presented in Section 2, proceeds by computing families of conflict sets in an incremental way. On each step the analyzed sub-paths are extended by appending a new ICFG node, while updating the conflict sets accordingly. To improve the analysis, one could extend the sub-paths in a more coarse grained fashion, e.g., by concatenating whole sub-paths going through a sub-graph. Let's consider this in a small example:

Example 5. Assume that we have two sub-paths $p_1 = (\mathtt{n_1}, \mathtt{n_2})$ and $p_2 = (\mathtt{n_3}, \mathtt{n_4})$, where each $\mathtt{n_i}$ is associated with a matching memory block $\mathtt{m_i}$, $1 \leq i \leq 4$, and that we wish to analyze the conflict set of $\mathtt{m_1}$. The conflict set of these sub-paths are $\{\mathtt{m_1}, \mathtt{m_2}\}$ and $\{\mathtt{m_3}, \mathtt{m_4}\}$ respectively. Appending p_2 to p_1 gives a new path $(\mathtt{n_1}, \mathtt{n_2}, \mathtt{n_3}, \mathtt{n_4})$, whose conflict set corresponds to the union of the two conflict sets. However, if we append p_1 to p_2, the conflict set of the combined sub-path $(\mathtt{n_3}, \mathtt{n_4}, \mathtt{n_1}, \mathtt{n_2})$ is simply $\{\mathtt{m_1}, \mathtt{m_2}\}$, since the transfer function (Definition 14) resets the information to $\{\mathtt{m_1}\}$ at node $\mathtt{n_1}$ and then adds $\mathtt{m_2}$ to the conflict set.

Apparently one cannot simply take the conflict sets of sub-paths and combine them using a simple set union. This stems from the fact that accesses to the memory block under analysis actually *reset* the conflict set (cf. the first case of Definition 14). However, similar to traditional GEN/KILL data-flow problems (Khedker et al., 2009), one can try to summarize the behavior of these two scenarios separately. We use *two* kinds of *outer cache summaries* for this: $\mathcal{A}$ summaries capture the behavior of a sub-graph of the ICFG along paths that *access* the analyzed memory block, while $\mathcal{C}$ summaries capture paths through the sub-graph where the memory block is *not accessed*.

Definition 18. Given an ICFG $G = (V, E, MB)$ a **sub-ICFG** $G' = (V', E', MB)$ is a sub-graph, where $V' \subseteq V$ and $E' = \{(n, o) \in E \mid n \in V' \vee o \in V'\}$.

Definition 19. The **entry edges** of a sub-ICFG G' are edges that allow to enter the sub-graph: $\mathit{entry}(G') = \{(n, o) \in E' \mid n \notin V' \wedge o \in V'\}$.

Definition 20. The **exit edges** of a sub-ICFG G' are edges that allow to leave the sub-graph: $\mathit{exit}(G') = \{(n, o) \in E' \mid n \in V' \wedge o \notin V'\}$.

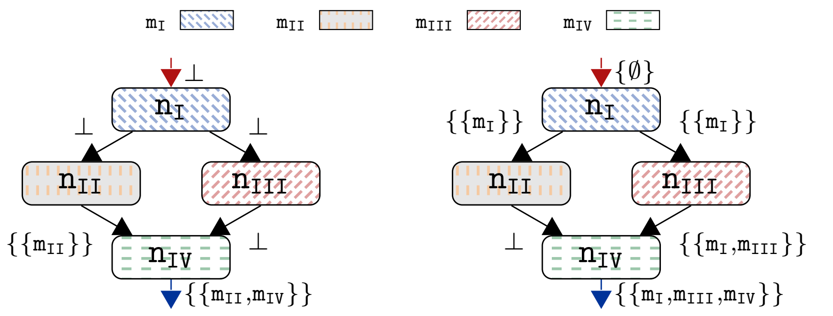
A sub-ICFG can be chosen arbitrarily. However, two classes of sub-graphs appear to be particularly interesting: functions and loops. This work will primarily focus on functions, where the entry and exit edges simply correspond to the corresponding `CALL` and `RET` edges respectively. The summaries are then computed through function-local abstract interpretation.

5.1 $\mathcal{C}$ Summaries for Paths Without Accesses

The objective of $\mathcal{C}$ summaries is to obtain a family of conflict sets that represents the impact of executing any path through a sub-graph, i.e., the impact on the cache state after leaving the sub-graph. For this, we need to consider all sub-paths through the sub-graph that start at an *entry edge*, end at an *exit edge*, and do not access the analyzed memory block. We do not consider summaries of nested sub-graphs, for now.

The analysis reuses the abstract domain and meet operator from Section 2, only the transfer function needs to be modified. A first insight is that the conflict sets for a $\mathcal{C}$ summary evolve quite similarly to the regular conflict sets, i.e., whenever a new ICFG node is encountered its memory block is added to the conflict sets, while respecting the cache characteristics $\langle a, s\rangle$. The main difference is that accesses to the memory block under analysis (m) have to be *filtered*. Instead of producing a valid conflict set it suffices to simply produce an *invalid* ($\perp$) value in the **transfer function** for $\mathcal{C}$ summaries:

$$T_m^{\mathcal{C}(G')\langle a,s\rangle}(\mathbb{I}, n) = \begin{cases} \perp & \text{if } mb(n) = m \\ \mathbb{I} \overset{\langle a,s\rangle}{\cdot} \{\{mb(n)\}\} & \text{otherwise.} \end{cases} \tag{2}$$



(a) $\mathcal{A}$ summary: $\mathcal{A}_{\mathtt{m_{II}}}^{\mathtt{F}\langle 4,4\rangle} = \{\{\mathtt{m_{II}}, \mathtt{m_{IV}}\}\}$ (b) $\mathcal{C}$ summary: $\mathcal{C}_{\mathtt{m_{II}}}^{\mathtt{F}\langle 4,4\rangle} = \{\{\mathtt{m_{I}}, \mathtt{m_{III}}, \mathtt{m_{IV}}\}\}$

Fig. 3 Cache summaries for the memory block of $\mathtt{n_{II}}$ within a simple function F (see Examples 6 and 7).

The usual fixed-point computation is performed on the sub-graph G', while also considering the sub-graph's entry and exit edges. This is important in order to initialize the data-flow equations, which are set to $\{\emptyset\}$ for all entry edges (not to confuse with $\bot = \emptyset$). This means that conflict sets are initially empty when entering the sub-graph, then incrementally grow larger or are reset to $\bot$, and are eventually propagated all the way to the exit edges. The final summary of the sub-graph can then be obtained for each exit edge individually or can be combined over all exit edges $exit(G') = \{e_1, \ldots, e_k\}$ and their respective analysis information $\mathbb{C}_i$, $1 \leq i \leq k$, using the meet operator: $\mathcal{C}_m^{G'\langle a,s\rangle} = M(\mathbb{C}_1, \ldots, \mathbb{C}_k)$.

The $\mathcal{C}$ summaries are specific to a sub-graph G' and the analyzed memory block m. However, it is easy to see that the same summary is computed for all memory blocks that are not accessed within G', i.e., if $\nexists n \in V'\colon mb(n) = m$.

Example 6. Consider the sub-ICFG of function F from Figure 3b, where each node $\mathtt{n_i}$ is associated with a memory block of unit size. The $\mathcal{C}$ summary $\mathcal{C}_{\mathtt{m_{II}}}^{\mathtt{F}\langle 4,4\rangle}$ for this function needs to be computed for memory block $\mathtt{m_{II}}$, accessed by node $\mathtt{n_{II}}$, and the cache configuration $\langle 4, 4\rangle$.

The analysis on the entry edge leading to $\mathtt{n_I}$ is initialized to a family containing only the empty set ($\{\emptyset\}$). Starting from this *empty* conflict set the analysis adds memory blocks $\mathtt{m_I}$, $\mathtt{m_{III}}$, and $\mathtt{m_{IV}}$ along the path on the right side. On the left, the analyzed memory block $\mathtt{m_{II}}$ is accessed, resulting in the analysis information $\bot$ on the edge $(\mathtt{n_{II}}, \mathtt{n_{IV}})$. The conflict set from this path is consequently *filtered* from the cache summary, resulting in a $\mathcal{C}$ summary $\mathcal{C}_{\mathtt{m_{II}}}^{\mathtt{F}\langle 4,4\rangle} = \{\{\mathtt{m_I}, \mathtt{m_{III}}, \mathtt{m_{IV}}\}\}$.

5.2 $\mathcal{A}$ Summaries for Paths With Accesses

$\mathcal{A}$ summaries are similar to $\mathcal{C}$ summaries, except that this time we need to consider all paths through the sub-graph that enter the sub-graph on an *entry edge*, leave the sub-graph on an *exit edge*, and *access* the memory block under analysis.

The analysis is performed on a sub-graph G', including the entry and exit edges, considering a cache configuration $\langle a, s\rangle$ and a memory block m. This time, however, the analysis domain, meet operator, and even the transfer function from the baseline analysis are reused without any modification.

The only difference to the baseline analysis is the initialization of the data-flow equations. The initial value at the entry edges is set to $\perp$. This *filters* the conflict sets from paths that *do not* access the memory block under analysis and only retains the conflict sets of paths that actually do access it.

The final summary of the sub-graph, as before, can be obtained by combining the analysis information over all exit edges $\mathit{exit}(G') = \{e_1, \ldots, e_k\}$ and their respective analysis information $\mathbb{A}_i$, $1 \le i \le k$: $\mathcal{A}_m^{G'\langle a,s\rangle} = M(\mathbb{A}_1, \ldots, \mathbb{A}_k)$.

Example 7. Consider again the sub-ICFG of function `F` from Figure 3a, assuming the same setup as for Example 6. The analysis information at the entry is initialized to $\perp$. Adding new memory blocks consequently does not modify the conflict sets (cf. Definition 13). This only changes after reaching an access to the memory block under analysis at $\mathtt{n_{II}}$, which first produces the conflict set $\{\{\mathtt{m_{II}}\}\}$. Subsequent accesses to other memory blocks on the path on the left-hand side are then added to the conflict set. Combining the conflict sets from the paths on the left and right using the meet operator finally results in the $\mathcal{A}$ summary $\mathcal{A}_{\mathtt{m_{II}}}^{\mathtt{F}\langle 4,4\rangle} = \{\{\mathtt{m_{II}}, \mathtt{m_{IV}}\}\}$.

5.3 $\mathcal{A}/\mathcal{C}$ Summaries for Nested Sub-ICFGs

The analyses defined in the previous subsections allow us to obtain a cache summary for a function. However, functions typically call other functions, for which summaries might exist. This can be seen as an instance of a nested sub-ICFG. The problem is then to exploit the summaries of the nested sub-ICFG instance to compute new summaries for the enclosing sub-graph.

For now, assume that a single nested sub-graph exists. We can collapse this sub-graph G'' by a summary node $\overline{n}_{G''}$ and redirect the entry/exit edges as follows:

Definition 21. Given a (sub-)ICFG $G' = (V', E', \mathit{MB})$ and a nested sub-ICFG $G'' = (V'', E'', \mathit{MB})$, with $V'' \subset V'$ and $E'' = E' \cap V'' \times V''$, the **collapsed sub-ICFG** $\overline{G'} = (\overline{V'}, \overline{E'}, \mathit{MB})$ is defined by: $\overline{V'} = (V' \cup \{\overline{n}_{G''}\}) \setminus V''$ and $\overline{E'} = (E' \cup \{(o, \overline{n}_{G''}) \mid \exists (o,n) \in \mathit{entry}(G'')\} \cup \{(\overline{n}_{G''}, o) \mid \exists (n,o) \in \mathit{exit}(G'')\}) \setminus E''$.

Several nested sub-graphs can easily be handled by collapsing each instance of a nested sub-graph and replacing it by a dedicated summary node. The analyses can then simply be applied to the final collapsed sub-ICFG. However, transfer functions have to be defined for the summary nodes. Assume that several nested sub-ICFGs G_i'' were replaced by summary nodes $\overline{n}_{G_i''}$ to form a collapsed sub-ICFG $\overline{G'}$, the transfer functions for the $\mathcal{A}$ and $\mathcal{C}$ summaries of $\overline{G'}$ then become:

$$T_m^{\overline{\mathcal{C}(G')}\langle a,s\rangle}(\mathbb{I}, n) = \begin{cases} \mathbb{I} \overset{\langle a,s\rangle}{\cdot} \mathcal{C}_m^{G_i''\langle a,s\rangle} & \text{if } \exists i : n = \overline{n}_{G_i''} \\ T_m^{\mathcal{C}(G')\langle a,s\rangle}(\mathbb{I}, n) & \text{otherwise,} \end{cases} \tag{3}$$

$$T_m^{\overline{\mathcal{A}(G')}\langle a,s\rangle}(\mathbb{I}, n) = \begin{cases} M\big(\mathbb{I} \overset{\langle a,s\rangle}{\cdot} \mathcal{C}_m^{G_i''\langle a,s\rangle}, \mathcal{A}_m^{G_i''\langle a,s\rangle}\big) & \text{if } \exists i : n = \overline{n}_{G_i''} \\ T_m^{\langle a,s\rangle}(\mathbb{I}, n) & \text{otherwise.} \end{cases} \tag{4}$$

Both cases refer to the transfer functions (T) defined for simple sub-ICFGs and only perform special actions on the summary nodes representing nested sub-graphs ($\exists i : n = \overline{n}_{G_i''}$).

5.4 Analysis Using Outer Cache Summaries

The ICFG representation, (cf. Section 2), is particularly well suited to compute outer summaries at the level of functions and does not require to explicitly collapse the sub-graphs of functions. Starting from the entry point of a function, it suffices to simply follow the `LINK` edges where the $\mathcal{A}$ and $\mathcal{C}$ summaries of callees are applied, while ignoring `CALL`/`RET` edges.

It remains to exploit the outer summaries in a *regular* analysis. Classifying accesses requires information on conflict sets *before* every access to a memory block, i.e., analysis information at the source node of every `FILL`, `CALL`, and `RET` edge. The outer summaries do not provide this information. One solution would be to use the outer cache summaries only to improve the analysis precision when calls contexts are merged by adopting the transfer function from Equations 3 and 4 and propagating information related to calls only across `LINK` and `CALL` edges (`RET` edges are simply ignored). This would allow to compute the complete analysis information at every access to a given memory block, while eliminating the propagation of bogus analysis information.

An obvious optimization is to *skip* functions that are not relevant to the classification, i.e., functions that do not access the memory block. Note that the $\mathcal{A}$ summary for such functions evaluates to $\perp$, which can be checked efficiently before processing `CALL` edges. Furthermore, only the analysis information on exit edges of a sub-ICFG needs to be retained. Intermediate results can be discarded in order to reduce memory consumption. For functions it generally suffices to only store the combined analysis information over all of the function's `RET` edges. The amount of memory required to store the outer cache summaries is then proportional to the number of functions instead of program points.

Analysis information is still propagated through functions, which leads to intermediate conflict sets that might not be relevant for the cache hit/miss classification. The outer summaries lack information on the conflict sets within sub-ICFGs. The next section proposes a solution to this shortcoming.

6 Inner Cache Summaries

Inner cache summaries describe how the conflict sets for a memory block evolve *up to* some access of that block *within* a sub-ICFG. Two cases have to be distinguished: the memory block is accessed for the *first* time after entering the sub-graph ($\mathcal{B}^{\mathcal{C}}$) and the memory block is accessed *again* after a previous access within the sub-graph ($\mathcal{B}^{\mathcal{A}}$).

6.1 $\mathcal{B}$ Summaries for Simple Sub-ICFGs

The first case corresponds to sub-paths from some entry edge of the sub-ICFG to an ICFG node that accesses the analyzed memory block, without any intermediate accesses to that block. These paths are readily covered by the analysis of the $\mathcal{C}$ summaries. The conflict set at the first access to a memory block can then be computed using the dot product (cf. Definition 13) between the conflict sets before entering the sub-graph and the conflict sets from the analysis of the $\mathcal{C}$ summary right before the access. The second case corresponds to sub-paths within the sub-ICFG starting with an access to the memory block under analysis and leading up to another access to the same memory block. These paths are readily covered by the $\mathcal{A}$ summaries. The conflict sets of these accesses are independent from the initial conflict sets when entering the sub-ICFG and thus do not need any further computation.

Given a simple sub-ICFG G' and a cache configuration $\langle a, s\rangle$, the $\mathcal{B}$ summaries can be derived by a post-processing step from the $\mathcal{A}$ and $\mathcal{C}$ summary analyses. It suffices to retain the analysis information $\mathbb{A}_i$ and $\mathbb{C}_i$ respectively at `FILL` and `LINK` edges that cause an access to the analyzed memory block m: $A = \{e_i \in E' \mid e_i = (u,v)\colon \mathit{kind}(e_i) \in \{\texttt{FILL}, \texttt{LINK}\} \wedge \mathit{mb}(v) = m\}$. One can either store the information individually for each edge or combine it: $\mathcal{B}_m^{\mathcal{A}(G')\langle a,s\rangle} = M(\mathbb{A}_1, \ldots, \mathbb{A}_{|A|})$ and $\mathcal{B}_m^{\mathcal{C}(G')\langle a,s\rangle} = M(\mathbb{C}_1, \ldots, \mathbb{C}_{|A|})$.

Example 8. Consider the ICFG from Figure 3 with a cache configuration $\langle 4, 4\rangle$. The $\mathcal{B}$ summaries for `F` are given by $\mathcal{B}_{\mathtt{m_{II}}}^{\mathcal{A}(\mathtt{F})\langle 4,4\rangle} = \perp$ and $\mathcal{B}_{\mathtt{m_{II}}}^{\mathcal{C}(\mathtt{F})\langle 4,4\rangle} = \{\{\mathtt{m_I}\}\}$, cf. edge $(\mathtt{n_I}, \mathtt{n_{II}})$ in Subfigure 3a and 3b respectively. The former indicates that $\mathtt{m_{II}}$ is not accessed within `F` before reaching $\mathtt{n_{II}}$, while the latter indicates that $\mathtt{m_I}$ is always accessed before reaching $\mathtt{n_{II}}$.

6.2 $\mathcal{B}$ Summaries for Nested Sub-ICFGs

Nested sub-ICFGs G''_i are replaced by summary nodes $\overline{n}_{G''_i}$ in a collapsed sub-ICFG $\overline{G'}$, while redirecting the entry and exit edges. The $\mathcal{B}$ summaries are computed for the analyzed memory block m and the given cache configuration $\langle a, s\rangle$ in a post-processing step. At each edge leading to a summary node of a nested ICFG G''_i, i.e., an edge $\overline{e}_j$ in the set $\{\overline{e}_j \in \overline{E}' \mid \exists i : \overline{e}_j = (u, \overline{n}_{G''_i})\}$ the inner cache summaries of the respective nested sub-ICFG is combined with the function-local analysis information of the $\mathcal{A}$ ($\mathbb{A}_j$) and $\mathcal{C}$ ($\mathbb{C}_j$) summaries at that edge:

$$\overline{\mathbb{A}_j} = M\big(\mathcal{B}_m^{\mathcal{A}(G''_i)\langle a,s\rangle},\ \mathbb{A}_j \overset{\langle a,s\rangle}{\cdot} \mathcal{B}_m^{\mathcal{C}(G''_i)\langle a,s\rangle}\big) \tag{5}$$

$$\overline{\mathbb{C}_j} = \mathbb{C}_j \overset{\langle a,s\rangle}{\cdot} \mathcal{B}_m^{\mathcal{C}(G''_i)\langle a,s\rangle} \tag{6}$$

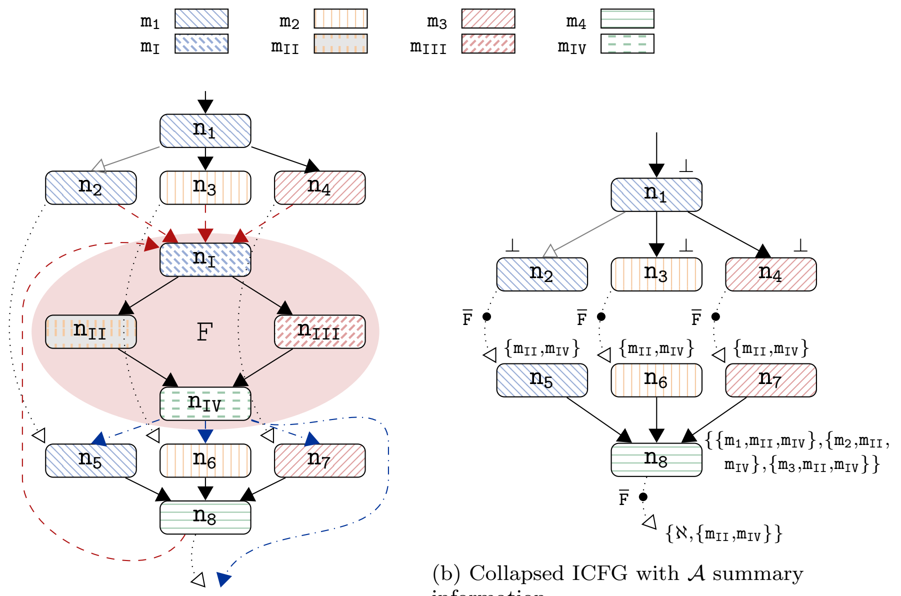
The information from the entry edges can be retained individually or combined using the usual meet operator:

$$\mathcal{B}_m^{\overline{\mathcal{A}(G')}\langle a,s\rangle} = M(\overline{\mathbb{A}_1}, \ldots, \overline{\mathbb{A}_{|\overline{A}|}}) \tag{7}$$

$$\mathcal{B}_m^{\overline{\mathcal{C}(G')}\langle a,s\rangle} = M(\overline{\mathbb{C}_1}, \ldots, \overline{\mathbb{C}_{|\overline{A}|}}). \tag{8}$$

6.3 $\mathcal{B}$ Summaries and Persistence

The $\mathcal{B}^{\mathcal{A}}$ summary information of a sub-graph allows us to derive two kinds of *persistence* classifications: either with regard to a scope covering the sub-graph alone (using the sub-graph's $\mathcal{B}^{\mathcal{A}}$ summary) or a scope covering also parts of the surrounding ICFG (via Equation 5). If the respective summary information evaluates



(a) Complete ICFG of a program

(b) Collapsed ICFG with $\mathcal{A}$ summary information

Fig. 4 Analysis of memory block $\mathtt{m_{II}}$ (see Example 9) considering summaries of the highlighted function F: $\mathcal{A}^{\mathtt{F}\langle 4,4\rangle}_{\mathtt{m_{II}}} = \{\{\mathtt{m_{II}}, \mathtt{m_{IV}}\}\}$, $\mathcal{C}^{\mathtt{F}\langle 4,4\rangle}_{\mathtt{m_{II}}} = \{\{\mathtt{m_{I}}, \mathtt{m_{III}}, \mathtt{m_{IV}}\}\}$, $\mathcal{B}^{\mathcal{A}(\mathtt{F})\langle 4,4\rangle}_{\mathtt{m_{II}}} = \perp$, and $\mathcal{B}^{\mathcal{C}(\mathtt{F})\langle 4,4\rangle}_{\mathtt{m_{II}}} = \{\{\mathtt{m_{I}}\}\}$.

to $\perp$, the analyzed memory block is not reused in the scope. If the summary is $\{\aleph\}$, the block is reused, but definitely evicted. If the analysis information contains $\aleph$, alongside other conflict sets, the block is potentially evicted, while the block is persistent otherwise.

Example 9. Consider Figure 4, which shows a variation of the motivating example from Section 3. The ICFG here consists of the original `main` function, which calls the function `F`, highlighted by the ellipse, several times. `F`'s local control flow is identical to the function from Example 7 and 8. In addition let's assume that `F` is called again at the confluence point at node $\mathtt{n_8}$ at the bottom of Subfigure 4a, as indicated by the **CALL**, **RET**, and **LINK** edges.

We wish to compute persistence information relative to `main` concerning memory block $\mathtt{m_{II}}$, which is only accessed within `F`. The usual cache configuration $\langle 4, 4\rangle$ is considered for the analysis.

The analysis first computes the various summaries of the called function `F`, namely: $\mathcal{A}^{\mathtt{F}\langle 4,4\rangle}_{\mathtt{m_{II}}} = \{\{\mathtt{m_{II}}, \mathtt{m_{IV}}\}\}$, $\mathcal{C}^{\mathtt{F}\langle 4,4\rangle}_{\mathtt{m_{II}}} = \{\{\mathtt{m_{I}}, \mathtt{m_{III}}, \mathtt{m_{IV}}\}\}$, $\mathcal{B}^{\mathcal{A}(\mathtt{F})\langle 4,4\rangle}_{\mathtt{m_{II}}} = \perp$, and $\mathcal{B}^{\mathcal{C}(\mathtt{F})\langle 4,4\rangle}_{\mathtt{m_{II}}} = \{\{\mathtt{m_{I}}\}\}$. The outer summaries ($\mathcal{A}$ and $\mathcal{C}$) are computed from the subgraph representing only `F` as illustrated through Example 7 and 8. The inner cache summaries ($\mathcal{B}^{\mathcal{A}}$ and $\mathcal{B}^{\mathcal{C}}$) are obtained during a post-processing step from the analysis information of the outer cache summaries – more precisely the inner cache summaries are obtained from the ICFG edge ($\mathtt{n_I}$, $\mathtt{n_{II}}$) as depicted by Subfigure 3a and Subfigure 3b respectively.

The analysis then proceeds by computing the cache summaries for the `main` function. However, the analysis does not process the complete ICFG. Instead function calls are – at least conceptually – collapsed as shown in Subfigure 4b. The respective `CALL` and `RET` edges to/from `F` are removed from the graph and replaced by summary nodes labeled $\bar{\mathtt{F}}$, which represent the called function at the various call sites. Note that it is not necessary to actually perform this transformation since the `LINK` edges themselves already represent these call sites.

The analyzed memory block $\mathtt{m_{II}}$ is not accessed in the `main` function itself. Persistence thus only changes at calls to `F`, i.e., the `LINK` edges originating from $\mathtt{n_2}$, $\mathtt{n_3}$, $\mathtt{n_4}$, and $\mathtt{n_8}$. Since persistence is obtained from the $\mathcal{A}$ summary at those edges (cf. Equation 5), we briefly sketch its evolution here (see Subfigure 4b).

The $\mathcal{A}$ summary evaluates to $\perp$ for the `LINK` edges originating from $\mathtt{n_2}$, $\mathtt{n_3}$ and $\mathtt{n_4}$, due to the initialization to $\perp$ at `main`'s entry and the fact that $\mathtt{m_{II}}$ is not accessed before any call to `F`. The function calls evidently have an impact on the $\mathcal{A}$ summary after the respective calls. The corresponding analysis information, shown above nodes $\mathtt{n_5}$, $\mathtt{n_6}$ and $\mathtt{n_7}$ in Subfigure 4b, is derived by applying the outer cache summaries ($\mathcal{A}$ and $\mathcal{C}$) to $\perp$. This yields $\{\mathtt{m_I}, \mathtt{m_{IV}}\}$ (cf. Equation 4).

Next, the transfer functions of the nodes after the calls are applied and the resulting conflict sets combined using the meet operator. The result is shown next to node $\mathtt{n_8}$: $\{\{\mathtt{m_1}, \mathtt{m_{II}}, \mathtt{m_{IV}}\}, \{\mathtt{m_2}, \mathtt{m_{II}}, \mathtt{m_{IV}}\}, \{\mathtt{m_3}, \mathtt{m_{II}}, \mathtt{m_{IV}}\}\}$. Subsequently $\mathtt{m_4}$, accessed by node $\mathtt{n_8}$, is added to all the conflict sets, which represents the analysis information of the $\mathcal{A}$ summary right before the last call to `F`.

At this point all the conflict sets of the analysis have a cardinality of 4, which indicates that $\mathtt{m_{II}}$ is persistent up to this point, i.e., once loaded it is not evicted on any path up to this point. The $\mathcal{A}$ summary analysis now combines the current analysis information with `F`'s outer cache summaries. This results in the following conflict sets: $\{\aleph, \{\mathtt{m_{II}}, \mathtt{m_{IV}}\}\}$, as shown at the bottom of Subfigure 4b. This indicates that $\mathtt{m_{II}}$ is not persistent with regards to the first function calls at $\mathtt{n_5}$, $\mathtt{n_6}$, and $\mathtt{n_7}$, but might be reloaded into the cache by the last call at $\mathtt{n_8}$.

The question now is whether $\mathtt{m_{II}}$ was actually persistent before any of its accesses within `F`. This can be computed by applying the inner cache summaries ($\mathcal{B}^{\mathcal{A}}$ and $\mathcal{B}^{\mathcal{C}}$) to the analysis information right before the various calls to `F`. For the calls at $\mathtt{n_5}$, $\mathtt{n_6}$, and $\mathtt{n_7}$ the $\mathcal{A}$ summary information before the respective calls evaluates to $\perp$, which indicates that $\mathtt{m_{II}}$ was not loaded into the cache between entering `main` and the respective accesses to $\mathtt{m_{II}}$.

For the last call at $\mathtt{n_8}$ the various conflict sets of the $\mathcal{A}$ summary information have a cardinality of 4, as explained in the preceding paragraphs. Appending the $\mathcal{B}^{\mathcal{C}}$ summary ($\{\{\mathtt{m_I}\}\}$) to these conflict sets would yield sets whose cardinality would be larger than the cache size (cf. Equation 5). The analysis thus yields $\{\aleph\}$ right before the accesses to $\mathtt{m_{II}}$ within `F` for this call. Consequently, the analyzed memory block is not persistent.

Combining the analysis information over all call sites to `F` yields $\overline{\mathcal{B}_{\mathtt{m_{II}}}^{\mathcal{A}(\mathtt{main})\langle 4,4\rangle}} = \{\aleph\}$, which indicates that the analyzed memory block is definitely *not persistent* within the `main` function.

6.4 Analysis Using Inner Cache Summaries

Inner cache summaries capture the accesses to the analyzed memory block with regard to a given sub-ICFG. We assume that functions are a typical class of such sub-ICFGs. The information can then be computed in a context-sensitive manner for each call site – similar to the scope graph from Huber et al. (2014). The analysis information is simply propagated upwards from the leaves of the call graph (Aho et al., 2006) to its root.

It remains to show how the hit/miss classification can be derived from the $\mathcal{B}^{\mathcal{C}}$ summaries. The problem here is that the conflict sets are incomplete during the upward propagation, since conflicting accesses up to the respective call sites are missing. This information is only available once the $\mathcal{B}$ summary of the main function is computed. However, Equations 7 and 8 only indicate how this information is merged into a single summary – which corresponds to an analysis without context sensitivity. Two options are possible. The $\mathcal{B}$ summaries can be stored explicitly for edges leading to an access of the memory block under analysis along with the various (nested) call sites. This represents a fully context-sensitive analysis. Alternatively, it is possible to store the $\mathcal{C}$ summaries for the various call sites (cf. Equation 6) and only compute the desired context-sensitive information on-demand by traversing the call graph. The latter is attractive, as it causes minimal memory overhead proportional to the number of functions and call sites.

Note, however, that the upward propagation of analysis information for $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ summaries, based on individual functions, is only possible in acyclic call graphs. Programs containing *recursive* functions, which are usually discouraged in real-time software, thus cannot be handled by the proposed function-based approach. However, it is possible to define sub-ICFGs for the strongly-connected components (SCCs) of the program's call graph, for which outer and inner cache summaries can be derived as a whole.

7 Correctness

Since the summary-based analyses essentially reuse large parts of the baseline analysis from Section 2, we will start by investigating the correctness of this baseline analysis with regards to a specification of an LRU-based method cache and then prove the correctness of the summary-based analyses. As a first step we have to extend the specification of the hardware implementation of the standard LRU replacement policy (cf. Definition 2 and 3) to take associativity and size into account as required for the method cache.

Definition 22. The set of **method cache states** for a cache configuration $\langle a, s\rangle$ is given by $MCS^{\langle a,s\rangle} \subseteq \mathcal{P}(MB \times \{0, \ldots, a\} \times \{1, \ldots, s\})$. The function $mage\colon MCS^{\langle a,s\rangle} \times MB \rightarrow \{0, \ldots, a\}$ provides the age of a given cache block, while $mpos\colon MCS^{\langle a,s\rangle} \times MB \rightarrow \{1, \ldots, s\}$ provides the block's position relative to the cache size.

The update function to determine the cache state after an access is given by:

Definition 23. The **method cache update** function takes a cache state $S \in MCS^{\langle a,s\rangle}$ and a cache block $m \in MB$ as argument and returns a new state in $MCS^{\langle a,s\rangle}$ representing the cache state after the access to m:

$$update\_MLRU^{\langle a,s\rangle}(S, m) = \{(o, i, j) \mid \forall o \in MB\colon (i, j) = update\_mage^{\langle a,s\rangle}(S, m, o)\}$$

$$update\_mage^{\langle a,s\rangle}(S, m, o) =$$

$$\textbf{let } \mathit{shift\_pos} = \mathit{mpos}(S,o) + \mathit{size}(m) \textbf{ in}$$
$$\begin{cases} (0, \mathit{size}(m)) & \text{if } m = o \\ (\mathit{mage}(S,o)+1, \mathit{shift\_pos}) & \text{if } \mathit{shift\_pos} \leq s \wedge \mathit{mage}(S,o) < \mathit{mage}(S,m) \\ (a, \_) & \text{if } \mathit{shift\_pos} > s \wedge \mathit{mage}(S,o) < \mathit{mage}(S,m) \\ (\mathit{mage}(S,o), \mathit{mpos}(S,o)) & \text{otherwise} \end{cases}$$

The main difference with the original specification of the LRU policy (cf. Definition 3) are an additional condition in the 2nd case of the function $\mathit{update\_mage}^{\langle a,s\rangle}$. This condition verifies that a possibly newly loaded memory block fits into the total cache size along with all the blocks that have been present beforehand. The newly added case then handles evictions when loading m into the cache exceeds the total cache size and one or more blocks have to be evicted from the cache.

Example 10. Assume a cache configuration $\langle 4, 8\rangle$ and an access sequence to memory blocks given by $(\mathtt{m_1}, \mathtt{m_2}, \mathtt{m_3})$, where the blocks have a size of 2, 1, and 6 respectively. Starting from an empty cache, where the age of all memory blocks is initialized to 4, the cache state after accessing $\mathtt{m_2}$ is $\{(\mathtt{m_2}, 0, 1), (\mathtt{m_1}, 1, 3), (\mathtt{m_3}, 4, \_)\}$. The final access to $\mathtt{m_3}$ causes the eviction of $\mathtt{m_1}$, since the combined size of $\mathtt{m_1}$, $\mathtt{m_2}$, and $\mathtt{m_3}$ exceeds the cache size $(2 + 1 + 6 > 8)$. This is reflected by the value of the variable *shift_pos* for $\mathtt{m_1}$ $(3 + 6 > 8)$, resulting in the final cache state $\{(\mathtt{m_3}, 0, 6), (\mathtt{m_2}, 1, 7), (\mathtt{m_1}, 4, \_)\}$.

It is easy to see that this extended specification is equivalent to Definition 3 when memory blocks have unit size (i.e., $\forall m \in \mathit{MB}\colon \mathit{size}(m) = 1$) and the cache's size is at least as large as its associativity (i.e., $a \leq s$). The standard LRU policy consequently is merely a special case of the method cache's policy. The update function also retains the same properties as the original policy. Notably, the formal definition of persistence (Definition 17) also applies to the method cache, when considering the modified update function. Furthermore, memory blocks that are present in the cache (i.e., whose age is smaller than a) is unique:

Lemma 1. The age and position of any memory block $m \in \mathit{MB}$, present in the cache, is unique, for any cache state $S \in \mathit{MCS}^{\langle a,s\rangle}$ reachable using the method cache update function from an initially empty cache:

$$\begin{aligned} \forall m, n \in \mathit{MB}, \mathit{mage}(S,m) < a\colon\ & \mathit{mage}(S,m) = \mathit{mage}(S,n) \Longrightarrow m = n\ \wedge \\ & \mathit{mpos}(S,m) = \mathit{mpos}(S,n) \Longrightarrow m = n \end{aligned}$$

Proof. The proof proceeds by induction over the length of an access sequence $(\mathtt{m_1}, \ldots, \mathtt{m_k})$, $\mathtt{m_i} \in \mathit{MB}$:

Induction base:
The lemma trivially holds for the first memory access of any sequence starting from an empty cache, i.e., only $\mathtt{m_1}$ is present in the cache and thus has a unique age (0) and position ($\mathit{size}(\mathtt{m_1})$).

Induction step:
Assuming that all memory blocks present in cache state $S_{i-1} \in \mathit{MCS}^{\langle a,s\rangle}$ have a unique age and position, applying the method cache update function (cf. Definition 23) results in a cache state $S_i \in \mathit{MCS}^{\langle a,s\rangle}$, where all memory blocks have a unique age and position.

We prove this by contradiction. Assume that two memory blocks $m, n \in MB$, $m \neq n$, had different ages in state S_{i-1}, but now have the same age in S_i that is smaller than a. This can only occur in two scenarios:

1. One of the memory blocks (say m) obtained the age 0 in S_i via the first case of the update function, while n retained its age of 0 from the previous state S_{i-1} via the last case of the update function. This implies that $m = \mathtt{m_i}$ and $\mathit{mage}(S_{i-1}, n) \geq \mathit{mage}(S_{i-1}, \mathtt{m_i})$ (cf. the 2nd and 3rd cases of the update function). However, the latter is only possible if both had the age 0 in state S_{i-1}, contradicting the induction base and/or the initial hypothesis that n initially had age 0.
2. One of the memory blocks (say m) increases its age via the 2nd case of the update function, while memory block n retains its age from the previous cache state via the last case. It follows that $\mathit{mage}(S_{i-1}, n) = \mathit{mage}(S_i, n) = \mathit{mage}(S_{i-1}, m) + 1$. Furthermore, it follows that $\mathit{mage}(S_{i-1}, m) < \mathit{mage}(S_{i-1}, \mathtt{m_i})$ as well as $\mathit{mage}(S_{i-1}, n) \geq \mathit{mage}(S_{i-1}, \mathtt{m_i})$. This gives us:
$$\mathit{mage}(S_{i-1}, m) < \mathit{mage}(S_{i-1}, \mathtt{m_i}) \leq \mathit{mage}(S_{i-1}, n) \leq \mathit{mage}(S_{i-1}, m) + 1$$
It follows that $n = \mathtt{m_i}$, due to the induction base and the fact that both, n and $\mathtt{m_i}$, had the same age in state S_{i-1}. This contradicts the initial hypothesis that n retained its age via the last case.

The proof for the positions of memory blocks is analogous. □

7.1 Concrete Conflict Sets

From the method cache states, reflecting an underlying hardware implementation, we may now derive conflict sets quite naturally:

Definition 24. The **concrete conflict set** for a given memory block $m \in MB$ is derived from a method cache state $S \in MCS^{\langle a,s \rangle}$ as follows:

$$\mathit{MCCS}(S, m) = \begin{cases} \aleph & \text{if } \mathit{mage}(S, m) = a \\ \{o \mid \forall o \in MB\colon \mathit{mage}(S, o) \leq \mathit{mage}(S, m)\} & \text{otherwise.} \end{cases}$$

Conflict sets clearly only capture memory blocks that are actually present in the cache for the respective cache state. We can thus state the following invariant, which we will use in a later proof:

Lemma 2. For any cache state $S \in MCS^{\langle a,s \rangle}$, reachable through the method cache update function from an initially empty cache state, the following holds:

$$\forall m \in MB\colon \mathit{MCCS}(S, m) \neq \aleph \implies$$
$$|\mathit{MCCS}(S, m)| = \mathit{mage}(S, m) + 1 \ \wedge \sum_{n \in \mathit{MCCS}(S,m)} \mathit{size}(n) = \mathit{mpos}(S, m).$$

Proof. The proof proceeds by induction over the length of an access sequence $(\mathtt{m_1}, \ldots, \mathtt{m_k})$, $\mathtt{m_i} \in MB$:

Induction base:
Only $\texttt{m}_1$ is present in the cache for the first memory access of any sequence starting from an empty cache, thus: $|\{\texttt{m}_1\}| = \mathit{mage}(S_1, \texttt{m}_1) + 1 = 0 + 1$ as well as $\mathit{mpos}(S_1, \texttt{m}_1) = \mathit{size}(\texttt{m}_1)$. All other memory blocks are not in the cache and their age remains a. The lemma consequently holds.

Induction step:
Assuming that the lemma holds for an access sequence of length $i - 1$, yielding a cache state $S_{i-1} \in \mathit{MCS}^{\langle a,s\rangle}$, we have to show that it also holds in state $S_i \in \mathit{MCS}^{\langle a,s\rangle}$ after an access to $\texttt{m}_\texttt{i}$.

For this we to consider the different cases of the method cache update function (cf. Definition 23):

- $\texttt{m}_\texttt{i} = m$:
  Due to Lemma 1 we know that ages and positions of memory blocks are unique. Similar to the argument for the induction base, the conflict set derived from S_i for $m = \texttt{m}_\texttt{i}$ evaluates to $\{\texttt{m}_\texttt{i}\}$ and the lemma trivially holds.
- $\texttt{m}_\texttt{i} \neq m$:
  - $\mathit{mage}(S_{i-1}, \texttt{m}_\texttt{i}) = \mathit{mage}(S_{i-1}, m)$:
    This case is irrelevant, due to Lemma 1, since $\mathit{mage}(S_{i-1}, \texttt{m}_\texttt{i}) = \mathit{mage}(S_{i-1}, m) = a$, which in turn implies that the conflict sets evaluate to $\aleph$. The lemma hence trivially holds.
  - $\mathit{mage}(S_{i-1}, \texttt{m}_\texttt{i}) < \mathit{mage}(S_{i-1}, m)$:
    The age and position of m do not change (cf. Definition 23, last case). Furthermore, all memory blocks younger than m in S_{i-1} remain younger than m in S_i. Consequently, the conflict set for m does not change and the lemma still holds after the access to $\texttt{m}_\texttt{i}$.
  - $\mathit{mage}(S_{i-1}, m) < \mathit{mage}(S_{i-1}, \texttt{m}_\texttt{i})$:
    This means that $\texttt{m}_\texttt{i}$ was not in the conflict set so far (cf. Definition 24). However, the age of $\texttt{m}_\texttt{i}$, currently accessed, becomes 0 and thus smaller than the age of m. Consequently, m's age has to increase (at least by 1) depending on its blocks position in S_{i-1}:
    - $\mathit{mpos}(S_{i-1}, m) + \mathit{size}(\texttt{m}_\texttt{i}) > s$:
      In this case m is evicted from the cache and its age becomes a. Consequently, the lemma no longer applies to m (cf. Definition 23, 3rd case).
    - $\mathit{mpos}(S_{i-1}, m) + \mathit{size}(\texttt{m}_\texttt{i}) \leq s$:
      In this case the age of m increases by 1, as does the age of all other blocks younger than m (cf. Definition 23, 2nd case). The age of $\texttt{m}_\texttt{i}$ at the same time becomes 0, which implies that $\texttt{m}_\texttt{i}$ is added to the conflict set. The lemma thus holds for the cardinality of the conflict set. Likewise, the position of m (and all blocks younger than m) increases by $\mathit{size}(\texttt{m}_\texttt{i})$, as does the total size of the blocks in the conflict set. □

We may now investigate how the conflict sets evolve, starting from an empty cache with a cache configuration $\langle a, s\rangle$, by comparing the cache states S_1 through S_k corresponding to an access sequence $(\texttt{m}_1, \ldots, \texttt{m}_\texttt{k})$, $\texttt{m}_\texttt{i} \in \mathit{MB}$. More precisely, we are interested in the relationship between the conflict sets derived from these states for every memory block $m \in \mathit{MB}$, i.e., $\mathit{MCCS}(S_i, m)$ for $i \in \{1, \ldots, k\}$. We show that the conflict set can be derived from the preceding conflict set through a simple recursive definition.

Lemma 3. Given an empty cache with a cache configuration $\langle a, s\rangle$, a memory block $m \in MB$, and a sequence of accesses to memory blocks $f = (\mathtt{m_1}, \ldots, \mathtt{m_k})$, $\mathtt{m_i} \in MB$, the concrete conflict set $MCCS(S_i, m)$ of cache state S_i can be computed recursively as follows:

$$C(f, m, i) = \begin{cases} \{\mathtt{m_i}\} & \text{if } \mathtt{m_i} = m \\ C(f, m, i-1) \cup \{\mathtt{m_i}\} & \text{if } C(f, m, i-1) \neq \aleph \wedge \\ & \quad \mathit{fits}^{\langle a,s\rangle}(C(f, m, i-1) \cup \{\mathtt{m_i}\}) \\ \aleph & \text{otherwise} \end{cases}$$

Proof. The proof proceeds by induction on the length of the access sequence:
Induction base:
The same arguments as for Lemma 2 apply.

Induction step:
We again consider the different cases of the method cache update function (cf. Definition 23):

- $\mathtt{m_i} = m$:
  The cache state will yield $\mathit{mage}(S_i, m) = 0$ and consequently result in the conflict set $MCCS(S_i, m) = \{m\}$ (cf. Lemma 1), which is equal to $C(f, m, i)$ due to the first case in the recursive definition.
- $\mathtt{m_i} \neq m$:
  - $\mathit{mage}(S_{i-1}, \mathtt{m_i}) = \mathit{mage}(S_{i-1}, m)$:
    It follows $\mathit{mage}(S_{i-1}, \mathtt{m_i}) = \mathit{mage}(S_{i-1}, m) = a$ due to Lemma 1. As a consequence m does not change its age or position resulting in the following equality $C(f, m, i-1) = MCCS(S_{i-1}, m) = MCCS(S_i, m) = \aleph$. Lemma 3 trivially holds, due to case 3 of the recursive definition (in combination with the condition of case 2).
  - $\mathit{mage}(S_{i-1}, \mathtt{m_i}) < \mathit{mage}(S_{i-1}, m)$:
    The age and position of memory block m do not change (cf. Definition 3, last case). Consequently, the conflict set derived from that cache state does not change and $MCCS(S_{i-1}, m) = C(f, m, i)$ has to hold.
    - $\mathit{mage}(S_{i-1}, m) = a$:
      It follows that $\mathit{mage}(S_i, m) = a$ and consequently $C(f, m, i-1) = MCCS(S_{i-1}, m) = MCCS(S_i, m) = \aleph$. Lemma 3 holds due to the 3rd case and the condition of the 2nd case of the recursive definition.
    - $\mathit{mage}(S_{i-1}, m) < a$:
      It follows that $\mathtt{m_i} \in MCCS(S_{i-1}, m)$ and thus $C(f, m, i-1) = MCCS(S_{i-1}, m) \cup \{\mathtt{m_i}\} = MCCS(S_i, m)$. The condition of the 2nd case of the recursive definition is satisfied (notably, $\mathit{fits}^{\langle a,s\rangle}(C(f, m, i-1) \cup \{\mathtt{m_i}\})$) and Lemma 3 consequently holds.
  - $\mathit{mage}(S_{i-1}, \mathtt{m_i}) > \mathit{mage}(S_{i-1}, m)$:
    This means that the age and position of m have to change (cf. Definition 23, 2nd and 3rd case). In addition, we know that $\mathit{mage}(S_{i-1}, m) < a$, i.e. $C(f, m, i-1) = MCCS(S_{i-1}, m) \neq \aleph$ and thus $|C(f, m, i-1)| < a$ as well as $\mathtt{m_i} \notin C(f, m, i-1)$. The next cache state thus only depends on the new position of m:

- $mpos(S_{i-1}, m) + size(\mathtt{m_i}) \leq s$:
  This means that m still remains in the cache and that $\mathtt{m_i}$ has to be added to the conflict set: $MCCS(S_{i-1}, m) \cup \{\mathtt{m_i}\} = MCCS(S_i, m)$. For the recursive definition we similarly know that $fits^{\langle a,s \rangle}(C(f, m, i-1) \cup \{\mathtt{m_i}\})$ has to be satisfied (cf. Lemma 2 and the fact that $\mathtt{m_i} \notin C(f, m, i-1)$). Consequently, the lemma holds due to its 2nd case.
- $mpos(S_{i-1}, m) + size(\mathtt{m_i}) > s$:
  This means that m is evicted from the cache, resulting in $mage(S_i, m) = a$ (cf. Definition 23, case 3). Lemma 2 ensures that $mpos(S_{i-1}, m)$ corresponds to the total size of the memory blocks in $C(f, m, i-1)$, and, since $\mathtt{m_i} \notin C(f, m, i-1)$, this implies that the predicate $fits^{\langle a,s \rangle}$ of the recursive definition cannot be satisfied. The conflict set derived from S_{i-1} and the recursive definition thus match (cf. its 3rd case). □

7.2 Correctness of the Baseline Analysis

The recursive definition given in the previous section is already very similar to the transfer function of the baseline analysis, which was defined in Section 2. The notable difference is that the transfer function operates on ICFG nodes and families, i.e., sets of conflict set. We will thus first show that the recursive definition always matches the result of the transfer function for a given path in the ICFG.

Lemma 4. Given an empty cache with cache configuration $\langle a, s \rangle$, an ICFG $G = (V, E, MB)$, a memory block $m \in MB$, a path $(\mathtt{n_1}, \ldots, \mathtt{n_k})$ in G leading to an ICFG node $\mathtt{n_k} \in V$, and the corresponding access sequence $f = (mb(\mathtt{n_1}), \ldots, mb(\mathtt{n_k}))$, it follows:

$$\{C(f, m, k)\} = T_m^{\langle a,s \rangle}(T_m^{\langle a,s \rangle}(\aleph, \mathtt{n_1}), \ldots, \mathtt{n_k})$$

Proof. The proof proceeds by induction on the length of the path/access sequence:
Induction base:
For a path of length 1 only $mb(\mathtt{n_1})$ is present in the cache at the end of that path. The transfer function thus yields $T_m^{\langle a,s \rangle}(\aleph, \mathtt{n_1}) = \{\{mb(\mathtt{n_1})\}\}$ or $T_m^{\langle a,s \rangle}(\aleph, \mathtt{n_1}) = \{\aleph\}$, which trivially matches $\{C(f, mb(\mathtt{n_1}), 1)\}$.

Induction step:
Assuming that $\{C(f', m, k-1)\} = T_m^{\langle a,s \rangle}(T_m^{\langle a,s \rangle}(\aleph, \mathtt{n_1}), \ldots, \mathtt{n_{k-1}})$ for a path of length $k-1$, i.e., $(\mathtt{n_1}, \ldots, \mathtt{n_{k-1}})$ and its access sequence f', we have to show that the lemma holds for any path of the form $(\mathtt{n_1}, \ldots, \mathtt{n_{k-1}}, \mathtt{n_k})$ and its access sequence f. For this we have to consider the various cases of the recursive definition of $C(f, m, k)$.

- $mb(\mathtt{n_i}) = m$:
  It follows that $\{C(f, m, k)\} = \{\{mb(\mathtt{n_i})\}\}$, which matches the result of the transfer function (cf. Definition 14, 1st case).
- $mb(\mathtt{n_i}) \neq m$:
  - $C(f, m, k-1) = \aleph$:
    This means that the result of the transfer function yields $T_m^{\langle a,s \rangle}(\aleph, \mathtt{n_k}) = \{\aleph\}$ (cf. Definition 13), which matches the result of the recursive definition due to it's 3rd case.

– $C(f, m, k-1) \neq \aleph$:
The conflict set after the k-th access then depends on the predicate $\mathit{fits}^{\langle a,s\rangle}(C(f, m, k-1) \cup \{mb(\mathtt{n_k})\})$:
  - $\neg\mathit{fits}^{\langle a,s\rangle}(C(f, m, k-1) \cup \{mb(\mathtt{n_k})\})$:
    The transfer function computes the dot product between $C(f, m, k-1)$ and $\{mb(\mathtt{n_k})\}$. This yields $\{\aleph\}$ according to Definition 13. The same goes for the recursive definition due to its 3rd case.
  - $\mathit{fits}^{\langle a,s\rangle}(C(f, m, k-1) \cup \{mb(\mathtt{n_k})\})$:
    The transfer function now also computes the dot product between $C(f, m, k-1)$ and $\{mb(\mathtt{n_k})\}$, which has to yield $\{C(f, m, k-1) \cup \{mb(\mathtt{n_k})\}\}$ (cf. Definition 13). This also matches the result of the recursive definition due to its 2nd case. □

The transfer function thus correctly reflects the conflict sets of individual paths within the ICFG. We can now simply compute the union over these individual conflict sets for each path reaching an ICFG node. This reflects the well known *Meet-Over-all-Path*, or MOP, solution of the analysis problem (Khedker et al., 2009).

Theorem 1. Given an empty cache with cache configuration $\langle a, s\rangle$, an ICFG $G = (V, E, MB)$, a memory block $m \in MB$, the MOP solution of the analysis framework instantiated from the transfer function $T_m^{\langle a,s\rangle}$ (Definition 14) and meet operator M (Definition 15), precisely computes the conflict sets derived from a specification of the LRU replacement policy for the method cache (Definition 23).

Proof. This follows immediately from Lemma 4 and the definition of the meet operator, which simply computes the union over all possible conflict sets. □

In practice it is difficult to compute the MOP solution directly. As indicated in Section 2, it is more common to perform a fixed-point computation. From data-flow analysis theory we know that for certain analysis problems the fixed-point solution is identical to the MOP solution (Khedker et al., 2009). This, in particular, applies to the baseline analysis considered here.

Lemma 5. The meet operator M is distributive over the transfer function $T_m^{\langle a,s\rangle}$, i.e., for any cache configuration $\langle a, s\rangle$, memory block m, ICFG node n, as well as families of memory blocks $\mathbb{A}$ and $\mathbb{B}$:

$$T_m^{\langle a,s\rangle}(M(\mathbb{A}, \mathbb{B}), n) = M(T_m^{\langle a,s\rangle}(\mathbb{A}, n), T_m^{\langle a,s\rangle}(\mathbb{B}, n)).$$

Proof. This follows immediately from Definition 13 and 15. □

Theorem 2. Given an empty cache with cache configuration $\langle a, s\rangle$, an ICFG $G = (V, E, MB)$, a memory block $m \in MB$, the fixed-point solution of the analysis framework instantiated from the transfer function of Definition 14 and meet operator from Definition 15, is identical to the MOP solution.

Proof. This follows from Lemma 5 – see Khedker et al. (2009) for additional discussion. □

7.3 Summary-Based Analyses on Simple Well-Formed Paths

The theorems so far show that the baseline analysis is indeed precise with regard to a MOP solution as well as the underlying specification of the LRU replacement policy for the method cache. However, there is an issue: the proofs consider *all* paths through the ICFG, irrespective of the semantics of `CALL` and `RET` edges. When we restrict the MOP solution to well-formed paths only (Definition 8), the fixed-point computation merely provides a safe *over-approximation*. We will now show that the various summaries proposed in Sections 6 and 5 allow us to perform a precise analysis taking well-formed paths into consideration.

We will start with individual well-formed paths and then generalize to all well-formed paths in an ICFG. Let's first consider a simple well-formed path $p = (\mathtt{n_1}, \dots, \mathtt{n_c}, \mathtt{n_{c+1}}, \dots, \mathtt{n_{r-1}}, \mathtt{n_r}, \dots, \mathtt{n_k})$, with a single function call, i.e., with a single `CALL` and `RET` edge at nodes $\mathtt{n_c}$ and $\mathtt{n_r}$ respectively. The summary-based analysis then simply operates on an equivalent path, where a `LINK` edge replaces the sub-path between `CALL` and `RET` edges. The cache summary is computed precisely on that sub-path. We then have to prove:

– that outer cache summaries allow to correctly compute the conflict set at $\mathtt{n_r}$ for any memory block $m \in MB$,
– that inner cache summaries allow to correctly compute the conflict set for any memory block accessed by the ICFG nodes between $\mathtt{n_c}$ and $\mathtt{n_r}$,
– and, finally, that $\mathcal{B}^{\mathcal{A}}$ summaries correctly represent persistence with regard to the called function for memory blocks accessed by the ICFG nodes between $\mathtt{n_c}$ and $\mathtt{n_r}$.

Before proceeding we will state an auxiliary lemma used later on:

Lemma 6. The dot product with cardinality and size constraints (Definition 13) is associative, i.e., for all values of a and s and families of memory blocks $\mathbb{A}$, $\mathbb{B}$, $\mathbb{C}$:

$$(\mathbb{A} \overset{\langle a,s\rangle}{\cdot} \mathbb{B}) \overset{\langle a,s\rangle}{\cdot} \mathbb{C} = \mathbb{A} \overset{\langle a,s\rangle}{\cdot} (\mathbb{B} \overset{\langle a,s\rangle}{\cdot} \mathbb{C}).$$

Proof. This follows immediately from the definition of the dot product. □

7.3.1 Outer Cache Summaries

For the baseline analysis the regular transfer function is repeatedly applied for each ICFG node along the path p. For the summary-based analyses the regular transfer function is applied until the call at node $\mathtt{n_c}$ is reached. Then the summary, computed from the nodes up the the return at $\mathtt{n_r}$, is applied, which yields the precise conflict sets after the call. The analysis from here on proceeds by applying the regular transfer function again.

Lemma 7. Assume a cache configuration $\langle a, s\rangle$, an ICFG $G = (V, E, MB)$ a well-formed path in G of the form $p = (\mathtt{n_1}, \dots, \mathtt{n_c}, \mathtt{n_{c+1}}, \dots, \mathtt{n_{r-1}}, \mathtt{n_r}, \dots, \mathtt{n_k})$, $\mathtt{n_i} \in V$, $(\mathtt{n_i}, \mathtt{n_{i+1}}) \in E$, where $\mathtt{n_c}$ is the origin of a call edge, i.e., $\mathit{kind}((\mathtt{n_c}, \mathtt{n_{c+1}})) = \mathtt{CALL}$, and $\mathtt{n_r}$ the destination of a return edge, i.e., $\mathit{kind}((\mathtt{n_{r-1}}, \mathtt{n_r})) = \mathtt{RET}$, and all other edges are either `FILL` or `FLOW` edges. The summary-based analysis operates on an equivalent path $p' = (\mathtt{n_1}, \dots, \mathtt{n_c}, \mathtt{n_r}, \dots, \mathtt{n_k})$, where $\mathit{kind}((\mathtt{n_c}, \mathtt{n_r})) = \mathtt{LINK}$ using a summary computed on the sub-path $f = (\mathtt{n_{c+1}}, \ldots, \mathtt{n_{r-1}})$ representing the called function.

The conflict set derived by the baseline analysis (Section 2) before $\mathtt{n_r}$ is identical to the conflict set computed by the summary-based analysis (Section 5) for any memory block $m \in MB$.

Proof. We have to consider two scenarios:

- m is accessed by some ICFG node on p:
  Assume that $\mathtt{n_i}$ denotes the last ICFG node before $\mathtt{n_r}$ that accessed m, i.e., $mb(\mathtt{n_i}) = m$ and $\nexists j, i < j < r\colon mb(\mathtt{n_j}) = m$.
  The conflict set computed by the baseline analysis for m then simply corresponds to an expression of the form $\{mb(\mathtt{n_i})\} \overset{\langle a,s\rangle}{\cdot} \ldots \overset{\langle a,s\rangle}{\cdot} \{mb(\mathtt{n_{r-1}})\}$.
  The $\mathcal{C}$ summary is obtained by repeatedly applying the modified transfer function from Equation 2 starting from an initial conflict set $\{\emptyset\}$, while the $\mathcal{A}$ summary is obtained by applying the regular transfer function (Definition 14) starting from an initial conflict set $\bot$ (i.e., $\emptyset$).
  We then have to distinguish two sub-scenarios:
  - $\nexists j, c < j < r\colon mb(\mathtt{n_j}) = m$:
    This means that the memory block under consideration is not accessed within the called function. Both transfer functions, used during the computation of the outer cache summaries, append the memory block of the next ICFG node to the current conflict set using the dot product (cf. the 2nd case of Equation 2 and Definition 14. For the $\mathcal{A}$ summary this yields $\mathcal{A}^f_m = \bot$, while for the $\mathcal{C}$ summary this simply yields $\mathcal{C}^f_m = \{mb(\mathtt{n_c})\} \overset{\langle a,s\rangle}{\cdot} \ldots \overset{\langle a,s\rangle}{\cdot} \{mb(\mathtt{n_{r-1}})\}$. These summaries are then applied according to Equation 3 (1st case) at $\mathtt{n_c}$, which yields:

    $$M(\{mb(\mathtt{n_i})\} \overset{\langle a,s\rangle}{\cdot} \ldots \overset{\langle a,s\rangle}{\cdot} \{mb(\mathtt{n_c})\} \overset{\langle a,s\rangle}{\cdot} \mathcal{C}^f_m, \mathcal{A}^f_m) = \\ \{mb(\mathtt{n_i})\} \overset{\langle a,s\rangle}{\cdot} \ldots \overset{\langle a,s\rangle}{\cdot} \{mb(\mathtt{n_c})\} \overset{\langle a,s\rangle}{\cdot} \mathcal{C}^f_m.$$

    Due to the associativity of the dot product (Lemma 6) the final result matches that of the baseline analysis.
  - $\exists j, c < j < r\colon mb(\mathtt{n_j}) = m$:
    This means that the memory block is accessed within the called function and thus $\mathtt{n_j} \in f$. Furthermore, it follows that $\mathtt{n_i} \in f$, since, by definition, it is the last access before $\mathtt{n_r}$ on p. We then obtain $\mathcal{C}^f_m = \bot$ and $\mathcal{A}^f_m = \{mb(\mathtt{n_i})\} \overset{\langle a,s\rangle}{\cdot} \ldots \overset{\langle a,s\rangle}{\cdot} \{mb(\mathtt{n_{r-1}})\}$. These summaries are again applied according to Equation 3 at $\mathtt{n_c}$, yielding:

    $$M(\{mb(\mathtt{n_i})\} \overset{\langle a,s\rangle}{\cdot} \ldots \overset{\langle a,s\rangle}{\cdot} \{mb(\mathtt{n_c})\} \overset{\langle a,s\rangle}{\cdot} \mathcal{C}^f_m, \mathcal{A}^f_m) = \mathcal{A}^f_m.$$

    This trivially corresponds to the result obtained from the baseline analysis.
- m is not accessed by any ICFG node on p:
  This means that the baseline analysis computes the conflict set $\{\aleph\}$. The reminder of the proof then proceeds as for the first sub-scenario, with the only difference that the prefix representing the state at the function call $\mathtt{n_c}$ is given by: $\{\aleph\} \overset{\langle a,s\rangle}{\cdot} \{mb(\mathtt{n_1})\} \overset{\langle a,s\rangle}{\cdot} \ldots \overset{\langle a,s\rangle}{\cdot} \{mb(\mathtt{n_{r-1}})\}$. □

7.3.2 Inner Cache Summaries

It remains to show that the conflict sets within the called function, i.e., before ICFG nodes of f are computed correctly. These conflict sets are computed by the baseline analysis through repeated applications of the regular transfer function. This is also true for the summary-based analyses, up to the function call at node $\mathtt{n_c}$, there we apply the inner summary in order to obtain the desired conflict sets. We assume that the inner cache summaries are stored for each ICFG node accessing the memory block m under analysis.

Lemma 8. Assume a cache configuration $\langle a, s\rangle$, an ICFG $G = (V, E, MB)$ a well-formed path in G of the form $p = (\mathtt{n_1}, \ldots, \mathtt{n_c}, \mathtt{n_{c+1}}, \ldots, \mathtt{n_{r-1}}, \mathtt{n_r}, \ldots, \mathtt{n_k})$, $\mathtt{n_i} \in V$, $(\mathtt{n_i}, \mathtt{n_{i+1}}) \in E$, where $\mathtt{n_c}$ is the origin of a call edge, i.e., $\mathit{kind}((\mathtt{n_c}, \mathtt{n_{c+1}})) = \mathtt{CALL}$, and $\mathtt{n_r}$ the destination of a return edge, i.e., $\mathit{kind}((\mathtt{n_{r-1}}, \mathtt{n_r})) = \mathtt{RET}$, and all other edges are either `FILL` or `FLOW` edges. The summary-based analysis operates on an equivalent path $p' = (\mathtt{n_1}, \ldots, \mathtt{n_c}, \mathtt{n_r}, \ldots, \mathtt{n_k})$, where $\mathit{kind}((\mathtt{n_c}, \mathtt{n_r})) = \mathtt{LINK}$ using a summary computed on the sub-path $f = (\mathtt{n_{c+1}}, \ldots, \mathtt{n_{r-1}})$ representing the called function.

The conflict set derived by the baseline analysis (Section 2) before any node $n \in f$ is identical to the conflict set computed by the summary-based analysis (Section 5) for the memory block m accessed by n ($m = mb(n)$).

Proof. The proof is similar to that of Lemma 7 with the only exception that paths to n are considered instead of $\mathtt{n_r}$. The correctness then follows from the constructed expressions by applying the dot product and associativity (Lemma 6). □

7.3.3 Persistence

We first have to revisit Definition 17, which defines persistence using the standard LRU update function and refers to the age of a given memory block. It is obvious that the standard LRU update function can simply be replaced by the method cache update function. Furthermore, it is possible to directly reason about the conflict sets associated with the cache states along the execution paths concerned by the definition. Instead of checking the age of the analyzed memory block, it suffices then to check whether the conflict set of a given path evaluates to $\aleph$ (cf. Lemma 2). For this section we consider this adapted definition for the method cache based on conflict sets.

As for the preceding proofs in this section we will reason about simple well-formed execution paths (p), performing a single function call. The scope of the persistence analysis is precisely the called function, i.e., f. It is obvious that not all of these well-formed paths are relevant for persistence. Definition 16 actually structurally restricts the set of paths to consider only those paths that contain a **reuse** of the analyzed memory block within a specific scope. Recalling that persistence information is derived from $\mathcal{B}^{\mathcal{A}}$ summaries, we have to show that only those paths are actually considered by the persistence analysis:

Lemma 9. Assume a cache configuration $\langle a, s\rangle$, an ICFG $G = (V, E, MB)$ a well-formed execution path in G of the form $p = (\mathtt{n_1}, \ldots, \mathtt{n_c}, \mathtt{n_{c+1}}, \ldots, \mathtt{n_{r-1}}, \mathtt{n_r}, \ldots, \mathtt{n_k})$, $\mathtt{n_i} \in V$, $(\mathtt{n_i}, \mathtt{n_{i+1}}) \in E$, where $\mathtt{n_c}$ is the origin of a call edge, i.e., $\mathit{kind}((\mathtt{n_c}, \mathtt{n_{c+1}})) = \mathtt{CALL}$, and $\mathtt{n_r}$ the destination of a return edge, i.e., $\mathit{kind}((\mathtt{n_{r-1}}, \mathtt{n_r})) = \mathtt{RET}$, and all other edges are either FILL or FLOW edges. The summary-based analysis operates on an equivalent path $p' = (\mathtt{n_1}, \ldots, \mathtt{n_c}, \mathtt{n_r}, \ldots, \mathtt{n_k})$, where $\mathit{kind}((\mathtt{n_c}, \mathtt{n_r})) = \mathtt{LINK}$ using a summary computed on the sub-path $f = (\mathtt{n_{c+1}}, \ldots, \mathtt{n_{r-1}})$ representing the called function.

For every **reuse** of m at positions i and j on p (cf. Definition 16), a corresponding conflict set exists in $\mathcal{B}_m^{\mathcal{A}(f)\langle a,s\rangle}$, while, inversely, no conflict set exists in $\mathcal{B}_m^{\mathcal{A}(f)\langle a,s\rangle}$ for any other path containing an access to m that is not a reuse.

Proof. Recall that the $\mathcal{B}^{\mathcal{A}}$ summaries are in fact derived from the $\mathcal{A}$ summaries defined in Section 5. Lemma 8 already proves that using these summaries the correct conflict set is computed for every access to m within f. However, these conflict sets are derived either directly from the $\mathcal{A}$ summary or the $\mathcal{C}$ summary, which is combined with the conflict set at the call using the dot product. We thus have to exclude the following two scenarios:

– A conflict set in $\mathcal{A}$ does not give rise to a reuse:
  Assume that m is accessed by the ICFG node $\mathtt{n_j} \in p$, where $c < j < r$, that is not a reuse, i.e., there is no other node $\mathtt{n_i} \in p$, with $c < i < j$, that also accesses m. Furthermore assume that the summary computation yielded a non-empty conflict set for the ICFG node $\mathtt{n_{j-1}}$, i.e., that is not $\bot$, which is consequently included in the $\mathcal{A}$ summary and thus in $\mathcal{B}^{\mathcal{A}}$. This, however, is impossible: the analysis information at the beginning of f is initialized to $\bot$ and remains this way until an ICFG node is reached that accesses m. However, our hypothesis excluded the existence of such a node.
– A conflict set in $\mathcal{C}$ does give rise to a reuse:
  Assume that m is accessed by the ICFG node $\mathtt{n_j} \in p$, where $c < j < r$, that is a reuse. This similarly leads to a contradiction. A reuse would imply the existence of a node $\mathtt{n_i}$, $c < i < j$. However, this is impossible since a conflict set in the $\mathcal{C}$ summary has to exist, which would in turn evaluate to $\bot$ due to the first access of the reuse.

It follows that the conflict sets for all actual reuses are indeed derived from the $\mathcal{A}$ summary and thus are covered by $\mathcal{B}_m^{\mathcal{A}(f)\langle a,s\rangle}$. □

7.4 Correctness of the Summary-Based Analyses

The correctness proofs for the summary-based analyses are, so far, limited to simple well-formed paths. This section will briefly discuss how to generalize these proofs.

Inter-Procedural Control-Flow Graphs: As a first generalization consider passing from individual paths to an entire ICFG – for now limited to a single call on each execution path.

Lemma 10. The summary-based analyses provide correct conflict sets for every ICFG node of any ICFG $G = (V, E, \mathit{MB})$, where every execution path only contains a single function call.

Proof. Lemma 7, 8, and 9 apply to all simple well-formed paths. Applying the meet operator thus gives a MOP solution. The correctness of the fixed-point solution then follows from the distributivity of the meet operator (Lemma 5). □

Sequential Composition Next consider ICFGs that may contain a sequence of simple function calls – excluding nested function calls for now.

Lemma 11. The summary-based analyses provide correct conflict sets for every ICFG node of any ICFG $G = (V, E, MB)$, where every execution path may contain a sequence of simple function calls.

Proof. Lemma 7 proves that the first function call on any of these paths can be handled correctly. Lemma 10 can then be generalized by induction over the sequential composition of function calls. Lemma 7, 8, and 9 apply almost without any change, only the considered path prefixes have to be adapted. □

Nested Function Calls: Finally, consider general ICFGs containing any combination of sequential and nested function calls:

Theorem 3. The summary-based analyses provide correct conflict sets for every ICFG node of any ICFG $G = (V, E, MB)$.

Proof. The proof proceeds by induction on the depth of the nested functions calls. It suffices to peel-off the outer-most function calls, i.e., consider the well-formed path previously denoted by f for the inner-most call, and adapt the proofs of Lemma 7, 8, and 9 for the initial cache states $\{\emptyset\}$ and $\perp$. This proves that the summaries obtained from nested function calls are correct – thus proving the analysis overall correct. □

8 Experiments

Our analyses were evaluated for the method cache and standard instruction caches using the TACLe suite (Falk et al., 2016), i.e., benchmarks commonly used to

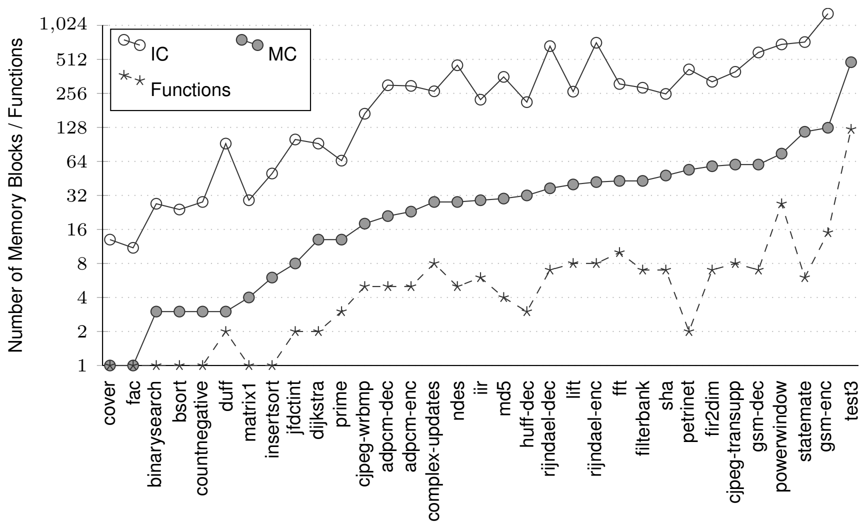


Fig. 5 Number of memory blocks and functions per program of non-recursive TACLe benchmarks (log-2-scale).

evaluate WCET analyzers. We used Patmos' LLVM compiler (version 5.0) with default optimizations (`-O2`). The minimal alignment of basic blocks is 8 B for both kinds of caches, while cache blocks of 16 B are assumed for the instruction cache. For the method cache the compiler was configured to form memory blocks of up to 1 KB, where profitable, and otherwise limit the size to 256 B (Hepp and Brandner, 2014). The compiler was furthermore instructed to form memory blocks having a size that is a multiple of 16 B. During the generation of the benchmark executables the compiler exports the call-context-insensitive ICFGs for the analysis.

Figure 5 shows the number of memory blocks and functions for the TACLe benchmarks that do not contain recursive functions (33 out of 53). For the instruction cache (IC) the programs consist of between 11 and 3466 memory blocks. These numbers are consistent with those of Touzeau et al. (2019), albeit slightly lower. The number of memory blocks for the instruction cache is on average 10× larger than for the method cache (MC). Here, all, but one, benchmarks consist of less than 128 memory blocks. The variable-sized blocks of the method cache thus represents a considerably smaller state space.

Other work (Hepp and Brandner, 2014; Huber et al., 2014; Schoeberl et al., 2018) on the method cache used cache sizes between 1 KB and 16 KB, with 4 to 16 tag entries (associativity). We thus conduct experiments considering cache sizes of 2, 4, 8, and 16 KB and tag memory sizes of 4, 8, 16, and 32 entries. For the standard instruction cache the same configurations are used, resulting in caches having between 4 and 256 cache sets.

The analysis tool relies on *Zero-Suppressed Decision Diagrams* (ZDDs) (Minato, 1993) in order to represent the analysis information. Only simple performance optimizations, based on caching, were applied to the library (improvements should be easy to attain). The tool was compiled with GCC (8.2.1) with standard optimizations (`-O2`). All experiments were carried out on an unloaded workstation, with an Intel Core2 Duo E8500 at 3.16 GHz, a 6 MB L2 cache, and 4 GB of DDR2 main memory, running Linux (Kernel 4.12).

Analysis times were measured using the standard high-resolution clock from the `C++` library (`chrono::high_resolution_clock`) and only comprises the actual analysis time. As the number of potential cache states is quite large, all analysis runs are terminated after a timeout of 90 minutes.

We compare three analyses: a) Baseline, which performs the naive analysis from Section 2, b) Outer, which propagates analysis information throughout the entire program and only relies on outer cache summaries (Section 5.4), and c) Full, which relies on outer and inner cache summaries to compute fully context-sensitive persistence information (Section 6.4). For the outer cache summaries the analysis information over all exit points of the considered functions are retrained and stored permanently, i.e., a pair of pointers representing the $\mathcal{A}$ and $\mathcal{C}$ summary are stored in a look-up table (for the 299 considered functions over all benchmarks). Inner cache summaries are derived as a post-processing step and the analysis information over all potential uses of a memory block are merged. However, the analysis information is retained for each memory block at every call context. For the method cache 718 198 contexts are considered over all benchmarks, whereas 4 211 191 context are distinguished for the standard instruction cache. The number of contexts is higher for the standard cache due to the larger number of memory blocks. The data structures are freed only at the very end of the analysis, the presented numbers consequently represents the peak memory consumption when *all* analysis information is stored in memory.

Note, we never fall back to heuristics, i.e., the three analyses are applied to all memory blocks of a program as described in the previous sections.

8.1 Analysis Complexity

Figure 6 summarizes the average analysis times over all benchmarks for all cache configurations and analyses. As one might expect, analysis complexity heavily increases with the size of the conflict sets, which primarily depends on the cache associativity, the number of memory blocks, and the branching behavior of the considered benchmark. This trend is clearly visible for the method cache (MC). For standard caches (IC) the evolution of the analysis time is not steady. The total cache size here has an important impact, as it tends to reduce the size of the conflict sets by dispersing the memory blocks over a larger number of cache sets.

The analyses based on cache summaries (Outer/Full) clearly outperform the Baseline analyses – by up to a factor of 200. Note furthermore that the Full analysis is considerably faster, despite the fact that it also computes fully-context sensitive persistence. For the method cache the gains increase with the size and associativity. For the instruction cache the gains stay rather constant. Notably, this is also true for 16-way set-associative standard caches. The Baseline (IC) analysis here experiences a much larger number of timeouts than the summary based analysis – which narrows the gap in the plot.

The speedups stem from the fact that the summaries allow us to *skip* the analysis of large portions of the program that are not relevant to the cache hit/miss classification. This means that the corresponding *intermediate* cache states are not computed, which reduces analysis time but also the number of operations on the underlying data structures. Figure 7 shows the average number of calls to an

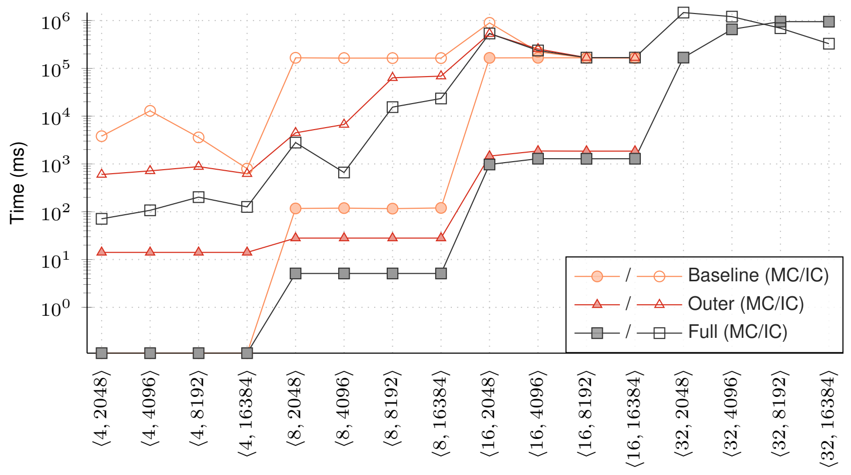


Fig. 6 Total analysis time averaged over all benchmarks for all considered cache configurations (log-scale, lower is better).

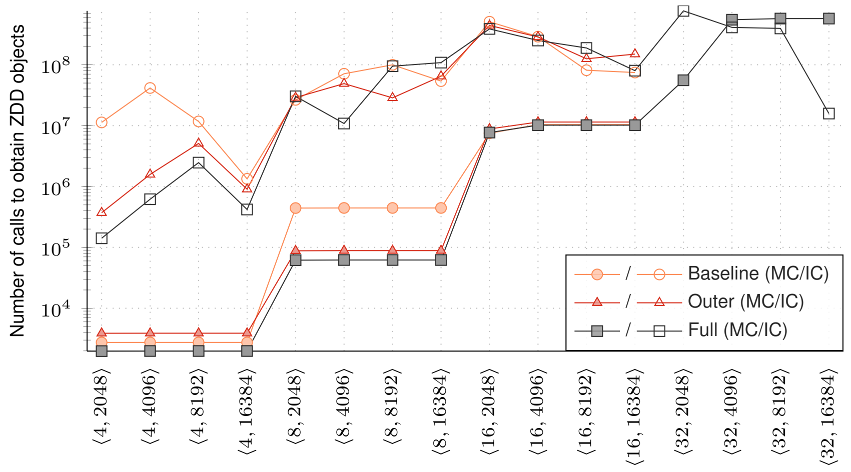


Fig. 7 Average number of calls to the function `get_node` on the ZDD data structure, over all benchmarks for all considered cache configurations (log-scale, lower is better).

elementary function of the ZDD data structure (`get_node`), which performs a look-up and allocates a new object within the ZDD data structure, if needed. All complex ZDD operations (e.g., union, concatenation) used during the analysis call this function very frequently. A correlation between analysis times and the calls to that function is clearly visible. Note, however, that other operations on the ZDDs are also considerably contributing to the total analysis time, but are not captured by this figure.

A timeout of 90 minutes is enforced in terms of analysis time. As indicated before this has an impact on the aggregated data. This applies notably to the Baseline analysis for the method cache, but, most importantly, to the standard instruction cache, which suffers from timeouts for all but the smallest cache configurations as can be seen in Figure 8. For the largest cache configurations with an associativity

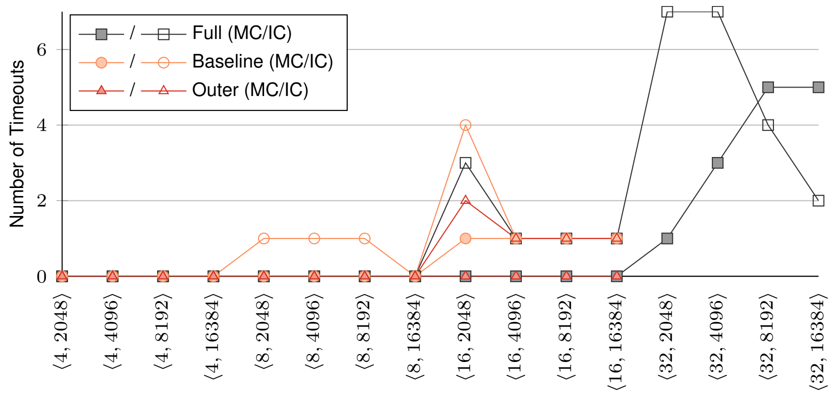


Fig. 8 Number of benchmarks where the analysis runs into a timeout (90 minutes) for all cache configurations (lower is better).

of 32, we only show the Full analyses as the other analysis variants experience too many timeouts. The Full analysis for the method cache experiences between 1 to 5 timeouts with increasing cache size, while the analysis for the standard cache experiences between 2 to 7 timeouts depending on cache size. Profiling showed that the analysis spends most of the time in a few essential functions of the ZDD data structure. It thus should be feasible to improve the respective functions, e.g., through caching or improved organization of the data structure. Some, though not all, of the timeouts might thus be avoidable.

In terms of memory consumption the ZDD representation (Minato, 1993) of the analysis information is, however, highly efficient. The average memory consumption over all configurations peaks slightly above 256 megabytes (MB) – see Figure 9. Whereas the Full analyses for the method and standard caches peak at merely 72 MB (IC) and 15 MB (MC). The gains of the summary-based analyses follow a similar trend as execution times, albeit less pronounced. Summaries reduce memory consumption by up to a factor of 42× (IC) and 7× (MC) respectively. These reductions are, as indicated before, due to the fact that certain intermediate cache states are not computed, which not only reduces the number of calls to the aforementioned `get_node` function, but also reduces the number of objects eventually allocated by that function, as illustrated by Figure 10. This figure is virtually the same as the total memory consumption, which indicates that the ZDD data structure dominates memory consumption and that other data structure, such as look-up tables for the data flow analysis, only marginally contribute. The figures also indicate that, on average, the memory reduction is more pronounced for standard caches. This is due to the fact that the number of memory blocks is much higher for standard caches. This means that more objects have to be allocated within the ZDD data structure in order to represent those memory blocks. This is further exacerbated by the fact that the ZDD data structure relies on a strict ordering of the memory blocks. Performing the various updates on the cache state according to this order is much more efficient. The summary-based analyses con-

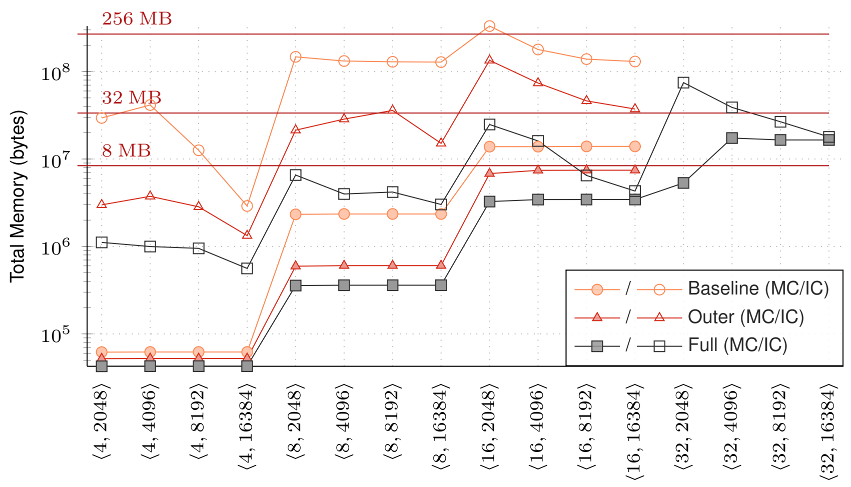


Fig. 9 Total memory consumption averaged over all benchmarks for all cache configurations (log-scale, lower is better).

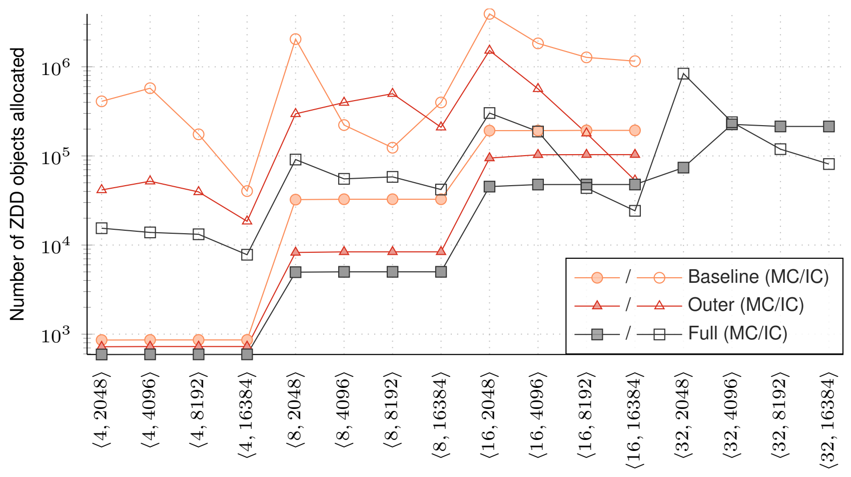


Fig. 10 Total number of objects allocated within the ZDD data structure averaged over all benchmarks for all cache configurations (log-scale, lower is better).

sequently profit from the fact that the summary information intrinsically respects this order. The Baseline analysis, on the other hand, updates the cache state in an order imposed by the ICFG structure. It should be noted that we have optimized the data-flow analysis and numbering of memory blocks in order to minimize this effect.

A more detailed view is given in Figure 11 and 12, which show the memory consumption and analysis time required by the individual benchmarks for a 16 KB method cache with a tag memory size of 8 entries (dashed lines indicate

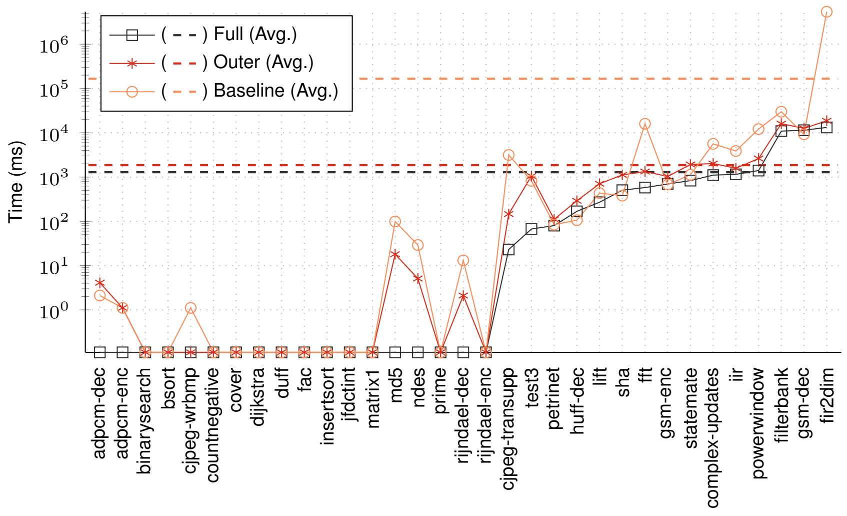


Fig. 11 Total analysis time for all benchmarks for a 16 KB method cache with associativity 16 (log-scale, lower is better).

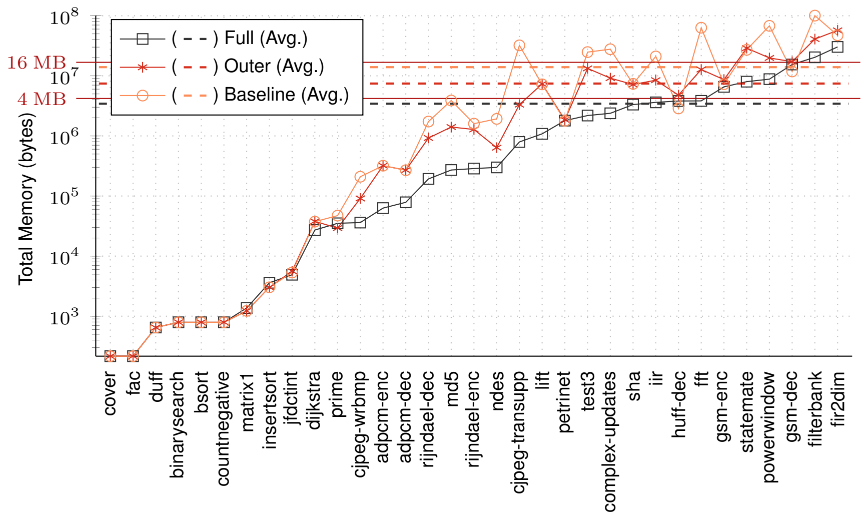


Fig. 12 Total memory consumption for all benchmarks for a 16 KB method cache with associativity 16 (log-scale, lower is better).

the average). These figures confirm the general trend observed over all cache configurations: cache summaries yield considerable improvements up to two orders of magnitudes. In rare cases the summary-based analyses perform worse though, as illustrated by `huff-dec`. This slowdown can be explained by the simple structure of the benchmark, which consists of only 3 functions that are only called once outside of loop structures. Consequently each function has only a single call context and the computations and bookkeeping related to the summaries cause a slight overhead. Our measurements, however, indicate that only moderate slowdowns are incurred in these cases, compared to the gains observed otherwise. For instance, for this cache configuration the degradation measured for the `huff-dec` benchmark amounts to *merely* 59% and 33% in terms of analysis time and memory consumption respectively.

The presented results show that cache summaries proposed in this work successfully reduce analysis complexity, both in terms of analysis time and memory consumption, by orders of magnitudes.

8.2 Comparison with the State of the Art

As pointed out before, the proposed analyses and in particular the Baseline analysis (Section 2) are similar to the work of Touzeau et al. (2019). The main difference is that Touzeau et al. compute maximum/minimum conflict sets in two passes, while the analyses presented here compute *all* conflict sets in a single pass. This difference has two important implications. For one, the state space is much larger when considering *all* conflict sets. This may increase analysis time and memory consumption. On the other hand, more information is available in the analyses presented here – since all conflict sets are retained. This might prove interesting. For instance, the analysis information can be used to determine *eviction points*, i.e., program locations where memory blocks are evicted from the cache. This might allow us to prove refined bounds on the number of cache misses, which are intrinsically linked to the number of evictions.

In order to evaluate the impact on analysis time we compare the analysis time of the Baseline analysis with the original data from the paper of Touzeau et al. (2019, Figure 10) – kindly provided by the authors. Note that this figure shows the analysis time of an *optimized* analysis flow that combines a fast, but imprecise, age-based analysis (Alt et al., 1996), with a second imprecise refinement to classify accesses as definitely unused (Touzeau et al., 2017). Their ZDD approach is thus only applied to a fraction of the memory blocks – which, in theory, should be highly favorable for this comparison. According to the authors, applying the ZDD approach without these heuristics is roughly 3 times slower, which is also illustrated by a figure in their paper (Touzeau et al., 2019, Figure 9a).

Note that the subsequent comparison should be taken with a grain of salt. The computer platform used here is an Intel Core2 Duo E8500 (released in 2008), while in their work a more powerful Intel Xeon E5-2650 (2012, 2/2.8 GHz, 20 MB L2 cache, 64 GB DDR3 main memory) was used. The options to compile the analysis tools as well as the ZDD libraries used in the measurements are vastly different. This likewise applies to the analysis input, including the binary programs (Patmos ISA vs. ARM) and options used to compile the benchmarks. Despite these differences, the numbers should be comparable, since both approaches were applied to binary programs derived from the same source code, which in practice yields comparable ICFGs. This is notably confirmed by the number of memory blocks of the considered benchmarks (see Figure 5). In both cases the analyses do not take context-sensitivity into consideration. Still, the numbers are ballpark figures, where only the *orders of magnitudes* should be taken into consideration.

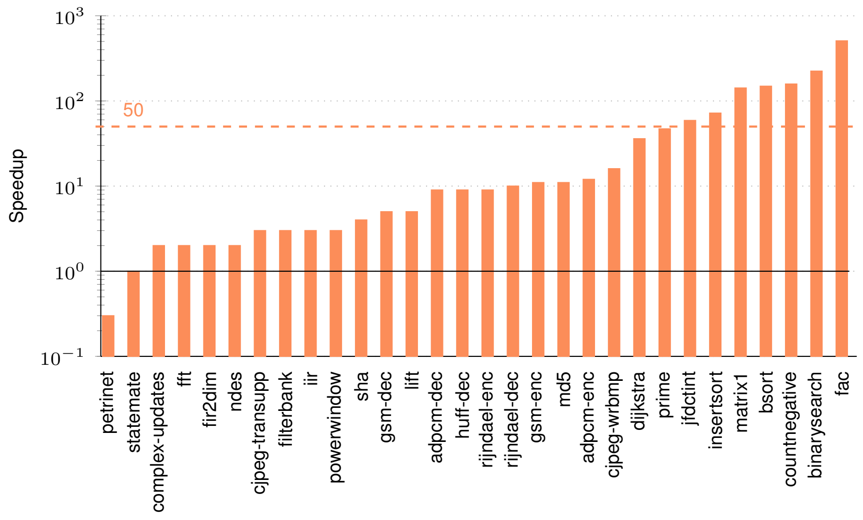


Fig. 13 Speedup of the Baseline analysis compared to the work of Touzeau et al. (2019) considering a 4 KB stabdard cache with an associativity of 8. (log-scale, higher is better).

We compared equivalent cache configurations, assuming a standard instruction cache with a total size of 4 KB, 16 B cache blocks, and an associativity of 4, 8, and 16 respectively. All non-recursive benchmark programs of the TACLe suite were considered, except `cover`, `duff`, and `test3`, which have not been considered by Touzeau et alii. The Baseline analysis seems to outperform the state-of-the-art in most cases. This is particularly true for small associativity numbers. For instance, the Baseline analysis for an associativity of 4 terminates instantly ($< 1\,\text{ms}$) for 25 out of the 30 benchmarks, while Touzeau et al.'s analysis requires up to about 1 second for these benchmarks. For the remaining benchmarks Baseline appears to be faster by a factor of 100 on average. For higher levels of associativity the analysis speedup goes down to a factor of 40 and 20 respectively. This change is partially explained by the fact that the number of benchmarks where the Baseline analysis terminates instantly drops from 25 to 17 and finally 10. Figure 13 shows the speedup for all considered benchmarks for the cache configuration with an associativity of 8. Only the `petrinet` benchmark showed a considerable slowdown (by a factor of $3.3\times$). It is difficult to draw definitive conclusions, however, it appears that this can be explained by the particular characteristics of the `petrinet` benchmark. This benchmark is dominated by a single large function that contains a large number of branches in the ICFG. This leads to a large number of possible combinations of conflict sets, many of which are subsumed by the minimum/maximum sets considered by Touzeau et alii. This is confirmed by the fact that the three worst performing benchmarks (`petrinet`, `statemate`, and `complex-updates`) have more than 1.17 ICFG edges per ICFG node, which indicates a high number of branches. The benchmarks performing best all have a ration below 1.03, which indicates a rather simple code structure. Note, however, that other characteristics, such as loops and nested function calls, may also increase the state space of the cache analysis without necessarily increasing the ratio between edges and nodes in the ICFG.

Despite the differences in terms of the experimental setup, this coarse comparison indicates that even the naive Baseline analysis is competitive against the state-of-the-art.

8.3 Predictability Considerations

The method cache was designed for the Patmos processor, which aims for predictability and analyzability. However, only the average performance was compared with mainstream architectures (Schoeberl et al., 2018), such as LEON3,[1] found in industrial real-time systems. The results here allow us to shed some light on this matter in terms of analyzability, i.e., which cache is simpler to analyze?

Cache configurations are not directly comparable. The method cache operates on fewer, but larger, memory blocks, which promises to reduce the analysis' state space. Its space utilization is usually limited by its associativity, i.e., small associativity combined with small memory blocks may cause evictions (conflict misses) despite the fact that only a fraction of the cache memory is used. Standard caches, on the other hand, operate on disjoint cache sets, which allows to decompose the cache's state. Cache utilization here depends on the distribution of memory blocks

[1] `https://www.gaisler.com/index.php/products/processors/leon3`

over cache sets, i.e., evictions may occur in one cache set (conflict misses), while other sets are not yet full. When comparing the maximum cache utilization across cache configurations one can observe that the 4-way set-associative standard caches have a slightly lower cache utilization than method caches with 16 sets. We thus compare these two configurations with a cache size of 4 KB.

The average (maximum) analysis time for the method cache amounts to 1.3 s (13.2 s), while for the standard cache the average (max.) analysis time amounts to 107 ms (3.5 s). Similarly, the average (max.) memory consumption amounts to 3.3 MB (28.9 MB) and 1 MB (19.2 MB) for the method and standard cache respectively. This indicates a slight advantage for the standard caches in terms of analysis complexity. However, due to its simpler design (full associativity) the method cache's behavior appears easier to predict, e.g., during the development of real-time software. This, for instance, allows to analyze the method cache accesses

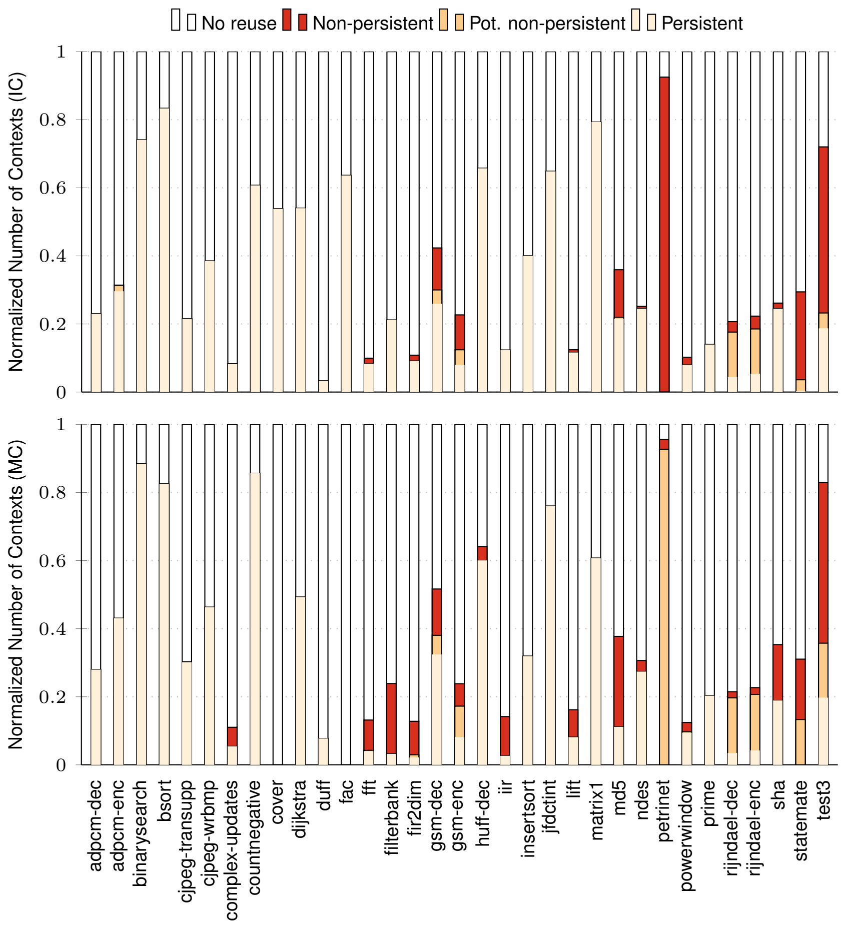


Fig. 14 Normalized total calling contexts with *not reused* (top), *NC-persistent*, *non-persistent*, and *persistent* (bottom) memory blocks (cache configuration IC:$\langle 4, 4\,\text{KB}\rangle$, MC:$\langle 16, 4\,\text{KB}\rangle$).

without knowing the precise address of the code. The mapping of memory blocks to cache sets of standard caches is more difficult to predict/control, as proven by the unsteady plots in Figure 6, since conflicts depend on the cache set, which, in turn, is derived from the memory block's address.

Another factor of analyzability is analysis precision, which in our case is best evaluated through persistence. Figure 14 summarizes the fully-context-sensitive persistence information over all memory blocks and benchmarks for both kinds of caches (IC top, MC bottom) according to the classification from Section 6.3 with regard of the scope of the *called* function. The results are normalized to the number of calling contexts and the size of the respective memory blocks. Overall the results follow very similar trends: a considerable portion of the memory blocks are not reused, while many blocks are persistent and only a small fraction is generally marked non-persistent. The method cache achieves better results for 9 out of 33 benchmarks, while the standard cache shows better results for 10 benchmarks. Major gains for the standard cache, e.g., for `complex-updates`, `filterbank`, `iir`, `lift`, `md5`, and `sha` are, to a large part, due to the compiler forming too large memory blocks, e.g., when an entire loop as well as code before/after that loop are placed inside a single memory block. This strategy is successful in terms of average-case performance, but appears to inflate the size of non-persistent regions of the ICFG. The gains for the method cache, on the other hand, (`gsm-enc`, `petrinet`, `rijndael-dec`, `rijndael-enc`, `statemate`, `test3`) can be explained by a better cache utilization, i.e, the cache sets of the standard cache are not ideally utilized. Note that the memory block formation by the compiler also explains the different height of the cumulative bars, i.e., code that normally is not reused is sometimes placed in a memory block with code that is reused. The inverse might also appear, as illustrated by `fac`, the compiler placed the benchmark's loop into a single memory block: the block is loaded once and then remains in the cache (i.e., the loop consists entirely of `FLOW` edges and will never cause a cache miss).

The comparison between the two cache kinds is rather mixed. The method cache does not significantly reduce complexity nor does it yield vastly superior precision. However, as with average performance (Schoeberl et al., 2018), it is able to compete with standard caches and still remains an interesting alternative to study, due to its simple design.

9 Related Work

A classical approach to cache analysis using abstract interpretation goes back to Alt et al. (1996). They proposed to classify memory accesses as `AH`, `AM`, or `NC`, based on an abstract domain that associates minimum/maximum age bounds with each memory block. The approach has proven quite successful for conventional caches. However, it is an ill fit for the method cache, due to the fact that the cache is fully associative. This is problematic for loops, where the age of all memory blocks steadily increases until it reaches the largest age of any memory block in the cache *before* the loop. This often means that all memory blocks – including those within, but also those outside of the loop – are essentially flushed from the cache in terms of the analysis. Later work added support for persistence (Ferdinand and Wilhelm, 1999; Ballabriga and Casse, 2008) that was proven incorrect. Corrections were proposed by independent teams (Huynh et al., 2011; Cullmann, 2013).

Recent work proposed exact analyses (Touzeau et al., 2019) to compute the minimum/maximum age of memory blocks. The age is represented indirectly through minimum/maximum conflict sets, which are computed similar to the baseline analysis (see Section 2). The work here relies on a single analysis that computes all conflict sets. The overhead induced by retaining all conflict sets is compensated by decomposing the analysis problem into smaller problems using inner and outer cache summaries. Note, however, that we could also define minimum/maximum summaries similar to their work. This would be compatible with the method cache presented here, but not necessarily with variants of the method cache, currently under development, that exploit meta-information (mentioned in Subsection 2.2) in order to modify the replacement policy. The work was soon afterwards extended to handle persistence Stock et al. (2019). Similar to the analysis for the $\mathcal{A}$ summaries, the transfer function of the traditional hit/miss classification is essentially applied to sub-graphs representing scopes. The authors in addition propose an extension of the data structure in order to represent – in addition to the memory blocks explicitly stored in the conflict sets of a family – a number of *anonymous* memory blocks. The analyses presented here should be equivalent to the approach proposed by Stock et al. (2019) in terms of analysis precision.

The notion of conflict sets was introduced by Mueller (2000) and later applied in various contexts (Huynh et al., 2011; Cullmann, 2013; Huber et al., 2014). A common limitation of these approaches is that a single conflict set over-approximates all possible cache states, which can quickly become pessimistic for large functions with disjoint control-flow paths. The approach of Huber et al. (2014) can be applied to caches with other cache replacement policies than LRU, notably FIFO. The traversal of the scope graph in their work is similar to the way summaries are computed here.

Compositional analysis techniques have been developed based on age- (Rakib et al., 2004; Ballabriga et al., 2008) and conflict-set-based (Patil et al., 2004) approaches. The aim here is to decompose the analysis of real-time programs at the level of object files, assuming incomplete information on the final program and its code layout (addresses). To achieve this, the various approaches define some form of *damage* function, which over-approximates the impact of calling a function (potentially from another object file). Ballabriga et al. (2008) proposed to split this damage function into two components – corresponding to the $\mathcal{A}$ and $\mathcal{C}$ summaries in this work. None of the past approaches defines a concept comparable to the *inner cache summaries* ($\mathcal{B}$). Also note that the method cache design favors compositionality: address and layout information is not needed, due to the fact that it is fully associative, i.e., the analysis can be symbolic.

Chu et al. (2016) applied symbolic execution in combination with SMT solving to precisely model cache states. The approach not only covers abstract cache states, but also takes infeasible paths into account. However, this comes at a price: high analysis time and memory consumption. The authors thus explore, similar to this work, the use of *summaries* that combine the age-based abstraction (Alt et al., 1996) with conditions (constraints), capturing the execution conditions under which the abstract cache states apply. The approach is evaluated using a standard 4 KB 4-way set-associative cache. Even for this small cache configuration the analysis times go up to 709 s, with a memory usage in the order of gigabytes. The analysis presented here appears to scale much better, even for cache configurations that are considerably larger.

Other approaches focused on refining the results of a fast, but imprecise, classical analysis – focusing on accesses classified as `NC`. One option is to explicitly keep track of paths where cache misses occur (Nagar and Srikant, 2017) and bound the number of misses by the number of executions on those paths. Another approach is to refine the `NC` classification by proving the existence of at least one path where a cache hit and another path where a cache miss occurs. Touzeau et al. (2017) propose an analysis based on abstract interpretation and a precise analysis based on model checking to accomplish this (Touzeau et al., 2017). Chattopadhyay and Roychoudhury (2011) similarly propose to use model checking.

10 Conclusion and Future Work

This work presented a novel technique to compute cache summaries based on the notion of conflict sets. These summaries can be computed for sub-graphs (e.g., functions) of an inter-procedural control-flow graph. The analysis allows to compute precise conflict sets by reusing summaries of nested sub-graphs that can be used to derive fully-call-context sensitive classical cache hit/miss classification and persistence information. The experiments indicate that the approach scales reasonably for realistic cache configurations.

Large cache sizes still cause considerable analysis time overhead. However, the experiments revealed several ways for improvements: the use of a fast pre-analysis to classifying simple cases, the use of minimum/maximum conflict sets, analysis-specific optimizations to the ZDD library, and the pruning of call contexts where memory blocks are not live.

Open research questions concern the composition of cache summaries for loops from their loop bodies and programs with recursion. For the former it appears feasible to define summaries of the loop body, treating back edges as special forms of entry and exit edges. This would allow us to precisely model the cache state across a loop's iteration space – similar to Huynh et al. (2011). The latter can be resolved by defining large sub-graphs covering cyclic regions of the call graph. However, inspired from the handling of loops, it might also be possible to define summaries for functions within these cycles.

Other possible applications of cache summaries could be in the context of compositional timing analysis, e.g., of third-party or obfuscated code, and (in combination) with parametric timing analysis. In both cases the ability to quickly derive precise states from a given cache state promises improved analysis performance and precision. In a similar vain, this may allow to improve the analysis precision when the set of initial cache states of a task are known (or can be characterized to some extent). In this case the cache analysis could abandon the conservative hypothesis to start from an empty cache state – which is known to trigger the worst-case scenario for caches with an LRU replacement policy Reineke and Grund (2013). This may even allow to derive cache states across successive job instances of a real-time task, similar to the concept of persistent cache blocks Rashid et al. (2016).

Acknowledgment

The author would like to thank Thomas Robert for sharing his understanding on BDDs/ZDDs and the associated libraries as well as Mihail Asavoae for the insight-full discussions leading up to this work. The author would like to thank in particular Amine Naji for his contributions to the Odyssey WCET analysis framework.

References

Aho AV, Lam MS, Sethi R, Ullman JD (2006) Compilers: Principles, Techniques, and Tools, 2nd edn. Addison-Wesley

Alt M, Ferdinand C, Martin F, Wilhelm R (1996) Cache behavior prediction by abstract interpretation. In: Proc. of the International Symposium on Static Analysis, Springer, SAS '96, pp 52–66

Ballabriga C, Casse H (2008) Improving the first-miss computation in set-associative instruction caches. In: Proc. of the Euromicro Conference on Real-Time Systems, IEEE, ECRTS '08, pp 341–350, DOI 10.1109/ECRTS.2008.34

Ballabriga C, Casse H, Sainrat P (2008) An improved approach for set-associative instruction cache partial analysis. In: Proc. of the Symposium on Applied Computing, ACM, SAC '08, pp 360–367, DOI 10.1145/1363686.1363778

Brandner F, Noûs C (2020) Precise and efficient analysis of context-sensitive cache conflict sets. In: Proc. of the International Conference on Real-Time Networks and Systems, ACM, RTNS '20, p 44–55, DOI 10.1145/3394810.3394811

Chattopadhyay S, Roychoudhury A (2011) Scalable and precise refinement of cache timing analysis via model checking. In: Proc. of the Real-Time Systems Symposium, IEEE, RTSS '11, pp 193–203, DOI 10.1109/RTSS.2011.25

Chu D, Jaffar J, Maghareh R (2016) Precise cache timing analysis via symbolic execution. In: Proc. of the Real-Time and Embedded Technology and Applications Symposium, RTAS '16, pp 1–12, DOI 10.1109/RTAS.2016.7461358

Cousot P, Cousot R (1977) Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of the Symposium on Principles of Programming Languages, ACM, POPL '77, pp 238–252, DOI 10.1145/512950.512973

Cullmann C (2013) Cache persistence analysis: Theory and practice. ACM Trans Embed Comput Syst 12(1s):40:1–40:25, DOI 10.1145/2435227.2435236

Degasperi P, Hepp S, Puffitsch W, Schoeberl M (2014) A method cache for Patmos. In: Proc. of the International Symposium on Object/Component-Oriented Real-Time Distributed Computing, IEEE, ISORC '14, pp 100–108, DOI 10.1109/ISORC.2014.47

Dvorak DL (2009) NASA study on flight software complexity. Technical excellence initiative, NASA Office of Chief Engineer

Falk H, Altmeyer S, Hellinckx P, Lisper B, Puffitsch W, Rochange C, Schoeberl M, Sørensen RB, Wägemann P, Wegener S (2016) TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In: Proc. of the Int. Workshop on Worst-Case Execution Time Analysis, Schloss Dagstuhl, OASIcs, vol 55, pp 1–10

Ferdinand C, Wilhelm R (1999) Efficient and precise cache behavior prediction for real-time systems. Real-Time Syst 17(2-3):131–181, DOI 10.1023/A:1008186323068

Hahn S, Reineke J (2018) Design and analysis of SIC: A provably timing-predictable pipelined processor core. In: Proc. of Real-Time Systems Symposium, RTSS '18, pp 469–481, DOI 10.1109/RTSS.2018.00060

Hepp S, Brandner F (2014) Splitting functions into single-entry regions. In: Proc. of the Int. Conference on Compilers, Architecture and Synthesis for Embedded Systems, ACM, CASES '14, pp 17:1–17:10, DOI 10.1145/2656106.2656128

Huber B, Hepp S, Schoeberl M (2014) Scope-based method cache analysis. In: Int. Workshop on Worst-Case Execution Time Analysis, Schloss Dagstuhl, OASIcs, vol 39, pp 73–82

Huynh BK, Ju L, Roychoudhury A (2011) Scope-aware data cache analysis for WCET estimation. In: Proc. of the Real-Time and Embedded Technology and Applications Symposium, IEEE, RTAS '11, pp 203–212, DOI 10.1109/RTAS.2011.27

Jordan A, Brandner F, Schoeberl M (2013) Static analysis of worst-case stack cache behavior. In: Proc. of the Conf. on Real-Time Networks and Systems, ACM, RTNS '13, pp 55–64

Kadota H, Miyake J, Okabayashi I, Maeda T, Okamoto T, Nakajima M, Kagawa K (1987) A 32-bit cmos microprocessor with on-chip cache and tlb. IEEE Journal of Solid-State Circuits 22(5):800–807

Khedker U, Sanyal A, Karkare B (2009) Data Flow Analysis: Theory and Practice, 1st edn. CRC Press

Li YTS, Malik S (1995) Performance analysis of embedded software using implicit path enumeration. In: Proc. of the Design Automation Conference, ACM, DAC '95, pp 456–461, DOI 10.1145/217474.217570

Lv M, Guan N, Reineke J, Wilhelm R, Yi W (2016) A survey on static cache analysis for real-time systems. Leibniz Transactions on Embedded Systems 3(1):05–1–05:48, DOI 10.4230/LITES-v003-i001-a005

Minato Si (1993) Zero-suppressed BDDs for set manipulation in combinatorial problems. In: Proc. of the International Design Automation Conference, ACM, DAC '93, pp 272–277, DOI 10.1145/157485.164890

Mishchenko A (2001) An introduction to zero-suppressed binary decision diagrams. Tech. rep., University of California, Berkeley

Mueller F (1994) Static cache simulation and its applications. PhD thesis, Florida State University

Mueller F (2000) Timing analysis for instruction caches. Real-Time Syst 18(2/3):217–247, DOI 10.1023/A:1008145215849

Nagar K, Srikant YN (2017) Refining cache behavior prediction using cache miss paths. ACM Trans Embed Comput Syst 16(4):103:1–103:26, DOI 10.1145/3035541

Naji A, Brandner F (2015) A comparative study of the precision of stack cache occupancy analyses. In: Proc. of the Junior Researcher Workshop on Real-Time Computing, JRWRTC '15, pp 13–16

Patil K, Seth K, Mueller F (2004) Compositional static instruction cache simulation. In: Proc. of the Conference on Languages, Compilers, and Tools for Embedded Systems, ACM, LCTES '04, pp 136–145, DOI 10.1145/997163.997183

Puschner PP, Schedl AV (1997) Computing maximum task execution times - a graph-based approach. Real-Time Systems 13(1):67–91, DOI 10.1023/A:1007905003094

Rakib A, Parshin O, Thesing S, Wilhelm R (2004) Component-wise instruction-cache behavior prediction. In: Proc. of Automated Technology for Verification and Analysis, Springer, ATVA '04, pp 211–229

Rashid SA, Nelissen G, Hardy D, Akesson B, Puaut I, Tovar E (2016) Cache-persistence-aware response-time analysis for fixed-priority preemptive systems. In: Proc. of the Euromicro Conference on Real-Time Systems, ECRTS'16, pp 262–272, DOI 10.1109/ECRTS.2016.25

Reineke J, Grund D (2013) Sensitivity of cache replacement policies. ACM Trans Embed Comput Syst 12(1s), DOI 10.1145/2435227.2435238

Schoeberl M, Schleuniger P, Puffitsch W, Brandner F, Probst C, Karlsson S, Thorn T (2011) Towards a time-predictable dual-issue microprocessor: The patmos approach. In: Proc. of Bringing Theory to Practice: Predictability and Performance in Embedded Systems, OASICS, vol 18, pp 11–21

Schoeberl M, Brandner F, Hepp S, Puffitsch W, D P (2013) Patmos Reference Handbook. Technical University of Denmark, URL `http://patmos.compute.dtu.dk/patmos_handbook.pdf`

Schoeberl M, Puffitsch W, Hepp S, Huber B, Prokesch D (2018) Patmos: A time-predictable microprocessor. Real-Time Syst 54(2):389–423, DOI 10.1007/s11241-018-9300-4

Smith AJ (1982) Cache memories. ACM Comput Surv 14(3):473—-530, DOI 10.1145/356887.356892

Stein IJ (2010) ILP-based path analysis on abstract pipeline state graphs. PhD thesis, Universität des Saarlandes

Stock G, Hahn S, Reineke J (2019) Cache persistence analysis: Finally exact. In: Proc. of the Real-Time Systems Symposium, RTSS '19, pp 481–494

Theiling H, Ferdinand C (1998) Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In: Proc. of the Real-Time Systems Symposium, IEEE, RTSS '98, pp 144–153

Touzeau V, Maïza C, Monniaux D, Reineke J (2017) Ascertaining uncertainty for efficient exact cache analysis. In: Computer Aided Verification, Springer, CAV '17, pp 22–40

Touzeau V, Maïza C, Monniaux D, Reineke J (2019) Fast and exact analysis for LRU caches. Proc ACM Program Lang 3(POPL):54:1–54:29, DOI 10.1145/3290367