Precise and Efficient Analysis of Context-Sensitive Cache Conflict Sets

Florian Brandner
florian.brandner@telecom-paris.fr
LTCI, Télécom Paris, Institut Polytechnique de Paris

Camille Noûs
camille.nous@cogitamus.fr
Laboratoire Cogitamus

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.

CCS CONCEPTS

• **Computer systems organization → Real-time systems**; • **Software and its engineering → Automated static analysis**; *Formal software verification.*

KEYWORDS

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

ACM Reference Format:
Florian Brandner and Camille Noûs. 2020. Precise and Efficient Analysis of Context-Sensitive Cache Conflict Sets. In *28th International Conference on Real-Time Networks and Systems (RTNS 2020), June 9–10, 2020, Paris, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3394810.3394811

1 INTRODUCTION

The computation of tight *Worst-Case Execution Time* (WCET) bounds is challenging due to the increasing size of real-time software [10] 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 [34] 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) [19, 28] 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 [33]. 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 from the set.

Traditionally, memory accesses of a program (e.g., load, store, instr. fetch) are classified [2, 23] 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) [12, 23].

The classifications for the memory accesses of a program can be derived from *conflict sets* [8, 15, 24]. A conflict set is usually defined with regard to a *memory block m* [34], 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 along with 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 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 [35, 36]. 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. [36] 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. We also rely on *Zero-Suppressed Decision Diagrams* (ZDDs) [21] in

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
RTNS 2020, June 9–10, 2020, Paris, France
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7593-1/20/06…$15.00
https://doi.org/10.1145/3394810.3394811

order to efficiently represent families of conflict sets. However, instead of computing lower/upper bounds on conflict sets, we retain *all* possible conflict sets. This inevitably leads to a larger state space and potentially longer analysis times. The main contribution of this work is to reduce the analysis overhead, and to a minor extent also to improve analysis precision, by introducing *cache summaries*.

Cache summaries represent the impact on the conflict sets when a given portion of a program is executed, i.e., typically a sub-graph of the program's control-flow graph such as a function or loop. We distinguish two kinds of summaries: *outer cache summaries* allow to efficiently obtain the conflict sets at the exit points of sub-graphs, while *inner cache summaries* allow to efficiently obtain the conflict sets right before a memory access within the sub-graph. This improves analysis time by up to a factor of 200, since large parts of programs that only produce intermediate conflict sets that are irrelevant for the final cache hit/miss classification are *skipped*.

The paper is organized as follows. We first provide some essential background on the method cache of the Patmos architecture, inter-procedural control-flow graphs, and cache analysis using conflict sets 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 provide a detailed description of the proposed analysis based on outer and inner cache summaries. The approach is evaluated in Section 7 using the TACLe benchmark suite [11]. Finally, related work is discussed in Section 8 before concluding in Section 9.

2 BACKGROUND

This section introduces a precise analysis over families of conflict sets, similar to Touzeau et al. [36]. The analysis relies on a single analysis pass and is extended to support the method cache [9] of the *time-predictable* Patmos processor [32]. We refer interested readers to the review of Lv et al. [20] for an introduction to cache analysis.

2.1 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 [14] and may exhibit variable sizes. The size of a cache block is pre-pended to the block's code, along with complementary meta-information.

Like traditional associative caches the method cache [9] 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 cache memory) have to be considered. These limits are specified by cache configurations:

Definition 1. 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 [9, 32] – which eliminates timing anomalies known from traditional instruction caches [13]. 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.

2.2 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) [17, 26]:

Definition 2. An **Inter-Procedural Control-Flow Graph** is a graph $G = (V, E, MB)$ 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 $mb\colon V \longrightarrow MB$, while edges may represent different kinds of control flow via the function $kind\colon E \longrightarrow \{\texttt{FLOW}, \texttt{FILL}, \texttt{CALL}, \texttt{RET}, \texttt{LINK}\}$.

For the purpose of this work the code inside the CFG nodes is actually not relevant, only the memory block of a CFG node is considered. Apart from `LINK` edges, the various edge kinds actually correspond to different classes of Patmos' control-flow instructions [30, 32]. 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.

2.3 Analysis via Families of Conflict Sets

Based on these definitions we can formalize the analysis using abstract interpretation [7]. 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. [18], which gives an excellent introduction.

Abstract Domain. Before defining the abstract domain itself, we first provide some definitions. As usual in cache analysis, *cache blocks* have to be tracked even when they are not in the cache. We thus introduce the notion of memory blocks, i.e., "cache blocks" in the cache and/or memory:

Definition 3. 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 $size\colon MB \rightarrow \mathbb{N}^+$.

One can then define a test to check whether a conflict set fits into a cache according to its cache configuration:

Definition 4. A conflict set $C \subseteq MB$ **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:

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

We define an 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 5. The **abstract domain** of the static analysis is given by $\mathcal{D} = \mathcal{P}(\{\aleph\} \cup \mathcal{P}(MB))$. The special symbols $\perp = \emptyset \in \mathcal{D}$ indicates the absence of analysis information, while $\aleph$ indicates the presence of conflict sets that do not fit into the cache (c.f. Definition 4).

Definition 6. 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 [22], 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 from above. 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 7. The **dot product with cardinality and size constraints** for a given cache configuration $\langle a, s\rangle$ is given by:

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

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 8. The **transfer function** takes two arguments, a CFG 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 9. The **meet operator** takes two (or more) families of conflict sets $\mathbb{A}$ and $\mathbb{B}$ from disjoint sets of sub-paths as input and produces their union:

$$M(\mathbb{A}, \mathbb{B}) = \{S \mid S \in \mathbb{A} \vee S \in \mathbb{B}\}.$$

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 6 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 [18], 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. [35, 36] for additional discussion.

Example 1. Assume memory block m_1 is analyzed for a 4-way set-associative cache configuration $\langle 4, 4\rangle$ and an initial family $\mathbb{I} = \{\{m_1, m_2, m_3, m_4\}, \{m_1, m_2, m_4, m_5\}\}$. $\mathbb{I}$ at this point contains two conflict sets that both fit into the cache, which represents an *always hit* classification (AH). Applying the transfer function $T_{m_1}^{\langle 4,4\rangle}$ on $\mathbb{I}$ for CFG nodes n_3 and n_6, accessing memory blocks m_3 and m_6 respectively, yields: $T_{m_1}^{\langle 4,4\rangle}(\mathbb{I}, n_3) = \{\{m_1, m_2, \mathbf{m_3}, m_4\}, \aleph\}$ and $T_{m_1}^{\langle 4,4\rangle}(\mathbb{I}, n_6) = \{\aleph\}$. The results thus represent a *not classified* (NC) and an *always miss* (AM) classification.

3 MOTIVATING EXAMPLE

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

Example 2. Figure 1 shows the ICFG of a program's `main` function, calling another function `F` several times in a `switch` statement. The control flow internal to the called function is not shown due to space considerations. However, the program's memory blocks, CFG nodes ($\mathtt{n_i}$) and edges for the `main` function are depicted. We

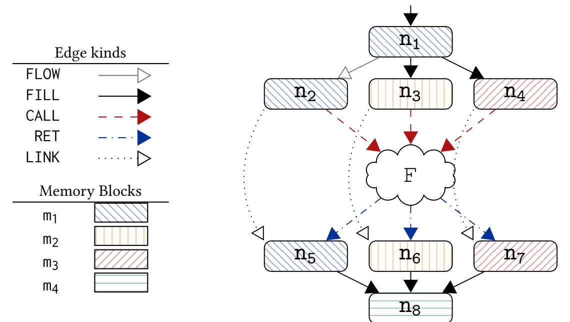


Figure 1: ICFG of a program (see Example 2).

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.

Let's assume that the called function contains highly complex control flow, (conditionally) accessing many different memory blocks. However, F does not access any of the memory blocks of main. The cache states within function F are thus irrelevant for the hit/miss classification of main's memory blocks.

Assume, for instance that memory block m_1, accessed by CFG 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 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 path in the middle ($n_3, \ldots, n_6$) and on the right ($n_4, \ldots, n_7$) 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 CFG nodes n_5, n_6, and n_7.

Function F is consequently analyzed 3 times in this example – despite the fact that none of the intermediate states are relevant for the hit/miss classification at n_5.

Another, minor, 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.

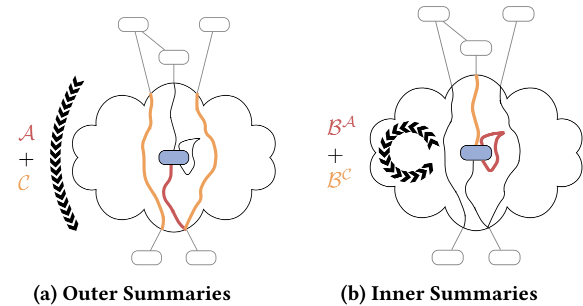
The previous example illustrates the high sensitivity of the precise conflict set analysis of Touzeau et al. [36] with regard to calling contexts: different cache states at different contexts may frequently trigger the computation of a large number of *irrelevant* cache states. The second issue, related 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 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



(a) Outer Summaries **(b) Inner Summaries**

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

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 classes: *outer* and *inner cache summaries*.

Sub-figure 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.

Sub-figure 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 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 [12, 23]. 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.

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 CFG 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, e.g., going through a sub-graph. Let's consider this in a small example:

Example 3. Assume that we have two sub-paths $p_1 = (n_1, n_2)$ and $p_2 = (n_3, n_4)$, where each n_i is associated with a matching memory block m_i, $1 \leq i \leq 4$, and that we wish to analyze the conflict set of m_1. The conflict set of these sub-paths are $\{m_1, m_2\}$ and $\{m_3, m_4\}$ respectively. Appending p_2 to p_1 gives a new sub-path (n_1, n_2, n_3, 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 (n_3, n_4, n_1, n_2) is simply $\{m_1, m_2\}$. The transfer function (Definition 8) resets the analysis information to $\{m_1\}$ at node n_1 and then adds 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 8). However, similar to traditional GEN/KILL data-flow problems [18], 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 C summaries capture paths through the sub-graph where the memory block is *not accessed.*

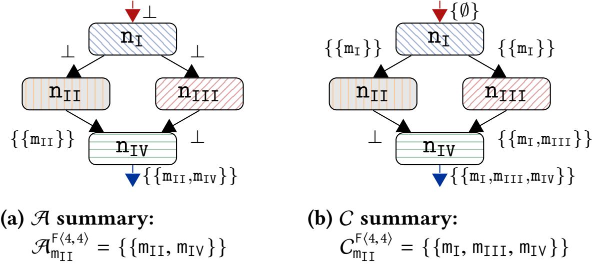
Definition 10. Given an ICFG $G = (V, E, MB)$ a **sub-ICFG** $G' = (V', E', MB)$ is a sub-graph, where $V' \subseteq V$ and $E' = \{(n, o) \in E \mid n \in V' \lor o \in V'\}$. The **entry edges** and **exit edges** of G' are edges that allow to enter/leave the sub-graph: $entry(G') = \{(n, o) \in E' \mid n \notin V' \land o \in V'\}$ and $exit(G') = \{(n, o) \in E' \mid n \in V' \land o \notin V'\}$.

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 C Summaries for Paths Without Accesses

The objective of 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 before, only the transfer function needs to be modified. A first insight is that the conflict sets for a C summary evolve quite similarly to the regular conflict sets, i.e., whenever a new CFG 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



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

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

Figure 3: Cache summaries for the memory block of $\mathtt{n_{II}}$ within a simple function F (see Examples 4 and 5).

filtered. Instead of producing a valid conflict set it suffices to simply produce an *invalid* ($\perp$) value in the **transfer function** for C summaries:

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

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 $\perp = \emptyset$). This means that conflict sets are initially empty when entering the sub-graph, then incrementally grow larger or are reset to $\perp$, 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 k-ary version of the meet operator: $C_m^{G'\langle a,s\rangle} = M(\mathbb{C}_1, \ldots, \mathbb{C}_k)$.

The 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 4. Consider the CFG of function F from Figure 3b, where each node $\mathtt{n_i}$ is associated with a memory block of unit size. The C summary $C^{\mathsf{F}\langle 4,4\rangle}_{\mathtt{m_{II}}}$ 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 $\perp$ on the edge $(\mathtt{n_{III}}, \mathtt{n_{IV}})$. The conflict set from this path is consequently *filtered* from the cache summary, resulting in a C summary $C^{\mathsf{F}\langle 4,4\rangle}_{\mathtt{m_{II}}} = \{\{\mathtt{m_I}, \mathtt{m_{III}}, \mathtt{m_{IV}}\}\}$.

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

$\mathcal{A}$ summaries are similar to 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 $\bot$. 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 $exit(G') = \{e_1, \dots, e_k\}$ and their respective analysis information $\mathbb{A}_i$, $1 \le i \le k$: $\mathcal{A}_m^{G'\langle a,s\rangle} = M(\mathbb{A}_1, \dots, \mathbb{A}_k)$.

Example 5. Consider once more the CFG of function `F` from Figure 3a, assuming the same setup as for Example 4. The analysis information at the entry is initialized to $\bot$. Adding new memory blocks consequently does not modify the conflict sets (cf. Definition 7). This only changes after reaching an access to the memory block under analysis at $\mathtt{n}_{\mathtt{II}}$, which first produces the conflict set $\{\{\mathtt{m}_{\mathtt{II}}\}\}$. Subsequent accesses to other memory blocks are then added to the conflict set, resulting in the $\mathcal{A}$ summary $\mathcal{A}_{\mathtt{m}_{\mathtt{II}}}^{\mathtt{F}\langle 4,4\rangle} = \{\{\mathtt{m}_{\mathtt{II}}, \mathtt{m}_{\mathtt{IV}}\}\}$.

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

The analyses from above 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 $n_{G''}$ and redirect the entry/exit edges as follows:

Definition 11. Given a (sub-)ICFG $G' = (V', E', MB)$ and a nested sub-ICFGs $G'' = (V'', E'', MB)$ the **collapsed sub-ICFG** $\overline{G'} = (\overline{V'}, \overline{E'}, MB)$ is defined by: $\overline{V'} = (V' \cup \{\overline{n}_{G''}\}) \setminus V''$ and $\overline{E'} = (E' \cup \{(o, \overline{n}_{G''}) \mid \exists (o,n) \in entry(G'')\} \cup \{(\overline{n}_{G''}, o) \mid \exists (n,o) \in exit(G'')\}) \setminus E''$.

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 from above 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 C summaries of $\overline{G'}$ then become:

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

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

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 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 C summaries of callees are applied, while ignoring `CALL`/`RET` edges.

It remains to exploit the 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 summaries do not provide this information. One solution would be to use the cache summaries only to improve the analysis precision 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 $\bot$, 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 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 ($\mathcal{B}^C$) 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 a CFG node that accesses the analyzed memory block, without any intermediate accesses to that block. These paths are readily covered by the analysis of the C summaries. The conflict set at the first access to a memory block can then be computed using the dot product (cf. Definition 7) between the conflict sets before entering the sub-graph and the conflict sets from the analysis of the 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 after the $\mathcal{A}$ and 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: $A = \{e_i \in E' \mid e_i = (u, v)\colon kind(e_i) \in \{\mathsf{FILL}, \mathsf{LINK}\} \wedge mb(v) = m\}$. One can either store the information individually for each edge or combine it as before: $\mathcal{B}_m^{\mathcal{A}(G')\langle a,s \rangle} = M(\mathbb{A}_1, \ldots, \mathbb{A}_{|A|})$ and $\mathcal{B}_m^{C(G')\langle a,s \rangle} = M(\mathbb{C}_1, \ldots, \mathbb{C}_{|A|})$.

Example 6. Consider the ICFG from Figure 3 with a cache configuration $\langle 4, 4 \rangle$. The $\mathcal{B}$ summaries for F are given by $\mathcal{B}_{\mathsf{m_{II}}}^{\mathcal{A}(\mathsf{F})\langle 4,4 \rangle} = \bot$ and $\mathcal{B}_{\mathsf{m_{II}}}^{C(\mathsf{F})\langle 4,4 \rangle} = \{\{\mathsf{m_I}\}\}$, cf. edge $(\mathsf{n_I}, \mathsf{n_{II}})$ in Subfigure 3a and 3b respectively. The former indicates that $\mathsf{m_{II}}$ is not accessed within F before reaching $\mathsf{n_{II}}$, while the latter indicates that $\mathsf{m_I}$ is always accessed before reaching $\mathsf{n_{II}}$.

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 as before. 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 C ($\mathbb{C}_j$) summaries at that edge:

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

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

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

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

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

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 to $\bot$, 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 7. Consider the ICFG of the main function from Figure 1, while assuming that F is called again at the confluence point $\mathsf{n_8}$ (the call is not shown in the figure). The setup is otherwise the same as for the previous examples.

We wish to compute persistence information for memory block $\mathsf{m_{II}}$, accessed within F, relative to main. For this the $\mathcal{B}$ summaries of function F are needed: $\mathcal{B}_{\mathsf{m_{II}}}^{\overline{\mathcal{A}(\mathsf{F})}\langle 4,4 \rangle} = \bot$ and $\mathcal{B}_{\mathsf{m_{II}}}^{\overline{C(\mathsf{F})}\langle 4,4 \rangle} = \{\{\mathsf{m_I}\}\}$ (cf. Example 6). These summaries are pre-computed and originate from the $\mathcal{A}/C$ analyses, depicted on edge $(\mathsf{n_I}, \mathsf{n_{II}})$ in Figure 3.

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

The $\mathcal{A}$ summary evaluates to $\bot$ for the LINK edges originating from $\mathsf{n_2}$, $\mathsf{n_3}$ and $\mathsf{n_4}$, due to the initialization to $\bot$ at main's entry and the fact that $\mathsf{m_{II}}$ is not accesses before any call to F. Combining this information with F's $\mathcal{B}^{\mathcal{A}}$ summary (also $\bot$), this indicates that $\mathsf{m_{II}}$ is not reused between the program start and the first return from F.

The situation changes for the additional call to F at $\mathsf{n_8}$. The $\mathcal{A}$ summary for main at this point is obtained from F's $\mathcal{A}$ summary ($\mathcal{A}_{\mathsf{m_{II}}}^{\mathsf{F}\langle 4,4 \rangle} = \{\{\mathsf{m_{II}}, \mathsf{m_{IV}}\}\}$, see Subfigure 3a) and by applying the transfer function (Equation 3) for CFG nodes $\mathsf{n_5}$, $\mathsf{n_6}$, and $\mathsf{n_7}$. This yields three conflict sets that are combined using the meet operator and updated using the transfer function for $\mathsf{n_8}$: $\{\{\mathsf{m_1}, \mathsf{m_4}, \mathsf{m_{II}}, \mathsf{m_{IV}}\}, \{\mathsf{m_2}, \mathsf{m_4}, \mathsf{m_{II}}, \mathsf{m_{IV}}\}, \{\mathsf{m_3}, \mathsf{m_4}, \mathsf{m_{II}}, \mathsf{m_{IV}}\}\}$. At this point $\mathsf{m_{II}}$ is still guaranteed to be in the cache – if loaded during the first call to F. However, when the $\mathcal{A}$ summary information of main is combined with $\mathcal{B}^C$ (cf. Equation 5), the conflict sets become too large (due to $\mathsf{m_I}$).

Combining the analysis information over all call sites to F yields $\mathcal{B}_{\mathsf{m_{II}}}^{\overline{\mathcal{A}(\mathsf{main})}\langle 4,4 \rangle} = \{\aleph\}$, 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. [15]. The analysis information is simply propagated upwards from the leaves of the call graph [1] to its root.

It remains to show how the hit/miss classification can be derived from the $\mathcal{B}^C$ 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 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 EXPERIMENTS

Our analyses were evaluated for the method cache and standard instruction caches using the TACLe suite [11], i.e., benchmarks commonly used to evaluate WCET analyzers. We used Patmos' LLVM compiler (version 5.0) with default optimizations (`-O2`). The minimal alignment of memory 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 [14]. The method cache is too organized in blocks, cached memory blocks thus occupy a multiple of 16 B. During the generation of the benchmark executables the compiler exports the call-context-insensitive ICFGs for the analysis.

Figure 4 shows the number of memory blocks 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 [36], 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 on the method cache used cache sizes between 1 KB and 16 KB, with 4 to 16 tag entries [14, 15, 31] (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 sets.

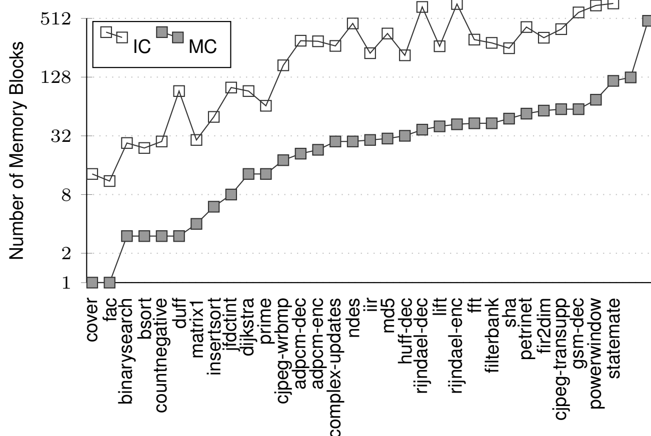


Figure 4: Number of memory blocks per program of non-recursive TACLe benchmarks (log-2-scale).

The analysis tool relies on *Zero-Suppressed Decision Diagrams* (ZDDs) [21] 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 at 3.16 GHz and 4 GB of main memory, running Linux (Kernel 4.12).

Analysis times were measured using the standard high-resolution clock (`chrono::high_resolution_clock`) from `C++` 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). Note, we never fall back to heuristics, i.e., the three analyses are applied to all memory blocks of a program as presented in the previous sections.

7.1 Analysis Complexity

Figure 5 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 and the number of memory blocks. 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. For the method cache the gains increase with the size and associativity. For the instruction cache the gains stay rather constant. Notably,

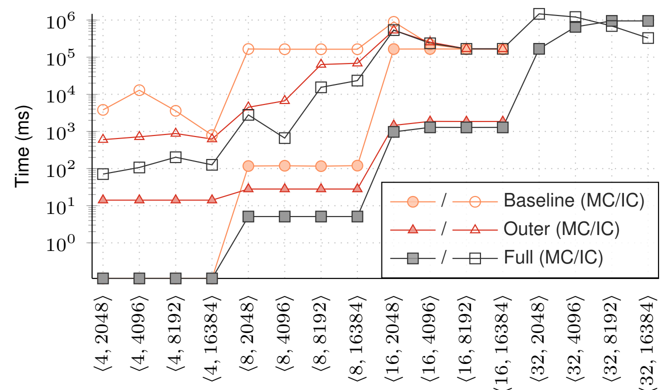


Figure 5: Total analysis time over all benchmarks for all considered cache configurations (log-scale, lower is better).

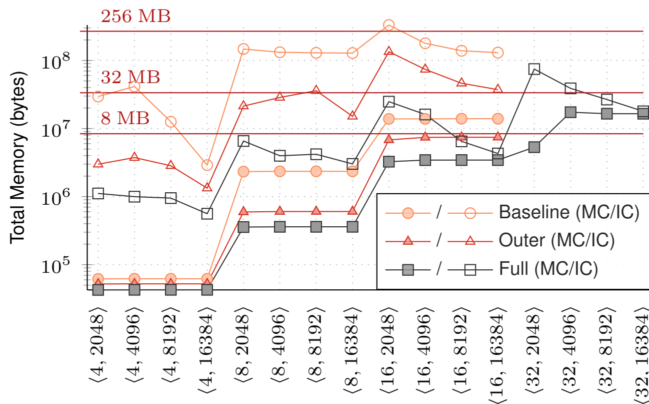


Figure 6: Total memory consumption over all benchmarks for all cache configurations (log-scale, lower is better).

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 to skip the analysis of large portions of the program that are not relevant to the cache hit/miss classification. Note furthermore that the Full analysis is considerably faster, despite the fact that it also computes fully-context sensitive persistence.

For the largest cache configurations with an associativity 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 and 5 timeouts with increasing associativity, while the analysis for the standard cache experiences between 6 and 2 timeouts with increasing size. These time outs are due to lacking optimizations in the ZDD library: the analysis spends most time in a look-up function, retrieving existing objects. Considerable improvements should be possible using analysis-specific caching.

The ZDD representation [21] of the analysis information is also highly efficient. The average memory consumption over all configurations peaks slightly above 256 megabytes (MB) – see Figure 6. 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. The summaries and optimizations proposed in this work thus successfully reduce analysis complexity by orders of magnitudes.

7.2 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. [36]. 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 briefly compare the analysis time of the Baseline analysis with the information in Touzeau et al.'s paper [36, Figure 10]. Note that this comparison should be taken with a grain of salt. The computer platforms (Intel Core2 Duo, 2006 vs. Intel Xeon, 2012), compiler options, and ZDD libraries used in the measurements are vastly different. This also applies to the analysis input, including the binary programs (Patmos ISA vs. ARM), benchmark compiler options, and considered call contexts. The subsequent numbers are thus ballpark figures, focusing on *orders of magnitudes*.

We compared equivalent cache configurations, assuming a standard instruction cache with a size of 4 KB and an associativity of 4, 8, and 16 respectively. All non-recursive benchmarks programs of the TACLe suite were considered, except `cover`, `duff`, and `test3`. The Baseline analysis seems to outperform Touzeau et al's analysis in most cases. This is particularly true for small associativity numbers. For instance, the Baseline analysis terminates instantly (0 ms) for 25 out of the 30 benchmarks, while Touzeau at 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. This coarse comparison indicates that even the naive Baseline analysis is competitive against the state-of-the-art.

7.3 Predictability Considerations

The method cache was designed for the Patmos processor, which aims for predictability and analyzability. However, only the average performance was compared [31] with mainstream architectures, 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 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

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

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. 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 5.

Another factor of analyzability is analysis precision, which in our case is best evaluated through persistence. Figure 7 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 [31], it is able to compete with standard caches and still remains an interesting alternative to study, due to its simple design.

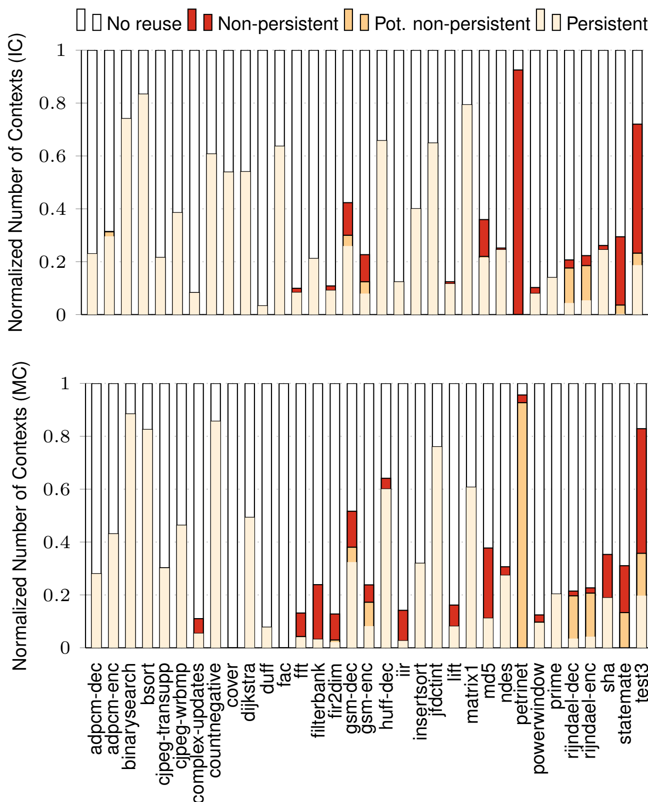


Figure 7: Normalized total calling contexts with *not reused* (top), *NC-persistent*, *non-persistent*, and *persistent* (bottom) memory blocks (cache configuration IC:⟨4, 4 KB⟩, MC:⟨16, 4 KB⟩).

8 RELATED WORK

A classical approach to cache analysis using abstract interpretation goes back to Ferdinand et al. [2]. 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 [3, 12] that was proven incorrect. Corrections were proposed later by independent teams [8, 16].

Recent work proposed exact analyses [36] 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 2.1) in order to modify the replacement policy.

The notion of conflict sets was introduced by Mueller [24] and later applied in various contexts [8, 15, 16]. 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. [15] 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- [4, 29] and conflict-set-based [27] 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. [4] proposed to split this damage function into two components – corresponding to the $\mathcal{A}$ and 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. [6] 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 [2] 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 [25] 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. [35] propose an analysis based on abstract interpretation and a precise analysis based on model checking to accomplish this [35]. Chattopadhyay et al. [5] similarly propose to use model checking.

9 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 [16]. 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.

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

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley.
[2] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. 1996. Cache Behavior Prediction by Abstract Interpretation. In *Proc. of the International Symposium on Static Analysis (SAS '96)*. Springer, 52–66.
[3] Clément Ballabriga and Hugues Casse. 2008. Improving the First-Miss Computation in Set-Associative Instruction Caches. In *Proc. of the Euromicro Conference on Real-Time Systems (ECRTS '08)*. IEEE, 341–350. https://doi.org/10.1109/ECRTS.2008.34
[4] C. Ballabriga, H. Casse, and P. Sainrat. 2008. An Improved Approach for Set-associative Instruction Cache Partial Analysis. In *Proc. of the Symposium on Applied Computing (SAC '08)*. ACM, 360–367. https://doi.org/10.1145/1363686.1363778
[5] Sudipta Chattopadhyay and Abhik Roychoudhury. 2011. Scalable and Precise Refinement of Cache Timing Analysis via Model Checking. In *Proc. of the Real-Time Systems Symposium (RTSS '11)*. IEEE, 193–203. https://doi.org/10.1109/RTSS.2011.25
[6] D. Chu, J. Jaffar, and R. Maghareh. 2016. Precise Cache Timing Analysis via Symbolic Execution. In *Proc. of the Real-Time and Embedded Technology and Applications Symposium (RTAS '16)*. 1–12. https://doi.org/10.1109/RTAS.2016.7461358
[7] Patrick Cousot and Radhia Cousot. 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 (POPL '77)*. ACM, 238–252. https://doi.org/10.1145/512950.512973
[8] Christoph Cullmann. 2013. Cache Persistence Analysis: Theory and Practice. *ACM Trans. Embed. Comput. Syst.* 12, 1s, Article 40 (March 2013), 25 pages. https://doi.org/10.1145/2435227.2435236
[9] Philipp Degasperi, Stefan Hepp, Wolfgang Puffitsch, and Martin Schoeberl. 2014. A Method Cache for Patmos. In *Proc. of the International Symposium on Object/Component-Oriented Real-Time Distributed Computing (ISORC '14)*. IEEE, 100–108. https://doi.org/10.1109/ISORC.2014.47
[10] D. L. Dvorak (editor). 2009. *NASA Study on Flight Software Complexity.* Technical Excellence Initiative. NASA Office of Chief Engineer.
[11] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. 2016. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *Proc. of the Int. Workshop on Worst-Case Execution Time Analysis (OASIcs)*, Vol. 55. Schloss Dagstuhl, 1–10.
[12] Christian Ferdinand and Reinhard Wilhelm. 1999. Efficient and Precise Cache Behavior Prediction for Real-Time Systems. *Real-Time Syst.* 17, 2-3 (Dec. 1999), 131–181. https://doi.org/10.1023/A:1008186323068
[13] Sebastian Hahn and Jan Reineke. 2018. Design and Analysis of SIC: A Provably Timing-Predictable Pipelined Processor Core. In *Proc. of Real-Time Systems Symposium (RTSS '18)*. 469–481. https://doi.org/10.1109/RTSS.2018.00060

[14] Stefan Hepp and Florian Brandner. 2014. Splitting Functions into Single-entry Regions. In *Proc. of the Int. Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '14)*. ACM, 17:1–17:10. https://doi.org/10.1145/2656106.2656128

[15] B. Huber, S. Hepp, and M. Schoeberl. 2014. Scope-Based Method Cache Analysis. In *Int. Workshop on Worst-Case Execution Time Analysis (OASIcs)*, Vol. 39. Schloss Dagstuhl, 73–82.

[16] Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. 2011. Scope-Aware Data Cache Analysis for WCET Estimation. In *Proc. of the Real-Time and Embedded Technology and Applications Symposium (RTAS '11)*. IEEE, 203–212. https://doi.org/10.1109/RTAS.2011.27

[17] A. Jordan, F. Brandner, and M. Schoeberl. 2013. Static Analysis of Worst-case Stack Cache Behavior. In *Proc. of the Conf. on Real-Time Networks and Systems (RTNS '13)*. ACM, 55–64.

[18] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. 2009. *Data Flow Analysis: Theory and Practice* (1st ed.). CRC Press.

[19] Yau-Tsun Steven Li and Sharad Malik. 1995. Performance Analysis of Embedded Software using Implicit Path Enumeration. In *Proc. of the Design Automation Conference (DAC '95)*. ACM, 456–461. https://doi.org/10.1145/217474.217570

[20] Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. 2016. A Survey on Static Cache Analysis for Real-Time Systems. *Leibniz Transactions on Embedded Systems* 3, 1 (2016), 05–1–05:48. https://doi.org/10.4230/LITES-v003-i001-a005

[21] Shin-ichi Minato. 1993. Zero-suppressed BDDs for Set Manipulation in Combinatorial Problems. In *Proc. of the International Design Automation Conference (DAC '93)*. ACM, 272–277. https://doi.org/10.1145/157485.164890

[22] Alan Mishchenko. 2001. *An introduction to zero-suppressed binary decision diagrams*. Technical Report. University of California, Berkeley.

[23] Frank Mueller. 1994. *Static Cache Simulation and its Applications*. Ph.D. Dissertation. Florida State University.

[24] Frank Mueller. 2000. Timing Analysis for Instruction Caches. *Real-Time Syst.* 18, 2/3 (May 2000), 217–247. https://doi.org/10.1023/A:1008145215849

[25] Kartik Nagar and Y. N. Srikant. 2017. Refining Cache Behavior Prediction Using Cache Miss Paths. *ACM Trans. Embed. Comput. Syst.* 16, 4, Article 103 (May 2017), 26 pages. https://doi.org/10.1145/3035541

[26] A. Naji and F. Brandner. 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)*. 13–16.

[27] Kaustubh Patil, Kiran Seth, and Frank Mueller. 2004. Compositional Static Instruction Cache Simulation. In *Proc. of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '04)*. ACM, 136–145. https://doi.org/10.1145/997163.997183

[28] Peter P. Puschner and Anton V. Schedl. 1997. Computing Maximum Task Execution Times - A Graph-Based Approach. *Real-Time Systems* 13, 1 (July 1997), 67–91. https://doi.org/10.1023/A:1007905003094

[29] Abdur Rakib, Oleg Parshin, Stephan Thesing, and Reinhard Wilhelm. 2004. Component-Wise Instruction-Cache Behavior Prediction. In *Proc. of Automated Technology for Verification and Analysis (ATVA '04)*. Springer, 211–229.

[30] M. Schoeberl, F. Brandner, S. Hepp, W. Puffitsch, and Prokesch D. 2013. *Patmos Reference Handbook*. Technical University of Denmark. http://patmos.compute.dtu.dk/patmos_handbook.pdf

[31] Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. 2018. Patmos: A Time-predictable Microprocessor. *Real-Time Syst.* 54, 2 (April 2018), 389–423. https://doi.org/10.1007/s11241-018-9300-4

[32] Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. 2011. Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach. In *Proc. of Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, Vol. 18. OASICS, 11–21.

[33] Ingmar Jendrik Stein. 2010. *ILP-based Path Analysis on Abstract Pipeline State Graphs*. Ph.D. Dissertation. Universität des Saarlandes.

[34] H. Theiling and C. Ferdinand. 1998. Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis. In *Proc. of the Real-Time Systems Symposium (RTSS '98)*. IEEE, 144–153.

[35] Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. 2017. Ascertaining Uncertainty for Efficient Exact Cache Analysis. In *Computer Aided Verification (CAV '17)*. Springer, 22–40.

[36] Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. 2019. Fast and Exact Analysis for LRU Caches. *Proc. ACM Program. Lang.* 3, POPL, Article 54 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290367