Cache Persistence Analysis: Finally Exact

Gregory Stock, Sebastian Hahn, and Jan Reineke
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
{g.stock, sebastian.hahn, reineke}@cs.uni-saarland.de

arXiv:1909.04374v1 [cs.PL] 10 Sep 2019

***Abstract*—Cache persistence analysis is an important part of worst-case execution time (WCET) analysis. It has been extensively studied in the past twenty years. Despite these efforts, all existing persistence analyses are approximative in the sense that they are not guaranteed to find all persistent memory blocks.**

In this paper, we close this gap by introducing the first exact persistence analysis for caches with least-recently-used (LRU) replacement. To this end, we first introduce an exact abstraction that exploits monotonicity properties of LRU to significantly reduce the information the analysis needs to maintain for exact persistence classifications. We show how to efficiently implement this abstraction using zero-suppressed binary decision diagrams (ZDDs) and introduce novel techniques to deal with uncertainty that arises during the analysis of data caches.

The experimental evaluation demonstrates that the new exact analysis is competitive with state-of-the-art inexact analyses in terms of both memory consumption and analysis run time, which is somewhat surprising as we show that persistence analysis is NP-complete. We also observe that while prior analyses are not exact in theory they come close to being exact in practice.

I. Introduction

Modern processors can perform several arithmetic and logic operations in a single cycle. On the other hand, a single access to main memory can take hundreds of cycles. To bridge this performance gap, modern processors include one or multiple levels of caches. Caches are small but fast memories that store parts of main memory to quickly serve accesses to commonly used instructions and data. Memory accesses that "hit" the cache are served from the cache at a low latency, while accesses that "miss" the cache are served from main memory at a much higher latency. The execution time of a program thus heavily depends on how effective the processor's caches are in hiding the high latency of main memory.

Real-time systems are systems that, in order to function correctly, have to perform their computations with limited amounts of wall-clock time. To verify a system's real-time behavior, a major task is to bound each software component's worst-case execution time (WCET). In the presence of caches, WCET analysis [1] has to account for the software's cache behavior. Simply assuming that each memory access could result in a cache miss would yield extremely pessimistic WCET bounds. Thus, static cache analyses [2] have been developed to soundly and precisely characterize a program's cache behavior on a particular cache architecture. These can broadly be categorized into two groups:

1) *Classifying cache analyses* aim to classify individual memory accesses as cache hits or cache misses.

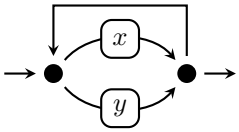


Fig. 1: Simple motivating example for persistence analysis.

2) *Quantitative cache analyses* aim to determine the number of cache misses resulting from a set of memory accesses.

In this paper we study *persistence analysis*, an instance of quantitative cache analysis. Persistence analysis considers all memory accesses in a program, or a fragment of a program such as a loop, that access the *same* memory block. A memory block is persistent if all memory accesses referring to this memory block may cumulatively result in at most one cache miss during any possible program execution.

For a motivating example, consider Figure 1, which contains the control-flow graph of a simple program. The program consists of a loop, in which, in each loop iteration either memory block x or memory block y is accessed. As neither block x nor block y is guaranteed to have been accessed in any loop iteration, it is impossible for a classifying cache analysis to classify any of the memory accesses in the program as a guaranteed cache hit, and so a WCET analysis would have to pessimistically account for misses upon all memory accesses. However, provided the cache is large enough to hold blocks x and y simultaneously, among all memory accesses to x (and similarly to y) only the very first may result in a cache miss. Both x and y are *persistent* and WCET analysis can safely account for at most two misses in total.

Given a program, the goal of persistence analysis is to determine which of the memory blocks accessed in the program are persistent. Persistence analysis has been extensively studied for caches with least-recently-used (LRU) replacement, starting with Mueller's [3]–[6] and Ferdinand's [7], [8] work in the 1990s up until today [9]–[17]. Notably, all prior persistence analyses are *approximative*, in the sense that they are not guaranteed to find all persistent memory blocks of a program.

In this paper, we close this gap by introducing the first *exact* persistence analysis. We develop this analysis via a sequence of three consecutive exact abstractions. The first abstraction is based on the observation that the persistence of a memory block can be determined by examining its possible *conflict sets*, i. e., the sets of blocks that may have been accessed since the last access to the block itself. The two following abstractions exploit a monotonicity property of LRU

replacement to further increase analysis efficiency without sacrificing exactness.

Next, we discuss how to efficiently implement the exact abstraction using zero-suppressed binary decision diagrams (ZDDs), a data structure that enables the compact representation of sets of conflict sets, sharing information across program points and between different memory blocks. We also introduce novel techniques to deal with uncertainty about the memory-access behavior that arises in data cache analysis.

We experimentally evaluate the new exact persistence analysis and a selection of prior persistence analyses and make the following high-level observations:

- Our exact analysis is competitive with prior analyses in terms of both memory consumption and analysis run time.
- While prior persistence analyses are not exact in theory, they come very close to being exact in practice.

Even though our exact persistence analysis is fairly efficient in practice, its worst-case complexity is exponential. We show that persistence analysis is NP-complete, which implies that a persistence analysis that is polynomial in all input parameters is not attainable, unless P=NP.

II. Background: Caches, Control-Flow Graphs, and Cache Persistence Analysis

A. Caches

Caches are fast but small memories that buffer parts of the large but slow main memory in order to bridge the speed gap between the processor and main memory. Caches operate at the granularity of memory blocks $b \in \mathcal{B}$, which are stored in the cache in *cache lines* of the same size. In order to facilitate the cache lookup, the cache is organized in *sets* such that each memory block maps to a unique cache set. The size k of a cache set is called the *associativity* of the cache. If an accessed block resides in the cache, the access *hits* the cache. Upon a cache *miss*, the block is loaded from main memory. To ease the formal presentation in this paper, we assume a fully-associative cache, i. e., with a single cache set. Set-associative caches with n sets can be treated as n independent fully-associative caches as described in [18].

Upon a cache miss, another memory block has to be evicted due to the limited size of the cache. The block to evict is determined by the *replacement policy*. In this paper, we assume the least-recently-used (LRU) policy that replaces the block that has not been accessed for the longest. A memory block b hits in an LRU cache of associativity k if b has been accessed before and less than k distinct blocks have been accessed since the last access to b. LRU is generally considered to be the most predictable replacement policy [19].

In this paper, we refer to the *age* of block b as the number of distinct blocks since the last access to b including the access to b itself[1]. Thus, a block b hits the cache if its age is less than or equal to the associativity k.

[1]This is subtly different from most of the related work in which the age of a block does not account for the access of the block itself.

B. Programs as Control-Flow Graphs

In this paper, we follow the common approach of representing the program under analysis by its control-flow graph (CFG). A CFG $\mathcal{G} = (V, E, i)$ consists of a set of vertices V, corresponding to control locations in the program; a set of edges $E \subseteq V \times \mathcal{B} \times V$, which represent the possible control flow between locations; and the initial control location $i \in V$. Each edge is annotated with a single memory block accessed between the source and target location.

The CFG is an abstraction of the program behavior as it does not capture the functional semantics of the instructions. In particular, all paths in the graph are assumed to be feasible even if, in reality, some are not, e. g., in case of a nested conditional statement with contradicting conditions. All our claims of exactness are relative to this control-flow graph abstraction. Incorporating the program semantics into the persistence analysis problem immediately renders it undecidable due to Rice's theorem. For example, it is undecidable whether a certain path in a program is feasible, and hence, it is also undecidable to collect all memory access sequences which are needed for exact persistence analysis.

To ease the visualization of a CFG, we allow empty edges on which no access is performed. For the sake of simplicity, we do not treat such empty edges in the formalization, but the extension would be trivial as such empty edges do not influence the cache state.

C. Notion of Persistence

A memory block is *persistent* during a program's execution if all accesses to the memory block collectively result in at most one cache miss. Assuming the cache is empty at the start of the program's execution, the first access to any memory block will always result in a cache miss. Thus, to be persistent, all accesses but the very first to a memory block must hit the cache.

In other words, a memory block is *persistent* during a program's execution if all accesses to the block hit the cache if the block has been accessed before. As an example, in Figure 1, as discussed before, blocks x and y are persistent in a cache of associativity two.

Due to its dependence on previous accesses, persistence is a property of execution traces. A *trace* $\tau = b_0 b_1 \ldots b_{n-1} \in \mathcal{B}^*$ is a sequence of memory blocks. We use τ_i to denote the i-th block b_i in the sequence and $|\tau| := n$ to denote its length. The trace of length zero is denoted by ϵ.

The *conflict set* of block b on trace τ is the set of all memory blocks accessed from the last access of block b onward:

$$CS(\tau, b) := \bigcap_{\substack{0 \le i < |\tau| \\ \tau_i = b}} \{\tau_j \mid j \ge i\}$$

If block b has not been accessed on trace τ, the empty intersection yields the universe $\mathcal{B}$.

The cardinality of the conflict set is referred to as the *age* of block b at the end of trace τ:

$$age(\tau, b) := |CS(\tau, b)|$$

In accordance with the discussion at the beginning of this subsection, we call block b persistent on trace τ if b is cached once it has been accessed before.

Definition II.1 (Persistence on Trace)**.** Memory block $b \in \mathcal{B}$ is *persistent on trace* $\tau \in \mathcal{B}^*$ if:

$$(\exists 0 \leq i < |\tau| : \tau_i = b) \rightarrow age(\tau, b) \leq k$$

The above definition captures persistence of a block on a single trace. In order to reason about the persistence of a memory block during a program's execution, we need to capture all possible traces of a given program. To this end, we define the *trace semantics*, which captures for each location in a control-flow graph all possible traces that end in this location.

Formally, the trace semantics is defined as the least solution R^C of the following set of equations where $R^C \colon V \rightarrow \mathcal{P}(\mathcal{B}^*)$ maps each location to a set of traces:

$$\forall w \in V : R^C(w) = \mathcal{I}^C(w) \cup \bigcup_{(v,b,w) \in E} update\big(R^C(v), b\big)$$

where $update(T, b)$ is defined as $\{\tau \circ b \mid \tau \in T\}$ and $\circ$ denotes the concatenation operator. The initial values, denoted by $\mathcal{I}^C(w)$, are the empty set for all locations except the program's entry point which is initialized to the set containing the empty trace $\mathcal{I}^C(i) = \{\epsilon\}$.

The equations can intuitively be understood as follows: Either a trace starts in location w, then it is given by $\mathcal{I}^C(w) = \{\epsilon\}$; or it starts elsewhere and reaches w from one of its predecessors v via edge (v, b, w). In the latter case, the trace reaching location w is obtained by concatenating the memory access b on the edge from v to w to the trace reaching location v.

A memory block b is persistent throughout a given program if b is persistent on each trace that may be immediately followed by an access to b. The set $V_b = \{v \in V \mid \exists w \in V : (v, b, w) \in E\}$ contains the program locations that can be followed by an access to block b.

Definition II.2 (Persistence)**.** Memory block $b \in \mathcal{B}$ is *persistent*, denoted by $persistent(b)$, if at each location $v \in V_b \subseteq V$ that can be followed by an access to b:

$$persistent\big(R^C(v), b\big) \coloneqq \forall \tau \in R^C(v) : persistent(\tau, b)$$

A memory block b may cause at most one miss during any possible execution through $\mathcal{G}$ if and only if it is persistent according to the definition above.

Scopes: A memory block might only persist in the cache during a certain phase of execution, e. g., in the innermost loop of a loop nest, and not during the overall program execution. To capture this behavior, *scopes* have been introduced in [9], [11] to describe a portion of program execution. A block that is persistent within a scope can cause at most one miss for each entrance of the scope during execution. For the sake of readability, we limit our formalization to persistence within the whole program. It is an easy exercise to extend the definitions to account for scopes. Indeed, the experimental evaluation in Section VI is performed by analyses at scope level.

D. Existing Cache Persistence Analyses

In general, the trace semantics is not computable as the number of traces may be infinite and the lengths of individual traces are unbounded. Persistence analyses thus rely on abstractions of cache traces that lead to finite representations.

A variety of cache persistence analyses has been proposed in the literature [3]–[16] with varying degrees of precision. In this section, we briefly discuss two of the existing persistence analyses, $C\text{-}Must$ and $Block\text{-}CS$ to convey how such persistence analyses operate in general, and also to illustrate that they are not exact.

Observe that according to Definition II.2, whether or not a memory block b is persistent is determined by the sizes of b's conflict sets at all program points that may be followed by accesses to block b. As a consequence, all existing persistence analyses can be seen as *approximating* the possible sets of conflict sets of each memory block at each program point.

Existing persistence analyses employ two different approaches to approximate the conflict sets of a memory block:

1) The $C\text{-}Must$ analysis [7], [8] maintains an *upper bound on the sizes* of all of the block's possible conflict sets.
2) The $Block\text{-}CS$ analysis [11], [14] maintains a *superset* of all of the block's possible conflict sets.

See Figure 2 for an example illustrating the results of the two analyses. For readability, the figure only includes the analysis information for memory block v. Following the access to v, the only possible conflict set of v is $\{v\}$ and so $C\text{-}Must$ maintains a bound of 1 and $Block\text{-}CS$ maintains $\{v\}$ as a superset of all of v's conflict sets. After the possible accesses to w and x, the possible conflict sets of v are $\{v, w\}$ and $\{v, x\}$, and $C\text{-}Must(v) = 2$. However, to overapproximate both conflict sets, $Block\text{-}CS(v) = \{v, w, x\} = \{v, w\} \cup \{v, x\}$. Thus, $C\text{-}Must$ is able to conclude that v is persistent in a cache of size 2, while $Block\text{-}CS$ is not, as $|\{v, w, x\}| > 2$. Clearly, $Block\text{-}CS$ is not an exact analysis.

Unfortunately, as the example in Figure 3 illustrates, $C\text{-}Must$ neither is exact: Both x and y are persistent in a cache of size 2, and indeed $Block\text{-}CS$ is able to show that, as $Block\text{-}CS(x) = \{x, y\}$. On the other hand, $C\text{-}Must$ is not able to derive any finite bound on the sizes of x's possible conflict sets. This is because $C\text{-}Must$ does not "remember" whether or not a given memory block has already been accounted for in its upper bounds. This may lead the analysis to account for the same block multiple times in loops.

E. A General Framework for Cache Persistence Analyses

Persistence analyses can be formalized within the framework of abstract interpretation [20] to reason about their correctness and precision. Here, we briefly present a simplified version of the persistence analysis framework developed in [17]. Persistence abstractions $\widehat{C}$ are characterized by

- an abstract update function $\widehat{update}$ to model the effect of a memory access,
- a join operator $\sqcup$ to combine multiple abstract traces into one at control-flow joins, and

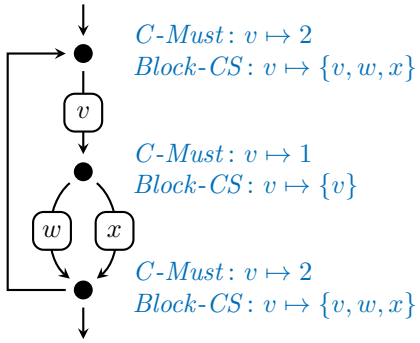


Fig. 2: Example illustrating *C-Must* and *Block-CS*, which shows that *C-Must* may be more precise than *Block-CS*.

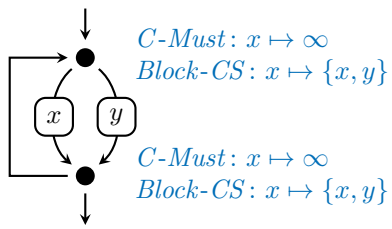


Fig. 3: Example illustrating *C-Must* and *Block-CS*, which shows that *Block-CS* may be more precise than *C-Must*.

- an abstract function $\widehat{persistent}$ to classify memory blocks as persistent.

Note that the join operator also defines a partial order $\sqsubseteq$ on the abstract traces as follows: $x \sqsubseteq y$ if and only if $x \sqcup y = y$. This partial order captures the relative precision of different abstract traces, where $x \sqsubseteq y$ implies that x is more precise analysis information than y.

In order to formally capture the meaning of abstract traces, abstraction and concretization functions, α and γ, can be defined to relate sets of concrete traces to abstract traces.

Analogously to the concrete semantics, the abstract semantics is captured as the least solution $\widehat{R}\colon V \to \widehat{C}$ of the following set of equations:

$$\forall w \in V : \widehat{R}(w) = \widehat{\mathcal{I}}(w) \sqcup \bigsqcup_{(v,b,w)\in E} \widehat{update}\big(\widehat{R}(v), b\big)$$

The initial value $\widehat{\mathcal{I}}(w)$ is $\bot$, the bottom element of the partial order $\sqsubseteq$, for all locations except for the program's entry point, where $\widehat{\mathcal{I}}(i) = \alpha\big(\mathcal{I}^C(i)\big)$.

A sound persistence analysis overapproximates the concrete trace semantics, i. e., $R^C(v) \subseteq \gamma\big(\widehat{R}(v)\big)$ for all locations v. Equivalently, given that (α, γ) form a Galois connection [20], we have $\alpha(R^C(v)) \sqsubseteq \widehat{R}(v)$ for all locations v.

Analogously to persistence in the concrete case, a memory block b is classified as persistent in a given program, denoted by $\widehat{persistent}(b)$, if b is classified as persistent at each control location in V_b, i. e., each location that might be immediately followed by an access to b. A sound persistence analysis never classifies a block as persistent that is not actually persistent on all concrete traces:

Definition II.3 (Soundness of Persistence Analysis)**.** A persistence analysis is *sound* if:

$$\widehat{persistent}(b) \to persistent(b)$$

Sound abstractions are not guaranteed to be exact, i. e., there can be persistent memory blocks that are *not* classified as persistent by the abstraction. Indeed, none of the existing persistence analyses is exact [17].

III. Exact Cache Persistence Analysis: A Sequence of Abstractions

In this paper, we do not just aim for yet another sound persistence analysis, but we aim for an *exact* persistence analysis that determines each and every persistent memory block.

Definition III.1 (Exactness of Persistence Analysis)**.** A persistence analysis is *exact* if:

$$\widehat{persistent}(b) \leftrightarrow persistent(b)$$

Note that by definition an exact analysis is also sound. How can we obtain an exact analysis? This requires an abstraction that satisfies the following two properties:

1) Applying the abstraction to "perfect" concrete information preserves enough information to precisely classify memory blocks as persistent or not. This property is comparably easy to achieve. It is captured formally by (3) in the theorem below. The *C-Must* abstraction, for example, satisfies this property, while *Block-CS* does not.
2) Abstract joins and abstract updates may not lose any additional information, beyond the information loss inherent to the abstraction itself. This is more difficult to achieve, and indeed none of the existing persistence analyses does. This property is captured formally by (1) and (2) in the theorem below.

Theorem III.2 (Exactness of Persistence Analysis)**.** *A persistence analysis over a finite abstract domain $\widehat{C}$ is* exact *if:*

$$\forall T, b : \alpha(update(T, b)) = \widehat{update}(\alpha(T), b) \tag{1}$$

$$\forall I, T_i : \alpha\Big(\bigcup_{i\in I} T_i\Big) = \bigsqcup_{i\in I} \alpha(T_i) \tag{2}$$

and the abstraction preserves the persistence classification:

$$\forall T, b : persistent(T, b) \leftrightarrow \widehat{persistent}(\alpha(T), b) \tag{3}$$

Proof. We will use standard arguments from abstract interpretation and fixpoint theory (along the lines of [21]) to show that (1) and (2) imply that:

$$\forall v \in V : \alpha\big(R^C(v)\big) = \widehat{R}(v) \tag{4}$$

i. e., the abstract semantics is precisely the abstraction of the concrete semantics. Applying (3) to (4) then yields the theorem.

To this end, we first define concrete and abstract transformers f and $\hat{f}$ as follows:

$$f(R) \coloneqq \lambda w.\ \mathcal{I}^C(w) \cup \bigcup_{(v,b,w)\in E} update(R(v), b)$$
$$\hat{f}(\widehat{R}) \coloneqq \lambda w.\ \widehat{\mathcal{I}}(w) \sqcup \bigsqcup_{(v,b,w)\in E} \widehat{update}\big(\widehat{R}(v), b\big)$$

By construction, the trace semantics R^C and the abstract semantics $\widehat{R}$ are the least fixed points of f and $\hat{f}$.

Applying (1) and (2) we show below that:

$$\alpha \circ f = \hat{f} \circ \alpha \tag{5}$$

where $\alpha\colon C \to \widehat{C}$ is lifted to the required domain $\alpha\colon (V \to C) \to \big(V \to \widehat{C}\big)$ as follows:

$$\alpha(h) \coloneqq \lambda v \in V.\ \alpha(h(v)) \tag{6}$$

The transformations to show (5) are as follows:

$$\begin{aligned}
& (\alpha \circ f)(R) \\
\overset{Def}{=}\ & \alpha(\lambda w.\ \mathcal{I}^C(w) \cup \bigcup_{(v,b,w)\in E} update(R(v), b)) \\
\overset{(6)}{=}\ & \lambda w.\ \alpha(\mathcal{I}^C(w) \cup \bigcup_{(v,b,w)\in E} update(R(v), b)) \\
\overset{(1),(2)}{=}\ & \lambda w.\ \widehat{\mathcal{I}}(w) \sqcup \bigsqcup_{(v,b,w)\in E} \widehat{update}(\alpha(R(v)), b) \\
\overset{(6)}{=}\ & \lambda w.\ \widehat{\mathcal{I}}(w) \sqcup \bigsqcup_{(v,b,w)\in E} \widehat{update}(\alpha(R)(v), b) \\
\overset{Def}{=}\ & \big(\hat{f} \circ \alpha\big)(R)
\end{aligned}$$

By construction, f is continuous [22], and $\hat{f}$ is continuous as it is monotone and the abstract domain is finite.
By Kleene's fixpoint theorem ($\star$) and (5), we have:

$$\begin{aligned}
\alpha\big(R^C\big) \overset{Def}{=} \alpha(lfp f) \overset{(\star)}{=} \alpha\Big(\bigcup_{i\in\mathbb{N}_0} f^i(\bot)\Big) \\
\overset{(2)}{=} \bigsqcup_{i\in\mathbb{N}_0} \alpha\big(f^i(\bot)\big) \overset{(5)}{=} \bigsqcup_{i\in\mathbb{N}_0} \hat{f}^i(\alpha(\bot)) \\
= \bigsqcup_{i\in\mathbb{N}_0} \hat{f}^i\big(\widehat{\bot}\big) \overset{(*)}{=} lfp \hat{f} \overset{Def}{=} \widehat{R}
\end{aligned} \tag{7}$$

Applying (3) to (7) then yields the theorem. ■

The remaining part of this section presents the abstraction of cache traces underlying our new exact persistence analysis. For pedagogical reasons, the abstraction is presented incrementally in three steps: First, an abstraction of concrete traces as a mapping from memory blocks to sets of conflict sets is defined. In the second step, a monotonicity property of LRU replacement is exploited to reduce the number of conflict sets the analysis needs to track. Third, conflict sets that exceed the cache's associativity are collapsed to further improve efficiency. The overall scheme is depicted in Figure 4.

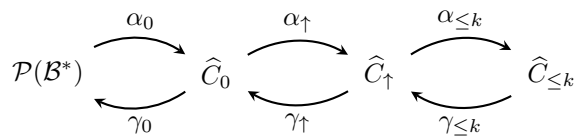


Fig. 4: Exact persistence abstractions developed in this section.

A. From Memory-Access Traces to Sets of Conflict Sets

The first abstraction *Exact-CS₀* maintains for each memory block the set of all possible *conflict sets* that appear in cache traces in which the memory block has been accessed. Recall that the conflict set of block b is the set of all distinct memory blocks accessed since the last access of b. Formally, the abstract domain is defined as:

$$\widehat{C}_0 \coloneqq \mathcal{B} \to \mathcal{P}(\mathcal{P}(\mathcal{B}))$$

An abstract trace $\widehat{S} \in \widehat{C}_0$ represents all concrete traces where for each memory block $b \in \mathcal{B}$ either b's conflict set is in $\widehat{S}(b)$ or b has not been accessed on the trace.

Upon a memory access to block b, its set of conflict sets is set to $\{\{b\}\}$, i. e., b conflicts just with itself. For all remaining memory blocks b', the accessed block b is added to *every* conflict set $s \in \widehat{S}(b')$:

$$\widehat{update}_0\big(\widehat{S}, b\big) \coloneqq \lambda b'.\ \begin{cases} \{\{b\}\} & \text{if } b' = b \\ \big\{s \cup \{b\} \mid s \in \widehat{S}(b')\big\} & \text{otherwise} \end{cases} \tag{8}$$

Note that as long as a block b' has not been accessed, the information $\widehat{S}(b') = \emptyset$ propagates.

At control-flow joins in the program, the union of the sets of conflict sets is taken:

$$\forall I \subseteq \mathbb{N}_0 : \bigsqcup_{i\in I}{}_0\, \widehat{S}_i \coloneqq \lambda b \in \mathcal{B}.\ \bigcup_{i\in I} \widehat{S}_i(b)$$

A drawback of the *Block-CS* abstraction is the loss of precision at joins as seen in Figure 2. Using sets of sets, *Exact-CS₀* keeps all conflict sets side by side without losing any precision.

Next, we define the abstraction function α_0 that relates a set of concrete traces to an abstract trace. For singleton sets, i. e., sets containing a single concrete trace, the abstraction function is recursively defined as follows:

$$\alpha_0(\{\epsilon\}) \coloneqq \lambda b \in \mathcal{B}.\ \emptyset$$
$$\alpha_0(\{\tau \circ b_{n-1}\}) \coloneqq \widehat{update}_0(\alpha_0(\{\tau\}), b_{n-1})$$

Abstractions of concrete traces are recursively constructed using the abstract update function with the accessed memory blocks. The base case, i. e., the abstraction of the empty trace ϵ, assigns an empty set of conflict sets to each block b since no memory block has been accessed.

The abstraction α_0 is lifted to arbitrary sets in the usual way:

$$\alpha_0(\{t_i \mid i \in I \subseteq \mathbb{N}_0\}) \coloneqq \bigsqcup_{i\in I}{}_0\, \alpha_0(\{t_i\})$$

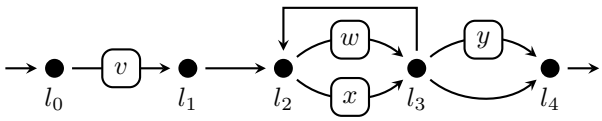


Fig. 5: Example illustrating *Exact-CS*.

A memory block b is classified as persistent if all conflict sets have cardinality less than or equal to the associativity k. This means that at most $k-1$ distinct other blocks have been accessed since the last access to the block itself:

$$\widehat{persistent}_0(\widehat{S}, b) \coloneqq \max\{|s| \mid s \in \widehat{S}(b)\} \leq k$$

Note that we assume the maximum over an empty set to be zero. As a consequence, a block that has never been accessed is classified as persistent.

See Figure 5 for an example of a control-flow graph and the corresponding *Exact-CS* cache trace abstractions in Table I. Note that only the analysis information for memory block v is shown as it highlights best the differences between the different exact analyses.

Theorem III.3 (Exactness of *Exact-CS*$_0$)**.** *The persistence analysis Exact-CS*$_0$ *is exact in the sense of Definition III.1.*

Due to space limitations we omit the proofs in this paper, but they can be found in the appendix of this technical report.

B. Exploiting Monotonicity

While *Exact-CS*$_0$ is exact, the number of conflict sets to track for a given block can be high. To classify a block as persistent only the largest conflict set of each block at each program location is relevant. It would be tempting to only keep the largest sets, but, unfortunately, this would yield an incorrect analysis, as these largest sets could not be correctly maintained across updates and joins.

It is, however, safe to remove conflict sets that are completely subsumed by others. This is because the update function $\widehat{update}_0$ is *monotonic*: For example, consider the set of conflict sets $\{\{a, b, c\}, \{a, b\}\}$. No matter which blocks are accessed and used to update these conflict sets, the first set $\{a, b, c\}$ and its successors will always subsume the second set $\{a, b\}$ and its successors. Removing such subsumed sets reduces the computational effort of the analysis, in particular the memory consumption, without any loss of precision. The resulting abstraction *Exact-CS*$_\uparrow$ is defined relative to the *Exact-CS*$_0$ abstraction as shown in Figure 4.

The abstraction function $\alpha_\uparrow$ takes an abstract trace and removes all conflict sets that are subsumed by larger sets:

$$\alpha_\uparrow(\widehat{S}) \coloneqq \lambda b \in \mathcal{B}.\ maxSet(\widehat{S}(b))$$

where *maxSet* is defined as follows:

$$maxSet(S) \coloneqq \{s \in S \mid \neg\exists s' \in S : s \subsetneq s'\}$$

In order to maintain a minimal set of conflict sets, the join operator takes the maximal conflict sets of the union of the sets of conflict sets to be joined:

$$\forall I \subseteq \mathbb{N}_0 : \bigsqcup_{i \in I}^{\uparrow} \widehat{S}_i \coloneqq \lambda b \in \mathcal{B}.\ maxSet\Big(\bigcup_{i \in I} \widehat{S}_i(b)\Big) \quad (9)$$

Similarly, the abstract update function removes all non-maximal conflict sets after adding b to the conflict sets:

$$\widehat{update}_\uparrow(\widehat{S}, b) \coloneqq \lambda b' \in \mathcal{B}.\ maxSet\big(\widehat{update}_0(\widehat{S}, b)(b')\big)$$

The classification function can be reused without modification.

Theorem III.4 (Exactness of *Exact-CS*$_\uparrow$)**.** *The persistence analysis Exact-CS*$_\uparrow$ *is exact in the sense of Definition III.1.*

C. Limit at Associativity

The abstraction described above allows the individual conflict sets to grow arbitrarily large. However, the persistence classification checks the existence of a single conflict set with cardinality greater than the associativity k. Thus, there is no need to distinguish conflict sets containing more than k elements.

Instead, all conflict sets with a cardinality larger than k can be collapsed into a single representative $\mathcal{B}$, i. e., the set of all memory blocks. Note that all conflict sets are trivially subsumed by $\mathcal{B}$. This fact and the monotonicity property from Section III-B allow to replace the whole set of conflict sets by $\{\mathcal{B}\}$ in the presence of an oversized conflict set. This reduces the computational effort of the analysis even further, in particular its memory consumption, without any loss of precision. The resulting abstraction *Exact-CS*$_{\leq k}$ is defined relative to the *Exact-CS*$_\uparrow$ abstraction as shown in Figure 4.

The abstraction function $\alpha_{\leq k}$ takes an *Exact-CS*$_\uparrow$ abstract trace and eliminates conflict sets with cardinality larger than k:

$$\alpha_{\leq k}(\widehat{S}) \coloneqq \lambda b \in \mathcal{B}.\ limit(\widehat{S}(b))$$

where

$$limit(S) \coloneqq \begin{cases} \{\mathcal{B}\} & \text{if } \exists s \in S : |s| > k \\ S & \text{otherwise} \end{cases}$$

The update function $\widehat{update}_{\leq k}$ is consequently defined as:

$$\widehat{update}_{\leq k}(\widehat{S}, b) \coloneqq \lambda b' \in \mathcal{B}.\ limit\big(\widehat{update}_\uparrow(\widehat{S}, b)(b')\big)$$

The join operator and the classification function can be reused as they do not increase the size of any conflict set.

Theorem III.5 (Exactness of *Exact-CS*$_{\leq k}$)**.** *The persistence analysis Exact-CS*$_{\leq k}$ *is exact in the sense of Definition III.1.*

D. Example of Superiority over Prior Persistence Analyses

Figure 6 contains an example control-flow graph on which all existing persistence analysis fail. In the example, block v is clearly persistent in a fully-associative cache of size $k = 3$: at most two distinct blocks are accessed between any two accesses to v.

TABLE I: Concrete traces as regular expressions and persistence abstractions for Figure 5 (with $k = 3$).

	Reachable access traces	$Exact\text{-}CS_0$	$Exact\text{-}CS_{\uparrow}$	$Exact\text{-}CS_{\leq k}$
l_0	ε	$v \mapsto \emptyset$	$v \mapsto \emptyset$	$v \mapsto \emptyset$
l_1	v	$v \mapsto \{\{v\}\}$	$v \mapsto \{\{v\}\}$	$v \mapsto \{\{v\}\}$
l_2	$v[w\|x]^*$	$v \mapsto \{\{v\}, \{v,w\}, \{v,x\}, \{v,w,x\}\}$	$v \mapsto \{\{v,w,x\}\}$	$v \mapsto \{\{v,w,x\}\}$
l_3	$v[w\|x]^+$	$v \mapsto \{\{v,w\}, \{v,x\}, \{v,w,x\}\}$	$v \mapsto \{\{v,w,x\}\}$	$v \mapsto \{\{v,w,x\}\}$
l_4	$v[w\|x]^+$, $v[w\|x]^+y$	$v \mapsto \{\{v,w\}, \{v,x\}, \{v,w,x\}, \{v,w,y\}, \{v,x,y\}, \{v,w,x,y\}\}$	$v \mapsto \{\{v,w,x,y\}\}$	$v \mapsto \{\mathcal{B}\}$

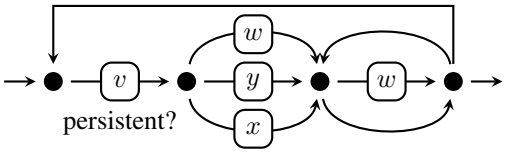


Fig. 6: Example illustrating exact analysis outperforming previous persistence analyses. Fully-associative cache with $k = 3$.

```
int arr[10];
for (int i = 0; i < 100; ++i)
    sum += arr[read_sensor()]
```

Fig. 7: Input-dependent data accesses.

The most precise known persistence analysis is the combination $C\text{-}Must \times Must \times Block\text{-}CS$ of the traditional must analysis, the $C\text{-}Must$, and the $Block\text{-}CS$ analysis [17].

Since neither x, y, nor w is *guaranteed* to get accessed, the must analysis does not gain any information about them. The $Block\text{-}CS$ analysis fails to classify v as persistent because there are three distinct blocks that may conflict with v in between two consecutive accesses to v. As a consequence, neither of the two analyses is able to support the $C\text{-}Must$ analysis. On its own, the $C\text{-}Must$ analysis ages v upon each access distinct from v. Since more than associativity many accesses might happen between the accesses to v (note the inner loop), the $C\text{-}Must$ analysis is also of no use.

IV. Exact Cache Persistence Analysis: Implementation

A. Implementation using Binary Decision Diagrams

The implementation of the $Exact\text{-}CS_{\leq k}$ analysis needs to maintain a set of conflict sets for each memory block in the program at every program point. Maintaining separate explicit representations of each set of conflict sets for each memory block and at each program point would likely be highly inefficient. An efficient implementation should implicitly (and thus hopefully more compactly) represent sets of conflict sets and it should share as much information as possible between different memory blocks and program points.

Observe that sets of conflict sets can be represented using Boolean functions: Let the set of memory blocks be $\mathcal{B} = \{b_1, \ldots, b_n\}$. Then a Boolean valuation $v = v_1, \ldots, v_n \in \mathbb{B}^n$ represents a set $set(v)$ of memory blocks as follows:

$$set(v_1, \ldots, v_n) = \{b_i \mid 1 \leq i \leq n \wedge v_i = 1\}$$

Extending upon this, a Boolean function $f \colon \mathbb{B}^n \to \mathbb{B}$ represents a set of conflict sets $CS(f)$ as follows:

$$CS(f) \coloneqq \{set(v) \mid v \in \mathbb{B}^n \wedge f(v) = 1\}$$

Binary decision diagrams [23] are a class of data structures that have been designed to efficiently represent and manipulate Boolean functions. They compactly represent individual Boolean functions and share information between the representations of different Boolean functions stored in the same data structure. Our implementation uses zero-suppressed binary decision diagrams (ZDDs) [24], [25], a variant of binary decision diagrams optimized to represent sets of sparse sets. This is beneficial in our setting as the sets of conflict sets used in our trace abstractions are typically sparse with respect to the universe $\mathcal{B}$; no conflict set in $Exact\text{-}CS_{\leq k}$ is greater than the associativity k.

To perform operations on ZDDs, the Colorado University Decision Diagram (`CUDD`) library[2] [26] is used in combination with the `EXTRA` library[3]. The latter offers extended ZDD procedures that facilitate the manipulation of ZDDs. More precisely, the following operators of the `EXTRA` library are used in the persistence analysis where S and T are sets containing sets:

1) $maxUnion$ calculates the maximum of the union of sets S and T:

$$maxUnion(S, T) \coloneqq maxSet(S \cup T)$$

This function is used to compute the abstract joins exploiting monotonicity in (9).

2) $maxDotProduct$ takes all pair-wise unions of subsets from S and T and computes the maximum of this set, i. e., it drops subsumed elements in the result:

$$maxDotProduct(S, T) \coloneqq maxSet\{s \cup t \mid s \in S \wedge t \in T\}$$

This function is involved in the abstract update (8) to add an accessed block b to every conflict set in a set. More precisely, the following property is exploited:

$$maxDotProduct(S, \{\{b\}\}) = maxSet\{s \cup \{b\} \mid s \in S\}$$

B. Extension to Data Caches

Until now, we have assumed that only a single known memory block is accessed on each control-flow edge. This

[2]Available at https://github.com/ivmai/cudd

[3]Available at https://people.eecs.berkeley.edu/~alanmi/research/extra/

is valid in the context of an instruction cache analysis. For data cache analysis, however, we have to deal with uncertainty about which memory blocks might be accessed. The memory blocks a single `load` or `store` instruction accesses might depend on dynamic aspects, e. g., the program input or a loop iteration counter. See Listing 7 for an example of a memory access depending on program input. Array `arr` is small compared to the number of loop iterations in the example. If `arr` completely fits into the cache, an exact persistence analysis has to classify it as persistent.

Instead of cache updates with a single accessed memory block, the update function for data caches has to consider a set of potentially accessed memory blocks $B \subseteq \mathcal{B}$. The straightforward solution is to lift the update function for single blocks to sets of blocks by performing an update for each block $b \in B$ individually and then joining the results:

$$\widehat{update}\left(\widehat{S}, B\right) := \bigsqcup_{b \in B} \widehat{update}\left(\widehat{S}, b\right) \tag{10}$$

Implementation: Prior to cache analysis, a preprocessing analysis determines a set $B \subseteq \mathcal{B}$ of potentially accessed memory blocks for each memory instruction in the program. Such sets can be large, e. g., when a large array is accessed. The preprocessing analysis might even fail to derive useful information for some accesses, resorting to $\mathcal{B}$. In such cases, performing a cache update according to (10) can be computationally expensive or infeasible.

To efficiently cope with large sets $B \subseteq \mathcal{B}$, we implemented a slightly different analysis that maps memory blocks to a *list* of k sets of conflict sets, i. e.,

$$\widehat{C} := \mathcal{B} \to \mathcal{P}(\mathcal{P}(\mathcal{B})) \times \cdots \times \mathcal{P}(\mathcal{P}(\mathcal{B}))$$

The list entry at index $i \in [0, \ldots, k-1]$ corresponds to the set of conflicting sets where each conflict set implicitly contains an additional i distinct unknown anonymous memory blocks.

Upon classification, these anonymous blocks have to be accounted for by adding them to the conflict set cardinalities:

$$\widehat{persistent}\left(\widehat{S}, b\right) := \forall i : \max\left\{|s| \;\middle|\; s \in \widehat{S}_i(b)\right\} + i \leq k$$

At updates with a single memory block b, the information for b is reset to the set with the conflict set $\{b\}$ as first list entry. For all other memory blocks, the original update is performed for each entry in the list:

$$\widehat{update}\left(\widehat{S}, b\right) := \lambda b'. \begin{cases} \left[\{\{b\}\}, \emptyset, \ldots, \emptyset\right] & \text{if } b' = b \\ \left[\widehat{update}\left(\widehat{S}_i, b\right)(b') \;\middle|\; 0 \leq i < k\right] & \text{otherwise} \end{cases}$$

In case of a completely unknown access, all entries are shifted by one position to the right. If there has been a non-empty conflict set at the rightmost position, the corresponding block is not guaranteed to be cached any more, which is indicated by $\mathcal{B}$ in the first position:

$$\widehat{update}\left(\widehat{S}\right) := \lambda b'. \begin{cases} \left[\;\emptyset\;, \widehat{S}_0(b'), \ldots, \widehat{S}_{k-2}(b')\right] & \text{if } \widehat{S}_{k-1}(b') = \emptyset \\ \left[\{\mathcal{B}\}, \;\emptyset\;, \ldots, \;\emptyset\;\right] & \text{otherwise} \end{cases}$$

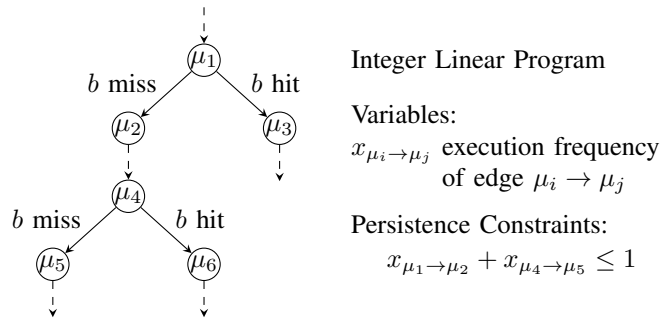


Fig. 8: Abstract execution graph snippet and persistence path analysis constraint.

For sets $B \subseteq \mathcal{B}$ of potentially accessed blocks larger than the associativity k, the above update can be performed without any loss of precision. For smaller sets $B \subseteq \mathcal{B}$, the update is performed as defined by Equation 10.

C. Validation

The two features described in this section are used for testing purposes and by default turned off during analysis. Their intention is to validate our implementation and spot bugs. One of the advantages of having an exact analysis is that it can be used to verify the correctness of other analyses to some extent as the exact one will always provide more precise results than the imprecise ones. We implemented a flag that allows us to run the exact persistence analysis alongside any other persistence analysis. After every operation, e. g., updates or joins, the sets of persistent memory blocks are compared. The set of the exact analysis must always include the other set due to its exactness. This proves by no means the implementation of the exact analysis to be correct but gives some hint that the analyses are computing reasonable results.

Another critical aspect of the implementation is the manipulation of ZDDs to represent sets of conflict sets. Therefore, the analysis was extended with an explicit representation which uses the sets of the C++ Standard Template Library. Again, after each operation, the equality between the ZDDs and explicit set representations is checked to see whether the ZDD library is working as intended.

D. Integration with WCET Analysis

The information on the persistence of memory blocks is used in worst-case execution time (WCET) analysis to obtain more precise upper bounds on a program's execution time. WCET analysis is commonly performed in two phases: microarchitectural analysis and path analysis. Microarchitectural analysis constructs an *abstract execution graph* that represents the execution of a program on a given hardware platform at the granularity of processor cycles. Nodes correspond to the abstract (microarchitectural) state of the hardware platform, including information about the caches, and edges represent the actual execution. Figure 8 illustrates an example graph.

The path analysis determines the longest path through the abstract execution graph using an integer linear program [27].

Since persistence information is a property of execution traces, it is taken into account in the path analysis. To this end, for each persistent memory block $b \in \mathcal{B}$ in the program, a linear constraint is used to limit the number of cache misses of b to one. An example constraint is shown in Figure 8.

V. Related Work

For LRU caches, persistence analysis is strongly related to must analysis [8]: A memory block b must be cached at location v, if b's age is at most k, the associativity of the cache, on all traces ending in location v. This is the case *if and only if* block b is persistent on all traces ending in location v *and* block b has previously been accessed on all these traces. As a consequence, must analysis could be solved by a persistence analysis running alongside a "definitely-accessed" analysis that determines whether a block is guaranteed to have been accessed before reaching a given program location. Exploring this relation further is future work.

The development of the exact persistence analyses in this paper has been inspired by recent work of Touzeau et al. [28], in which they develop exact must and may analyses for LRU caches. Similarly to our persistence analysis, their implementation employs ZDDs to compactly represent sets of sets of memory blocks and their abstraction exploits the monotonicity of LRU replacement to reduce the number of sets to track. Besides solving a related but different problem, our analysis adds support for the efficient analysis of data caches as discussed in Section IV and unlike [28] we evaluate the analysis in the context of WCET analysis.

The presence of timing anomalies [29], [30] can often be traced back to the non-monotonicity of a system's dynamics. In contrast to LRU, other cache replacement policies such as FIFO, NMRU, and PLRU, do not behave monotonically, and have been found to exhibit timing anomalies [31], [32]. For that reason it seems unlikely that our analysis approach can be extended to such policies. The strictly in-order core [33] is a pipelined processor core that has been designed to be free of timing anomalies by eliminating dependencies in the pipeline that induce non-monotonicity.

A variety of cache persistence analyses has been proposed in the literature [3]–[16] with varying degrees of precision. Many persistence analyses can be seen as combinations of analyses from a small set of basic persistence analyses. We have already discussed two of these basic analyses in Section II.: $Block\text{-}CS$ and $C\text{-}Must$. Another basic analysis, called $Global\text{-}CS$ maintains a superset of the conflict sets of *all* memory blocks, rather than maintaining separate information for each memory block, as $Block\text{-}CS$ does. This simple approach has been particularly popular in the literature [3]–[6], [14], [15] and it constitutes the most efficient known analysis.

Reineke [17] has analyzed the landscape of persistence analyses and has shown how the different analyses relate to each other in terms of precision, and how they can be explained as combinations of basic persistence analyses. In Figure 9, we reproduce the landscape of persistence analyses from [17]. In the diagram, a node labeled $A \times B$ denotes an analysis that is obtained by the combination of the basic analyses A and B. Further, if analysis A is provably more precise than analysis B, then A and B are connected by an edge and A is higher up in the diagram.

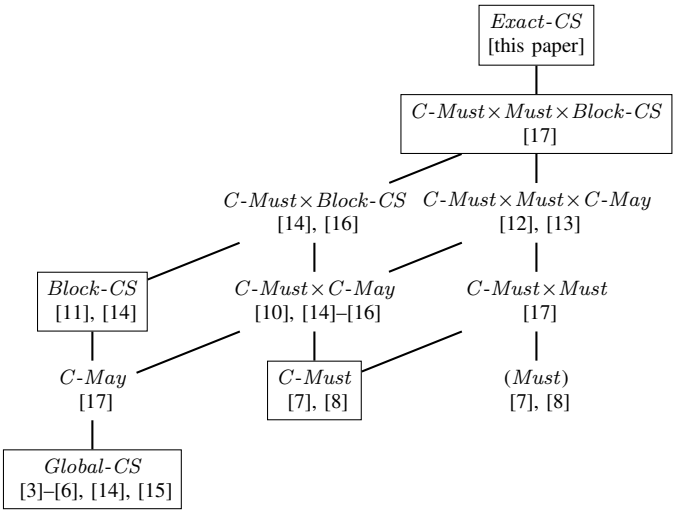


Fig. 9: Landscape of persistence analyses adapted from [17].

For the experimental evaluation in Section VI, we chose to evaluate the three basic analyses $Global\text{-}CS$, $Block\text{-}CS$, $C\text{-}Must$, and the most precise known combination of analyses $C\text{-}Must \times Must \times Block\text{-}CS$ as explained below in Section VI. These four analyses and the exact analysis developed in this paper are framed in Figure 9.

VI. Experimental Evaluation

In this evaluation, we compare the performance of the exact analysis $Exact\text{-}CS$ in terms of the calculated WCET bound as well as the run time and memory consumption with the four existing persistence analyses $Global\text{-}CS$, $Block\text{-}CS$, $C\text{-}Must$, and $C\text{-}Must \times Must \times Block\text{-}CS$. While $Global\text{-}CS$ represents the most efficient persistence analysis, $C\text{-}Must \times Must \times Block\text{-}CS$ provides the most precise results among the previously known analyses. In addition, we chose $Block\text{-}CS$ and $C\text{-}Must$ as they represent two basic but complementary ideas to approximate conflict sets as described in Section II-D. The results below also indicate that a comparison with the various possible other combinations (Figure 9) would not have revealed major further insights.

A. Experimental Setup

We implemented the exact persistence analysis as well as the previously known analyses within the WCET analysis framework LLVMTA, which is described in [34]. LLVMTA analyses the persistence of memory blocks at the level of scopes instead of the whole program. Precisely, every loop at any nesting level in the program is considered a separate persistence scope.

To evaluate the implementation of the exact analysis, we use the benchmarks of the TACLeBench suite [35]. TACLeBench

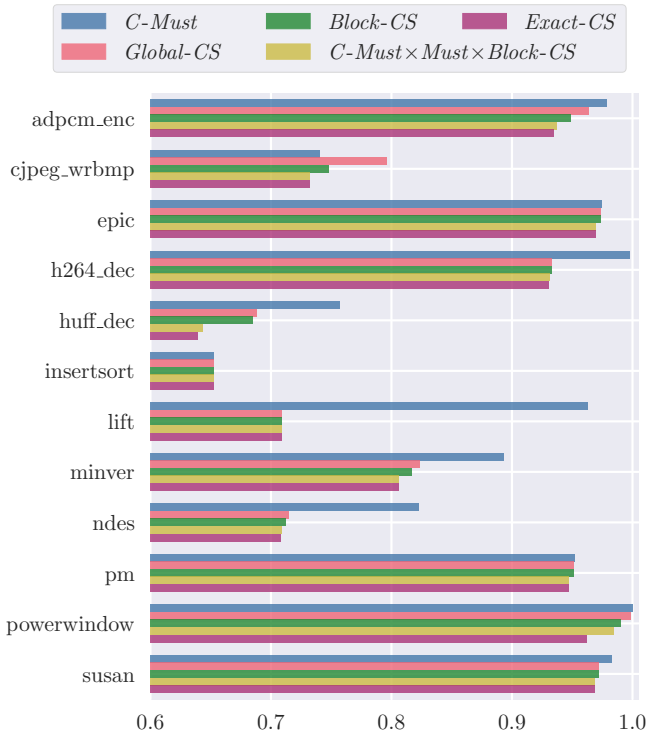


Fig. 10: WCET ratios of the persistence analyses compared with performing no persistence analysis at all.

consists of several open-source C programs commonly used to evaluate timing analysis.

In the following, results and measurements are shown for TACLeBench compiled *without* compiler optimizations enabled. Software for safety-critical embedded systems is often compiled without optimizations to ease the subsequent verification of the produced binary w. r. t. the underlying high-level model [36]. We also conducted experiments with enabled compiler optimizations. Since compiler optimizations generally reduce the amount of memory operations, they are less impacted by different cache persistence analyses and thus show slightly fewer differences among the different analyses. For the sake of completeness, all the numbers are made available in the appendix of this technical report.

The analyzed cache configuration has a total size of 4 kB and consists of 32 cache sets, 8 ways, and cache lines holding 16-byte-sized memory blocks, which is also used by [28] taking into account the relative small size of the benchmarks. Accessing a single word in main memory takes ten processor cycles with an additional cycle per consecutive word accessed. In total, the load of a cache line takes 13 cycles which is a realistic value for main memories such as the Automotive DRAM MT46V16M16 [37] clocked at 100 MHz. An additional evaluation of the exact analysis with a higher latency of 100 cycles showed no interesting differences.

All measurements have been performed on the same Linux machine, equipped with an Intel Core™ i5-4590 CPU (running at 3.30 GHz) and 20 GB of main memory.

B. Analysis Precision

This section answers how the exact, ZDD-based analysis compares with inexact alternatives in terms of the calculated WCET bounds. Figure 10 shows the WCET bounds obtained with the different persistence analyses for a selection of the TACLeBench benchmarks. All bounds are normalized to the WCET bounds obtained with no persistence analysis at all, i. e., only running the traditional age-based must and may analysis [7]. For instance, a value of 0.8 means that the WCET bound is improved by 20 % using the respective persistence analysis compared with the WCET bound obtained without any persistence analysis. A general observation is that persistence analysis is often important to obtain precise WCET bounds with improvements of more than 20 % in several cases.

We omitted a total of 27 benchmarks from the figure, among which 5 showed no differences between the analyses at all (as in `insertsort`); for 10 omitted benchmarks, only *C-Must* performed significantly worse than the rest (as in `lift`); and 12 benchmarks only showed negligible differences (as in `pm`). An unabridged chart can be found in the appendix of this technical report.

The chart shows that, in practice, the exact persistence analysis is only slightly more precise than existing inexact approaches. In most cases all persistence analyses perform almost identically (in particular in most of the omitted benchmarks), the only outlier being the *C-Must* analysis, e. g., in case of `lift` or `ndes`. Even the cheapest *Global-CS* analysis usually performs similar to the exact analysis, except for `cjpeg_wrbmp`. There are seven benchmarks in which the exact analysis obtains strictly better WCET bounds than all other analyses: the benchmarks `adpcm_enc` (0.2 %), `epic` (0.0006 %), `h264_dec` (0.04 %), `huff_dec` (0.6 %), `ndes` (0.1 %), `pm` (0.007 %), and `powerwindow` (2.4 %). The numbers in parentheses indicate the marginal improvements in terms of the computed WCET bound.

At its default settings, LLVMTA heuristically performs loop peeling, i. e., it distinguishes the initial loop iteration from the following loop iterations using trace partitioning [38]. The motivation for loop peeling is that the memory blocks used in the loop body miss the cache in the loop's first iteration, while the same memory blocks hit the cache in subsequent iterations. Without loop peeling, i. e., if the loop is considered as a whole, regular must and may analysis cannot classify such accesses. On the other hand, persistence analysis can cover such cases without loop peeling. Thus, to stress test the persistence analyses further, we performed another evaluation in which we deactivated loop peeling. As expected, without loop peeling, the average improvement of the exact persistence analysis over the plain must and may analysis in terms of WCET bounds rises to 52 %—compared to 17 % with loop peeling. The exact analysis, however, is again on par with the previously known persistence analyses showing improvements in the range of only 0.2 % to 0.3 % in most cases. An interesting insight from this experiment is that the *Global-CS* analysis performs significantly worse relative to

the remaining analyses once loop peeling is deactivated. This is likely due to the fact, that *Global-CS* is the only analysis whose analysis information for a block b is not "reset" upon an access to b itself. Loop peeling conceals this limitation because the must analysis is able to classify many accesses as always hit for all but the first loop iteration.

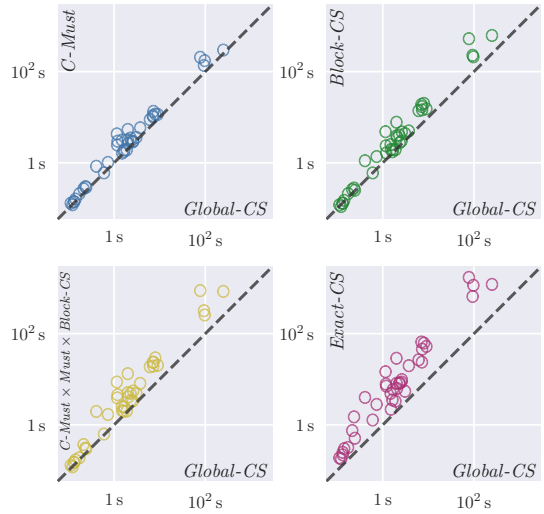
The execution time of a program can be seen as the sum of computation times and memory access latencies. The memory access latencies can further be split into contributions from data accesses and from instruction fetches. To focus the evaluation on the number of cache misses, we performed an analysis that separately bounds the maximum number of instruction and data cache misses. This experiment reveals that the observed differences between the persistence analyses are mainly due to the data cache.

C. Analysis Cost

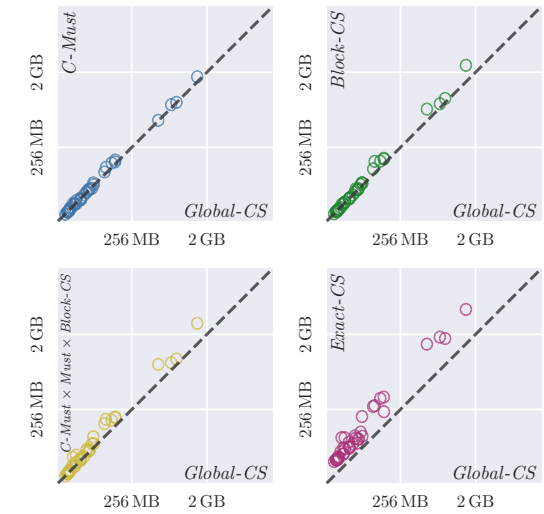
This section evaluates the analysis cost of the different persistence analyses in terms of analysis run time and memory consumption. The same set of persistence analyses as in the previous section is evaluated. Additional experimental results may be found in the appendix of this technical report.

The run time (Figure 11a) and memory consumption (Figure 11b) results are visualized in scatter plots. In each of the scatter plots the horizontal axis corresponds to the value obtained for the simplest and presumably cheapest analysis *Global-CS*. These values are compared with the remaining analyses on the vertical axis, respectively. A logarithmic scale is used for all scatter plots because the measured numbers vary greatly in size. Unsurprisingly, a general trend is that the exact analysis is the most expensive one regarding both analysis run time and memory consumption. However, even compared with the cheapest analysis *Global-CS*, the memory overhead of the exact analysis is less than $3\times$ for all benchmarks and the analysis time is at most $23\times$ higher.

In Figure 12, the exact analysis is compared directly with the most precise analysis from the literature, *C-Must*$\times$*Must*$\times$*Block-CS*. The data shows that the exact analysis is on the average $2\times$ slower and needs about $1.6\times$ more memory, which is indicated by the blue lines in the figure.



(a) Time



(b) Memory

Fig. 11: Run time and memory comparison of persistence analyses relative to *Global-CS*.

VII. Persistence Analysis is NP-complete

In this section, we show that persistence analysis is NP-complete. The *persistence problem* is defined as follows: given a control-flow graph $G = (V, E, i)$, a designated memory block b, and a cache size k, is there a path through G that yields an access trace that results in more than one miss upon accesses to block b in a fully-associative LRU cache of size k?

Theorem VII.1. *The* persistence problem *is NP-complete.*

Proof. First, we show that the problem is indeed in NP. To this end, we show that if there is a witness path π that shows that block b is not persistent, then there is also a *short* witness path π', i. e., a witness path of length polynomial in the size of the control-flow graph G. This proves that the problem is in NP, because a non-deterministic algorithm could first guess and then verify this witness path in polynomial time.

Let π be an arbitrary witness path through G containing at least two accesses to block b that are misses in an LRU cache of size k. Then π can be decomposed as follows: $\pi = \mathit{pre} \circ (s_i, b, s_{i+1}) \circ \mathit{mid} \circ (s_j, b, s_{j+1}) \circ \mathit{post}$, where the transition (s_j, b, s_{j+1}) corresponds to the *second miss* to b among all accesses to b in π, and *mid* does not contain accesses to b, i. e., the transition (s_i, b, s_{i+1}) corresponds to the last access to b before the second miss to b in π.

Clearly, the suffix *post* can be removed from π, and the resulting path is still a witness path. Next, we argue that *mid* can be replaced by mid', such that $|\mathit{mid}'| < |V| \cdot |E|$, maintaining that the subsequent transition (s_j, b, s_{j+1}) results in a miss: To this end, *mid* is further decomposed into $\mathit{mid} = \mathit{mid}_1 \circ \mathit{mid}_2 \circ \cdots \circ \mathit{mid}_n$, where each mid_i starts with

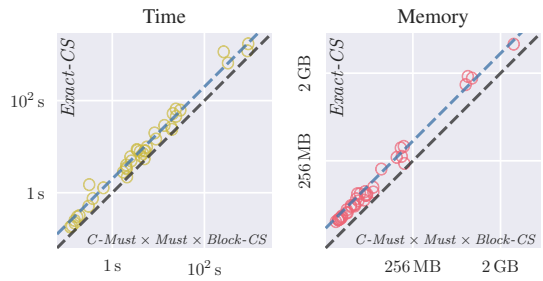
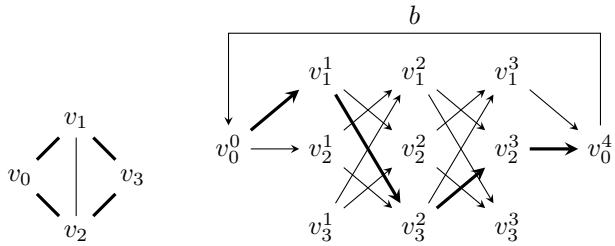


Fig. 12: Run time and memory comparison of *Exact-CS* and *C-Must* × *Must* × *Block-CS*.



(a) Graph with Hamiltonian circuit (thick).

(b) Control flow graph obtained by the reduction. Edge labels not shown, except for the back edge labeled b, which is to be classified. The thick path corresponds to the Hamiltonian circuit of the graph in (a).

Fig. 13: Reduction from the Hamiltonian circuit problem.

an access to a memory block that was not accessed previously in *mid*. Thus, the number of subpaths n is the number of distinct memory blocks accessed on the path *mid*. Clearly, $n < |E|$. Each mid_i can be replaced by a mid'_i, such that $|mid'_i| \leq |V|$: Such a mid'_i can be obtained by keeping the first transition of mid_i and then removing from mid_i any cycles, i. e., subpaths starting and ending in the same node. By construction, $mid' = mid'_1 \circ mid'_2 \circ \cdots \circ mid'_n$ does not contain accesses to b and consists of accesses to at least as many distinct memory blocks as *mid*. Finally, *pre* can be replaced by the shortest path pre' in G from the initial location i to s_i. Clearly, $|pre'| \leq |V|$. Also, the first access to b in $pre' \circ (s_i, b, s_{i+1})$, which must exist due to the final transition (s_i, b, s_{i+1}), results in a miss.

Thus, the path $\pi' = pre' \circ (s_i, b, s_{i+1}) \circ mid' \circ (s_j, b, s_{j+1})$ is also a witness to the fact that b is not persistent, and its length is bounded by $|V| + |V| \cdot |E| + 2$, i. e., it is polynomial in the size of the control-flow graph G.

Now we show that the persistence problem is NP-hard. This part of the proof is analogous to Touzeau et al.'s proof that LRU must analysis is NP-hard [28].

We reduce the Hamiltonian circuit problem to the persistence problem. Let (V, E) be a graph, let $n = |V|$, $V = \{v_0, \ldots, v_{n-1}\}$. We construct a control-flow graph $G = (V', E', i)$ for cache persistence analysis as follows:

- Create two copies v_0^0 and v_0^n of v_0 in V'.
- For each v_i, $i \geq 1$, create $|V| - 1 = n - 1$ copies v_i^j, $1 \leq j < n$ in V'. This arranges these vertices in layers indexed by j.
- For each pair v_j^l, $v_{j'}^{l+1}$ of nodes in consecutive layers, create an edge in E', labeled by the address j', if and only if there is an edge (j, j') in E.
- The initial control location i is v_0^0.

See Figure 13 for an example. There is a Hamiltonian circuit in (V, E) if and only if there is a path in G from v_0^0 to v_0^n such that no edge label is repeated, thus if and only if there exists a path from v_0^0 to v_0^n with at least n distinct edge labels.

Now assume an edge going from v_0^n back to v_0^0 labeled with the fresh memory block b. This memory block b is the one to classify. For cache size n there exists a path resulting in two or more misses to b if and only if there is a path from v_0^0 to v_0^n with at least n distinct edge labels, corresponding to a Hamiltonian circuit in the graph (V, E). ■

It is important to point out that the above NP-hardness proof critically relies on the cache size being an input parameter. In fact, it turns out to be possible to devise a persistence analysis that is exponential in the cache size but polynomial in the size of the control-flow graph [39] based on recent results in theoretical computer science [40].

VIII. Conclusions and Future Work

We have shown that it is possible to perform exact cache persistence analysis for caches with LRU replacement at a reasonable analysis cost. To this end, we introduced a sequence of exact abstractions, exploiting monotonicity properties inherent to LRU replacement; followed by an efficient implementation based on zero-suppressed binary decision diagrams (ZDDs). In addition, we introduced novel techniques to efficiently deal with uncertainty arising in the context of data cache analysis.

The motto of our paper could be: "in practice, theory and practice are different", as the following findings demonstrate: Our experimental evaluation reveals that the new exact analysis is only moderately more costly than existing inexact analyses; in particular, in our experiments it does not exhibit exponential complexity in terms of the input size. This is in spite of the fact that its worst-case complexity is indeed exponential and that persistence analysis is NP-hard, as we show in Section VII. Similarly, while even the most precise existing persistence analyses are not exact in theory, our experiments show that they are close to exact in practice.

Acknowledgements

This work was supported by the Deutsche Forschungsgemeinschaft as part of the project PEP – 289264719. We thank the anonymous reviewers for their helpful comments.

IX. Appendix

Theorem IX.1 (Exactness of Persistence Analysis)**.** *Let $\mathcal{P}$ be a persistence analysis over a finite abstract domain $\widehat{C}_{\mathcal{P}}$ that satisfies the equations of Theorem III.2. Furthermore let $\mathcal{Q}$ be a persistence analysis over a finite abstract domain $\widehat{C}_{\mathcal{Q}}$ defined relative to $\mathcal{P}$ by an abstraction function $\alpha_{\mathcal{Q}}$. The persistence analysis $\mathcal{Q}$ is* exact *if:*

$$\forall T, b : \alpha_{\mathcal{Q}}\big(\widehat{update}_{\mathcal{P}}(T, b)\big) = \widehat{update}_{\mathcal{Q}}(\alpha_{\mathcal{Q}}(T), b)$$
$$\forall I, T_i : \alpha_{\mathcal{Q}}\Big(\bigsqcup_{i\in I}{}_{\mathcal{P}}\, T_i\Big) = \bigsqcup_{i\in I}{}_{\mathcal{Q}}\, \alpha_{\mathcal{Q}}(T_i)$$

and the abstraction preserves the persistence classification:

$$\forall T, b : \widehat{persistent}_{\mathcal{P}}(T, b) \leftrightarrow \widehat{persistent}_{\mathcal{Q}}(\alpha_{\mathcal{Q}}(T), b)$$

Proof. Composing the abstraction functions $\alpha_{\mathcal{P}}$ and $\alpha_{\mathcal{Q}}$ we obtain the abstraction function $\alpha_{\mathcal{Q}\circ\mathcal{P}} := \alpha_{\mathcal{Q}} \circ \alpha_{\mathcal{P}}$ that relates the persistence analysis $\mathcal{Q}$ directly with the concrete trace semantics. If we show that the equations of Theorem III.2 are satisfied for $\alpha_{\mathcal{Q}\circ\mathcal{P}}$, then the exactness of $\mathcal{Q}$ follows.

1) $\forall T, b : \alpha_{\mathcal{Q}\circ\mathcal{P}}(update(T, b)) = \widehat{update}_{\mathcal{Q}}(\alpha_{\mathcal{Q}\circ\mathcal{P}}(T), b)$
2) $\forall I, T_i : \alpha_{\mathcal{Q}\circ\mathcal{P}}\big(\bigcup_{i\in I} T_i\big) = \bigsqcup_{i\in I}{}_{\mathcal{Q}}\, \alpha_{\mathcal{Q}\circ\mathcal{P}}(T_i)$
3) $\forall T, b : persistent(T, b) \leftrightarrow \widehat{persistent}_{\mathcal{Q}}(\alpha_{\mathcal{Q}\circ\mathcal{P}}(T), b)$

The proof is straightforward and follows immediately from the equations of Theorem III.2 for $\mathcal{P}$ and the premises of this theorem. ■

Theorem III.3 (Exactness of *Exact-CS$_0$*)**.** *The persistence analysis Exact-CS$_0$ is exact in the sense of Definition III.1.*

Proof. It suffices to prove the equations of Theorem III.2.

1) $\forall T, b : \alpha_0(update(T, b)) = \widehat{update}_0(\alpha_0(T), b)$

   Let T be a set of traces and $b \in \mathcal{B}$. By unfolding definitions, the term $\alpha_0(update(T, b))$ reduces to:

$$\lambda b'.\ \bigcup_{\tau\in T} \begin{cases} \{\{b\}\} & \text{if } b' = b \\ \{s \cup \{b\} \mid s \in \alpha_0(\{\tau\})(b')\} & \text{otherwise} \end{cases}$$

   Next, the union over all traces can be drawn in:

$$\lambda b'.\ \begin{cases} \{\{b\}\} & \text{if } b' = b \\ \big\{s \cup \{b\} \mid s \in \bigcup_{\tau\in T} \alpha_0(\{\tau\})(b')\big\} & \text{otherwise} \end{cases}$$

   This is equivalent to $\widehat{update}_0(\alpha_0(T), b)$ by definition.

2) $\forall T_i : \alpha_0\big(\bigcup_{i\in\mathbb{N}_0} T_i\big) = \bigsqcup_{i\in\mathbb{N}_0}{}_0\, \alpha_0(T_i)$

   The claim follows trivially from the definition of α_0.

3) $\forall T, b : persistent(T, b) \leftrightarrow \widehat{persistent}_0(\alpha_0(T), b)$

   Let T be an arbitrary set of traces and $b \in \mathcal{B}$. The left hand side is equivalent to $\forall \tau \in T : persistent(\tau, b)$ by definition while the right hand side can be reformulated as $\forall \tau \in T : \max\{|s| \mid s \in \alpha_0(\tau)(b)\} \leq k$. It is therefore sufficient to apply Lemma IX.4 on all $\tau \in T$. ■

Lemma IX.2.

$$\forall \tau, b : (\forall 0 \leq i < |\tau| : \tau_i \neq b) \rightarrow \alpha_0(\tau)(b) = \emptyset$$

Proof. Let τ be a trace and $b \in \mathcal{B}$. We are always in the second case of $\widehat{update}_0$ and propagate the initial information $\alpha_0(\epsilon)(b) = \emptyset$. ■

Lemma IX.3.

$$\forall \tau, b : (\exists 0 \leq i < |\tau| : \tau_i = b) \rightarrow \alpha_0(\tau)(b) = \{CS(\tau, b)\}$$

Proof. Let τ be a trace, $b \in \mathcal{B}$ and $m := \max\{0 \leq i < |\tau| \mid \tau_i = b\}$. We know that $\alpha_0(\tau_0 \circ \cdots \circ \tau_m)(b) = \{\{b\}\}$ by the definition of $\widehat{update}_0$. Furthermore, $\{CS(\tau, b)\} = \{t_j \mid j \geq m\}$. Now, all upcoming updates just add some block in $CS(\tau, b)$ to $\{b\}$ and the claim follows. ■

Lemma IX.4.

$$\forall \tau, b : persistent(\tau, b) \leftrightarrow \widehat{persistent}_0(\alpha_0(\{\tau\}), b)$$

Proof. Let τ be a trace and $b \in \mathcal{B}$. The left hand side reduces to $(\exists 0 \leq i < |\tau| : \tau_i = b) \rightarrow |CS(\tau, b)| \leq k$ and the right hand side to $\max\{|s| \mid s \in \alpha_0(\{\tau\})(b)\} \leq k$ by unfolding all definitions. Applying Lemmas IX.2 and IX.3 completes the proof. ■

Theorem III.4 (Exactness of *Exact-CS$_\uparrow$*)**.** *The persistence analysis Exact-CS$_\uparrow$ is exact in the sense of Definition III.1.*

Proof. It suffices to prove the equations of Theorem IX.1.

1) $\forall T, b : \alpha_\uparrow\big(\widehat{update}_0(T, b)\big) = \widehat{update}_\uparrow(\alpha_\uparrow(T), b)$

   Let $T \in \alpha_0$ be an abstract trace and $b \in \mathcal{B}$. We show that both functions agree on all $b' \in \mathcal{B}$. The case $b' = b$ is trivial.
   If $b' \neq b$, the left hand side $\alpha_\uparrow\big(\widehat{update}_0(T, b)\big)(b')$ reduces to:

$$maxSet(\{s \cup \{b\} \mid s \in T(b')\})$$

   while the right hand side $\widehat{update}_\uparrow\big(\alpha_\uparrow(T), b\big)(b')$ reduces to:

$$maxSet(\{s \cup \{b\} \mid s \in maxSet(T(b'))\})$$

   The missing step is closed by Lemma IX.5.

2) $\forall I, T_i : \alpha_\uparrow\big(\bigsqcup_{i\in I}{}_0\, T_i\big) = \bigsqcup_{i\in I}{}_\uparrow\, \alpha_\uparrow(T_i)$

   Let $I \subseteq \mathbb{N}_0$ and $T_i \in \alpha_0$ be arbitrary abstract traces. We show that both functions agree on all $b \in \mathcal{B}$. By definition, we can transform $\alpha_\uparrow\big(\bigsqcup_{i\in I}{}_0\, T_i\big)(b)$ to $maxSet\big(\bigcup_{i\in I} T_i(b)\big)$. This is equal to $maxSet\big(\bigcup_{i\in I} maxSet(T_i(b))\big)$ by Lemma IX.6 which is equal to $\big(\bigsqcup_{i\in I}{}_\uparrow\, \alpha_\uparrow(T_i)\big)(b)$ by definition.

3) $\forall T, b : \widehat{persistent}_0(T, b) \leftrightarrow \widehat{persistent}_\uparrow(\alpha_\uparrow(T), b)$

   Let $T \in \alpha_0$ be an abstract trace and $b \in \mathcal{B}$. The claim is

proven with Lemma IX.7 after unfolding all definitions. ■

Lemma IX.5.

$$\forall A, b : \mathit{maxSet}(\{s \cup \{b\} \mid s \in A\}) = \mathit{maxSet}(\{s \cup \{b\} \mid s \in \mathit{maxSet}(A)\})$$

Proof. Let A be a set and $b \in \mathcal{B}$. We prove the claim by showing mutual inclusion.

"$\subseteq$": Let $t \in \mathit{maxSet}(\{s \cup \{b\} \mid s \in A\})$, i. e., $t = s \cup \{b\}$ for some $s \in A$. We have to prove two statements:

1) $t \in \{s \cup \{b\} \mid s \in \mathit{maxSet}(A)\}$
   In the case $t \in A$, we claim that $t \in \mathit{maxSet}(A)$. By assuming the contrary, we immediately get a contradiction with $t \in \mathit{maxSet}(\{s \cup \{b\} \mid s \in A\})$.
   If $t \notin A$, i. e., $t = s \uplus \{b\}$, we claim that $s \in \mathit{maxSet}(A)$. Assume the contrary, i. e., there is $s' \in A$, $s \subsetneq s'$. Note that $s' \neq t$ as $t \notin A$. Then, $s' \cup \{b\} \supsetneq t$ contradicts $t \in \mathit{maxSet}(\{s \cup \{b\} \mid s \in A\})$.
2) $\neg\exists s' \in \{s \cup \{b\} \mid s \in \mathit{maxSet}(A)\} : t \subsetneq s'$
   If we assume otherwise, we get a contradiction to $t \in \mathit{maxSet}(\{s \cup \{b\} \mid s \in A\})$.

"$\supseteq$": Trivial, as $\mathit{maxSet}(A) \subseteq A$. ■

Lemma IX.6.

$$\forall I, A_i : \mathit{maxSet}\Big(\bigcup_{i \in I} A_i\Big) = \mathit{maxSet}\Big(\bigcup_{i \in I} \mathit{maxSet}(A_i)\Big)$$

Proof. Let $I \subseteq \mathbb{N}_0$, A_i be arbitrary sets and $\widehat{A}_i := \mathit{maxSet}(A_i)$. We prove the claim by showing mutual inclusion.

"$\subseteq$": Let $s \in \mathit{maxSet}\big(\bigcup_{i \in I} A_i\big)$, i. e., $s \in \bigcup_{i \in I} A_i$ and $\neg\exists s' \in \bigcup_{i \in I} A_i : s \subsetneq s'$. It is easy to see that $s \in \widehat{A}_j$ for some $j \in I$, and hence, $s \in \bigcup_{i \in I} \widehat{A}_i$. As $\big(\bigcup_{i \in I} \widehat{A}_i\big) \subseteq \big(\bigcup_{i \in I} A_i\big)$, we get that $s \in \mathit{maxSet}\big(\bigcup_{i \in I} \widehat{A}_i\big)$.

"$\supseteq$": Trivial, as $\big(\bigcup_{i \in I} \widehat{A}_i\big) \subseteq \big(\bigcup_{i \in I} A_i\big)$. ■

Lemma IX.7.

$$\forall A : \max\{|x| \mid x \in A\} = \max\{|x| \mid x \in \mathit{maxSet}(A)\}$$

Proof. Let A be a set. We prove the equality by showing mutual less or equal relations.

"$\leq$": Let $n := \max\{|x| \mid x \in A\}$ and $m := \max\{|x| \mid x \in \mathit{maxSet}(A)\}$. Assume that $n > m$ and there is $a \in A$ with $|a| = n$. There cannot exist $b \in A$ with $a \subsetneq b$ because otherwise $|b| > n$ is a contradiction. But then, $a \in \mathit{maxSet}(A)$ which contradicts $|a| = n > m$.

"$\geq$": Trivial, as $\mathit{maxSet}(A) \subseteq A$. ■

Theorem III.5 (Exactness of $\mathit{Exact\text{-}CS}_{\leq k}$)**.** *The persistence analysis $\mathit{Exact\text{-}CS}_{\leq k}$ is exact in the sense of Definition III.1.*

Proof. With the proof of Theorem III.4, we have already shown that the domain of $\alpha_\uparrow$ satisfies the equations of Theorem III.2. As $\alpha_{\leq k}$ is defined relative to this domain, is is sufficient to prove the equations of Theorem IX.1.

1) $\forall T, b : \alpha_{\leq k}\big(\widehat{\mathit{update}}_\uparrow(T, b)\big) = \widehat{\mathit{update}}_{\leq k}(\alpha_{\leq k}(T), b)$
   Let $T \in \alpha_\uparrow$ be an abstract trace and $b \in \mathcal{B}$. We show that both functions agree on all $b' \in \mathcal{B}$. The case $b' = b$ is trivial.
   If $b' \neq b$, the left hand side $\alpha_{\leq k}(\widehat{\mathit{update}}_\uparrow(T, b))(b')$ reduces to:
   $$\mathit{limit}(\mathit{maxSet}(\{s \cup \{b\} \mid s \in T(b')\})) \tag{11}$$
   We have to show equality with:
   $$\mathit{limit}(\mathit{maxSet}(\{s \cup \{b\} \mid s \in \mathit{limit}(T(b'))\})) \tag{12}$$
   which is the right hand side $\widehat{\mathit{update}}_{\leq k}(\alpha_{\leq k}(T), b)(b')$ where all definitions are unfolded.
   We distinguish the two cases of $\mathit{limit}(T(b'))$. The case $\mathit{limit}(T(b')) = T(b')$ is trivial. Therefore, assume $\exists s \in T(b') : |s| > k$. Equation 12 reduces to $\{\mathcal{B}\}$ by definition as $\mathit{limit}(T(b')) = \{\mathcal{B}\}$. For Equation 11, the union with $\{b\}$ cannot decrease the cardinality and we get $\exists s' \in \{s \cup \{b\} \mid s \in T(b')\} : |s'| > k$. The proof is completed by applying Lemma IX.7 and the definition of limit which prove that (11) is equal to $\{\mathcal{B}\}$, too.
2) $\forall I, T_i : \alpha_{\leq k}\big(\bigsqcup_{i \in I}^{\uparrow} T_i\big) = \bigsqcup_{i \in I}^{\leq k} \alpha_{\leq k}(T_i)$
   Let $I \subseteq \mathbb{N}_0$ and $T_i \in \alpha_\uparrow$ be arbitrary abstract traces. We show that both functions agree on all $b \in \mathcal{B}$. By definition, we can transform $\alpha_{\leq k}\big(\bigsqcup_{i \in I}^{\uparrow} T_i\big)(b)$ to $\mathit{limit}\big(\mathit{maxSet}\big(\bigcup_{i \in I} T_i(b)\big)\big)$. This is equal to $\mathit{maxSet}\big(\bigcup_{i \in I} \mathit{limit}(T_i(b))\big)$ by Lemma IX.8 which is equal to $\big(\bigsqcup_{i \in I}^{\leq k} \alpha_{\leq k}(T_i)\big)(b)$ by definition.
3) $\forall T, b : \widehat{\mathit{persistent}}_\uparrow(T, b) \leftrightarrow \widehat{\mathit{persistent}}_{\leq k}(\alpha_{\leq k}(T), b)$
   Let $T \in \alpha_\uparrow$ be an abstract trace and $b \in \mathcal{B}$. The claim is proven with Lemma IX.9 after unfolding all definitions. ■

Lemma IX.8.

$$\forall I, A_i : \mathit{limit}\Big(\mathit{maxSet}\Big(\bigcup_{i \in I} A_i\Big)\Big) = \mathit{maxSet}\Big(\bigcup_{i \in I} \mathit{limit}(A_i)\Big)$$

Proof. Let $I \subseteq \mathbb{N}_0$ and A_i be arbitrary sets. We prove the claim by showing mutual inclusion.

"$\subseteq$": Let $s \in \mathit{limit}\big(\mathit{maxSet}\big(\bigcup_{i \in I} A_i\big)\big)$.

- Assume $\exists s' \in \mathit{maxSet}\big(\bigcup_{i \in I} A_i\big) : |s'| > k$, i. e., $s = \mathcal{B}$. W. l. o. g. let $s' \in A_j$ for some $j \in I$. Then, $\mathit{limit}(A_j) = \{\mathcal{B}\}$ and $s \in \{\mathcal{B}\} = \mathit{maxSet}\big(\bigcup_{i \in I} \mathit{limit}(A_i)\big)$.
- Otherwise, $s \in \mathit{maxSet}\big(\bigcup_{i \in I} A_i\big)$ and $\neg\exists s' \in \mathit{maxSet}\big(\bigcup_{i \in I} A_i\big) : |s'| > k$. Therefore, $\forall i \in I : \neg\exists s' \in A_i : |s'| > k$, i. e., $\mathit{limit}(A_i) = A_i$.

"$\supseteq$": Let $s \in \mathit{maxSet}\big(\bigcup_{i \in I} \mathit{limit}(A_i)\big)$. If $\forall i \in I : \neg\exists s' \in A_i : |s'| > k$, then $s \in \mathit{maxSet}\big(\bigcup_{i \in I} A_i\big)$. Otherwise let w. l. o. g. $\mathit{limit}(A_j) = \{\mathcal{B}\}$ for some $j \in I$. Then, $s = \mathcal{B} \in \mathit{limit}\big(\mathit{maxSet}\big(\bigcup_{i \in I} A_i\big)\big)$. ■

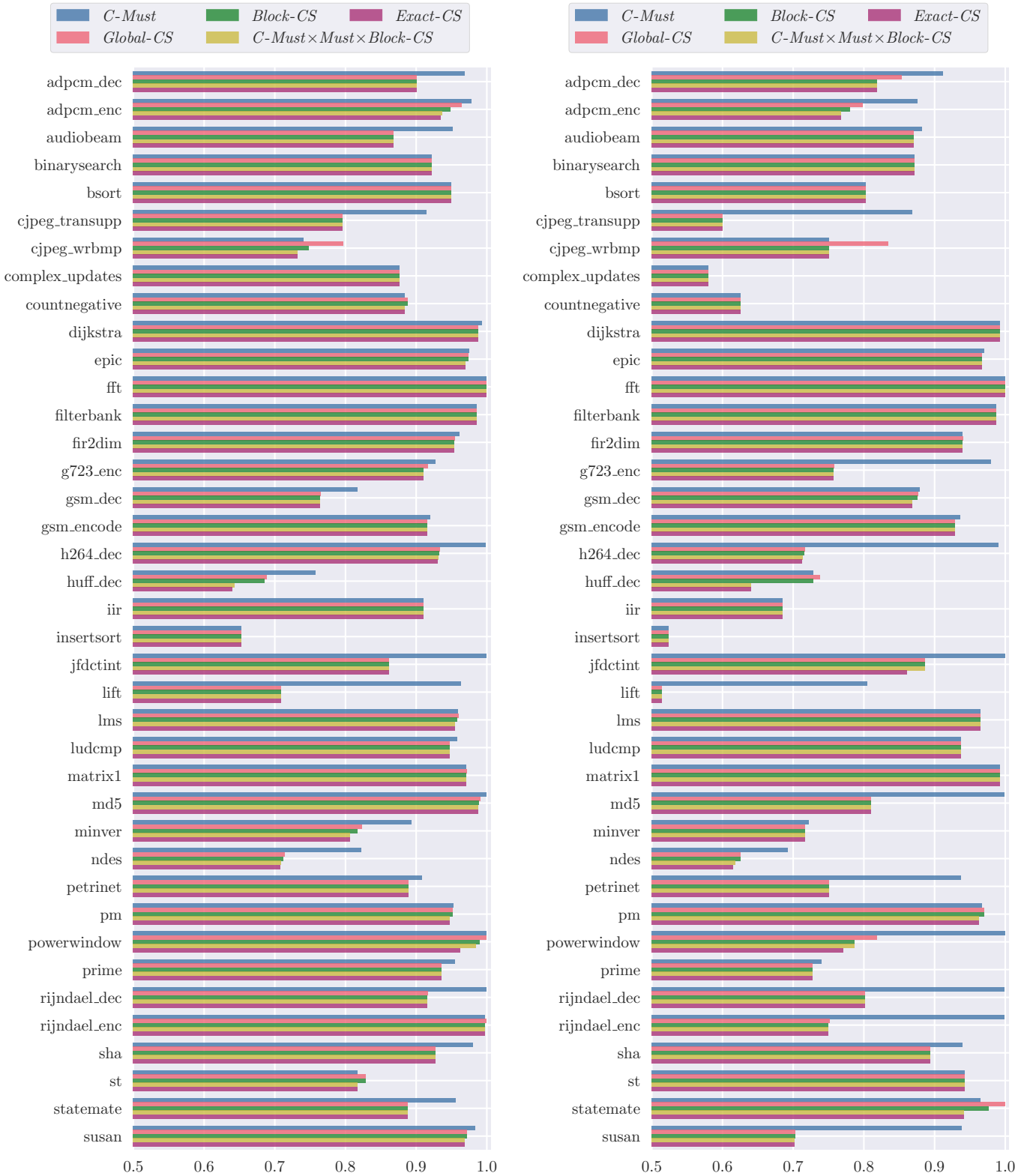
Lemma IX.9.

$$\forall A, k : \max\{|x| \mid x \in A\} \leq k \leftrightarrow \max\{|x| \mid x \in \mathit{limit}(A)\} \leq k$$

Proof. Let A be a set and $k \in \mathbb{N}$. We prove the claim by showing mutual implication.

"$\rightarrow$": Let $\max\{|x| \mid x \in A\} \leq k$, i. e., $\neg\exists y \in A : |y| > k$. But then we have $\mathit{limit}(A) = A$.

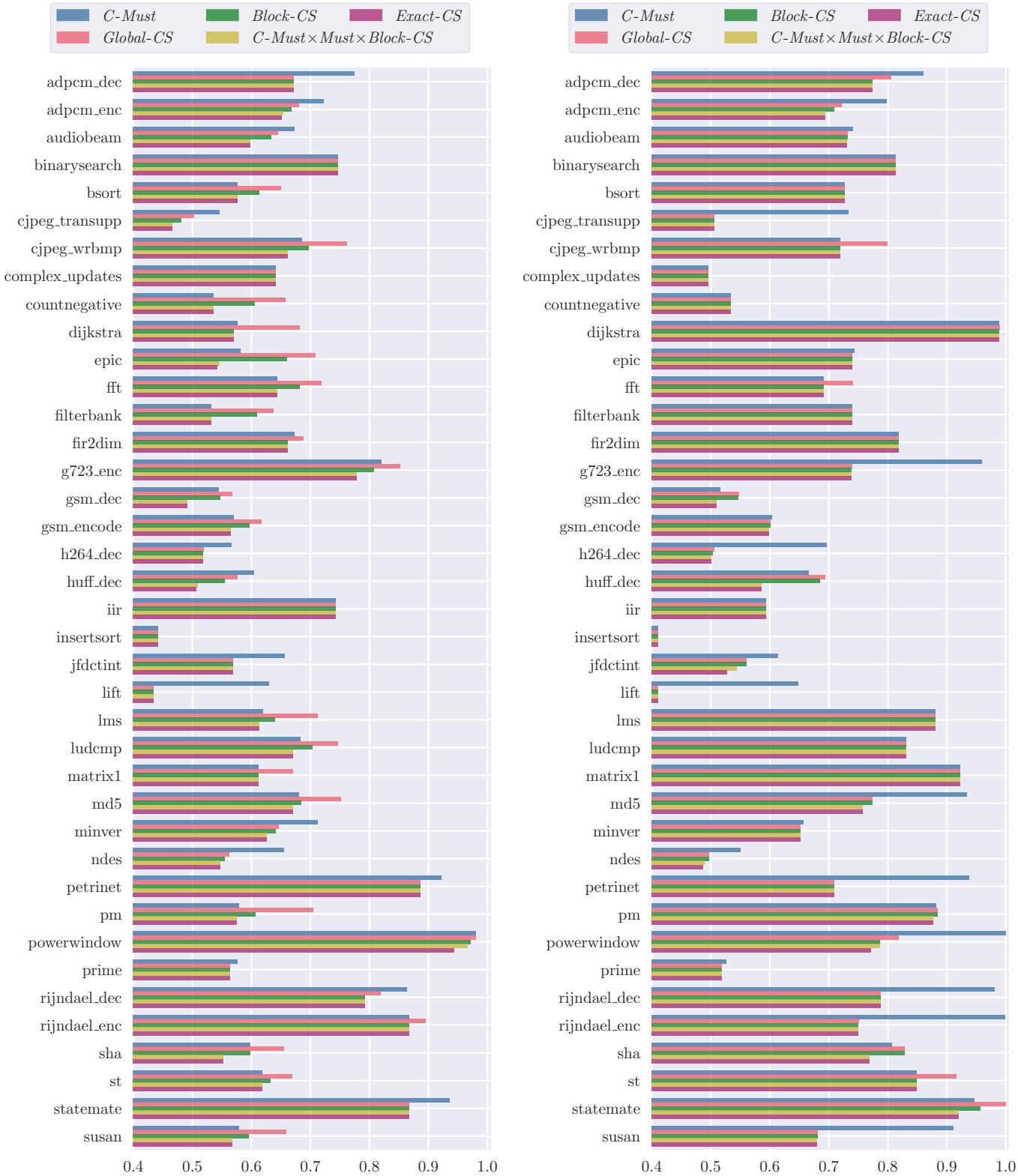
"$\leftarrow$": Let $\max\{|x| \mid x \in \mathit{limit}(A)\} \leq k$. Assume $\mathit{limit}(A) = \{\mathcal{B}\}$, then $\exists y \in A : |y| > k$, a contradiction. ■



(a) *Without* compiler optimizations.

(b) *With* compiler optimizations.

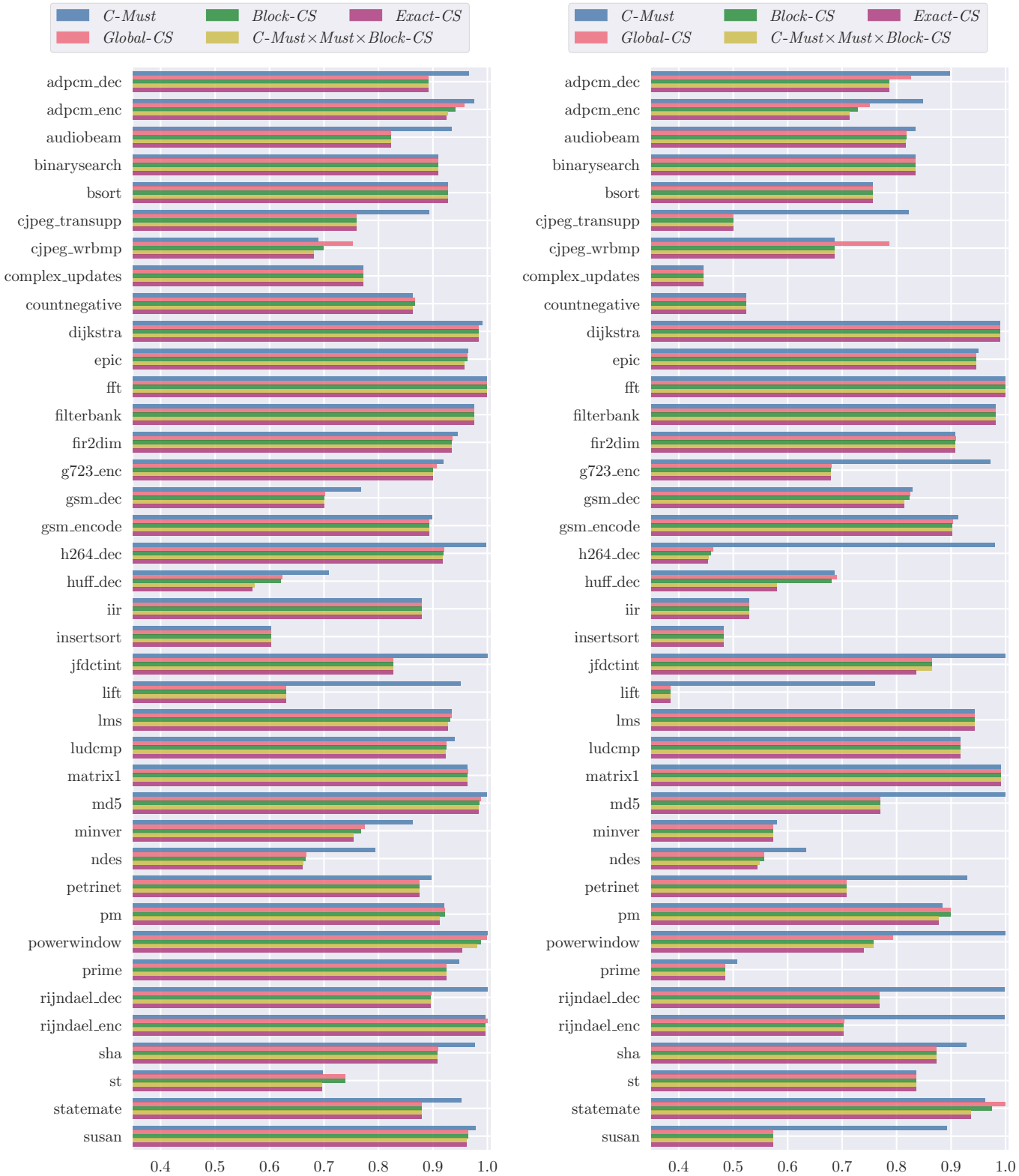
Fig. 14: WCET ratios of the persistence analyses compared with performing no persistence analysis at all. Cache configuration: 32 cache sets, 8 ways, 16 B line size.



(a) *Without* compiler optimizations.

(b) *With* compiler optimizations.

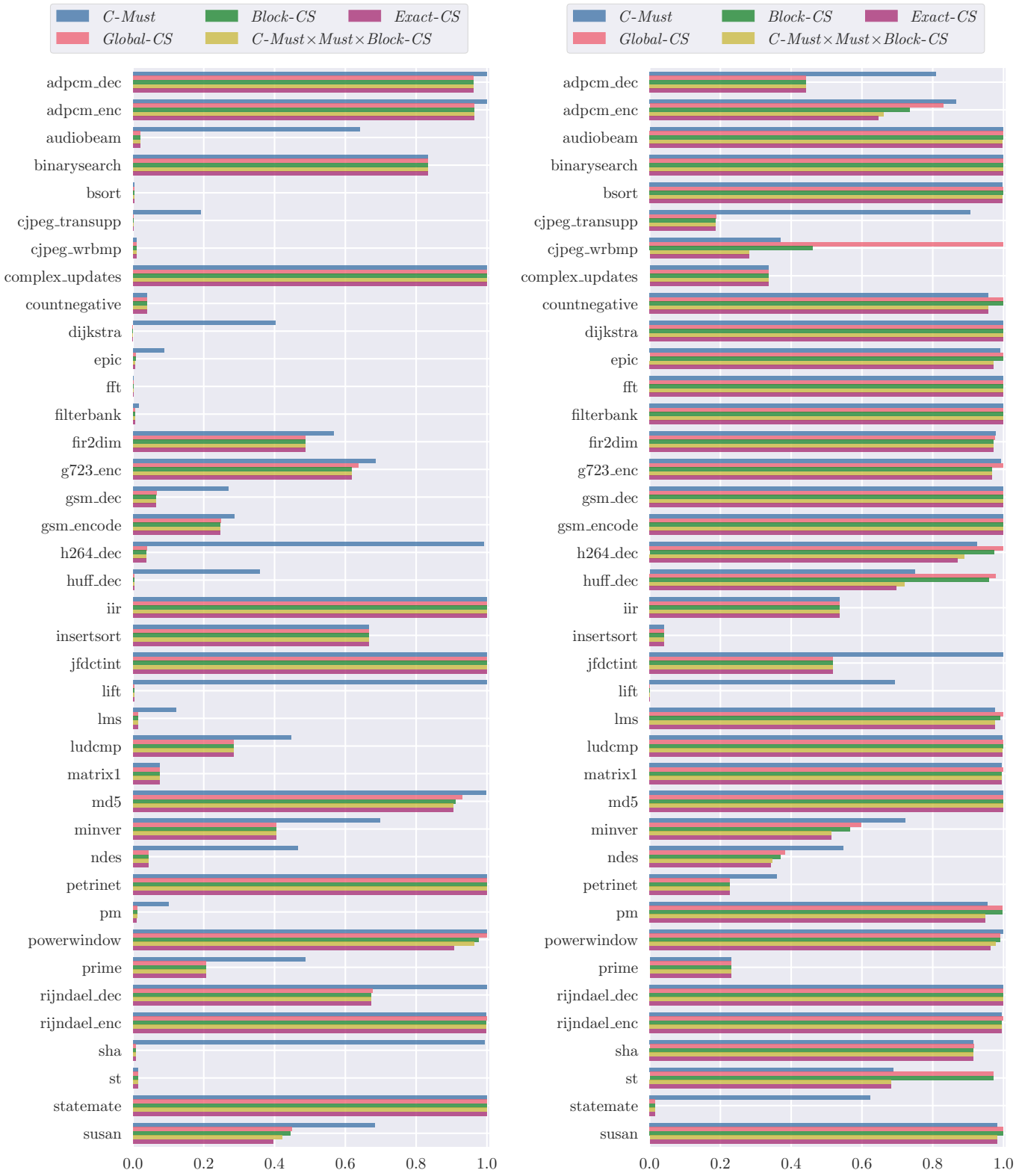
Fig. 15: WCET ratios of the persistence analyses compared with performing no persistence analysis at all. Cache configuration: 32 cache sets, 8 ways, 16 B line size, *loop peeling disabled*.



(a) *Without* compiler optimizations.

(b) *With* compiler optimizations.

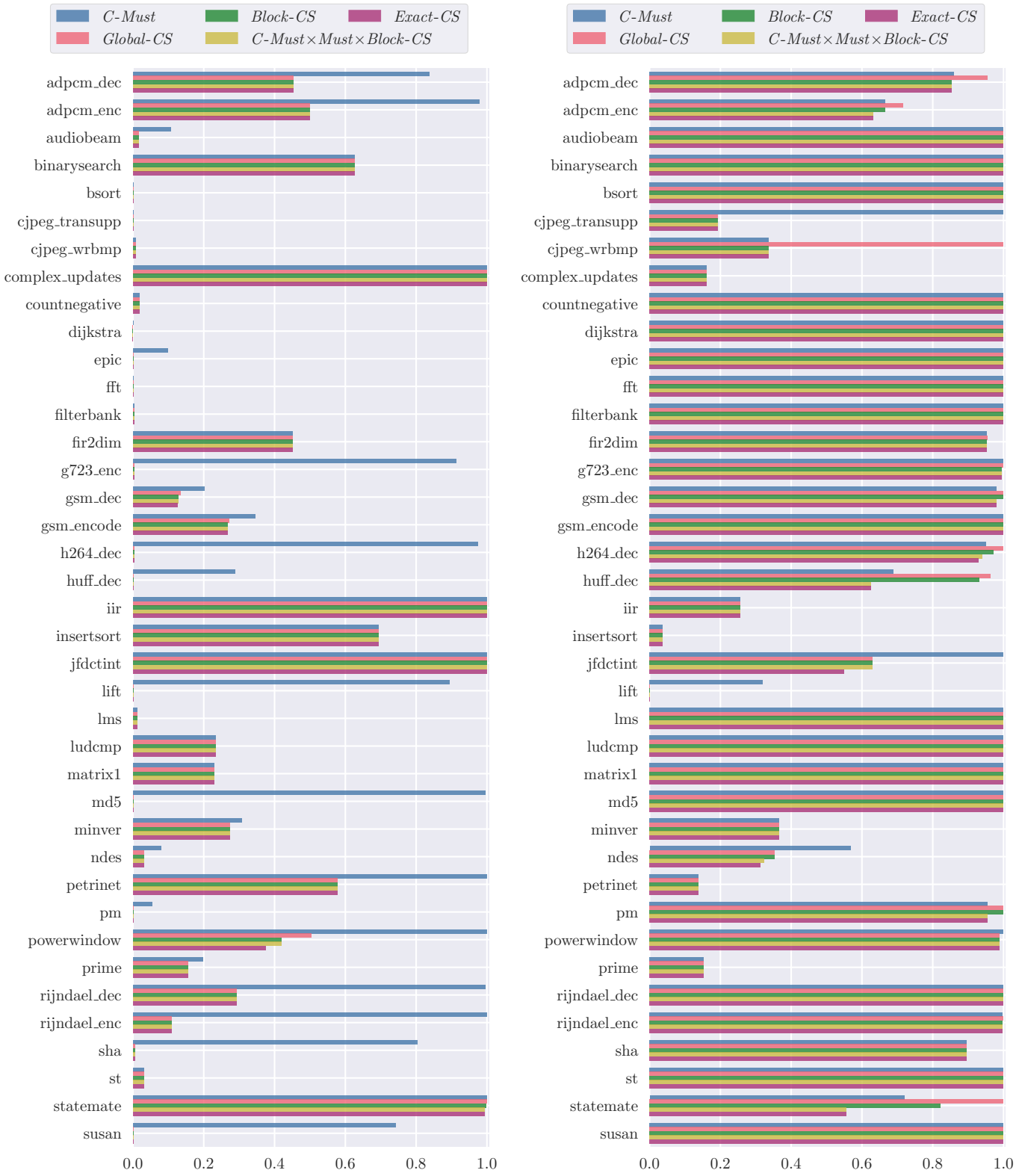
Fig. 16: WCET ratios of the persistence analyses compared with performing no persistence analysis at all. Cache configuration: 32 cache sets, 8 ways, 16 B line size, *higher latency of 100 cycles to deliver the first word*.



(a) *Instruction* cache misses.

(b) *Data* cache misses.

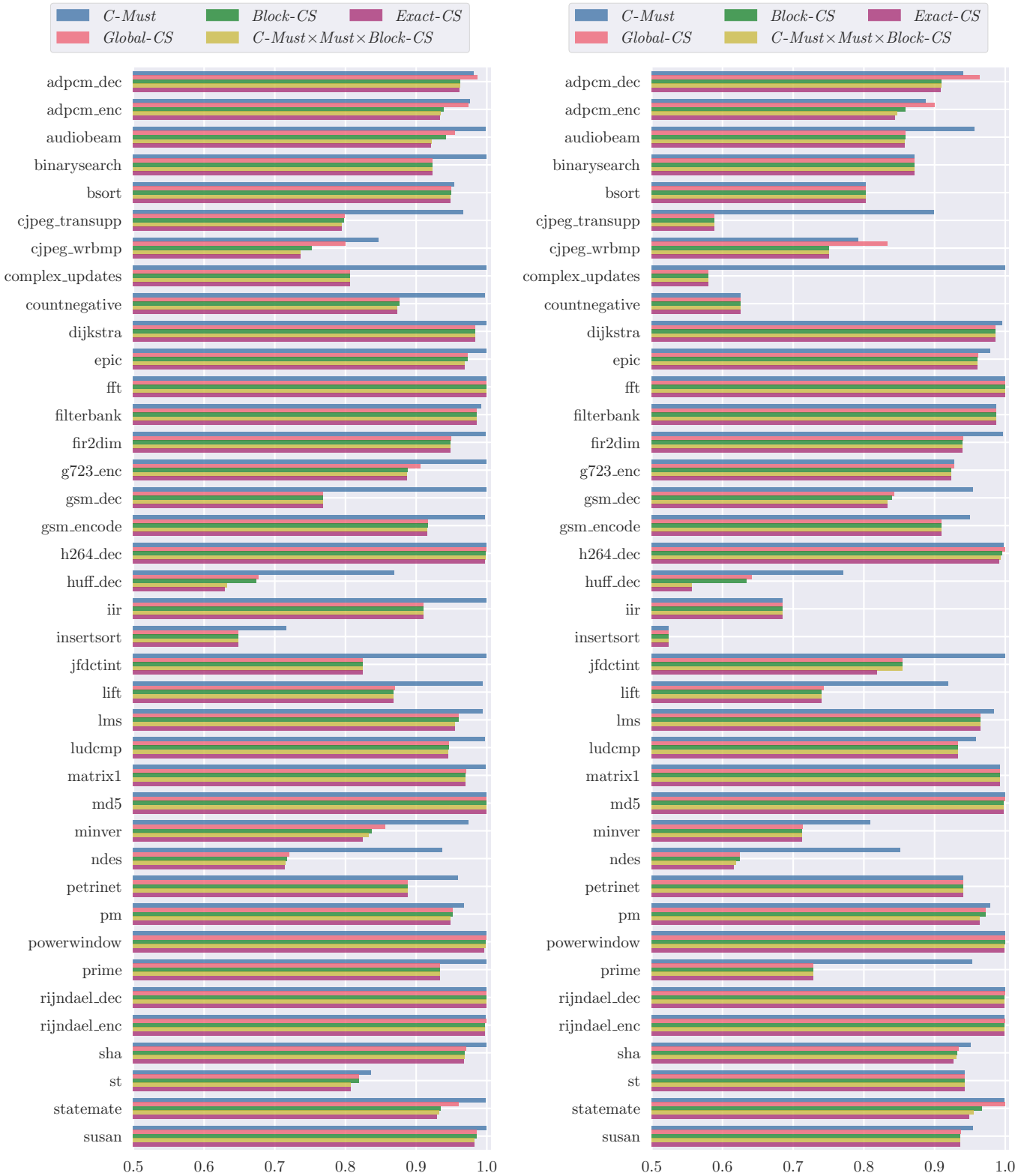
Fig. 17: Ratios of the maximum number of instruction and data cache misses of the persistence analyses compared with performing no persistence analysis at all.
Cache configuration: 32 cache sets, 8 ways, 16 B line size, *without* compiler optimizations.



(a) *Instruction* cache misses.

(b) *Data* cache misses.

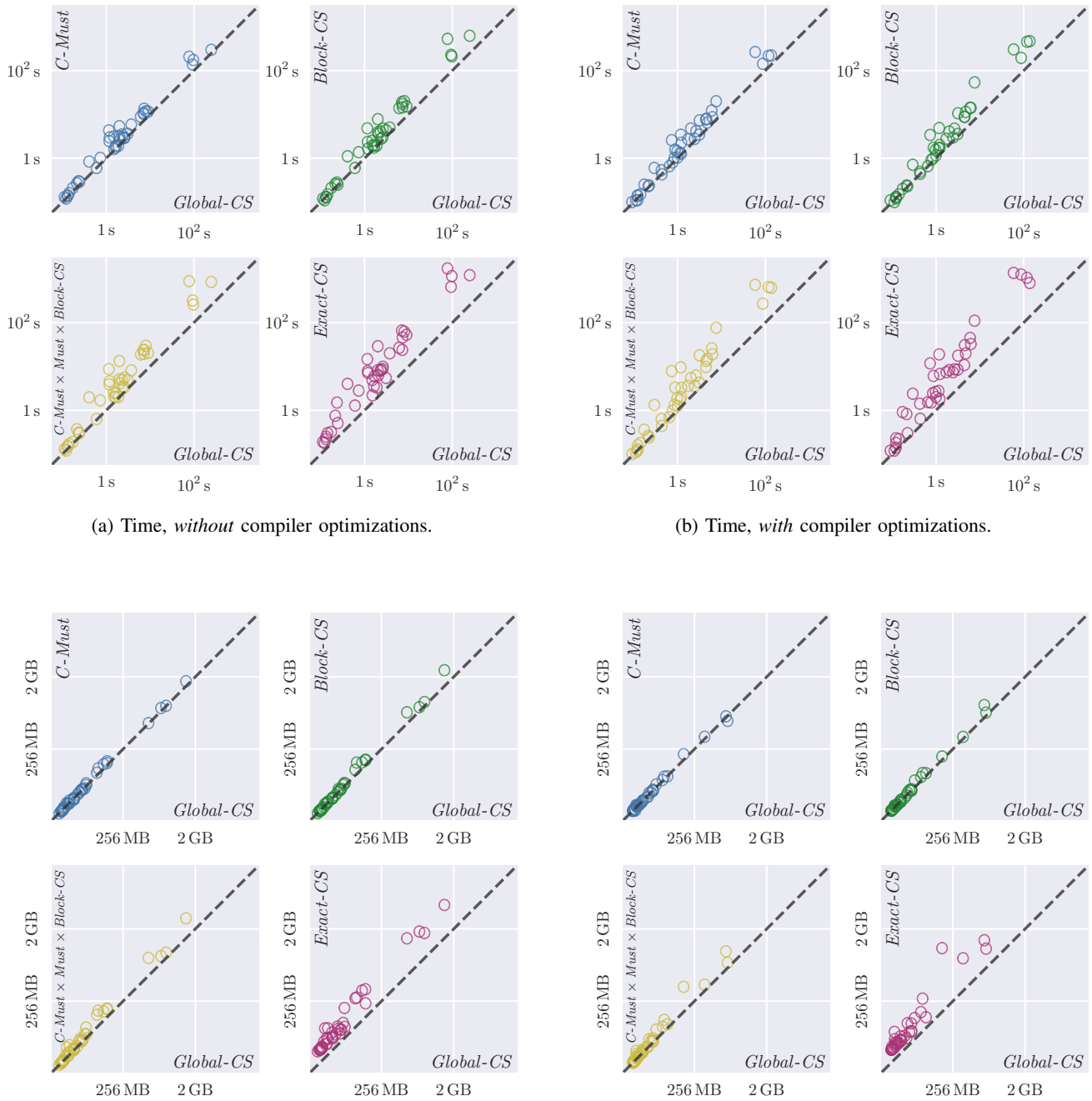
Fig. 18: Ratios of the maximum number of instruction and data cache misses of the persistence analyses compared with performing no persistence analysis at all.
Cache configuration: 32 cache sets, 8 ways, 16 B line size, *with* compiler optimizations.



(a) *Without* compiler optimizations.

(b) *With* compiler optimizations.

Fig. 19: WCET ratios of the persistence analyses compared with performing no persistence analysis at all. Cache configuration: 8 *cache sets*, 8 ways, 16 B line size.



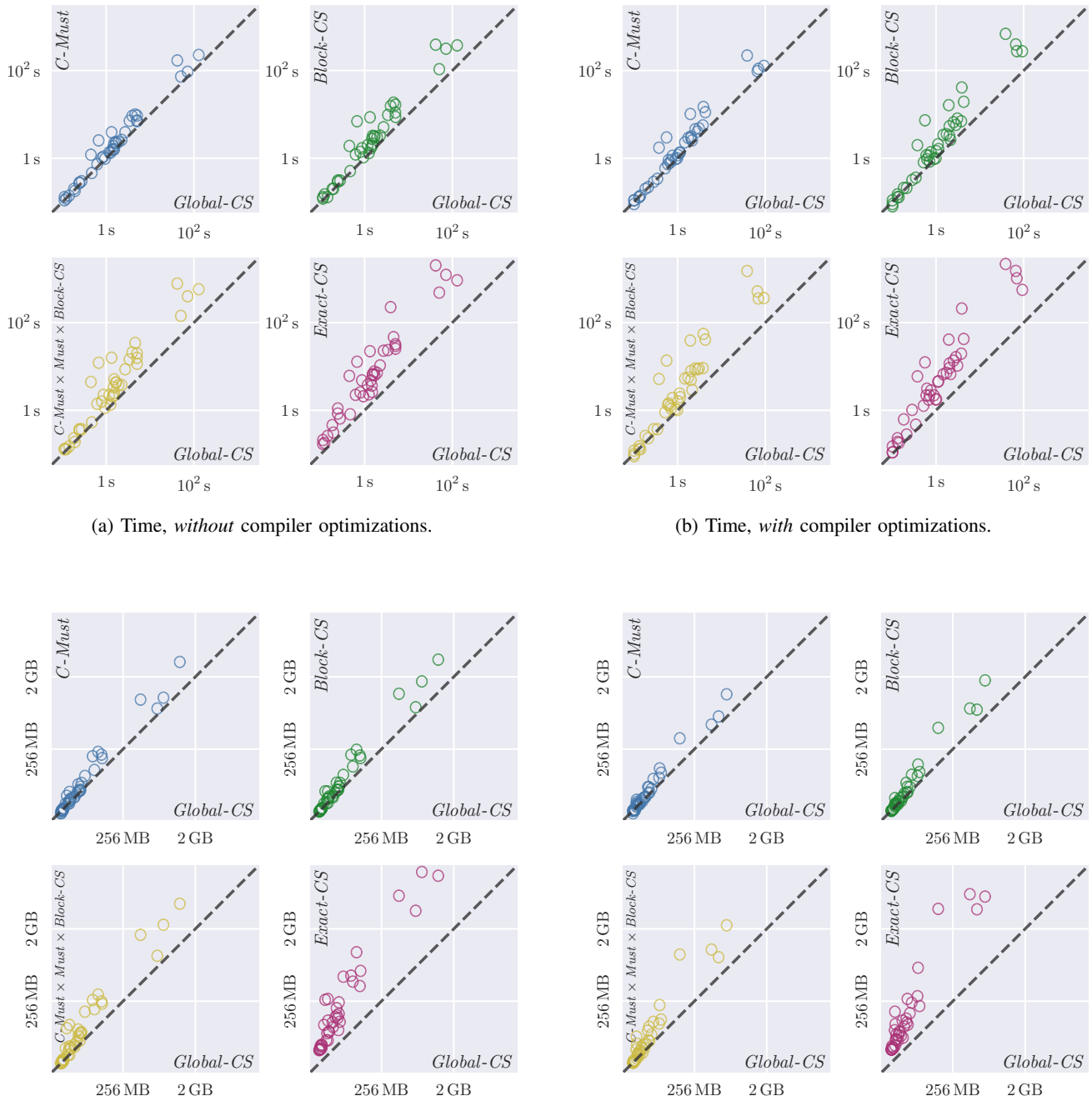
(a) Time, *without* compiler optimizations.

(b) Time, *with* compiler optimizations.

(c) Memory, *without* compiler optimizations.

(d) Memory, *with* compiler optimizations.

Fig. 20: Run time and memory comparison of persistence analyses relative to *Global-CS*.
Cache configuration: 32 cache sets, 8 ways, 16 B line size.



(a) Time, *without* compiler optimizations.

(b) Time, *with* compiler optimizations.

(c) Memory, *without* compiler optimizations.

(d) Memory, *with* compiler optimizations.

Fig. 21: Run time and memory comparison of persistence analyses relative to *Global-CS*.
Cache configuration: 8 *cache sets*, 8 ways, 16 B line size.

References

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, 2008. [Online]. Available: https://doi.org/10.1145/1347375.1347389

[2] M. Lv, N. Guan, J. Reineke, R. Wilhelm, and W. Yi, "A survey on static cache analysis for real-time systems," *Leibniz Transactions on Embedded Systems*, vol. 3, no. 1, pp. 05–1–05:48, 2016. [Online]. Available: https://doi.org/10.4230/LITES-v003-i001-a005

[3] F. Mueller, "Static cache simulation and its applications," Ph.D. dissertation, Florida State University, Tallahassee, United States, 1994. [Online]. Available: https://www.cs.fsu.edu/~whalley/papers/mueller_diss94.pdf

[4] R. D. Arnold, F. Mueller, D. B. Whalley, and M. G. Harmon, "Bounding worst-case instruction cache performance," in *Proceedings of the 15th IEEE Real-Time Systems Symposium, San Juan, Puerto Rico, December 7-9, 1994*, ser. RTSS 1994, 1994, pp. 172–181. [Online]. Available: https://doi.org/10.1109/REAL.1994.342718

[5] R. T. White, C. A. Healy, D. B. Whalley, F. Mueller, and M. G. Harmon, "Timing analysis for data caches and set-associative caches," in *3rd IEEE Real-Time Technology and Applications Symposium, Montreal, Canada, June 9-11, 1997*, ser. RTAS 1997, 1997, pp. 192–202. [Online]. Available: https://doi.org/10.1109/RTTAS.1997.601358

[6] F. Mueller, "Timing analysis for instruction caches," *Real-Time Systems*, vol. 18, no. 2, pp. 217–247, May 2000. [Online]. Available: https://doi.org/10.1023/A:1008145215849

[7] C. Ferdinand, "Cache behavior prediction for real-time systems," Ph.D. dissertation, Saarland University, Saarbrücken, Germany, 1997, iSBN: 3-9307140-31-0. [Online]. Available: https://d-nb.info/953983706

[8] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems," *Real-Time Systems*, vol. 17, no. 2-3, pp. 131–181, Nov. 1999. [Online]. Available: https://doi.org/10.1023/A:1008186323068

[9] C. Ballabriga and H. Casse, "Improving the first-miss computation in set-associative instruction caches," in *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, ser. ECRTS 2008, 2008, pp. 341–350. [Online]. Available: https://doi.org/10.1109/ECRTS.2008.34

[10] C. Cullmann, "Cache persistence analysis: a novel approach," in *Proceedings of the ACM SIGPLAN/SIGBED 2011 conference on Languages, compilers, and tools for embedded systems, Chicago, IL, USA, April 11-14, 2011*, ser. LCTES 2011, 2011, pp. 121–130. [Online]. Available: https://doi.org/10.1145/1967677.1967695

[11] B. K. Huynh, L. Ju, and A. Roychoudhury, "Scope-aware data cache analysis for WCET estimation," in *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS 2011, 2011, pp. 203–212. [Online]. Available: https://doi.org/10.1109/RTAS.2011.27

[12] K. Nagar and Y. N. Srikant, "Interdependent cache analyses for better precision and safety," in *Tenth ACM/IEEE International Conference on Formal Methods and Models for Codesign, Arlington, VA, USA, July 16-17, 2012*, ser. MEMOCODE 2012, 2012, pp. 99–108. [Online]. Available: https://doi.org/10.1109/MEMCOD.2012.6292306

[13] K. Nagar, "Cache analysis for multi-level data caches," Master's thesis, Indian Institute of Science, Bangalore, India, 2012. [Online]. Available: http://clweb.csa.iisc.ac.in/kartik.nagar/thesis.pdf

[14] C. Cullmann, "Cache persistence analysis for embedded real-time systems," Ph.D. dissertation, Saarland University, Saarbrücken, Germany, 2013. [Online]. Available: https://doi.org/10.22028/D291-26418

[15] ——, "Cache persistence analysis: Theory and practice," *ACM Trans. Embedded Comput. Syst.*, vol. 12, no. 1s, pp. 40:1–40:25, 2013. [Online]. Available: https://doi.org/10.1145/2435227.2435236

[16] Z. Zhang and X. D. Koutsoukos, "Improving the precision of abstract interpretation based cache persistence analysis," in *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems, Portland, OR, USA, June 18-19, 2015*, ser. LCTES 2015, 2015, pp. 10:1–10:10. [Online]. Available: https://doi.org/10.1145/2670529.2754967

[17] J. Reineke, "The semantic foundations and a landscape of cache-persistence analyses," *LITES*, vol. 5, no. 1, pp. 03:1–03:52, 2018. [Online]. Available: https://doi.org/10.4230/LITES-v005-i001-a003

[18] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm, "Cache behavior prediction by abstract interpretation," in *Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings*, 1996, pp. 52–66. [Online]. Available: https://doi.org/10.1007/3-540-61739-6_33

[19] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Timing predictability of cache replacement policies," *Real-Time Systems*, vol. 37, no. 2, pp. 99–122, Nov. 2007. [Online]. Available: https://doi.org/10.1007/s11241-007-9032-3

[20] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL 1977. New York, NY, USA: ACM, 1977, pp. 238–252. [Online]. Available: https://doi.org/10.1145/512950.512973

[21] ——, "Systematic design of program analysis frameworks," in *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '79. New York, NY, USA: ACM, 1979, pp. 269–282. [Online]. Available: https://doi.org/10.1145/567752.567778

[22] B. A. Davey and H. A. Priestley, *Introduction to Lattices and Order*, 2nd ed. Cambridge University Press, 2002. [Online]. Available: https://doi.org/10.1017/CBO9780511809088

[23] R. Drechsler and D. Sieling, "Binary decision diagrams in theory and practice," *International Journal on Software Tools for Technology Transfer*, vol. 3, no. 2, pp. 112–136, May 2001. [Online]. Available: https://doi.org/10.1007/s100090100056

[24] S. Minato, "Zero-suppressed BDDs and their applications," *International Journal on Software Tools for Technology Transfer*, vol. 3, no. 2, pp. 156–170, May 2001. [Online]. Available: https://doi.org/10.1007/s100090100038

[25] A. Mishchenko, *Introduction to Zero-Suppressed Decision Diagrams*. Morgan & Claypool, Dec. 2014, ch. 1. [Online]. Available: https://doi.org/10.2200/S00612ED1V01Y201411DCS045

[26] F. Somenzi, *CUDD: CU Decision Diagram Package*, 2018. [Online]. Available: https://add-lib.scce.info/assets/documents/cudd-manual.pdf

[27] Y. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, & Tools for Real-Time Systems (LCT-RTS 1995). La Jolla, California, June 21-22, 1995*, 1995, pp. 88–98. [Online]. Available: https://doi.org/10.1145/216636.216666

[28] V. Touzeau, C. Maïza, D. Monniaux, and J. Reineke, "Fast and exact analysis for LRU caches," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 54:1–54:29, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3290367

[29] T. Lundqvist and P. Stenström, "Timing anomalies in dynamically scheduled microprocessors," in *Proceedings of the 20th IEEE Real-Time Systems Symposium, Phoenix, AZ, USA, December 1-3, 1999*, 1999, pp. 12–21. [Online]. Available: https://doi.org/10.1109/REAL.1999.818824

[30] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A definition and classification of timing anomalies," in *Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Jul. 2006. [Online]. Available: https://doi.org/10.4230/OASIcs.WCET.2006.671

[31] C. Berg, "PLRU cache domino effects," in *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, ser. OpenAccess Series in Informatics (OASIcs), F. Mueller, Ed., vol. 4. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2006. [Online]. Available: https://doi.org/10.4230/OASIcs.WCET.2006.672

[32] G. Gebhard, "Timing anomalies reloaded," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, ser. OpenAccess Series in Informatics (OASIcs), B. Lisper, Ed., vol. 15. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 1–10. [Online]. Available: https://doi.org/10.4230/OASIcs.WCET.2010.1

[33] S. Hahn and J. Reineke, "Design and analysis of SIC: A provably timing-predictable pipelined processor core," in *2018 IEEE Real-Time Systems Symposium, RTSS 2018, Nashville, TN, USA, December 11-14, 2018*, 2018, pp. 469–481. [Online]. Available: https://doi.org/10.1109/RTSS.2018.00060

[34] S. Hahn, "On static execution-time analysis — compositionality, pipeline abstraction, and predictable hardware," Ph.D. dissertation, Saarland University, Saarbrücken, Germany, 2019. [Online]. Available: https://doi.org/10.22028/D291-27991

[35] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sorensen, P. Wägemann, and S. Wegener, "TACLeBench: A benchmark collection to support worst-case execution time research," in *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, 2016, pp. 2:1–2:10. [Online]. Available: https://doi.org/10.4230/OASIcs.WCET.2016.2

[36] R. B. França, D. Favre-Felix, X. Leroy, M. Pantel, and J. Souyris, "Towards Formally Verified Optimizing Compilation in Flight Control Software," in *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, ser. OpenAccess Series in Informatics (OASIcs), P. Lucas, L. Thiele, B. Triquet, T. Ungerer, and R. Wilhelm, Eds., vol. 18. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011, pp. 59–68. [Online]. Available: https://doi.org/10.4230/OASIcs.PPES.2011.59

[37] Micron Technology, Inc., *Automotive DDR SDRAM MT46V32M8, MT46V16M16*, *Available at* https://micron.com/~/media/documents/products/data-sheet/dram/mobile-dram/low-power-dram/lpddr/256mb_x8x16_at_ddr_t66a.pdf.

[38] L. Mauborgne and X. Rival, "Trace partitioning in abstract interpretation based static analyzers," in *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, 2005, pp. 5–20. [Online]. Available: https://doi.org/10.1007/978-3-540-31987-0_2

[39] H. Dell and C. Brand, Personal Communication, 2019.

[40] C. Brand, H. Dell, and T. Husfeldt, "Extensor-coding," in *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, ser. STOC 2018. New York, NY, USA: ACM, 2018, pp. 151–164. [Online]. Available: https://doi.org/10.1145/3188745.3188902