# Fast and exact analysis for LRU caches

Claire Maïza, Valentin Touzeau, David Monniaux, Jan Reineke

# Fast and exact analysis for LRU caches*†

Valentin Touzeau, Claire Maïza, and David Monniaux

Univ. Grenoble Alpes, CNRS, Grenoble INP‡, VERIMAG, 38000 Grenoble, France

Jan Reineke

Saarland University, Saarbrücken, Germany

December 18, 2018

**Abstract**

For applications in worst-case execution time analysis and in security, it is desirable to statically classify memory accesses into those that result in cache hits, and those that result in cache misses. Among cache replacement policies, the least recently used (LRU) policy has been studied the most and is considered to be the most predictable.

The state-of-the-art in LRU cache analysis presents a tradeoff between precision and analysis efficiency: The classical approach to analyzing programs running on LRU caches, an abstract interpretation based on a range abstraction, is very fast but can be imprecise. An exact analysis was recently presented, but, as a last resort, it calls a model checker, which is expensive.

In this paper, we develop an analysis based on abstract interpretation that comes close to the efficiency of the classical approach, while achieving exact classification of all memory accesses as the model-checking approach. Compared with the model-checking approach we observe speedups of several orders of magnitude. As a secondary contribution we show that LRU cache analysis problems are in general NP-complete.

*This work was partially supported by the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement nr. 306595 "STATOR", and by the Deutsche Forschungsgemeinschaft as part of the project PEP: Precise and Efficient Prediction of Good Worst-case Performance for Contemporary and Future Architectures.

†This is an extended version of work presented at POPL 2019: it includes appendices not present in the ACM publication.

‡Institute of Engineering Univ. Grenoble Alpes

# 1 Introduction

## 1.1 Motivation

Due to technological developments, the latency of an access to DRAM-based main memory has long been much higher than the latency of an individual computation on the CPU. The most common solution to bridge this "memory gap" is to include a hierarchy of cache memories between the CPU and main memory, meant to speed up accesses to frequently required code and operands.

In the presence of caches, the latency of an individual memory access may vary considerably depending on whether the access is a *cache hit*, i.e., it can be served from an on-chip cache memory, or a *cache miss*[1], i.e., it has to be fetched from the next level cache or DRAM-based main memory.

The purpose of *cache analysis* is to statically classify every memory access at every machine-code instruction in a program into one of the following three classes:

1. "always hit": each dynamic instance of the memory access results in a cache hit;
2. "always miss": each dynamic instance of the memory access results in a cache miss;
3. there exist dynamic instances that result in a cache hit and others that result in a cache miss.

This is of course, in general, an undecidable question; so all analyses involve some form of abstraction, and may classify some accesses as "unknown". An analysis is deemed more precise than another if it produces fewer unknowns.

For the certification of safety-critical real-time applications, it is often necessary to bound a program's *worst-case execution time* (WCET). For instance, if a control loop runs at 100 Hz, then it is imperative to show that the program's WCET is less than 0.01 seconds. In architectures involving caches, i.e., any modern architecture except the lowest-performance ones, such WCET analysis [39][2] must thus take into account caches. For pipelined and superscalar architectures, it is very important to have precise information about the cache behavior, since pipeline analysis must consider the two cases "cache hit" and "cache miss" for any memory access that cannot be shown to "always hit" or "always miss" [28, 34], leading to a state explosion. Thus, imprecise cache analysis may have two adverse effects on WCET analysis: (a) excessive overestimation of the WCET

[1] An often quoted figure is that a cache miss is 100 times slower than a cache hit. We have a simple program whose memory access pattern can be changed by a numeric parameter while keeping exactly the same computations; we timed it with a cache-favorable sequential access pattern compared to a cache-unfavorable one. Depending on the machine and processor clocking configuration, the ratio of the two timings varies between 13 and 40 if one core is used, up to 50 with two cores.

[2] WCET static analysis tools include, among others, aiT, an industrial tool from Absint GmbH, and Otawa, an academic tool.

compared to the true WCET[3] (b) excessively high analysis time due to state explosion. Improvements to cache analysis precision are thus of high importance in this respect; but they must come at reasonable cost.

Caches also give rise to side channels that can be used to extract or transmit sensitive information. For example, cache timing attacks on software implementations of the Advanced Encryption Standard [3] were one motivation for adding specific hardware support for that cipher to the x86 instruction set [32]. Cache analysis may help identify possibilities for such *side-channel attacks* and quantify the amount of information leakage [11, 10]; improved precision in cache analysis translates into fewer false alarms and tighter leakage bounds.

## 1.2 Cache Organization, Cache Analysis, and the State-of-the-Art

Instruction and data caches are usually set-associative and thus partitioned into *cache sets*. Each memory block may reside in exactly one of these cache sets, determined by a simple computation from its address. Each cache set consists of multiple *cache lines*, which may each be used to store a single memory block. The number of cache lines $N$ in each cache set is known as the *number of ways* or the *associativity* of the cache. Upon a cache miss to a memory block that maps into a full cache set, one of the $N$ cached memory blocks must be evicted from the set. There exist several *policies* for choosing which memory block to evict. In this article, we focus on the *least recently used* policy (LRU): the least recently used memory block within a cache set is evicted. LRU has been extensively studied in the literature and is frequently used in practice, e.g., in processors such as the MPC603E, the TriCore17xx, or the TMS320C3x.

As noted before, cache analysis is in general undecidable; so all analyses involve some form of abstraction. Most work on cache analysis separates the concerns of

(a) control-flow analysis, determining which execution paths can be taken,

(b) pointer analysis, determining which memory locations may be accessed by instructions, and

(c) cache analysis proper.

Concerns (a) and (b) are in general undecidable, so safe and terminating analyses use some form of over-approximation — they may consider some paths to be feasible when actually they are not, or that pointers may point to memory locations when actually they cannot. In our case, as in many other works, we assume we are given a control-flow graph $G$ of the program decorated with the memory locations that are possibly accessed by the program, but without the

[3] An industrial user may suspect this when the upper bound on WCET given by the tools is far from experimental timings. This may discourage the user from using static analysis tools.

functional semantics (arithmetic, guards etc.); the executions of this control-flow graph are thus a superset of those of the program. This is what we will consider as a concrete semantics — though we shall sketch, as future work, in Section 10, how to recover some of the precision lost by using that semantics by reintroducing information about infeasible paths.

In this paper, we focus on (c) cache analysis proper for LRU caches. When it comes to LRU cache analysis, the state-of-the-art currently presents a tradeoff between *precision* and *analysis efficiency*:

The classical approach to the static analysis of LRU caches [14] is a highly-efficient abstract interpretation that essentially keeps for each block a range of how many other blocks have been used more recently. This analysis is exact for straight-line programs, but loses precision in general when tests are involved: the join operation adds spurious states, which may translate into classification of memory accesses as "unknown," whereas, with respect to the concrete semantics of $G$, they should be classified as "always hit" or "always miss".

Recently, Touzeau et al. [38] proposed an exact analysis, i.e., it exactly classifies memory accesses with respect to the concrete semantics, into "always hit", "always miss", or "hits or misses depending on the execution". Their approach encodes the concrete cache state transitions into a reachability problem, fed to a symbolic model checker. Since that approach was slow, they also proposed a fast abstract pre-analysis able to detect cases where an access "hits or misses depending on the execution". The model-checking algorithm is then only applied to the relatively infrequent cases where accesses are still classified as "unknown" by their new analysis and the classical ones by Ferdinand and Wilhelm.

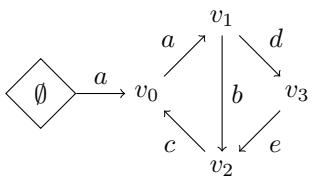
### 1.3 Contributions

In this paper, we develop an analysis based on abstract interpretation that comes close to the efficiency of the classical approach by Ferdinand and Wilhelm [14] while achieving exact classification of all memory accesses as the model-checking approach by Touzeau et al. [38]. In other terms, we introduce an exact and scalable analysis by carefully refining the abstraction and using suitable algorithms and data structures.

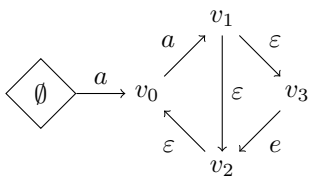
Our main contribution is the introduction of a new exact abstraction for LRU caches that is based on a partial order of cache states. To classify cache misses (cache hits), it is sufficient to only keep minimal (maximal) elements w.r.t. this partial order. As a consequence, the abstraction may be exponentially more succinct than the model-checking approach followed by Touzeau et al. [38].

We improve the focused semantics of Touzeau et al. [38] by removing subsumed elements with upward and downward closures. This form of convergence acceleration preserves the precision of the final classification.

We discuss a suitable data structure for this abstraction based on zero-suppressed binary decision diagrams (ZDDs), and an implementation on top of Otawa [1] and Cudd [37]. Our experimental evaluation shows an analysis speedup of up to 950 compared with the prior exact approach by Touzeau et al. [38]. The geometric mean of the speedup across all studied benchmarks




(a) Original control-flow graph for two cache sets: $\{a, e\}$ and $\{b, c, d\}$.




(b) The same control-flow graph focused for cache set $\{a, e\}$.

Figure 1: Slicing of a control-flow graph according to a cache set

is at least $9^4$. On the other hand, compared with the imprecise age-based analysis of Ferdinand and Wilhelm [14] we observe an average slowdown across all benchmarks of only 3.46.

Our secondary contribution is a proof that both of the problems that we address (existence of a trace leading to a cache hit, existence of a trace leading to a cache miss) are NP-complete. To the best of our knowledge, this was not known previously, whereas it justifies using imprecise abstractions, as in the traditional age-based analyses, and/or algorithms with non-polynomial worst-case complexity, as in our analysis.

## 1.4 Outline

In Section 2 we define the static analysis problem for LRU caches. In Section 3 we illustrate how the results of our analysis are more precise than those of classical analysis on a small example. In Section 4 we reformulate this problem as a least fixpoint, then give two exact abstractions, one for "always hit", the other for "always miss" results, each of which can be implemented by computations over antichains. In Section 5 we explain the algorithms and data structures used for the upward and downward closures, using and extending zero-suppressed binary decision diagrams (ZDDs). In Section 6 we describe a few possible extensions and variants of our approach. In Section 7 we discuss complexity issues and show that the analysis problems we solve are NP-hard, thus justifying the use of potentially exponentially large ZDDs. In Section 8 we describe our implementation and our experimental results. In Section 9 we discuss related work. We conclude the paper sketching possible avenues for future work in Section 10.

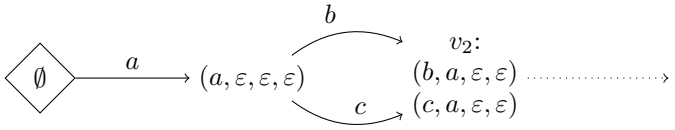



Figure 2: At $v_2$, two cache states are possible, according to the path of arrival. In both of them $a$ is present, so $a$ is a "must hit" on edges going out of $v_2$. $b$ is present in one of them, so it is a "may hit". $d$ is present in neither, so it is a "must miss".

# 2 Problem Setting

As is common in cache analysis, we assume the following analyses have already been performed:

(a) A control-flow graph has been reconstructed from the machine code of the program under analysis (perhaps with some knowledge about the compiler and/or dumps of its intermediate representation).

(b) For instruction cache analysis, all code addresses are known. This is the normal situation for embedded software running on platforms with no operating system or a lightweight one; it also applies to logical addresses[5] for non-relocatable programs running in full-fledged operating systems.

(c) For data cache analysis, a points-to analysis has been run to obtain, for all memory accesses, a superset of the memory locations that may be affected. In programs using pointer arithmetic or array accesses, this points-to analysis may need a value analysis. Note that our analysis gives exact results for write-through write-allocate data caches, and that adaptation for other write policies is future work.

(d) For mixed instruction/data caches, both of the above must have been done.

(e) For caches addressed by physical addresses, the mapping from logical (virtual) to physical addresses must be known.[6]

The result is a control-flow graph $G$ with vertices representing program locations. An edge from $v_1$ to $v_2$ is labeled with the address of the memory block accessed when control steps from $v_1$ to $v_2$, or with $\varepsilon$ if no access is made (e.g.

[4] On a number of benchmarks the prior exact approach timed out at 12 hours. For these benchmarks, we conservatively assume an analysis time of 12 hours, and may thus underestimate the actual speedup, had the analysis been run to completion.

[5] For systems with a memory management unit (MMU) and "virtual memory" we distinguish the logical addresses, as seen from the program, and the physical addresses, as seen from RAM.

[6] This is impossible for general operating systems with dynamic memory paging, but is possible for embedded systems: one may have a MMU to isolate processes from each other, e.g. high criticality from low criticality, with a fixed memory layout.

we are analyzing the data cache and the instruction corresponding to the edge accesses no data). Note that this is not, in general, the same thing as the address of the memory access: for instance, with 64-byte lines, an access to a byte at address 127 is considered to be an address to the line at address 64. One memory access can extend over several cache lines: for instance, if instead of a byte we access a 4-byte word at address 127, on a processor allowing unaligned accesses, then we access successively two lines at addresses 64 and 128. More generally, if several memory accesses are performed at the same program location, this location must be split into several sub-locations according to the order of the accesses.

Furthermore, this graph may actually be a multigraph, with several edges, labeled differently, between the same pair of vertices: if points-to analysis returns a set of several possible addresses for one access, there is one edge per address.

In our examples, for instance in Figure 1a, lowercase letters $a$, $b\ldots$ denote such addresses. $G$ has special start vertices labeled with either $\emptyset$, meaning that program execution is assumed to start with an empty cache, or $\top$, meaning that program execution may start with an arbitrary cache state (all legal combinations of memory blocks and empty lines are possible); other classes of initial vertices may be added if needed. All program executions must start at a start vertex. Without loss of generality, we assume all vertices and edges to be reachable from start vertices, and the start vertices not to be endpoints of any edges.

In an LRU cache, as with almost all replacement policies, each cache set is treated independently. One can thus analyze the behavior of the program completely independently on each cache set $S$: $G$ is *sliced* according to cache set $S$ by replacing each address not in $S$ by $\varepsilon$ (see Figure 1). In the rest of the article, unless noted otherwise, we shall thus assume a single cache set, without loss of generality. For efficiency, an implementation may wish to collapse vertices and edges making $\varepsilon$ transitions only.

An LRU cache encountering an access to memory block $a$ loads $a$ into the cache, for instance, after an access to $a$ the cache state of a 4-way LRU cache is $(a, x, y, z)$ where $a$ is the most recently used and $z$ the least recently used. Two cases may occur: **(Hit)** $a$ is already in the cache and is "rejuvenated", i.e., cache-internal status bits are updated to record that $a$ is the most-recently-used memory block; for instance, from the cache state $(x, y, a, z)$ and access to $a$ leads to the cache state $(a, x, y, z)$ and from the cache state $(a, x, y, z)$ an access to $a$ does not change the cache state[7]. **(Miss)** $a$ is not in the cache and the "oldest" memory block is evicted from $a$'s cache set, for instance, from a cache state $(w, x, y, z)$ an access to $a$ evicts block $z$ and leads to the cache state $(a, w, x, y)$.

As a consequence, along a program execution $E$, assuming an initially empty cache, an access to $a$ is a hit *if and only if* at most $N-1$ distinct memory blocks have been accessed along $E$ since the last access to $a$ (several accesses to the same memory block count as one).

[7]In FIFO caches, there is no rejuvenation: a "hit" does not change the cache.

An edge is said to be "always hit" if all executions passing through this edge encounter a cache hit at this edge; otherwise it is said to be "may miss". An edge is said to be "always miss" if all executions passing through this edge encounter a cache miss at this edge; otherwise it is said to be "may hit". See Figure 2 for an example. We shall propose two analyses, one for classifying edges as either "always hit" or "may miss", the other for classifying edges as "always miss" or "may hit".

# 3 Motivating Examples

## 3.1 Age-based Analysis vs Precise Analysis

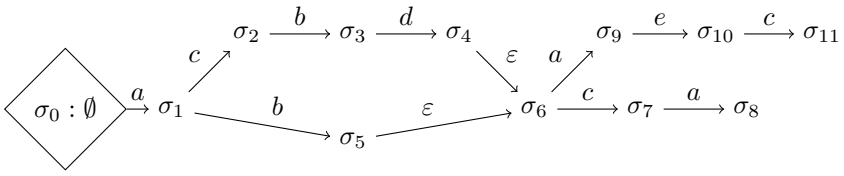



Figure 3: Example of control-flow graph where classical age-based "must-hit" analysis yields suboptimal results.

Table 1: Concrete and abstract states in the analysis of the control-flow graph in Figure 3. To avoid too wide a table we omit the analysis results for $e$. $\mathcal{A}$ means "absent".

| Reachable cache states | | Age-based analysis | | | | Block-focused analysis | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | a | b | c | d | a | b | c | d |
| $\sigma_0$ | $(\varepsilon, \varepsilon, \varepsilon, \varepsilon)$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\mathcal{A}$ | $\mathcal{A}$ | $\mathcal{A}$ | $\mathcal{A}$ |
| $\sigma_1$ | $(a, \varepsilon, \varepsilon, \varepsilon)$ | 0 | $\infty$ | $\infty$ | $\infty$ | $\emptyset$ | $\mathcal{A}$ | $\mathcal{A}$ | $\mathcal{A}$ |
| $\sigma_2$ | $(c, a, \varepsilon, \varepsilon)$ | 1 | $\infty$ | 0 | $\infty$ | $\{c\}$ | $\mathcal{A}$ | $\emptyset$ | $\mathcal{A}$ |
| $\sigma_3$ | $(b, c, a, \varepsilon)$ | 2 | 0 | 1 | $\infty$ | $\{b, c\}$ | $\emptyset$ | $\{b\}$ | $\mathcal{A}$ |
| $\sigma_4$ | $(d, b, c, a)$ | 3 | 1 | 2 | 0 | $\{b, c, d\}$ | $\{d\}$ | $\{b, d\}$ | $\emptyset$ |
| $\sigma_5$ | $(b, a, \varepsilon, \varepsilon)$ | 1 | 0 | $\infty$ | $\infty$ | $\{b\}$ | $\emptyset$ | $\mathcal{A}$ | $\mathcal{A}$ |
| $\sigma_6$ | $(d, b, c, a)$, $(b, a, \varepsilon, \varepsilon)$ | $[1, 3]$ | $[0, 1]$ | $[2, \infty]$ | $[0, \infty]$ | $\{b, c, d\}$, $\{b\}$ | $\{d\}$, $\emptyset$ | $\{b, d\}$, $\mathcal{A}$ | $\emptyset$, $\mathcal{A}$ |
| $\sigma_7$ | $(c, d, b, a)$, $(c, b, a, \varepsilon)$ | $[2, \infty]$ | $[1, 2]$ | 0 | $[1, \infty]$ | $\{b, c, d\}$, $\{b, c\}$ | $\{c, d\}$, $\{c\}$ | $\emptyset$ | $\{c\}$, $\mathcal{A}$ |
| $\sigma_8$ | $(a, c, d, b)$, $(a, c, b, \varepsilon)$ | 0 | $[2, 3]$ | 1 | $[2, \infty]$ | $\emptyset$ | $\{a, c, d\}$, $\{a, c\}$ | $\{a\}$ | $\{a, c\}$, $\mathcal{A}$ |
| $\sigma_9$ | $(a, d, b, c)$, $(a, b, \varepsilon, \varepsilon)$ | 0 | $[1, 2]$ | $[2, \infty]$ | $[1, \infty]$ | $\emptyset$ | $\{a, d\}$, $\{a\}$ | $\{a, b, d\}$, $\mathcal{A}$ | $\{a\}$, $\mathcal{A}$ |
| $\sigma_{10}$ | $(e, a, d, b)$, $(e, a, b, \varepsilon)$ | 1 | $[2, 3]$ | $[3, \infty]$ | $[2, \infty]$ | $\{e\}$ | $\{a, d, e\}$, $\{a, e\}$ | $\mathcal{A}$ | $\{a, e\}$, $\mathcal{A}$ |
| $\sigma_{11}$ | $(c, e, a, d)$, $(c, e, a, b)$ | 2 | $[3, \infty]$ | 0 | $[3, \infty]$ | $\{c, e\}$ | $\mathcal{A}$, $\{a, c, e\}$ | $\mathcal{A}$ | $\{a, c, e\}$, $\mathcal{A}$ |

To analyze LRU caches, it is convenient to introduce the following notion of the *age* of a memory block: The age of a block $x$ in a concrete cache state is the number of blocks younger than $x$, i.e., the number of blocks that have been accessed more recently than $x$, or $\infty$ if $x$ is not in the cache. For instance,

assuming a cache of associativity 4, in the cache state $(c, b, a, \varepsilon)$, from youngest to oldest, $c$ has age 0, $b$ has age 1, $a$ has age 2, $d$ has age $\infty$.

The classical age-based analysis [14] abstracts the set of possible cache states as follows: to each block $x$ it attaches a range of possible ages in the cache. This abstraction may lead to imprecise results, that is, it may fail to conclude that an access is always a hit (respectively, a miss) whereas it is truly always a hit (respectively, a miss) in all executions. Consider the control-flow graph in Figure 3, and the corresponding concrete and abstract cache states in Table 1. At $\sigma_4$, the age of $a$ is 3 and at $\sigma_5$ it is 1, so at $\sigma_6$ it is known to be in $[1, 3]$; similarly the age of $c$ is known to be in $[2, \infty]$. When analyzing $\sigma_6 \xrightarrow{c} \sigma_7$, it is thus unknown whether $c$ is younger or older than $a$ in the cache at $\sigma_6$: in the former case $a$'s age does not change, whereas in the latter case $a$'s age increases by one. The age-based analysis concludes that at $\sigma_7$, the age of $a$ is in $[2, \infty]$, whereas the exact range is $[2, 3]$. The age-based analysis thus cannot conclude that $\sigma_7 \xrightarrow{a} \sigma_8$ is a hit, which is the case in reality.

Instead of abstracting a concrete state with respect to a block $x$ by the *number* of blocks younger than $x$, we abstract it by the *set* of these blocks; thus a set of concrete states is abstracted by a set of sets of blocks. For instance, at $\sigma_7$, we consider the possible sets of blocks younger than $a$ in the cache: $\{b, c, d\}$ and $\{b, c\}$. This analysis is exact, in the sense that no precision is lost by performing analysis steps on the abstract states compared to performing the steps concretely and then abstracting the final result.

Note that the second set is included in the first. If $a$ is a hit after executing a sequence of steps from $\sigma_7$, from a cache state where $a$ is preceded by $\{b, c, d\}$, then *a fortiori* the same sequence of steps also results in a hit if started in a cache state where $a$ is preceded by $\{b, c\}$. We can thus discard $\{b, c, d\}$ from an analysis aimed at discovering the existence of hits ("may hit" analysis). Similarly, we can discard $\{b, c\}$ from an analysis aimed at discovering the existence of misses (the complement of an "always hit" analysis). Again, doing this does not impact precision.

Similarly, in the age-based analysis, when analyzing $\sigma_6 \xrightarrow{a} \sigma_9$, it is unknown whether $c$ is younger or older than $a$ in the cache at $\sigma_6$: in the former case $c$'s age does not change, whereas in the latter case $c$ ages by one. The age-based analysis concludes that at $\sigma_9$, the age of $c$ is in $[2, \infty]$, whereas the exact range is $[3, \infty]$. The step $\sigma_9 \xrightarrow{e} \sigma_{10}$, with a fresh letter $e$, results in an increase of the ages of all other blocks. The age-based analysis thus cannot conclude that $\sigma_{10} \xrightarrow{c} \sigma_{11}$ is a miss, which is the case in reality.

Note that the "definitely-unknown" abstract analysis proposed by Touzeau et al. [38] would not help in any way: it resolves some of the cases where there are execution traces leading both to a hit and a miss at the same location, which is not the case here. Their approach would then have to call a model checker to establish that $\sigma_7 \xrightarrow{a} \sigma_8$ is a "must hit" and $\sigma_{10} \xrightarrow{c} \sigma_{11}$ a "must miss". The purpose of our analysis is to replace this expensive call to a model checker by an abstract interpretation that yields the same result.

### 3.2 Collecting Semantics vs Focused Semantics vs Antichain

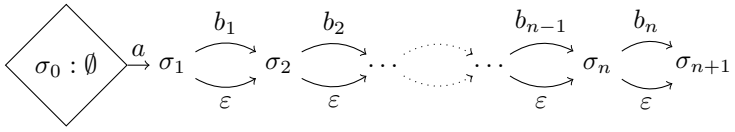



Figure 4: Example where collecting and focused semantics are unnecessarily detailed and inefficient. $\varepsilon$ means "no access".

Consider the control-flow graph of Figure 4, with an associativity $N > n$, and an empty initial cache. At the last control location $\sigma_{n+1}$, the possible cache states are all subsequences of $b_n, \ldots, b_1$ followed by $a$ and possibly empty lines, e.g. $(b_5, b_3, b_1, a, \varepsilon, \varepsilon)$. There are therefore $2^n$ reachable cache states at $\sigma_{n+1}$, all of which appear in the collecting semantics of the program composed with the cache.

The "block-focused" abstraction, which was also applied by Touzeau et al. [38] when encoding cache problems into model-checking reachability problems, records only the set of blocks younger than the block of interest. Here, if our block of interest is $a$, this abstraction thus yields at $\sigma_{n+1}$ the set of subsets of $\{b_1, \ldots, b_n\}$. Of course, symbolic set representation techniques may have a compact representation for such a set, but the main issue is that this set keeps too much information.

If our goal is to prove the existence of a "hit" on an access to memory block $a$ further down the execution, then it is sufficient to keep, in this set, $\emptyset$, corresponding to a path composed of the access to $a$ followed by a sequence of no accesses $\varepsilon$. More generally, it is sufficient to keep only the minimal elements (with respect to the inclusion ordering) from this set, which can be exponentially more succinct, as in this example. Similarly, if our goal is to prove the existence of a "miss" on $a$ further down the execution, then it is sufficient to keep, in this set, $\{b_1, \ldots, b_n\}$, corresponding to a path composed of the access to $a$ followed by a sequence of accesses to $b_1$, $\ldots$, $b_n$. More generally, it is sufficient to keep only the maximal elements from this set.

This is the main difference between our analysis and the "focused" model that Touzeau et al. [38] fed into the model checker: the "focused" model contains unnecessary information (non-minimal elements for the Always-miss analysis, non-maximal elements for the Always-hit analysis), which increases model-checking times. We discard these in our analysis.

The following section formally defines the collecting and focused semantics, and our new Always-hit and Always-miss analyses. The Always-hit analysis (respectively Always-miss) computes the antichain of maximal elements (respectively minimal elements) i.e. the downward (respectively upward) closure of reachable states.

# 4 Analyses as Fixed-Point Problems

## 4.1 Collecting Semantics

To each vertex $l$ we attach a set $C_l$ of possible cache states. Each cache state $s$ is a sequence $(s_1, \ldots, s_N)$ of addresses, from youngest to oldest, possibly ending with one or more special values $\varepsilon$, meaning that this cache line is empty, and without repetition of addresses except for $\varepsilon$. Let $S$ be the set of cache states.

*Example* 4.1. If $N = 4$, $(\varepsilon, \varepsilon, \varepsilon, \varepsilon)$ (empty cache), $(a, b, \varepsilon, \varepsilon)$, and $(a, d, c, b)$ are valid cache states; $(a, b, \varepsilon, d)$ and $(a, d, a, b)$ are not.

To any start vertex $l$ labeled $\emptyset$, we attach $C_l = \{(\varepsilon, \ldots, \varepsilon)\}$, meaning that the only possible cache state at this location is empty. To any start vertex $l$ labeled $\top$, we attach $C_l = S$, meaning that any cache state is possible at this location. The rest of the cache states are obtained as the least solution of:

Miss: if $(s_1, \ldots, s_N) \in C_l$, none of the $s_i$ is $a$, and there is an edge $l \xrightarrow{a} l'$, then $(a, s_1, \ldots, s_{N-1}) \in C_{l'}$: the oldest line is evicted;

Hit: if $(s_1, \ldots, s_N) \in C_l$, $a$ occurs at position $i$ ($s_i = a$), and there is an edge $l \xrightarrow{a} l'$, then $(a, s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_N) \in C_{l'}$ (if $i = N$, that is $(a, s_1, \ldots, s_{i-1}) \in C_{l'}$): block $a$ is "rejuvenated".

*Example* 4.2. If the cache contains $(a, b, c, d)$ and $b$ is accessed, then $b$ is rejuvenated and the cache then contains $(b, a, c, d)$. If instead $e$ is accessed, then $d$ is evicted and the cache then contains $(e, a, b, c)$.

## 4.2 Focused Semantics

We reuse the "focused semantics" proposed by Touzeau et al. [38], as well as their proof of exactness.

Let $a \in A$ be some address; we are interested in classifying accesses to $a$. We can focus the behavior of the cache with respect to $a$ as follows. A cache state $s_1, \ldots, s_N$ such that $s_i = a$ will be abstracted as the set $\{s_1, \ldots, s_{i-1}\}$ ($\emptyset$ if $i = 1$) of blocks present in the cache before $a$, i.e., younger than $a$. A cache state not containing $a$ is abstracted as the special value $\mathcal{A}$.

*Example* 4.3. When focusing on block $a$, cache states $(c, b, a, \varepsilon)$ and $(b, c, a, d)$ are both abstracted as the set $\{b, c\}$, and $(c, b, e, \varepsilon)$ as $\mathcal{A}$.

One can easily show that this focused semantics can be directly computed as follows. To each vertex $l$ we attach a set $C_{l,a}$ of $a$-focused states. These sets are ordered by inclusion. To any start vertex $l$ labeled $\emptyset$, we attach $C_{l,a} = \{\mathcal{A}\}$. To any start vertex $l$ labeled $\top$, we attach $C_{l,a} = \{\mathcal{A}\} \cup \{S \mid S \subseteq A \setminus \{a\} \wedge |S| \leq N - 1\}$. The rest of the cache states are obtained as the least solution of:

- if there is an edge $l \xrightarrow{a} l'$, then $\emptyset \in C_{l',a}$.
- if $\mathcal{A} \in C_{l,a}$ and there is an edge $l \xrightarrow{b} l'$, $b \neq a$ then $\mathcal{A} \in C_{l',a}$;

- if $S \in C_{l,a}$, $S \neq \mathcal{A}$, $|S \cup \{b\}| < N$ and there is an edge $l \xrightarrow{b} l'$, $b \neq a$ then $S \cup \{b\} \in C_{l',a}$;
- if $S \in C_{l,a}$, $S \neq \mathcal{A}$, $|S \cup \{b\}| = N$ and there is an edge $l \xrightarrow{b} l'$, $b \neq a$ then $\mathcal{A} \in C_{l',a}$;

An edge $l \xrightarrow{a} l'$ may result in a miss if and only if $\mathcal{A} \in C_{l,a}$. An edge $l \xrightarrow{a} l'$ may result in a hit if and only if there exists $S \in C_{l,a}$, $S \neq \mathcal{A}$.

Another intuitive characterization is that $C_{l,a}$ is the collection of the sets of addresses found along the paths from the nearest preceding occurrences of $a$, truncated at associativity; sets of cardinality greater than or equal to associativity are all abstracted to the special value $\mathcal{A}$.

## 4.3 Always-Hit Analysis

To get a better idea of what the Always-hit analysis computes let us first recall the definitions of antichain and upper set, and illustrate this analysis with an example.

**Definition 4.4.** An *antichain* is a subset of an ordered set such that no two distinct elements of that subset are comparable. An *upper set* (respectively *lower set*) is a set such that if an element is in this set, then all elements larger (respectively, smaller) than it are also in the set.

*Example* 4.5. Assume that a node $l$ may be reached with cache states $C_l = \{(c, b, e, a),\ (b, c, d, a),\ (b, a, \varepsilon, \varepsilon)\}$. The possible a-focused states are: $C_{l,a} = \{\{b, c, e\}, \{b, c, d\}, \{b\}\}$. Since $\{b\}$ is strictly included in $\{b, c, e\}$ (and in $\{b, c, d\}$), it may not contribute to cache misses that would not also occur following $\{b, c, e\}$ (and $\{b, c, d\}$) and can be removed without affecting soundness; the antichain of the maximal elements of $C_{l,a}$ is $\{\{b, c, e\}, \{b, c, d\}\}$, which will be called $C^{\max}_{l,a}$.

In all that follows, the ordering will be the inclusion ordering $\subset$.

Recall that $S \in C_{l,a}, S \neq \mathcal{A}$ means that at position $l$, there is a reachable cache state of the form $(s_1, \ldots, s_{|S|}, a, \ldots)$ where $S = \{s_1, \ldots, s_{|S|}\}$. An edge $l \xrightarrow{a} l'$ "always hits" if and only if it "may not miss", that is, if there is no execution trace leading to a miss at this location, i.e. $\mathcal{A} \notin C_{l,a}$.

**Definition 4.6.** Let $x \xrightarrow{b} y$ denote the transition "upon an access to block $b$, $b \neq a$, the cache may move from an $a$-focused state $x$ to an $a$-focused state $y$". Recall that $x$ may be $\mathcal{A}$ ($a$ is not in the cache) or a subset of cache blocks, not containing $a$, of cardinality at most $N-1$. This deterministic transition relation is defined as follows:

- $\mathcal{A} \xrightarrow{b} \mathcal{A}$;
- $x \xrightarrow{b} x \cup \{b\}$ for $|x \cup \{b\}| < N$;
- $x \xrightarrow{b} \mathcal{A}$ for $|x \cup \{b\}| = N$.

**Definition 4.7.** For $x, y \subseteq A \setminus \{a\}$, let $x \xrightarrow{\downarrow} y$ denote a downward closure step:

- $\mathcal{A} \xrightarrow{\downarrow} y$ for any $y$;
- $x \xrightarrow{\downarrow} y$ for any $x, y \neq \mathcal{A}$, $y \subseteq x$.

*Example* 4.8. $\{b, c, e\} \xrightarrow{\downarrow} y$ for any $y \in \{\emptyset, \{b\}, \{c\}, \{e\}, \{b, c\}, \{b, e\}, \{c, e\}, \{b, c, e\}\}$

**Lemma 4.9.** *Assume there are $x, y, z$, and $b$ such that $x \xrightarrow{\downarrow} y \xrightarrow{b} z$. Then there exists $y'$ such that $x \xrightarrow{b} y' \xrightarrow{\downarrow} z$.*

*Proof.* As the transition relation is deterministic, $y'$ is uniquely determined by $x$ and $b$.

We distinguish two cases based on the value of $y'$:

1. If $y' = \mathcal{A}$, then the results follows immediately, as $\mathcal{A} \xrightarrow{\downarrow} z$ for any $z$.
2. If $y' \neq \mathcal{A}$, then $y' = x \cup \{b\}$ with $|y'| < N$. Then $x \neq \mathcal{A}$ and $y \subseteq x$, and so $z = y \cup \{b\} \subseteq x \cup \{b\} = y'$.

□

**Corollary 4.10.** *There exists a sequence of the form $u_0 \xrightarrow{b_0} v_0 \xrightarrow{\downarrow} u_1 \xrightarrow{b_1} v_1 \xrightarrow{\downarrow} \dots u_n \xrightarrow{b_n} v_n \xrightarrow{\downarrow} u_{n+1}$ if and only if there exists a sequence of the form $u_0 \xrightarrow{b_0} u'_1 \xrightarrow{b_1} u'_2 \dots \xrightarrow{b_n} u'_n \xrightarrow{\downarrow} u_{n+1}$.*

It is thus equivalent to compute the reachable states of the $a$-focused semantics (for transitions different from $a$), then apply downward closure, and to apply downward closure at every step during the computation of reachable states. In addition, it is obvious that $\mathcal{A}$ is in a set if and only if it is in its downward closure. It is thus equivalent to test for a "may miss" on the reachable states of the $a$-focused semantics and on their downward closure.

This suggests two possible (and equivalent, in a sense) simplifications to the focused semantics if our goal is to find places where an access to $a$ may be a miss:

Closure Replace $C_{l,a}$ by its down-closure $C^{\downarrow}_{l,a}$: $S' \in C^{\downarrow}_{l,a}$ if and only if there exists $S \in C_{l,a}$ such that $S' \subseteq S$.

Subsumption removal Replace $C_{l,a}$ by the antichain of its maximal elements: $S \in C^{\max}_{l,a}$ if and only if $S \in C_{l,a}$ and there is no $S' \in C_{l,a}$ such that $S \subsetneq S'$.

Note that $C^{\downarrow}_{l,a}$ is the down-closure of $C^{\max}_{l,a}$, and that $C^{\max}_{l,a}$ is the antichain of maximal elements of $C^{\downarrow}_{l,a}$; thus $C^{\max}_{l,a}$ is just an alternative representation for $C^{\downarrow}_{l,a}$. Our idea is to directly compute $C^{\max}_{l,a}$.

## 4.4 Always-Miss Analysis

This subsection presents the Always-miss analysis which is the dual of the Always-hit analysis of Section 4.3. A control location "always misses" if and only if it "may not hit", that is, if there is no execution trace leading to a hit at this location.

**Definition 4.11.** For $x, y \subseteq A \setminus \{a\}$, let $x \xrightarrow{\uparrow} y$ denote an upward closure step:

- $x \xrightarrow{\uparrow} \mathcal{A}$ for any $x$;
- $x \xrightarrow{\uparrow} y$ for any $x, y \neq \mathcal{A}$, $x \subseteq y$.

**Lemma 4.12.** *Assume there are $x, y, z$, and $b$ such that $x \xrightarrow{\uparrow} y \xrightarrow{b} z$. Then there exists $y'$ such that $x \xrightarrow{b} y' \xrightarrow{\uparrow} z$.*

*Proof.* We distinguish two cases based on the value of $z$:

1. If $z = \mathcal{A}$, then the results follows immediately, as $y' \xrightarrow{\uparrow} \mathcal{A}$ for any $y'$.
2. If $z \neq \mathcal{A}$, then $z = y \cup \{b\}$ with $|z| < N$. Then $y \neq \mathcal{A}$ and $x \subseteq y$, and so $y' = x \cup \{b\} \subseteq y \cup \{b\} = z$.

□

**Corollary 4.13.** *There exists a sequence of the form $u_0 \xrightarrow{b_0} v_0 \xrightarrow{\uparrow} u_1 \xrightarrow{b_1} v_1 \xrightarrow{\uparrow} \dots u_n \xrightarrow{b_n} v_n \xrightarrow{\uparrow} u_{n+1}$ if and only if there exists a sequence of the form $u_0 \xrightarrow{b_0} u'_1 \xrightarrow{b_1} u'_2 \dots \xrightarrow{b_n} u'_n \xrightarrow{\uparrow} u_{n+1}$.*

It is thus equivalent to compute the reachable states of the $a$-focused semantics (for transitions different from $a$), then apply upward closure, and to apply upward closure at every step during the computation of reachable states. In addition, it is obvious that there exists $x$ in $X$, $x \neq \mathcal{A}$, if and only if there exists $y$ in the upward closure of $X$ such that $y \neq \mathcal{A}$. It is thus equivalent to test for a "may hit" on the reachable states of the $a$-focused semantics and on their upward closure.

This again suggests two possible simplifications to the focused semantics if our goal is to find places where an access to $a$ may be a hit:

Closure Replace $C_{l,a}$ by its up-closure $C^{\uparrow}_{l,a}$: $S' \in C^{\uparrow}_{l,a}$ if and only if there exists $S \in C_{l,a}$ such that $S \subseteq S'$.

Subsumption removal Replace $C_{l,a}$ by the antichain of its minimal elements: $S \in C^{\min}_{l,a}$ if and only if $S \in C_{l,a}$ and there is no $S' \in C_{l,a}$ such that $S' \subsetneq S$.

Note that $C^{\uparrow}_{l,a}$ is the up-closure of $C^{\min}_{l,a}$, and that $C^{\min}_{l,a}$ is the antichain of minimal elements of $C^{\uparrow}_{l,a}$; thus $C^{\min}_{l,a}$ is just an alternative representation for $C^{\uparrow}_{l,a}$. Our idea is to directly compute $C^{\min}_{l,a}$.

### 4.5 A Remark on Lattice Height

We replace the focused semantics by its upward or downward closure; this is a form of convergence acceleration, albeit one that preserves the precision of the final result. We shall see in Section 8 that this improves practical performance considerably compared to a version that checks the focused semantics in a model checker. It is however unlikely that this improvement translates to the worst case; let us see why.

The number of iterations of a data-flow or abstract interpretation analysis is bounded by the height of the analysis lattice, that is, the maximal length of a strictly increasing sequence. However, this height does not change by imposing that the sets should be lower (respectively upper) closed: just apply the following lemma to $T$, the set of subsets of $A$ of cardinality at most $N-1$ (plus $\mathcal{A}$) ordered by inclusion (respectively, reverse inclusion).

**Lemma 4.14.** *Let $(T, \leq)$ be a partially ordered finite set. The lattice of lower subsets of $T$, ordered by inclusion, has height $|T|$, the same height as the lattice of subsets of $T$.*

*Proof.* Order $T$ topologically: $t_1, \ldots, t_{|T|}$, such that $\forall i, j : \; t_i \leq t_j \implies i \leq j$. The sequence $(u_i)_{i=0,\ldots,|T|}$, with $u_i = \{t_1, \ldots, t_i\}$, is a strictly ascending sequence of lower sets. □

## 5 Data Structures and Algorithms

In Section 4 we defined a collecting semantics for concrete cache states, then, in two steps (1. focused semantics, 2. closures), showed that there is a cache hit (respectively, a cache miss) in the concrete semantics if and only if there is a cache hit (respectively, a cache miss) in an upward-closed (respectively, downward-closed) semantics, and that upward-closed (respectively, downward-closed) sets may be represented by the antichains of their minimal (respectively, maximal) elements.

### 5.1 Computation by Abstract Interpretation

The abstracted semantics in upward-closed (or, downward-closed) sets may be computed by a standard data-flow/abstract interpretation algorithm, by upward iterations, as follows.

To each initial control point we initially attach an initialization value (see below). For a semantics focused on accesses to $a$, we consider that each edge $x \xrightarrow{a} y$ is replaced by an initial edge $\emptyset \xrightarrow{a} y$, pushing $\{\emptyset\}$ as the value associated to the control state $y$. Then, we iterate in the usual abstract interpretation fashion: we maintain a "working set", initially containing the initial locations and the targets of the $\xrightarrow{a}$ edge; we take a control point $x$ from the working set, update the abstract values at the end point of edges going out of $x$ (using the union operation on upper or lower sets), and add these end points to the

working set if their value has changed (equality testing). The iterations stop when the working set becomes empty. It is a classical result [9, §2.9] that the final result of such iterations does not depend on the iteration ordering, and in fact several elements from the working set may be treated in parallel; the only requirement is that all elements from the working set are eventually treated.

The sequence of updates to the set decorating a given control location is strictly ascending, in a finite lattice; thus its length is bounded by the height $h$ of that lattice. If $V$ is the set of control locations, then the total number of updates is bounded by $|V| \cdot h$. Recall that the height of the lattice of subsets of a set $X$ is $|X|$.

If we implement the focused semantics directly, then we compute over sets of subsets of size at most $N-1$ of $A \setminus \{a\}$, completed with $\mathcal{A}$; the number of such subsets is bounded by $\sum_{k=0}^{N-1}(|A|-1)^k$ and thus $h \leq \frac{(|A|-1)^N}{|A|-2}+1$. The cost could thus be exponential in the associativity; we shall see in Section 7 that a polynomial-time algorithm is unlikely, since the problems are NP-complete.

## 5.2 Closed Sets Implementation

We initially attempted adding closure steps to the focused semantics, and running a model checker on the resulting systems. The performance was however disappointing, worse than model-checking the focused semantics itself as was proposed by Touzeau et al. [38]. The model checker (nuXmv) was representing its sets of sets of blocks using state-of-the-art binary decision diagrams; we thus did not expect any gain by going to our own implementation of iterations over the same structure. We thus moved from representing a closed set by its content to representing it by the antichain of its minimal (respectively, maximal) elements. There remains the question of how to store and compute upon the antichains representing those sets.

We then tried storing an antichain simply as a sorted set of subsets of $A$, each subset being represented as the list of its elements. Experimentally, this approach was inefficient; let us explain why, algorithmically. For once, when computing the antichain for the union of two upward or downward closed sets $S$ and $S'$, one takes the antichains $W$ and $W'$ representing $S$ and $S'$ and eliminates redundancies; if such a naive representation is used, one needs to enumerate all pairs of items from $W \times W'$ — there is no way to immediately identify which parts of $W$ and $W'$ are subsumed, or even to identify which parts are identical. Furthermore, there is no sharing of representation between related antichains.

Binary decision diagrams are one well-known data structure for representations of sets of states; they share identical subsets, and allow fast equality testing. All operations over such diagrams can be "memoized", meaning that when an operation is run twice between identical subparts of existing diagrams, the result may be cached. We store an antichain, a set $S$ of sets of addresses, as a *zero-suppressed decision diagram* (ZDD) [30, 31] [23, §7.1.4, p.249], a variant of binary decision diagrams optimized for representing sets of sparse sets of items.

### 5.3 Basic Functions for May-Hit and May-Miss Analyses

We assume that all control states are reachable (unreachable states are easily discarded by a graph traversal). The starting points of the analyses focused on $a$ are the initial control points as well as all accesses to $a$.

The operations that we need for antichains defining upper sets, for the may-hit analysis, are

Initialization to empty cache Return $\emptyset$.

Initialization to undefined cache state Return $\{\emptyset\}$.

Initialization to unreachable state Return $\emptyset$.

Access to address $b \neq a$: return $\{s \cup \{b\} \mid s \in S\}$.

Access to tracked block $a$: return $\{\emptyset\}$

Limitation to associativity Return $\{s \mid s \in S \wedge |s| \leq N-1\}$.[8]

Union of upper sets represented by antichain of minimal elements of $S$ and $S'$: return $\{s \mid s \in S \wedge \neg\exists s' \in S'\ s' \subsetneq s\} \cup \{s' \mid s' \in S' \wedge \neg\exists s \in S\ s \subsetneq s'\}$.

Equality testing given $S$ and $S'$, return whether $S = S'$.

*Example* 5.1. Let $S$ be the upper set generated by the antichain $\big\{\{a\},\{b,c\}\big\}$, and $S'$ the upper set generated by the antichain $\big\{\{b\},\{a,c\},\{d\}\big\}$. The union of the two upper sets is an up-set generated by the union of these two antichains. However, this union is not an antichain because it contains redundant items: $\{a,c\}$ is subsumed by $\{a\}$, $\{b,c\}$ is subsumed by $\{b\}$. The antichain of minimal elements of $S \cup S'$ is thus $\big\{\{a\},\{b\},\{d\}\big\}$.

The operations that we need for antichains defining lower sets, for the may-miss analysis, are

Initialization to empty or undefined cache state Return $\{\mathcal{A}\}$.

Initialization to unreachable state Return $\emptyset$.

Accesses Same as with upper sets.

Test for eviction Returns whether there exists $s \in S$ such that $|s| \geq N$, in which case $S$ is replaced by $\{\mathcal{A}\}$ (again, this is an optimization).

Union of lower sets represented by antichains of maximal elements $S$ and $S'$: return $\{s \mid s \in S \wedge \neg\exists s' \in S'\ s \subsetneq s'\} \cup \{s' \mid s' \in S' \wedge \neg\exists s \in S\ s' \subsetneq s\}$.

Equality testing Same as with upper sets.

[8] This means that execution traces that cannot lead to a "hit" on the next access to $a$ are discarded. This is correct since execution is assumed to start from all accesses to $a$ as well as initial control states. If one wishes to combine the analysis with others which need to distinguish between "no hit at the next access to $a$" and "unreachable", the special value $\mathcal{A}$ may be added when elements of too large associativity are discarded.

The union of antichains with subsumption removal was supported by an extension [31] of the ZDD library that we used. The only operations not supported were the test for eviction and the limitation to associativity. We implemented them by recursive descent over the structure of the ZDD, with an extra parameter for the current depth (number of items already seen in the set), and memoization of the results. As in the CUDD library, we call "then" the branch where the top variable is true (i.e. the branch that contains the cache block associated to the current node) and "else" the branch associated to value false (i.e. the branch that does not contain that cache block). As shown in Algorithm 1, the general case (case 3) of the algorithm simply consists in truncating the "then" and "else" branches of the current nodes. When the number of "then" branches taken reaches the associativity (case 2), we remove all further "then" branches (they only lead to sets of cardinality greater than the associativity). Finally, the algorithm may stop exploring a branch for two different reasons: a) either the node treated is a leaf of the ZDD (case 0), or b) the result of the truncate function has already been computed and memoized (case 1).

**Algorithm 1** Truncate(zdd, n) as a recursive function

```
1:  function TRUNCATE(zdd, n)
2:      if zdd = ∅ or zdd = {∅} then
3:          return zdd                                  ▷ Case 0. Leaf of the ZDD DAG
4:      end if
5:      res ← CACHELOOKUP(Truncate, zdd, n)    ▷ Case 1. Already computed
6:      if res then
7:          return res
8:      end if
9:      if n = 0 then                              ▷ Case 2. Associativity is reached
10:         return TRUNCATE(zdd.else, 0)          ▷ Case 2. Else branch recursion
11:     else                                                  ▷ Case 3. General case
12:         then ← TRUNCATE(zdd.then, n − 1) ▷ Case 3. Then branch recursion
13:         else ← TRUNCATE(zdd.else, n)          ▷ Case 3. Else branch recursion
14:         return ZDD(zdd.var, then, else)
15:     end if
16: end function
```

# 6 Variants and Extensions

**Combination with classical abstract interpretation** When classical abstract interpretation [14], or its combination with the "definitely unknown" abstract analysis [38], can correctly classify all accesses to a given block $a$ into "always hit", "always miss" and "definitely unknown", there is no use in running our analysis for that block. We have implemented this combination, which improves performance (see Section 8).

**Simultaneous computation** We have explained our analyses for classifying accesses to each address $a$ separately. It is also possible to simultaneously classify all addresses together, by updating the abstractions (e.g. $C^{\min}_{l,a}$) for all $a$ all together when updating the abstract state at location $l$.

This simultaneous computation, including across cache sets, is likely to be compulsory if the cache analysis is integrated with a microarchitectural analysis: if the sequence of memory accesses depends on whether some previous accesses are hits or misses, e.g. due to out-of-order execution or opportunistic prefetching.

**On-demand backward analysis** We have presented our analysis in a forward fashion: to classify hits and misses to $a$, we compute at each location the collection of the set of addresses found along path $\pi$ for all paths $\pi$ from the nearest preceding occurrences of $a$ (truncated at length $N$). We could formulate our analysis in a backward fashion: given a specific location $l$ in the control-flow graph, we compute at each location $l'$ the collection of the set of addresses found along path $\pi$ for all paths $\pi$ from $l'$ to $l$. This computation stops at other edges labeled with $a$, start vertices, or when computing the special value $\mathcal{A}$. Then, an edge going out of $l$ and labeled by $a$ may result in a miss if and only if at least one value $\mathcal{A}$ or an $\emptyset$ start vertex was reached during this backward propagation, and it may result in a hit if and only if at least one edge $a$ or a $\top$ start vertex was reached during this backward propagation.

# 7 Complexity and NP-Hardness

The cache contains at most $N$ cache lines chosen among $|A|$ memory blocks; the number of cache states is thus bounded by $\sum_{k=0}^{N} |A|^k = \frac{|A|^{N+1}-1}{|A|-1}$. Consequently, the total number of program states is bounded by $|V|\frac{|A|^{N+1}-1}{|A|-1}$ where $V$ is the set of vertices. Recall that $A$ is the set of possible addresses, which are used to label the edges $E$; thus $|A| \leq |E|$. For a fixed associativity $N$, an explicit model-checking approach, enumerating all cache states, thus has polynomial complexity in the size of the control-flow graph under analysis; however its complexity is exponential in the cache's associativity. Furthermore, for program analysis, any effective complexity beyond almost-linear in the size of the program is generally considered prohibitive. This explains the development of abstract interpretation, with some imprecision [14], as well as clever pre-analyses and model reductions before applying symbolic model checking [38]. We shall now see that cache analysis problems for LRU caches are NP-hard, even if the control-flow graph is acyclic.

We here assume an empty initial cache. The *may-hit* (respectively, *may-miss*) problem is: given a control-flow multigraph and a designated edge $e$, does there exist a path through the graph such that the access on edge $e$ is a hit (respectively, a miss)? The *always hit* (respectively, *always miss*) problem is its complement: is a given access in a control-flow graph always a hit (respectively,

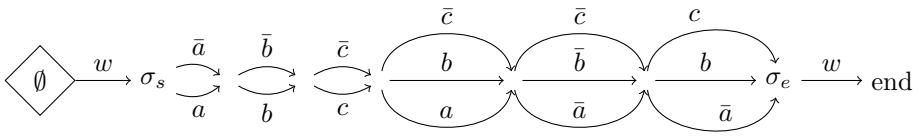



Figure 5: Reduction from Theorem 7.1. There is a path from $\sigma_s$ to $\sigma_e$ with at most $N-1=3$ different labels if and only if the formula $(\bar{c} \vee b \vee a) \wedge (\bar{c} \vee \bar{b} \vee \bar{a}) \wedge (c \vee b \vee \bar{a})$ has a model. This is equivalent to the existence of a path with a "hit" at the last $w$ edge for associativity $N=4$.

a miss) irrespective of how it is reached?

The input problem is given as (a) as in preceding sections, the control-flow multigraph, with edges labeled with the addresses of the data being accessed, (b) the designated edge to classify, and (c) the cache's associativity[9].

**Theorem 7.1.** *The may-hit problem is NP-complete for acyclic control-flow graphs.*

*Proof.* Obviously, the problem is in NP: a path may be chosen nondeterministically, then checked in polynomial time.

Now consider the following reduction from CNF-SAT (see Figure 5 for an example). To each variable $v$ in the SAT problem we associate two memory blocks $v$ and $\bar{v}$. The control-flow graph is a sequence of switches:

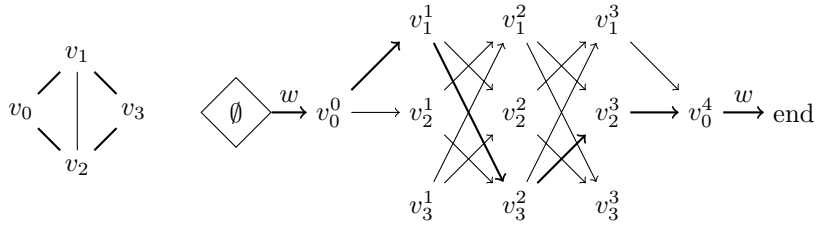
- For all variables $v$ in the SAT problem, a switch between two edges labeled with $v$ and $\bar{v}$, respectively.
- For each clause in the SAT problem, a switch between edges labeled with the literals present in the clause.

Let $n$ be the number of variables in the SAT problem. Each path through the sequence of switches with at most $n$ different labels corresponds to a satisfying assignment. Such a path exists if and only if the input formula is satisfiable.

Now add to the control-flow graph an incoming edge into the first node and an outgoing edge from the last node, both labeled with the same fresh letter $w$. The outgoing edge is the designated edge to classify. If the associativity of the cache is $n+1$, then the final access to $w$ may be a hit if and only if the SAT problem is satisfiable. □

**Theorem 7.2.** *The may-miss problem is NP-complete for acyclic control-flow graphs.*

[9]Note that if the associativity is larger than the set of possible edge labels, the two problems reduce to simple reachability problems in a directed graph. We can thus assume the associativity to be less than the number of edge labels. Whether the associativity is written in unary or binary form then is of no importance for the complexity of the problem.




(a) Graph with (thick) Hamiltonian cycle

(b) Acyclic control flow graph obtained by the reduction. Edge labels not shown. Path corresponding to the Hamiltonian cycle (thick).

Figure 6: Reduction from Theorem 7.2.

*Proof.* Obviously, the problem is in NP: a path may be chosen nondeterministically, then checked in polynomial time.

We reduce the Hamiltonian circuit problem to the may-miss problem (see Figure 6 for an example). Let $(V, E)$ be a graph, let $n = |V|$, $v = \{v_0, \ldots, v_{n-1}\}$ (the ordering is arbitrary). Let us construct an acyclic control-flow graph $G$ suitable for cache analysis as follows:

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

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 a start node into $v_0^0$, and an edge going from $v_0^n$ into an end node, both labeled with the same fresh letter $w$. The edge going from $v_0^n$ is the designated edge to classify. For associativity $n$ there exists an access missing the cache at that last edge if and only if there is a path from $v_0^0$ to $v_0^n$ with at least $n$ distinct edge labels. □

We have shown in this section how to construct CFGs for which solving the exist-miss and exist-hit problems is hard. Note that this implies that both problems are NP-complete in the general case, but not that there is no algorithm for efficiently dealing with ordinary CFGs.

# 8 Implementation and Experiments

We have implemented our antichain-based analysis, as well as the classical age-based analysis [14], and the "definitely unknown" (DU) and "focused" model-

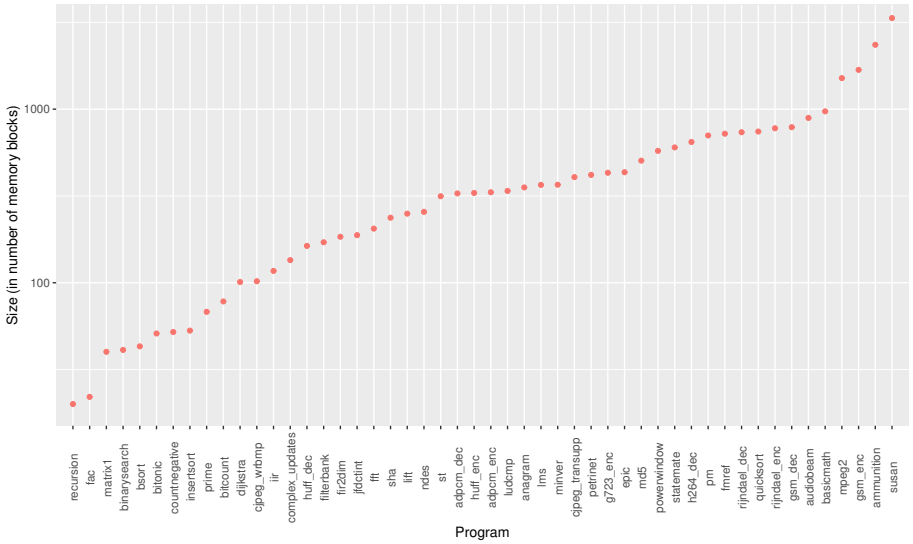



Figure 7: Size of the analyzed binary code.

checking analyses proposed by Touzeau et al. [38]. We did not implement the naive collecting semantics approach (model checking with "unfocused" cache states) since Touzeau et al. [38, §6.3] note that then the models become so large and complex that the model checker timed out on all of their examples. Furthermore, initial experiments with concretely represented antichains (an antichain being represented as a concrete set of arrays of block identifiers) scaled very poorly, so we did not pursue that direction further and pursued a fully symbolic representation using ZDDs.

Our experiments are performed on a server with 64 GB of memory, and an Intel Xeon CPU E5-2650 (32 logical cores running at 2.0 GHz). The tested implementation is fully sequential, and thus does not benefit from the high number of cores available. Note that the approach could however easily be implemented in parallel, by analyzing a different cache block on each core[10]. We analyze a 4 KB cache with 32 cache sets, 8 ways[11] and cache lines holding 16-byte-sized memory blocks.

We evaluate our approach on all sequential benchmarks, i.e., excluding par-

[10] Using threads if the ZDD library is capable of dealing with one different ZDD manager per thread, or separate processes.

[11] Note that associativities of 8 or even 16 are common in modern microarchitectures. For instance, in the AMD Ryzen microarchitecture [36], the L1 data cache and the unified L2 cache are 8-way set-associative, while the shared L3 cache even consists of 16 ways. Similarly, in the Intel Skylake microarchitecture [22], the L1 data and instruction caches are both 8-way set-associative and, depending on the specific model, the shared L3 cache consists of up to 16 ways.

allel benchmarks, from the TacleBench[12] [12] suite, which is also used by Touzeau et al. [38]. The benchmarks vary in size from 70 to 13000 lines of C code, and the size of the binary files obtained when compiling for ARM 5 (supported by Otawa) are shown in Figure 7. Sizes are given in the number of memory blocks, and range from 20 blocks for the smallest benchmark to 3348 blocks for the biggest benchmark. When measuring the time and memory consumption of analyses, we use a timeout of 12 hours. Consequently, when this timeout is reached (i.e. the approach did not finish classifying accesses in the available amount of time), the associated point is plotted as if the corresponding analysis had terminated after 12 hours.

We have implemented our analyses on top of Otawa [1], an open-source WCET analysis tool.[13] Computations over ZDDs are performed by Cudd 2.3.1 [37] together with an extension [31] for computing over antichains[14]. In order to compare our new analysis to the previous analysis by Touzeau et al. [38], we reimplemented it within Otawa, as our previous experiments were conducted at the level of the intermediate representation of the LLVM compiler suite rather than on machine code, as our present analysis. Recall that our analyses and theirs compute exactly the same classifications and differ only in memory and time consumption; this enabled us to test and debug our implementation.

There are several comparisons that the interested reader would have perhaps appreciated, but that we were unable to perform. We are not able to directly confront our implementation to Touzeau et al.'s because theirs operates, as a proof of concept, upon LLVM's intermediate representation, using a fake memory mapping, while ours operate upon machine code with the true memory mapping. We were not able to measure the precision gained on the WCET upper bound computed by Otawa by replacing the imprecise age-based static analysis [14] by our precise analysis, due to engineering issues — our analysis is implemented on top of Otawa version 2, which is under development and constantly evolving. We expect to be able to connect our analysis to the WCET computation in Otawa in a matter of months. Moreover, our experiments are performed on an instruction cache. As mentioned in Section 2, analyzing data caches is possible but would require further engineering effort to connect to Otawa's pointer analyses.

## 8.1 Implementation and Evaluation of our Analysis

Figure 8 shows how we integrated our analyses into the Otawa framework, and how the operations described in Section 5 interact with each other. The main component of our Otawa plugin is the "ZDD Analysis" box, which classifies accesses by abstract interpretation by calling the generic abstract interpretation engine provided by Otawa. This iterates on the CFG and calls the update and join operation we provide when needed. The join operation is realized by

[12] TacleBench is available at `https://github.com/tacle/tacle-bench.git`.

[13] We have used the version 2 obtained at `https://www.tracesgroup.net/otawa/download/otawa-v2/`.

[14] This extension has not been ported to more recent versions of Cudd.

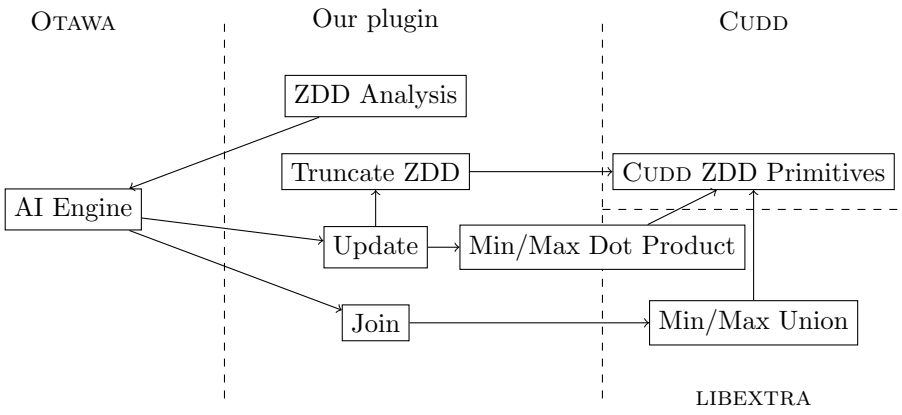


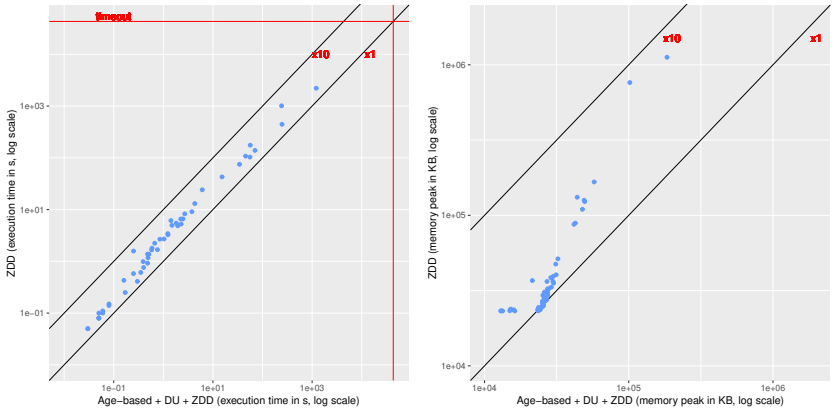

Figure 8: Overview of our framework. Edges represent dependencies ($u \to v$ means that code in $u$ calls some methods in $v$).

computing the union of a given pair of ZDDs, and then removing the subsumed sets (as explained in section Section 5). This is done by using the MinUnion (respectively MaxUnion) function provided by libextra, which compute the minimal elements of the union of two ZDDs. The update function models the effect of accessing a block: to this end, the accessed block is added to all sets represented by the current ZDD. This operation could be performed by the DotProduct function of libextra which, given two ZDDs $S_1$ and $S_2$, computes the set $S = \{s_1 \cup s_2, s_1 \in S_1, s_2 \in S_2\}$. In practice, we use the MaxDotProduct provided by libextra which only keeps the maximal elements of $S$. However, libextra does not provide the dual MinDotProduct, whose implementation we added. Once the new block is added to the current ZDD, we truncate the ZDD, keeping only those sets whose size is below the associativity. This is achieved by the Truncate algorithm described in Section 5.

We implemented two different versions of our analysis:

- *ZDD*: The version described in Section 5. One analysis is performed for every memory block in the program to classify all accesses to this block. When an analysis terminates, all the structures it used are freed and Cudd cache and memoization tables are flushed. This approach is referred to as *ZDD* in the following.

- *Age-based + DU + ZDD*: The last version uses the *Age-based + DU* analyses to classify memory accesses, and refines the accesses left "unknown" by running the associated ZDD analysis. In other words, this approach is the same as Touzeau et al. [38], where the model-checking phase is replaced by our *ZDD* approach.




(a) Time comparison (b) Memory comparison

Figure 9: ZDD vs. *Age-based* + DU + ZDD.

### 8.1.1 Comparison of the two Variants of our Analysis

Remember that the two versions of our analysis obtain the same access classifications. They differ only in the way the classification is obtained. Figure 9 shows that the approach combining *ZDDs* with a pre-analysis based on the age-based and DU analyses, is more efficient in terms of analysis execution time and memory consumption than the *ZDD* approach alone. The pre-analysis is performed using a single pass over the whole program for all memory blocks. As it successfully classifies most accesses as "always hit", "always miss", or "definitely unknown", the ZDD approach needs to be run only on a relatively small subset of all memory blocks. We will keep this *Age-based + DU + ZDD* variant as a basis for the following experiments.

### 8.1.2 Scalability with Respect to the Associativity

As mentioned in Section 7, the *may-hit* and *may-miss* problems are NP-complete, when the cache's associativity is considered an input parameter. In practice, however, repeated accesses to the same block are infrequent and the CFGs' branching structures are simple. We thus evaluate the running time of our analysis when increasing the associativity of the cache, while keeping the same cache size (thus decreasing the number of cache sets). Figure 10 shows that our analysis scales well for usual values of associativity; the increase in analysis time is usually proportional to the increase in associativity.

In the next section we experimentally evaluate how our approach compares with previous work. Using a first set of experiments, we compare our approaches to the model-checking approach described in Touzeau et al. [38]. Then, we show that our analysis is similarly efficient as the age-based analysis of Ferdinand and Wilhelm [14] and the DU analysis of Touzeau et al. [38] in terms of memory

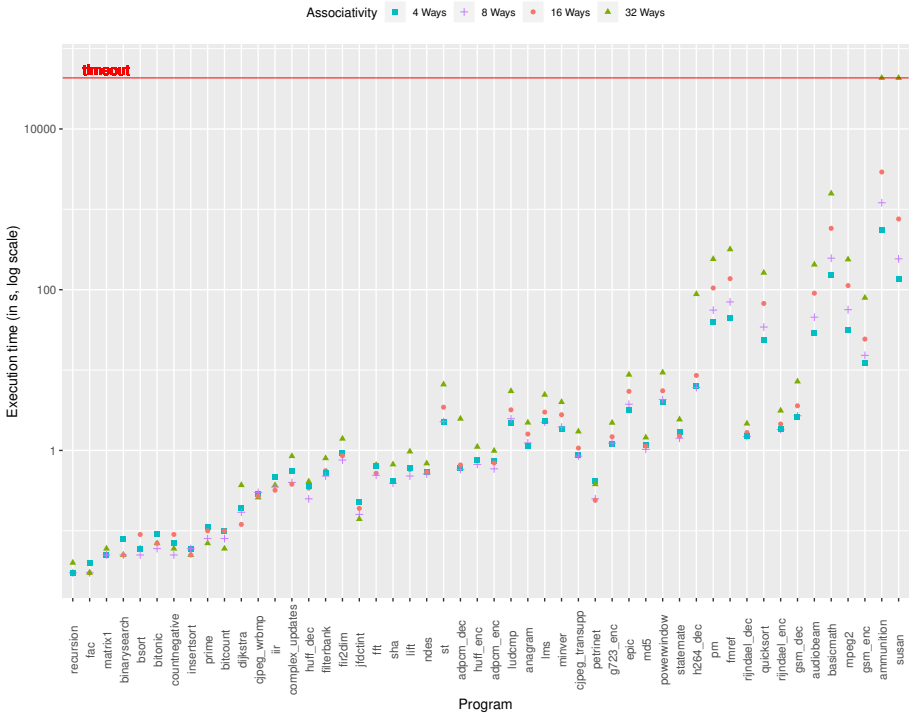



Figure 10: Evolution of execution time of our analysis when increasing associativity.

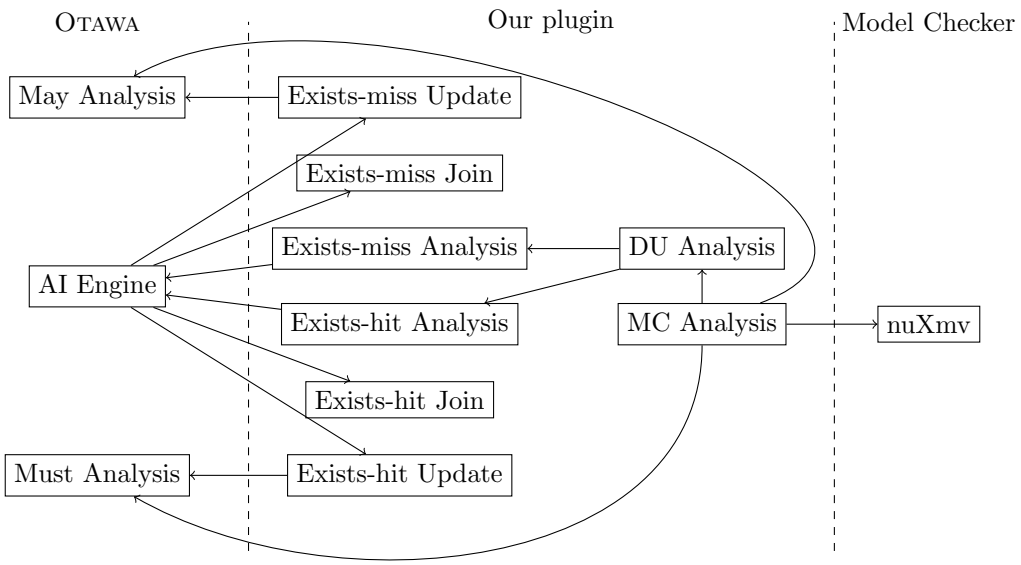



Figure 11: Integration of Touzeau et al. [38] in Otawa.

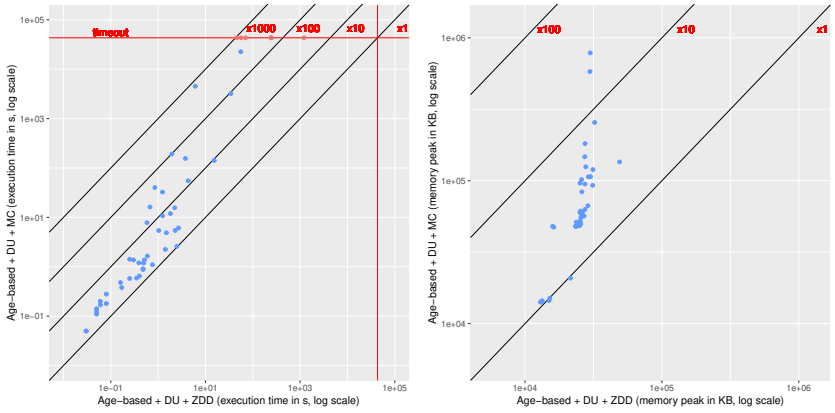
usage and analysis time.

## 8.2 Comparison with Prior Work

Among the existing cache analyses, the approach of Touzeau et al. [38] is the closest to our work. It consists of the following three steps:

- First, the usual analysis of Ferdinand and Wilhelm [14] (which we refer to as *Age-based* analysis in following) is performed and classifies accesses as "always hit", "always miss" or "unknown".
- Then, a second analysis, called definitely unknown, classifies a subset of the "unknown" accesses as "definitely unknown" when it finds both a path leading to hit and a path leading to a miss for a given access (this is an approximated analysis — it may fail to identify such paths).
- Finally, the remaining unknown accesses are classified using a model checker (MC) and marked as "always hit", "always miss" or "definitely unknown".

Our implementation of this approach in Otawa is illustrated in Figure 11. Note that the DU analysis is based on two approximate analyses: an "exists hit" analysis, which determines whether a path leading to a hit exists, and an




(a) Time comparison (b) Memory comparison

Figure 12: ZDD approach vs. model-checking approach from [38].

"exists miss" analysis which determines whether a path leading to a miss exists. Note that the DU analysis reuses the abstract cache states from *Age-based* May/Must analyses: in our implementation, *Age-based* analysis is provided by Otawa. Finally, the accesses that are not classified precisely by the two abstract interpretation phases are refined using a call to the nuXmv model checker, which processes a model focussed on the memory block under analysis.

### 8.2.1 Comparison to Touzeau et al. [38]

As the prior work of Touzeau et al. and ours provide the same classification of memory accesses, we compare the analyses according to two criteria:

a) The full analysis time, including every analysis step from CFG reconstruction to memory-access classification, is compared in Figure 12a.

b) The peak memory usage of the two approaches is compared in Figure 12b.

The scatter plots are both on a log. scale. Each dot corresponds to the resource consumption of one benchmark from TacleBench under the two analysis approaches.

The figures clearly show that the ZDD approach is significantly faster than the model-checking approach (more than a hundred times faster for the largest benchmarks) and that the benefits increase with the size of the benchmarks. The peak memory usage is also generally smaller with our ZDD approach than with model checking.

### 8.2.2 Overhead of ZDD over *Age-based + DU*

We evaluate the scalability of our approach comparing it to the usual *Age-based* analysis and DU analysis. Figure 13 and Figure 14 show the execution time

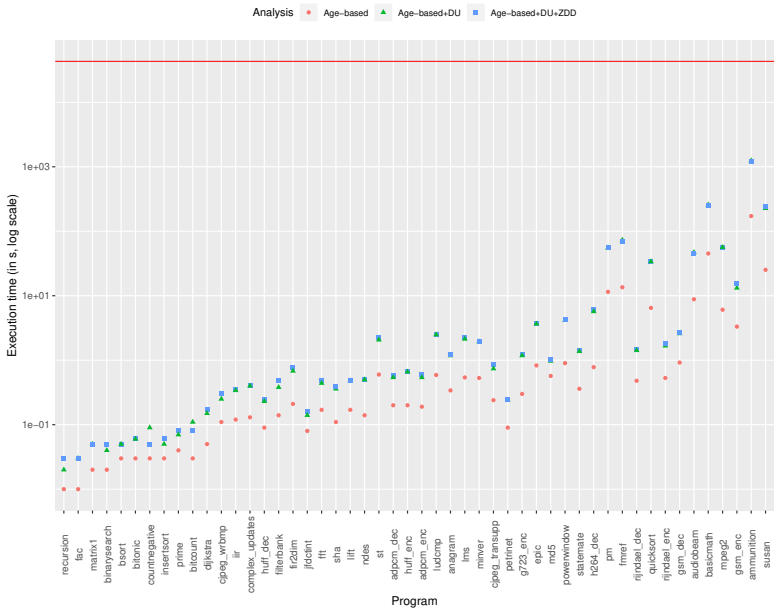



Figure 13: Execution time overhead.

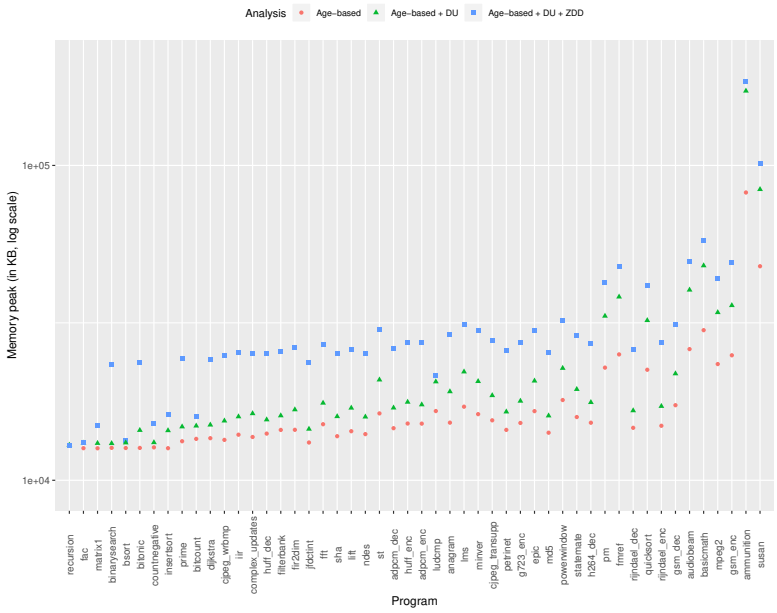



Figure 14: Memory overhead.

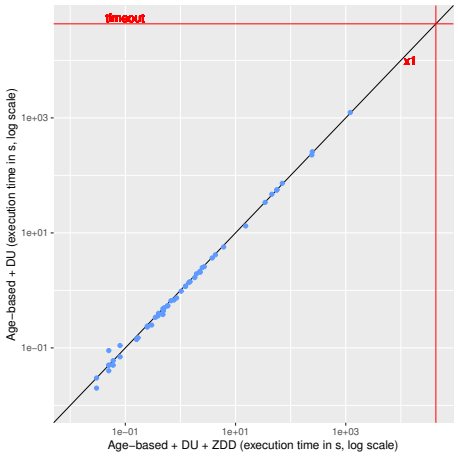



Figure 15: Execution time overhead.

of our approach compared to *Age-based* and *Age-based + DU* on a logarithmic scale. The vertical distance between two points can be interpreted as the slowdown due to the additional analysis[15]. On average, our approach is 3.46 times more costly than the usual *Age-based* analysis, and only adds a 4.6% overhead over the *Age-based + DU* analysis, as shown in Figure 15.

To conclude, the ZDD approach is significantly faster than the exact analysis by Touzeau et al. [38]. This validates our choice of algorithms and data structures. The results demonstrate that the ZDD approach offers good scalability that could lead to an industrial use.

### 8.2.3 Scalability Comparison with Other Previous Work

The approaches of Chattopadhyay and Roychoudhury [7] and Chu, Jaffar, and Maghareh [8] analyze programs at the source code level and are thus not directly comparable to our work (which analyzes binary code), if only because we map statements to cache blocks differently.

The only common benchmark with ours is STATEMATE: for this benchmark their analysis stops after 100 calls to the model checker and the analysis spends 195 seconds, where we analyze the whole benchmark in less than 2 seconds. Similarly, in Chu, Jaffar, and Maghareh [8] the paper states an analysis time of 350 seconds for STATEMATE and 38 seconds for the benchmark NDES where we analyze them in less than 2 seconds and less than 1 second.

Unfortunately, there are no other common benchmarks; it however seems our analysis scales much better than these two previous works.

[15] Note that for very small benchmarks the time measurements are not very reliable, as the reported values correspond to single measurements. Experiments with COUNTNEGATIVE showed variations of up to 30% from one measurement to another.

# 9 Related Work

**Cache analysis for the verification of real-time systems** Static cache analysis was first studied in the context of real-time systems [29]. Mueller and Whalley [33] introduced a data-flow analysis for direct-mapped caches, i.e., for caches with associativity 1. Based on abstract interpretation (AI), Ferdinand and Wilhelm [14] proposed the classical age-based analysis for set-associative caches with LRU replacement. Their analysis is still in widespread use in commercial and academic WCET analysis tools, e.g. [21, 1].

As discussed in Section 3, Ferdinand and Wilhelm's analysis can be seen as computing a range of possible ages for each memory block in a given program. On straight-line code this analysis is exact. However, at control-flow joins the relation between ages of different blocks may be lost. As a consequence, the analysis may classify some accesses as "unknown" that are in fact always hits or always misses. There have been several attempts to improve upon the precision of the classical AI-based cache analysis:

Chattopadhyay and Roychoudhury [7] refines memory accesses classified as "unknown" by AI using a software model-checking step: when abstract interpretation cannot classify an access, the source program is enriched with annotations for counting conflicting accesses and run through a software model checker. Their approach, in contrast to ours, takes into account program semantics during the refinement step; it is thus likely to be more precise on programs where many paths are infeasible for semantic reasons. Our approach however scales considerably better, as shown in Section 8. (Our approach can also be combined with analyses for checking the feasibility of paths, as sketched in Section 10; we however have not experimented with it yet). They advocated applying their method with a time bound. Accesses classified as "unknown" by AI are refined until the time bound is reached. Chu, Jaffar, and Maghareh [8] present a WCET analysis framework based on symbolic execution, where an SMT solver is used to prune infeasible paths. They employ the age-based abstraction of Ferdinand and Wilhelm within symbolic execution, but never join states, thus avoiding any imprecision. See Section 8.2.3 for a performance comparison of these two analyses with ours.

Touzeau et al. [38] refine accesses classified as "unknown" by AI using model checking similarly to Chattopadhyay and Roychoudhury [7]. In order to reduce the number of calls to the model checker, they introduce an AI-based analysis that can classify accesses as "definitely-unknown". For a "definitely-unknown" access both a hit and a miss are possible depending on the path taken through the control flow graph to reach the access. The classification of such accesses cannot be refined by model checking. To reduce the effort of the model checker in classifying a particular access, they also introduce a "focused semantics", which we reuse and extend in this paper, as discussed in Section 4. We compare the efficiency of their approach with ours in Section 8.

Abstract interpretation has also been applied to the analysis of caches with other popular replacement policies found in modern microarchitectures, such as *first-in, first-out* (FIFO) [16, 17, 19], *not most-recently-used* (NMRU) [20],

and *pseudo-LRU* (PLRU) [18]. No exact analysis has been proposed for these policies, which are considered to be harder to analyze than LRU [35]. It is doubtful whether our approach can be extended to these policies, as they do not seem to exhibit any useful monotonicity properties as LRU does.

**Compiler optimizations** Optimizing compilers may apply loop transformations to maximize parallelism and data locality [13, 25]. To support such optimizations, various approaches have been proposed to compute or approximate the number of cache misses of a given loop nest [15, 6, 5, 4, 2]. This line of work is limited to restricted classes of programs, usually affine loop nests with no input-dependent control flow or input-dependent memory accesses. The advantage of such methods is that they distinguish each dynamic instance of an instruction in a loop nest, while our approach classifies all dynamic instances together, thereby introducing pessimism. It would be interesting to investigate whether the exact abstraction developed in this paper could be combined with analytical approaches such as [2] to support input-dependent program behavior.

**Cache side-channel analysis** Caches can be exploited as covert channels [24, 26] and in side-channel attacks [3, 32, 27, 42]. Static cache analysis has been applied to quantify the vulnerability of implementations of cryptographic protocols [11, 10] to cache side-channel attacks. More accurate cache analyses, such as the one developed in this paper, may yield more accurate vulnerability quantifications. Applying our analysis in this context is future work.

**Antichains** Antichains have been used in verification to represent lower and upper sets, which occur in many contexts (automata, LTL satisfiability, games, etc.). Wulf et al. [41, 40] proposed two succinct representations: (i) *fully symbolic*: a binary decision diagram (BDD) represents sets of sets of states: each automaton state is mapped to a BDD variable, a set of states is thus a valuation; this representation is thus similar to ours except that they use BDDs and not ZDDs; (ii) *semi symbolic*: a state is encoded as an integer, thus as a vector of bits, a set of states is thus encoded as a BDD, and an antichain is thus a sequence of BDDs; this representation is thus similar to our explicit list of sets of blocks, which we tried then discarded due to inefficiency.

They report that, on their examples, the fully symbolic representation is less efficient than the semi symbolic one, particularly on large automata; the explain this by the linear growth of the number of variables in the BDDs for the fully symbolic representation with respect to automaton size, as opposed to logarithmic growth in the semi symbolic representation. We explain this difference with our own findings as follows: (1) Their sets of states correspond to sets of blocks in our problem. We limit the cardinality of the sets we handle to associativity, whereas, as far as we understand, their sets can be very large. (2) We use ZDDs, which are more compact than BDDs for sets of small sets (no need for nodes "this element is not in the set"). (3) Their antichains are smaller than ours. Our ZDD approach thus seems efficient for large, often similar

antichains of small sets, whereas their semi symbolic approach seems efficient for small antichains of large sets.

# 10 Conclusion and Future Work

For decades, it was believed that only rough abstractions (Ferdinand's analysis) could scale up for cache analyses. We show here that it is actually possible to perform an exact analysis by carefully refining the abstraction and using good algorithms and data structures.

We have demonstrated how it is possible to obtain a cache analysis as precise as one obtained by analyzing the cache replacement policy using a model checker, but at a much lower cost (hundreds of times faster). The main difference between the two analysis is that we abstract the problem further while preserving the exact same results.

Our results are the strongest possible (exactly classifying accesses as "always hit", "always miss", "hits or misses depending on the execution") with respect to a model where all paths inside the control-flow graph may be taken. This includes paths that cannot be taken inside the program, e.g. ones with conflicting tests $x < 1$ and $x > 2$ with no change to $x$ in between. It is impossible to remove all spurious paths: this is an undecidable question. We can however combine our analysis with others to improve the precision in this respect.

Our analysis computes an abstract state, as opposed to encoding everything in a model checker. As such, it can be combined simultaneously with other abstract interpretations (program variables, pointers, micro-architecture...) and approaches such as the *abstract reachability graph*, with nodes adorned by pairs (location, abstract state), edges adorned by instructions and where each test, at least up to a certain depth, is split into two branches; two nodes may be merged, as in a directed acyclic graph, if one abstract state is subsumed by the other; cycles in the graph may be introduced, perhaps after widening operations, to account for loops. Such an analysis would exclude some infeasible paths.

Our approach applies both to instruction and data caches. If the address of a data fetch or write is only known to lie within a set of addresses $\{a_1, \ldots, a_n\}$, then we consider n parallel edges labeled with $a_1, \ldots, a_n$ in the analysis graph. As future work, trace partitioning may be used to improve the precision of such an approach, and can be implemented as a layer above our analysis.

# References


[1] Clément Ballabriga et al. "OTAWA: An Open Toolbox for Adaptive WCET Analysis". In: *Software Technologies for Embedded and Ubiquitous Systems - 8th IFIP WG 10.2 International Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings*. Ed. by Sang Lyul Min et al. Vol. 6399. Lecture Notes in Computer Science. Springer, 2010, pp. 35–46. DOI: 10.1007/978-3-642-16256-5_6.

[2] Wenlei Bao et al. “Analytical Modeling of Cache Behavior for Affine Programs”. In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 32:1–32:26. ISSN: 2475-1421. DOI: 10.1145/3158120. URL: http://doi.acm.org/10.1145/3158120.

[3] Daniel J. Bernstein. *Cache-timing attacks on AES*. 2005. URL: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[4] Kristof Beyls and Erik H. D'Hollander. “Generating cache hints for improved program efficiency”. In: *Journal of Systems Architecture* 51.4 (2005), pp. 223 –250. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2004.09.004. URL: http://www.sciencedirect.com/science/article/pii/S1383762104001213.

[5] Calin Cascaval and David A. Padua. “Estimating Cache Misses and Locality Using Stack Distances”. In: *Proceedings of the 17th Annual International Conference on Supercomputing*. ICS '03. San Francisco, CA, USA: ACM, 2003, pp. 150–159. ISBN: 1-58113-733-8. DOI: 10.1145/782814.782836. URL: http://doi.acm.org/10.1145/782814.782836.

[6] Siddhartha Chatterjee et al. “Exact Analysis of the Cache Behavior of Nested Loops”. In: *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. PLDI '01. Snowbird, Utah, USA: ACM, 2001, pp. 286–297. ISBN: 1-58113-414-2. DOI: 10.1145/378795.378859. URL: http://doi.acm.org/10.1145/378795.378859.

[7] Sudipta Chattopadhyay and Abhik Roychoudhury. “Scalable and precise refinement of cache timing analysis via path-sensitive verification”. In: *Real-Time Systems* 49.4 (2013), pp. 517–562. DOI: 10.1007/s11241-013-9178-0. URL: http://dx.doi.org/10.1007/s11241-013-9178-0.

[8] Duc-Hiep Chu, Joxan Jaffar, and Rasool Maghareh. “Precise Cache Timing Analysis via Symbolic Execution”. In: *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*. IEEE Computer Society, 2016, pp. 293–304. ISBN: 978-1-4673-8639-5. DOI: 10.1109/RTAS.2016.7461358. URL: http://dx.doi.org/10.1109/RTAS.2016.7461358.

[9] Patrick Cousot. “Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes”. Thèse d'état ès sciences mathématiques. Grenoble, France: Université scientifique et médicale de Grenoble, Mar. 1978. URL: https://tel.archives-ouvertes.fr/tel-00288657/en/.

[10] Goran Doychev and Boris Köpf. “Rigorous Analysis of Software Countermeasures Against Cache Attacks”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 406–421. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062388. URL: http://doi.acm.org/10.1145/3062341.3062388.

[11] Goran Doychev et al. “CacheAudit: A Tool for the Static Analysis of Cache Side Channels”. In: *ACM Trans. Inf. Syst. Secur.* 18.1 (June 2015), 4:1–4:32. ISSN: 1094-9224. DOI: 10.1145/2756550. URL: http://doi.acm.org/10.1145/2756550.

[12] Heiko Falk et al. “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, 2:1–2:10. DOI: 10.4230/OASIcs.WCET.2016.2. URL: http://dx.doi.org/10.4230/OASIcs.WCET.2016.2.

[13] Paul Feautrier. “Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time”. In: *International Journal of Parallel Programming* 21.6 (1992), pp. 389–420. ISSN: 1573-7640. DOI: 10.1007/BF01379404. URL: https://doi.org/10.1007/BF01379404.

[14] Christian Ferdinand and Reinhard Wilhelm. “Efficient and Precise Cache Behavior Prediction for Real-Time Systems”. In: *Real-Time Systems* 17.2-3 (1999), pp. 131–181. DOI: 10.1023/A:1008186323068.

[15] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. “Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior”. In: *ACM Trans. Program. Lang. Syst.* 21.4 (July 1999), pp. 703–746. ISSN: 0164-0925. DOI: 10.1145/325478.325479. URL: http://doi.acm.org/10.1145/325478.325479.

[16] Daniel Grund and Jan Reineke. “Abstract Interpretation of FIFO Replacement”. In: *Static Analysis, 16th International Symposium, SAS 2009*. Ed. by Jens Palsberg and Zhendong Su. Vol. 5673. LNCS. Springer-Verlag, 2009, pp. 120–136. DOI: 10.1007/978-3-642-03237-0. URL: http://rw4.cs.uni-saarland.de/~grund/papers/sas09-AI_FIFO.pdf.

[17] Daniel Grund and Jan Reineke. “Precise and Efficient FIFO-Replacement Analysis Based on Static Phase Detection”. In: *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS '10)*. Brussels, Belgium, 2010, pp. 155–164. ISBN: 978-1-4244-7546-9. DOI: 10.1109/ECRTS.2010.8. URL: http://rw4.cs.uni-saarland.de/~grund/papers/ecrts10-fifo_phases.pdf.

[18] Daniel Grund and Jan Reineke. “Toward Precise PLRU Cache Analyis”. In: *Proceedings of 10th International Workshop on Worst-Case Execution Time (WCET) Analysis*. Ed. by Björn Lisper. Austrian Computer Society, 2010, pp. 28–39. URL: http://rw4.cs.uni-saarland.de/~grund/papers/wcet10-plru.pdf.

[19] Nan Guan et al. “FIFO Cache Analysis for WCET Estimation: A Quantitative Approach”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE 2013. Grenoble, France: EDA Consortium, 2013, pp. 296–301. ISBN: 978-1-4503-2153-2. URL: http://dl.acm.org/citation.cfm?id=2485288.2485362.

[20] Nan Guan et al. “WCET Analysis with MRU Cache: Challenging LRU for Predictability”. In: *ACM Trans. Embed. Comput. Syst.* 13.4s (Apr. 2014), 123:1–123:26. ISSN: 1539-9087. DOI: 10.1145/2584655. URL: http://doi.acm.org/10.1145/2584655.

[21] Reinhold Heckmann and Christian Ferdinand. “Worst-Case Execution Time Prediction by Static Program Analysis”. In: *The whitepaper of aiT* (2014). URL: http://www.absint.com/aiT_{W}{C}{E}{T}.pdf.

[22] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Order Number: 248966-033. Intel Corporation. June 2016. URL: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[23] Donald E. Knuth. *The Art of Computer Programming: Combinatorial Algorithms, part 1*. Vol. 4A. Pearson, 2011.

[24] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *CoRR* abs/1801.01203 (2018). arXiv: 1801.01203. URL: http://arxiv.org/abs/1801.01203.

[25] Amy W. Lim and Monica S. Lam. “Maximizing Parallelism and Minimizing Synchronization with Affine Transforms”. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’97. Paris, France: ACM, 1997, pp. 201–214. ISBN: 0-89791-853-3. DOI: 10.1145/263699.263719. URL: http://doi.acm.org/10.1145/263699.263719.

[26] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[27] Fangfei Liu et al. “Last-Level Cache Side-Channel Attacks are Practical”. In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 605–622. DOI: 10.1109/SP.2015.43.

[28] Thomas Lundqvist and Per Stenström. “Timing Anomalies in Dynamically Scheduled Microprocessors”. In: *20th IEEE Real-Time Systems Symposium (RTSS)*. 1999. ISBN: 0-7695-0475-2.

[29] Mingsong Lv et al. “A Survey on Static Cache Analysis for Real-Time Systems”. In: *Leibniz Transactions on Embedded Systems* 3.1 (2016), 05–1–05:48. ISSN: 2199-2002. DOI: 10.4230/LITES-v003-i001-a005. URL: https://ojs.dagstuhl.de/index.php/lites/article/view/LITES-v003-i001-a005.

[30] Shin-ichi Minato. “Zero-suppressed BDDs and their applications”. In: *Int. J. on Software Tools for Technology Transfer (STTT)* 3.2 (2001), pp. 156–170. DOI: 10.1007/s100090100038. URL: http://hdl.handle.net/2115/16895.

[31] Alan Mishchenko. "An introduction to zero-suppressed binary decision diagrams". In: *Applications of Zero-Suppressed Decision Diagrams*. Ed. by Tsutomu Sasao and Jon T. Butler. Morgan Claypool, 2014. ISBN: 9781627056496. URL: https://people.eecs.berkeley.edu/~alanmi/publications/2001/tech01_zdd_.pdf.

[32] Keaton Mowery, Sriram Keelveedhi, and Hovav Shacham. "Are AES x86 Cache Timing Attacks Still Feasible?" In: *Cloud Computing Security Workshop*. Raleigh, North Carolina, USA: ACM, 2012, pp. 19–24. ISBN: 978-1-4503-1665-1. DOI: 10.1145/2381913.2381917.

[33] Frank Mueller and David B. Whalley. "Fast Instruction Cache Analysis via Static Cache Simulation". In: *Proceedings of the 28th Annual Simulation Symposium*. 1995, pp. 105–114.

[34] Jan Reineke et al. "A Definition and Classification of Timing Anomalies". In: *6th International Workshop on Worst-Case Execution Time Analysis (WCET)*. 2006.

[35] Jan Reineke et al. "Timing Predictability of Cache Replacement Policies". In: *Real-Time Systems* 37.2 (2007), pp. 99–122. DOI: 10.1007/s11241-007-9032-3. URL: http://rw4.cs.uni-saarland.de/~grund/papers/rts07-predictability.pdf.

[36] *Software Optimization Guide for AMD Family 17h Processors*. Publication No. 55723, Revision 3.00. Advanced Micro Devices. June 2017. URL: https://developer.amd.com/wordpress/media/2013/12/55723_3_00.ZIP.

[37] Fabio Somenzi. "Efficient manipulation of decision diagrams". In: *Int J. Software Tools for Technology Transfer (STTT)* 3.2 (2001), pp. 171–181. DOI: 10.1007/s100090100042.

[38] Valentin Touzeau et al. "Ascertaining Uncertainty for Efficient Exact Cache Analysis". In: *Computer-aided verification (CAV)*. Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10427. Lecture Notes in Computer Science. Springer, 2017, pp. 22–40. DOI: 10.1007/978-3-319-63390-9_2. arXiv: 1709.10008.

[39] Reinhard Wilhelm et al. "The worst-case execution-time problem - overview of methods and survey of tools". In: *ACM Trans. Embedded Comput. Syst.* 7.3 (2008), 36:1–36:53. DOI: 10.1145/1347375.1347389.

[40] Martin De Wulf et al. "Alaska: Antichains for Logic, Automata and Symbolic Kripke structures Analysis". In: *Automated Technology for Verification and Analysis, 6th International Symposium, ATVA 2008, Seoul, Korea, October 20-23, 2008. Proceedings*. Ed. by Sung Deok Cha et al. Vol. 5311. Lecture Notes in Computer Science. Springer, 2008, pp. 240–245. DOI: 10.1007/978-3-540-88387-6_21.

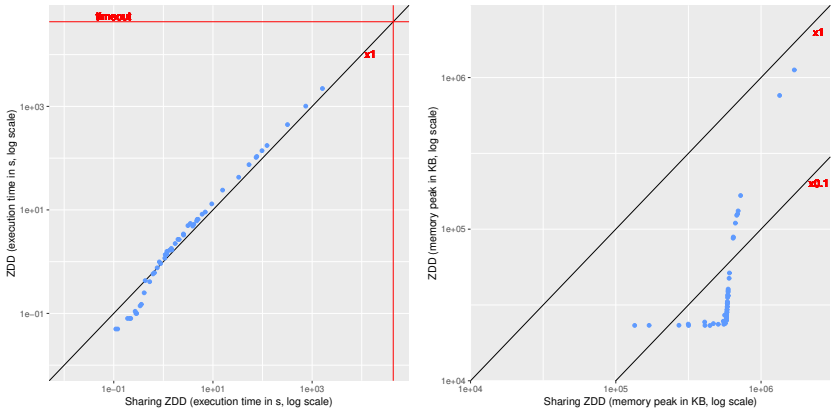
[41] Martin De Wulf et al. “Antichains: A New Algorithm for Checking Universality of Finite Automata”. In: *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Ed. by Thomas Ball and Robert B. Jones. Vol. 4144. Lecture Notes in Computer Science. Springer, 2006, pp. 17–30. DOI: 10.1007/11817963_5.

[42] Yuval Yarom, Daniel Genkin, and Nadia Heninger. “CacheBleed: a timing attack on OpenSSL constant-time RSA”. In: *Journal of Cryptographic Engineering* 7.2 (2017), pp. 99–112. ISSN: 2190-8516. DOI: 10.1007/s13389-017-0152-y. URL: https://doi.org/10.1007/s13389-017-0152-y.

# A Sharing ZDD implementation

BDD (or ZDD) libraries intensively use hash tables for **hash-consing** the BDD nodes: when the library wishes to create a node isomorphic to one that already exists in the system, that one is used instead; **memoizing** the BDD operations: during a recursive operation $f$ on $n$-tuples of BDDs nodes, the computed values of $f$ are stored into a hash table and are retrieved if the same $n$-tuple is encountered instead of recursing; this ensures that an operation of BDDs $D_1, \ldots, D_n$ is (roughly) in time $\prod_i |D_i|$. The requirements are that all currently reachable BDD nodes should be retained in the hash-consing table (but other nodes may be retained), and that (at least for ensuring polynomial computations) during one BDD operation the memoizing hash tables should not be flushed (but they may be retained longer). Depending on the BDD library in use and its parameters, "garbage" in these tables may be collected eagerly (at the risk of having to recreate or recompute collected data) or late (at the risk of storing useless data).

In our initial implementation, the analyses for various focus blocks $a$ were run completely separately; all ZDD tables were flushed in between. We implemented a variation of the *ZDD* approach that does *not* free all structures and CUDD's data between two analysis runs for different memory blocks. Doing so, the analysis may benefit from computations done in a previous analysis run if the result of a particular computation has been memoized.




(a) Time comparison (b) Memory comparison

Figure 16: *ZDD* vs. *Sharing ZDD*

Comparing the *ZDD* and *Sharing ZDD* variants, we observe that sharing of ZDDs between analysis runs is not very beneficial in our case (Figure 16): *Sharing ZDD* does not significantly improve the analysis execution time but is more costly in terms of peak memory usage. It seems that there is not much sharing between successive analyses and that the library just fills the memory up to a certain threshold before collecting garbage; our knowledge of CUDD's

internals is however insufficient to check whether this explanation is correct.

This motivates our yet unimplemented idea of parallelizing the analysis by running it for different values of $a$ on different cores, completely separate from each other. Note that this is much easier than running analyses sharing a single ZDD manager, due to contention on the hash tables.

# B Comparison of all exact analyses

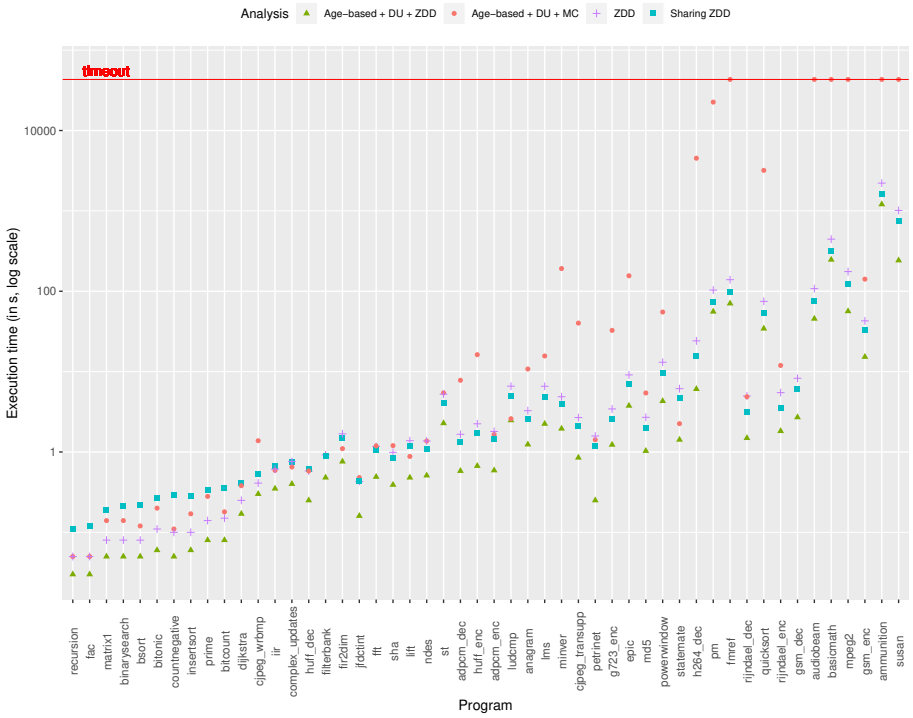



Figure 17: Execution time of all exact analyses.

Figures Figure 17 and Figure 18 show the execution time and memory consumption all fully-precise analyses we have implemented.

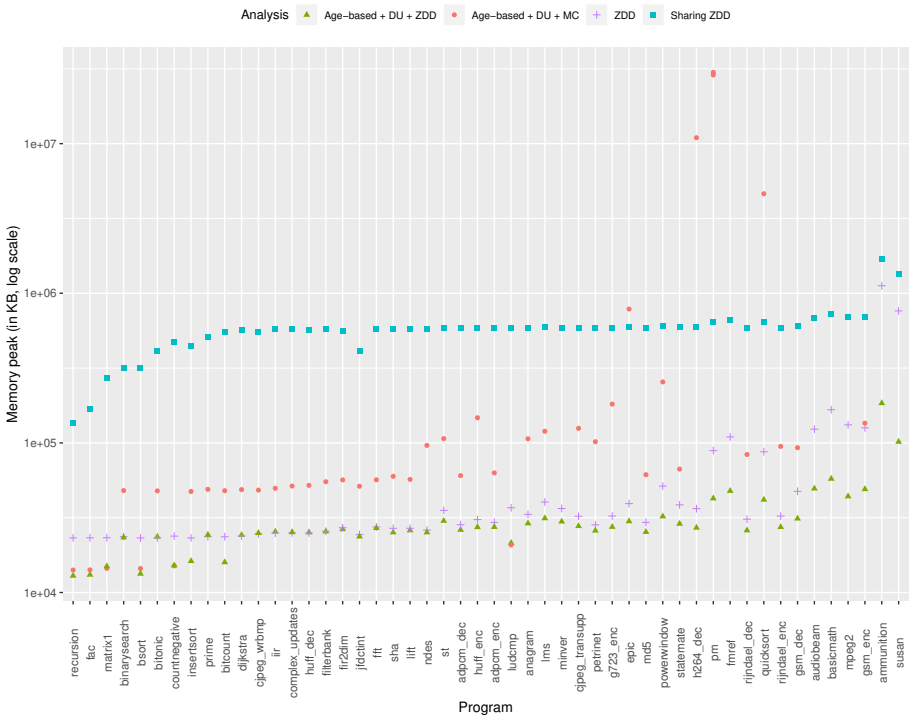



Figure 18: Memory consumption of all exact analyses.