Ascertaining Uncertainty for Efficient Exact Cache Analysis

Valentin Touzeau[1], Claire Maïza[1], David Monniaux[1], and Jan Reineke[2]

[1] Univ. Grenoble Alpes, VERIMAG, F-38000 Grenoble, France
CNRS, VERIMAG, F-38000 Grenoble, France
firstname.lastname@univ-grenoble-alpes.fr
[2] Saarland University, Saarland Informatics Campus
Saarbrücken, Germany
reineke@cs.uni-saarland.de

Abstract. Static cache analysis characterizes a program's cache behavior by determining in a sound but approximate manner which memory accesses result in cache hits and which result in cache misses. Such information is valuable in optimizing compilers, worst-case execution time analysis, and side-channel attack quantification and mitigation.
Cache analysis is usually performed as a combination of "must" and "may" abstract interpretations, classifying instructions as either "always hit", "always miss", or "unknown". Instructions classified as "unknown" might result in a hit or a miss depending on program inputs or the initial cache state. It is equally possible that they do in fact always hit or always miss, but the cache analysis is too coarse to see it.
Our approach to eliminate this uncertainty consists in (i) a novel abstract interpretation able to ascertain that a particular instruction may definitely cause a hit and a miss on different paths, and (ii) an exact analysis, removing all remaining uncertainty, based on model checking, using abstract-interpretation results to prune down the model for scalability.
We evaluated our approach on a variety of examples; it notably improves precision upon classical abstract interpretation at reasonable cost.

1 Introduction

There is a large gap between processor and memory speeds termed the "memory wall" [21]. To bridge this gap, processors are commonly equipped with caches, i.e., small but fast on-chip memories that hold recently-accessed data, in the hope that most memory accesses can be served at a low latency by the cache instead of being served by the slow main memory. Due to temporal and spatial locality in memory access patterns caches are often highly effective.

In hard real-time applications, it is important to bound a program's *worst-case execution time* (WCET). For instance, if a control loop runs at 100 Hz, one must show that its WCET is less than 0.01 s. In some cases, measuring the program's execution time on representative inputs and adding a safety margin may be enough, but in safety-critical systems one may wish for a higher degree

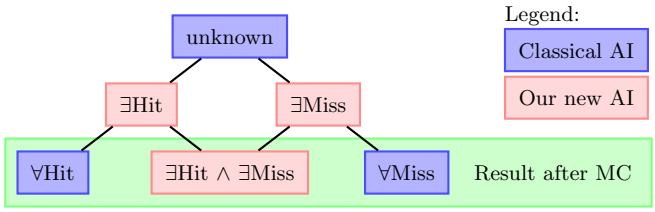


Fig. 1. Possible classifications of classical abstract-interpretation-based cache analysis, our new abstract interpretation, and after refinement by model checking.

of assurance and use static analysis to cover all cases. On processors with caches, such a static analysis involves classifying memory accesses into cache hits, cache misses, and unclassified [20]. Unclassified memory accesses that in reality result in cache hits may lead to gross overestimation of the WCET.

Tools such as OTAWA[3] and AIT[4] compute an upper bound on the WCET of programs after first running a static analysis based on abstract interpretation [11] to classify memory accesses. Our aim, in this article, is to improve upon that approach with a refined abstract interpretation and a novel encoding into finite-state model checking.

Caches may also leak secret information [2] to other programs running on the same machine—through the shared cache state—or even to external devices—due to cache-induced timing variations. For instance, cache timing attacks on software implementations of the Advanced Encryption Standard [1] were one motivation for adding specific hardware support for that cipher to the x86 instruction set [15]. Cache analysis may help identify possibilities for such *side-channel attacks* and quantify the amount of information leakage [7]; improved precision in cache analysis then translates into fewer false alarms and tighter bounds on leakage.

An ideal cache analysis would statically classify every memory access at every machine-code instruction in a program into one of three cases: i) the access is a cache hit in all possible executions of the program ii) the access is a cache miss in all possible executions of the program iii) in some executions the access is a hit and in others it is a miss. However, no cache analysis can perfectly classify all accesses into these three categories.

One first reason is that perfect cache analysis would involve testing the reachability of individual program statements, which is undecidable.[5] A simplifying assumption often used, including in this article, is that all program paths are feasible—this is safe, since it overapproximates possible program behaviors. Even with this assumption, analysis is usually performed using sound but incomplete

[3] http://www.otawa.fr/: an academic tool developed at IRIT, Toulouse.
[4] https://www.absint.com/ait/: a commercial tool developed by Absint GmbH.
[5] One may object that given that we consider machine-level aspects, memory is bounded and thus properties are decidable. The time and space complexity is however prohibitive.

abstractions that can safely determine that some accesses always hit ("∀Hit" in Figure 1) or always miss ("∀Miss" in Fig. 1). The corresponding analyses are called *may* and *must* analysis and referred to as "classical AI" in Fig. 1. Due to incompleteness the status of other accesses however remains "unknown" (Fig. 1).

Contributions In this article, we propose an approach to eliminate this uncertainty, with two main contributions (colored red and green in Figure 1):

1. A novel abstract interpretation that safely concludes that certain accesses are hits in some executions ("∃Hit"), misses in some executions ("∃Miss"), or hits in some and misses in other executions ("∃Hit ∧ ∃Miss" in Fig. 1). Using this analysis and prior must- and may- cache analyses, most accesses are precisely classified.
2. The classification of accesses with remaining uncertainty ("unknown", "∃Hit", and "∃Miss") is refined by model checking using an exact abstraction of the behavior of the cache replacement policy. The results from the abstract interpretation in the first analysis phase are used to dramatically reduce the complexity of the model.

Because the model-checking phase is based on an exact abstraction of the cache replacement policy, our method, overall, is *optimally precise*: it answers precisely whether a given access is always a hit, always a miss, or a hit in some executions and a miss in others (see "Result after MC" in Fig. 1).[6] This precision improvement in access classifications can be beneficial for tools built on top of the cache analysis: in the case of WCET analysis for example, a precise cache analysis not only improves the computed WCET bound; it can also lead to a faster analysis. Indeed, in case of an unclassified access, both possibilities (cache hit and cache miss) have to be considered [10,17].

The model-checking phase would be sufficient to resolve all accesses, but our experiments show this does not scale; it is necessary to combine it with the abstract-interpretation phase for tractability, thereby reducing (a) the number of model-checker calls, and (b) the size of each model-checking problem.

2 Background: Caches and Static Cache Analysis

Caches Caches are fast but small memories that store a subset of the main memory's contents to bridge the latency gap between the CPU and main memory. To profit from spatial locality and to reduce management overhead, main memory is logically partitioned into a set of *memory blocks* M. Each block is cached as a whole in a cache line of the same size.

When accessing a memory block, the cache logic has to determine whether the block is stored in the cache ("cache hit") or not ("cache miss"). For efficient look up, each block can only be stored in a small number of cache lines known as

[6] This completeness is relative to an execution model where all control paths are feasible, disregarding the functional semantics of the edges.

a *cache set*. Which cache set a memory block maps to is determined by a subset of the bits of its address. The cache is partitioned into equally-sized cache sets. The size k of a cache set in blocks is called the *associativity* of the cache.

Since the cache is much smaller than main memory, a *replacement policy* must decide which memory block to replace upon a cache miss. Importantly, replacement policies treat sets independently[7], so that accesses to one set do not influence replacement decisions in other sets. Well-known replacement policies are least-recently-used (LRU), used, e.g., in various Freescale processors such as the MPC603E and the TriCore17xx; pseudo-LRU (PLRU), a cost-efficient variant of LRU; and first-in first-out (FIFO). In this article we focus exclusively on LRU. The application of our ideas to other policies is left as future work.

LRU naturally gives rise to a notion of *ages* for memory blocks: The age of a block b is the number of pairwise different blocks that map to the same cache set as b that have been accessed since the last access to b. If a block has never been accessed, its age is ∞. Then, a block is cached if and only if its age is less than the cache's associativity k.

Given this notion of ages, the state of an LRU cache can be modeled by a mapping that assigns to each memory block its age, where ages are truncated at k, i.e., we do not distinguish ages of uncached blocks. We denote the set of cache states by $C = M \to \{0, \ldots, k\}$. Then, the effect of an access to memory block b under LRU replacement can be formalized as follows[8]:

$$update : C \times M \to C$$

$$(q, b) \mapsto \lambda b'. \begin{cases} 0 & \text{if } b' = b \\ q(b') & \text{if } q(b') \geq q(b) \\ q(b') + 1 & \text{if } q(b') < q(b) \wedge q(b') < k \\ k & \text{if } q(b') < q(b) \wedge q(b') = k \end{cases} \tag{1}$$

Programs as Control-flow Graphs As is common in program analysis and in particular in work on cache analysis, we abstract the program under analysis by its control-flow graph: vertices represent control locations and edges represent the possible flow of control through the program. In order to analyze the cache behavior, edges are adorned with the addresses of the memory blocks that are accessed by the instruction, including the instruction being fetched.

For instruction fetches in a program without function pointers or computed jumps, this just entails knowing the address of every instruction—thus the program must be linked with absolute addresses, as common in embedded code. For data accesses, a pointer analysis is required to compute a set of possible addresses for every access. If several memory blocks may be alternatively accessed by an instruction, multiple edges may be inserted; so there may be multiple edges between two nodes. We therefore represent a control-flow graph by a tuple

[7] To our best knowledge, the only exception to this rule is the *pseudo round-robin* policy, found, e.g., in the ARM Cortex A-9.

[8] Assuming for simplicity that all cache blocks map to the same cache set.

$G = (V, E)$, where V is the set of vertices and $E \subseteq V \times (M \cup \{\bot\}) \times V$ is the set of edges, where $\bot$ is used to label edges that do not cause a memory access.

The resulting control-flow graph G does not include information on the functional semantics of the instructions, e.g. whether they compute an addition. All paths in that graph are considered feasible, even if, taking into account the instruction semantics, they are not—e.g. a path including the tests $x \leq 4$ and $x \geq 5$ in immediate succession is considered feasible even though the two tests are mutually exclusive. All our claims of completeness are relative to this model.

As discussed above, replacement decisions for a given cache set are usually independent of memory accesses to other cache sets. Thus, analyzing the behavior of G on all cache sets is equivalent to separately analyzing its projections onto individual cache sets: a projection of G on a cache set S is G where only blocks mapping to S are kept. Projected control-flow graphs may be simplified, e.g. a self-looping edge labeled with no cache block may be removed. Thus, we assume in the following that the analyzed cache is fully associative, i.e. of a single cache set.

Collecting Semantics In order to classify memory accesses as "always hit" or "always miss", cache analysis needs to characterize for each control location in a program *all* cache states that may reach that location in any execution of the program. This is commonly called the *collecting semantics*.

Given a control-flow graph $G = (V, E)$, the *collecting semantics* is defined as the least solution to the following set of equations, where $R^C : V \to \mathcal{P}(C)$ denotes the set of reachable concrete cache configurations at each program location, and $R_0^C(v)$ denotes the set of possible initial cache configurations:

$$\forall v' \in V : R^C(v') = R_0^C(v') \cup \bigcup_{(v,b,v') \in E} update^C(R^C(v), b), \tag{2}$$

where $update^C$ denotes the cache update function lifted to sets of states, i.e., $update^C(Q, b) = \{update(q, b) \mid q \in Q\}$.

Explicitly computing the collecting semantics is practically infeasible. For a tractable analysis, it is necessary to operate in an abstract domain whose elements compactly represent large sets of concrete cache states.

Classical Abstract Interpretation of LRU Caches To this end, the classical abstract interpretation of LRU caches [9] assigns to every memory block at every program location an interval of ages enclosing the possible ages of the block during any program execution. The analysis for upper bounds, or *must analysis*, can prove that a block must be in the cache; conversely, the one for lower bounds, or *may analysis*, can prove that a block may not be in the cache.

The domains for abstract cache states under may and must analysis are $\mathcal{A}_{May} = \mathcal{A}_{Must} = C = M \to \{0, ..., k\}$, where ages greater than or equal to the cache's associativity k are truncated at k as in the concrete domain. For reasons of brevity, we here limit our exposition to the must analysis. The set of concrete

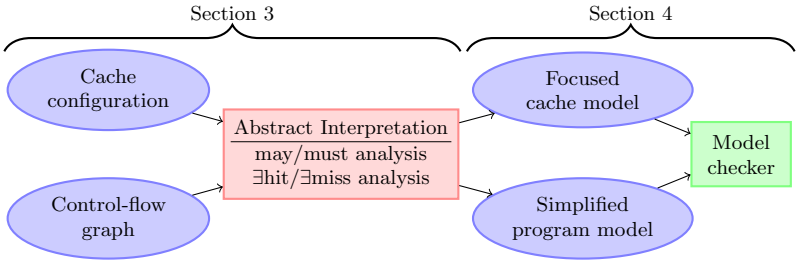


Fig. 2. Overall analysis flow.

cache states represented by abstract cache states is given by the concretization function: $\gamma_{Must}(\hat{q}_{Must}) = \{q \in C \mid \forall m \in M : q(m) \leq \hat{q}_{Must}\}$. Abstract cache states can be joined by taking their pointwise maxima: $\hat{q}_{M1} \sqcup_{Must} \hat{q}_{M2} = \lambda m \in M : \max\{\hat{q}_{M1}(m), \hat{q}_{M2}(m)\}$. For reasons of brevity, we also omit the definition of the abstract transformer $update_{Must}$, which closely resembles its concrete counterpart given in (1), and which can be found e.g. in [16].

Suitably defined abstract semantics R_{Must} and R_{May} can be shown to over-approximate their concrete counterpart:

Theorem 1 (Analysis Soundness [9]). *The may and the must abstract semantics are safe approximations of the collecting semantics:*

$$\forall v \in V : R^C(v) \subseteq \gamma_{Must}(R_{Must}(v)), R^C(v) \subseteq \gamma_{May}(R_{May}(v)). \tag{3}$$

3 Abstract Interpretation for Definitely Unknown

All proofs can be found in Appendix A of the technical report [19]. Together, may and must analysis can classify accesses as "always hit", "always miss" or "unknown". An access classified as "unknown" may still be "always hit" or "always miss" but not detected as such due to the imprecision of the abstract analysis; otherwise it is "definitely unknown". Properly classifying "unknown" blocks into "definitely unknown", "always hit", or "always miss" using a model checker is costly. We thus propose an abstract analysis that safely establishes that some blocks are "definitely unknown" under LRU replacement.

Our analysis steps are summarized in Figure 2. Based on the control-flow graph and on an initial cache configuration, the abstract-interpretation phase classifies some of the accesses as "always hit", "always miss" and "definitely unknown". Those accesses are already precisely classified and thus do not require a model-checking phase. The AI phase thus reduces the number of accesses to be classified by the model checker. In addition, the results of the AI phase are used to simplify the model-checking phase, which will be discussed in detail in Section 4.

An access is "definitely unknown" if there is a concrete execution in which the access misses and another in which it hits. The aim of our analysis is to prove

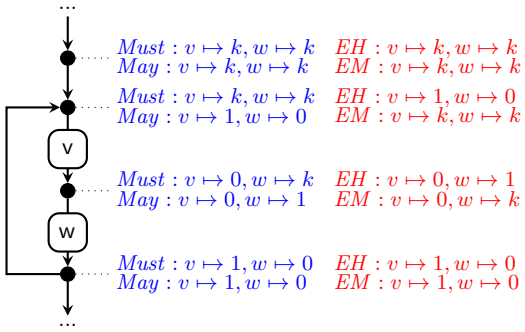


Fig. 3. Example of two accesses in a loop that are definitely unknown. May/Must and EH/EM analysis results are given next to the respective control locations.

the existence of such executions to classify an access as "definitely unknown". Note the difference with classical may/must analysis and most other abstract interpretations, which compute properties that hold *for all executions*, while here we seek to prove that *there exist* two executions with suitable properties.

An access to a block a results in a hit if a has been accessed recently, i.e., a's age is low. Thus we would like to determine the minimal age that a may have in a reachable cache state immediately prior to the access in question. The access can be a hit if and only if this minimal age is lower than the cache's associativity. Because we cannot efficiently compute exact minimal ages, we devise an *Exists Hit* (EH) analysis to compute safe upper bounds on minimal ages. Similarly, to be sure there is an execution in which accessing a results in a miss, we compute a safe lower bound on the maximal age of a using the *Exists Miss* (EM) analysis.

Example. Let us now consider a small example. In Figure 3, we see a small control-flow graph corresponding to a loop that repeatedly accesses memory blocks v and w. Assume the cache is empty before entering the loop. Then, the accesses to v and w are definitely unknown in fully-associative caches of associativity 2 or greater: they both miss in the first loop iteration, while they hit in all subsequent iterations. Applying standard may and must analysis, both accesses are soundly classified as "unknown". On the other hand, applying the EH analysis, we can determine that there are cases where v and w hit. Similarly, the EM analysis derives that there exist executions in which they miss. Combining those two results, the two accesses can safely be classified as definitely unknown.

We will now define these analyses and their underlying domains more formally. The EH analysis maintains upper bounds on the minimal ages of blocks. In addition, it includes a must analysis to obtain upper bounds on all possible ages of blocks, which are required for precise updates. Thus the domain for abstract

cache states under the EH analysis is $\mathcal{A}_{EH} = (M \to \{0, \ldots, k-1, k\}) \times \mathcal{A}_{Must}$. Similarly, the EM analysis maintains lower bounds on the minimal ages of blocks and includes a regular may analysis: $\mathcal{A}_{EM} = (M \to \{0, \ldots, k-1, k\}) \times \mathcal{A}_{May}$. In the following, for reasons of brevity, we limit our exposition to the EH analysis. The EM formalization is analogous and can be found in the technical report [19].

The properties we wish to establish, i.e. bounds on minimal and maximal ages, are actually *hyperproperties* [6]: they are not properties of individual reachable states but rather of the entire *set* of reachable states. Thus, the conventional approach in which abstract states concretize to sets of concrete states that are a superset of the actual set of reachable states is not applicable. Instead, we express the meaning, γ_{EH}, of abstract states by *sets of sets* of concrete states. A set of states Q is represented by an abstract EH state $(\hat{q}, \hat{q}_{Must})$, if for each block b, $\hat{q}(b)$ is an upper bound on b's minimal age in Q, $\min_{q \in Q} q(b)$:

$$\begin{aligned} &\gamma_{EH} : \mathcal{A}_{EH} \to \mathcal{P}(\mathcal{P}(C)) \\ &\quad (\hat{q}, \hat{q}_{Must}) \mapsto \left\{ Q \subseteq \gamma_{Must}(\hat{q}_{Must}) \mid \forall b \in M : \min_{q \in Q} q(b) \leq \hat{q}(b) \right\} \end{aligned} \tag{4}$$

The actual set of reachable states is an element rather than a subset of this concretization. The concretization for the must analysis, γ_{Must}, is simply lifted to this setting. Also note that—possibly contrary to initial intuition—our abstraction cannot be expressed as an underapproximation, as different blocks' minimal ages may be attained in different concrete states.

The abstract transformer $update_{EH}((\hat{q}_{EH}, \hat{q}_{Must}), b)$ corresponding to an access to block b is the pair $(\hat{q}'_{EH}, update_{Must}(\hat{q}_{Must}, b))$, where

$$\hat{q}'_{EH} = \lambda b'. \begin{cases} 0 & \text{if } b' = b \\ \hat{q}(b') & \text{if } \hat{q}_{Must}(b) \leq \hat{q}(b') \\ \hat{q}(b') + 1 & \text{if } \hat{q}_{Must}(b) > \hat{q}(b') \wedge \hat{q}(b') < k \\ k & \text{if } \hat{q}_{Must}(b) > \hat{q}(b') \wedge \hat{q}(b') = k \end{cases} \tag{5}$$

Let us explain the four cases in the transformer above. After an access to b, b's age is 0 in all possible executions. Thus, 0 is also a safe upper bound on its minimal age (case 1). The access to b may only increase the ages of younger blocks (because of the LRU replacement policy). In the cache state in which b' attains its minimal age, it is either younger or older than b. If it is younger, then the access to b may increase b''s actual minimal age, but not beyond $\hat{q}_{Must}(b)$, which is a bound on b's age in every cache state, and in particular in the one where b' attains its minimal age. Otherwise, if b' is older, its minimal age remains the same and so may its bound. This explains why the bound on b''s minimal age does not increase in case 2. Otherwise, for safe upper bounds, in cases 3 and 4, the bound needs to be increased by one, unless it has already reached k.

Lemma 1 (Local Consistency). *The abstract transformer $update_{EH}$ soundly approximates its concrete counterpart $update^C$:*

$$\begin{aligned} &\forall (\hat{q}, \hat{q}_{Must}) \in \mathcal{A}_{EH}, \forall b \in M, \forall Q \in \gamma_{EH}(\hat{q}, \hat{q}_{Must}) : \\ &\qquad update^C(Q, b) \in \gamma_{EH}(update_{EH}((\hat{q}, \hat{q}_{Must}), b)). \end{aligned} \tag{6}$$

How are EH states combined at control-flow joins? The standard must join can be applied for the must analysis component. In the concrete, the union of the states reachable along all incoming control paths is reachable after the join. It is thus safe to take the *minimum* of the upper bounds on minimal ages:

$$(\hat{q}_1, \hat{q}_{Must1}) \sqcup_{EH} (\hat{q}_2, \hat{q}_{Must2}) = (\lambda b. \min(\hat{q}_1(b), \hat{q}_2(b)), \hat{q}_{Must1} \sqcup_{Must} \hat{q}_{Must2}) \quad (7)$$

Lemma 2 (Join Consistency). *The join operator $\sqcup_{EH}$ is correct:*

$$\forall((\hat{q}_1, \hat{q}_{M1}), (\hat{q}_2, \hat{q}_{M2})) \in \mathcal{A}^2_{EH}, Q_1 \in \gamma_{EH}(\hat{q}_1, \hat{q}_{M1}), Q_2 \in \gamma_{EH}(\hat{q}_2, \hat{q}_{M2}) : \\ Q_1 \cup Q_2 \in \gamma_{EH}((\hat{q}_1, \hat{q}_{M1}) \sqcup_{EH} (\hat{q}_2, \hat{q}_{M2})). \quad (8)$$

Given a control-flow graph $G = (V, E)$, the *abstract EH semantics* is defined as the least solution to the following set of equations, where $R_{EH} : V \to \mathcal{A}_{EH}$ denotes the abstract cache configuration associated with each program location, and $R^C_0(v) \in \gamma_{EH}(R_{EH,0}(v))$ denotes the initial abstract cache configuration:

$$\forall v' \in V : R_{EH}(v') = R_{EH,0}(v') \sqcup_{EH} \bigsqcup_{(v,b,v') \in E} update_{EH}(R_{EH}(v), b). \quad (9)$$

It follows from Lemmas 1 and 2 that the abstract EH semantics includes the actual set of reachable concrete states:

Theorem 2 (Analysis Soundness). *The abstract EH semantics includes the collecting semantics: $\forall v \in V : R^C(v) \in \gamma_{EH}(R_{EH}(v))$.*

We can use the results of the EH analysis to determine that an access results in a hit in at least some of all possible executions. This is the case if the minimum age of the block prior to the access is guaranteed to be less than the cache's associativity. Similarly, the EM analysis can be used to determine that an access results in a miss in at least some of the possible executions.

Combining the results of the two analyses, some accesses can be classified as "definitely unknown". Then, further refinement by model checking is provably impossible. Classifications as "exists hit" or "exists miss", which occur if either the EH or the EM analysis is successful but not both, are also useful to reduce further model-checking efforts: e.g. in case of "exists hit" it suffices to determine by model checking whether a miss is possible to fully classify the access.

4 Cache Analysis by Model Checking

All proofs can be found in Appendix B of the technical report [19]. We have seen a new abstract analysis capable of classifying certain cache accesses as "definitely unknown". The classical "may" and "must" analyses and this new analysis classify a (hopefully large) portion of all accesses as "always hit", "always miss", or "definitely unknown". But, due to the incomplete nature of the analysis the exact status of some blocks remains unknown. Our approach is summarized

at a high level in Listing 1.1. Functions `May`, `Must`, `ExistsHit` and `ExistsMiss` return the result of the corresponding analysis, whereas `CheckModel` invokes the model checker (see Listing 1.2). Note that a block that is not fully classified as "definitely unknown" can still benefit from the *Exists Hit* and *Exists Miss* analysis during the model-checking phase. If the AI phase shows that there exists a path on which the block is a hit (respectively a miss), then the model checker does not have to check the "always miss" (respectively "always hit") property.

```
function ClassifyBlock(block) {
  if (Must(block)) //Must analysis classifies the block
    return AlwaysHit;
  else if (!May(block)) //May analysis classifies the block
    return AlwaysMiss;
  else if (ExistHit(block) && ExistMiss(block))
    return DefinitelyUnknown; //DU analysis classifies the block
  else // Otherwise, we call the model checker
    return CheckModel(block, ExistsHit(block), ExistsMiss(block));
}
```

Listing 1.1. Abstract-interpretation phase

```
function CheckModel(block, exist_hit, exist_miss) {
  if (exist_hit) { //block can not always miss
    if (CheckAH(block)) return AlwaysHit;
  }
  else if (exist_miss) { //block can not always hit
    if (CheckAM(block)) return AlwaysMiss;
  } else { //AI phase did not provide any information
    if (CheckAH(block)) return AlwaysHit;
    else if (CheckAM(block)) return AlwaysMiss;
  }
  return DefinitelyUnknown;
}
```

Listing 1.2. Model-checking phase

We shall now see how to classify these remaining blocks using model checking. Not only is the model-checking phase *sound*, i.e. its classifications are correct, it is also *complete* relative to our control-flow-graph model, i.e. there remain no unclassified accesses: each access is classified as "always hit", "always miss" or "definitely unknown". Remember that our analysis is based on the assumption that each path is semantically feasible.

In order to classify the remaining unclassified accesses, we feed the model checker a finite-state machine modeling the cache behavior of the program, composed of i) a model of the program, yielding the possible sequences of memory accesses ii) a model of the cache. In this section, we introduce a new cache model, focusing on the state of a particular memory block to be classified, which we further simplify using the results of abstract interpretation.

As explained in the introduction, it would be possible to directly encode the control-flow graph of the program, adorned with memory accesses, as one big

finite-state system. A first step is obviously to slice that system per cache set to make it smaller. Here we take this approach further by defining a model sound and complete with respect to a given memory block a: parts of the model that have no impact on the caching status of a are discarded, which greatly reduces the model's size. For each unclassified access, the analysis constructs a model focused on the memory block accessed, and queries the model checker. Both the simplified program model and the focused cache model are derived automatically, and do not require any manual interaction.

The *focused cache model* is based on the following simple property of LRU: a memory block is cached if and only if its age is less than the associativity k, or in other words, if there are less than k younger blocks. In the following, w.l.o.g., let $a \in M$ be the memory block we want to focus the cache model on. If we are only interested in whether a is cached or not, it suffices to track the set of blocks younger than a. Without any loss in precision concerning a, we can abstract from the relative ages of the blocks younger than a and of those older than a.

Thus, the domain of the focused cache model is $C_\odot = \mathcal{P}(M) \cup \{\varepsilon\}$. Here, ε is used to represent those cache states in which a is not cached. If a is cached, the analysis tracks the set of blocks younger than a. We can relate the focused cache model to the concrete cache model defined in Section 2 using an abstraction function mapping concrete cache states to focused ones:

$$\begin{aligned} &\alpha_\odot : C \to C_\odot \\ &\qquad q \mapsto \begin{cases} \varepsilon & \text{if } q(a) \geq k \\ \{b \in M \mid q(b) < q(a)\} & \text{if } q(a) < k \end{cases} \end{aligned} \tag{10}$$

The focused cache update $update_\odot$ models a memory access as follows:

$$\begin{aligned} &update_\odot : C_\odot \times M \to C_\odot \\ &\qquad (\widehat{Q}, b) \mapsto \begin{cases} \emptyset & \text{if } b = a \\ \varepsilon & \text{if } b \neq a \wedge \widehat{Q} = \varepsilon \\ \widehat{Q} \cup \{b\} & \text{if } b \neq a \wedge \widehat{Q} \neq \varepsilon \wedge |\widehat{Q} \cup \{b\}| < k \\ \varepsilon & \text{if } b \neq a \wedge \widehat{Q} \neq \varepsilon \wedge |\widehat{Q} \cup \{b\}| = k \end{cases} \end{aligned} \tag{11}$$

Let us briefly explain the four cases above. If $b = a$ (case 1), a becomes the most-recently-used block and thus no other blocks are younger. If a is not in the cache and it is not accessed (case 2), then a remains outside of the cache. If another block is accessed, it is added to a's younger set (case 3) unless the access causes a's eviction, because it is the k^{th} distinct younger block (case 4).

Example. Figure 4 depicts a sequence of memory accesses and the resulting concrete and focused cache states (with a focus on block a) starting from an empty cache of associativity 2. We represent concrete cache states by showing the two blocks of age 0 and 1. The example illustrates that many concrete cache states may collapse to the same focused one. At the same time, the focused cache model does not lose any information about the caching status of the focused block, which is captured by the following lemma and theorem.

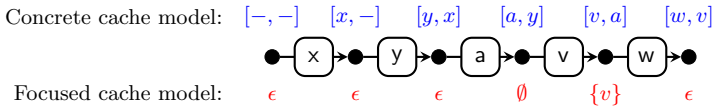


Fig. 4. Example: concrete vs. focused cache model.

Lemma 3 (Local Soundness and Completeness). *The focused cache update abstracts the concrete cache update exactly:*

$$\forall q \in C, \forall b \in M : \alpha_{\odot}(update(q, b)) = update_{\odot}(\alpha_{\odot}(q), b). \tag{12}$$

The *focused collecting semantics* is defined analogously to the *collecting semantics* as the least solution to the following set of equations, where $R^C_{\odot}(v)$ denotes the set of reachable focused cache configurations at each program location, and $R^C_{\odot,0}(v) = \alpha^C_{\odot}(R^C_0(v))$ for all $v \in V$:

$$\forall v' \in V : R^C_{\odot}(v') = R^C_{\odot,0}(v') \cup \bigcup_{(v,b,v') \in E} update^C_{\odot}(R^C_{\odot}(v), b), \tag{13}$$

where $update^C_{\odot}$ denotes the focused cache update function lifted to sets of focused cache states, i.e., $update^C_{\odot}(Q, b) = \{update_{\odot}(q, b) \mid q \in Q\}$, and $\alpha^C_{\odot}$ denotes the abstraction function lifted to sets of states, i.e., $\alpha^C_{\odot}(Q) = \{\alpha_{\odot}(q) \mid q \in Q\}$.

Theorem 3 (Analysis Soundness and Completeness). *The focused collecting semantics is exactly the abstraction of the collecting semantics:*

$$\forall v \in V : \alpha^C_{\odot}(R^C(v)) = R^C_{\odot}(v). \tag{14}$$

Proof. From Lemma 3 it immediately follows that the lifted focused update $update^C_{\odot}$ exactly corresponds to the lifted concrete cache update $update^C$.

Since the concrete domain is finite, the least fixed point of the system of equations of Def. 2 is reached after a bounded number of Kleene iterations. One then just applies the consistency lemmas in an induction proof. □

Thus we can employ the focused cache model in place of the concrete cache model without any loss in precision to classify accesses to the focused block as "always hit", "always miss", or "definitely unknown".

For the program model, we simplify the CFG without affecting the correctness nor the precision of the analysis: i) If we know, from may analysis, that in a given program instruction a is never in the cache, then this instruction cannot affect a's eviction: thus we simplify the program model by not including this instruction. ii) When we encode the set of blocks younger than a as a bit vector, we do not include blocks that the may analysis proved not to be in the cache at that location: these bits would anyway always be 0.

5 Related Work

Earlier work by Chattopadhyay and Roychoudhury [4] 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 (actually, a bounded 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 6: not only do we not keep the program semantics in the problem instance passed to the model checker, which thus has finite state as opposed to being an arbitrarily complex program verification instance, we also strive to minimize that instance by the methods discussed in Section 4.

Chu et al. [5] also refine cache analysis results based on program semantics, but by symbolic execution, where an SMT solver is used to prune infeasible paths. We also compare the scalability of their approach to ours.

Our work complements [12], which uses the classification obtained by classical abstract interpretation of the cache as a basis for WCET analysis on timed automata: our refined classification would increase precision in that analysis. Metta et al. [13] also employ model checking to increase the precision of WCET analysis. However, they do not take into account low-level features such as caches.

6 Experimental Evaluation

In industrial use for worst-case execution time, cache analysis targets a specific processor, specific cache settings, specific binary code loaded at a specific address. The processor may have a hierarchy of caches and other peculiarities. Loading object code and reconstructing a control-flow graph involves dedicated tools. For data caches, a pointer value analysis must be run. Implementing an industrial-strength analyzer including a pointer value analysis, or even interfacing in an existing complex analyzer, would greatly exceed the scope of this article. For these reasons, our analysis applies to a single-level LRU instruction cache, and operates at LLVM bitcode level, each LLVM opcode considered as an elementary instruction. This should be representative of analysis of machine code over LRU caches at a fraction of the engineering cost.

We implemented the classical may and must analyses, as well as our new definitely-unknown analysis and our conversion to model checking. The model-checking problems are produced in the NuSMV format, then fed to nuXmv [3].[9] We used an Intel Core i3-2120 processor (3.30 GHz) with 8 GiB RAM.

[9] https://nuxmv.fbk.eu/: nuXmv checks for reachability using Kleene iterations over sets of states implicitly represented by binary decision diagrams (BDDs). We also tried nuXmv's implementation of the IC3 algorithm with no speed improvement.

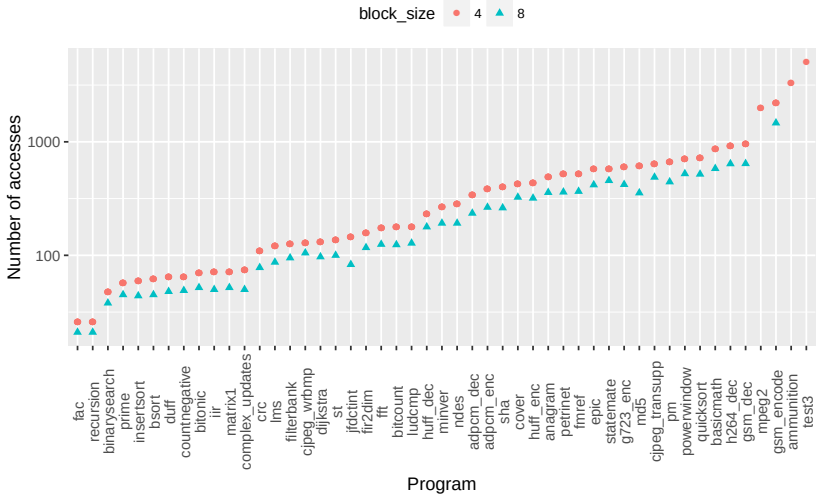


Fig. 5. Size of benchmarks in CFG blocks of 4 and 8 LLVM instructions.

Our experimental evaluation is intended to show i) precision gains by model checking (number of unknowns at the may/must stage vs. after the full analysis) ii) the usefulness of the definitely-unknown analysis (number of definitely-unknown accesses, which corresponds to the reduced number of MC calls, reduced MC cumulative execution time), iii) the global analysis efficiency (impact on analysis execution time, reduced number of MC calls).

As analysis target we use the TACLeBench benchmark suite [8][10], the successor of the Mälardalen benchmark suite, which is commonly used in experimental evaluations of WCET analysis techniques. Figure 5 (log. scale) gives the number of blocks in the control flow graph where a block is a sequence of instructions that are mapped to the same memory block. In all experiments, we assume the cache to be initially empty and we chose the following cache configuration: 8 instructions per block, 4 ways, 8 cache sets. More details on the sizes of the benchmarks and further experimental results (varying cache configuration, detailed numbers for each benchmark,...) may be found in the technical report [19].

6.1 Effect of Model Checking on Cache Analysis Precision

Here we evaluate the improvement in the number of accesses classified as "always hit" or "always miss". In Figure 6 we show by what percentage the number of such classifications increased from the pure AI phase due to model checking.

As can be observed in the figure, more than 60% of the benchmarks show an improvement and this improvement is greater than 5% for 45% of them.

[10] http://www.tacle.eu/index.php/activities/taclebench

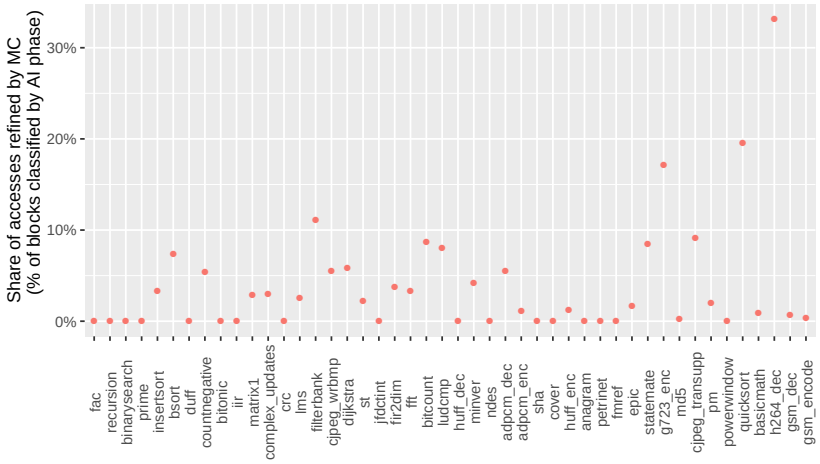


Fig. 6. Increase in hit/miss classifications due to MC relative to pure AI-based analysis.

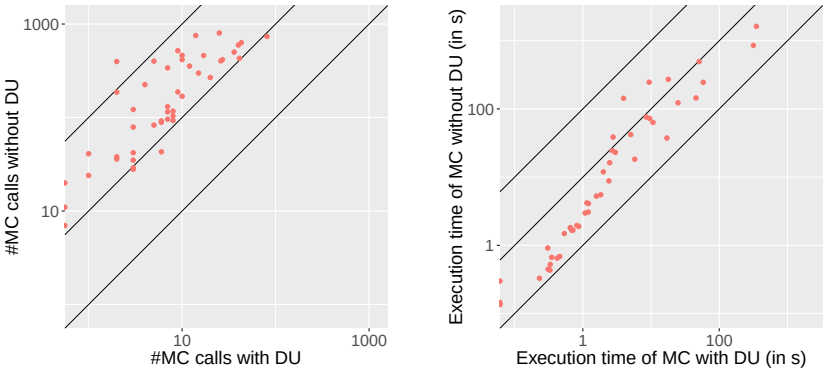
We performed the same experiment under varying cache configurations (number of ways, number of sets, memory-block size) with similar outcomes.

6.2 Effect of the Definitely-Unknown Analysis on Analysis Efficiency

We introduced the definitely-unknown analysis to reduce the number of MC calls: instead of calling the MC for each access not classified as either always hit or always miss by the classical static analysis, we also do not call it on definitely-unknown blocks. Figure 7(a) shows the number of MC calls with and without the definitely-unknown analysis. The two lines parallel to the diagonal correspond to reductions in the number of calls by a factor of 10 and 100. The definitely-unknown analysis significantly reduces the number of MC calls: for some of the larger benchmarks by around a factor of 100. For the three smallest benchmarks, the number of calls is even reduced to zero: the definitely-unknown analysis perfectly completes the may/must analysis and no more blocks need to be classified by model checking. For 28 of the 46 benchmarks, fewer than 10 calls to the model checker are necessary after the definitely-unknown analysis.

This reduction of the number of calls to the model checker also results in significant improvements of the whole execution time of the analysis, which is dominated by the time spent in the model checker: see Figure 7(b). On average (geometric mean) the total MC execution time is reduced by a factor of 3.7 compared with an approach where only the may and must analysis results are used to reduce the number of MC calls.

Note that the definitely-unknown analysis itself is very fast: it takes less than one second on all benchmarks.



(a) Number of calls to the MC. (b) Total MC time.

Fig. 7. Analysis efficiency improvements due to the definitely-unknown analysis.

6.3 Effect of Cache and Program Model Simplifications on Model-Checking Efficiency

In all experiments we used the focused cache model: without this focused model, the model is so large that a timeout of one hour is reached for all but the 6 smallest benchmarks. This shows a huge scalability improvement due to the focused cache model. It also demonstrates that building a single model to classify all the accesses at once is practically infeasible.

Figure 8 shows the execution time of individual MC calls (on a log. scale) with and without program-model simplifications based on abstract-interpretation results. For each benchmark, the figure shows the maximum, minimum, and mean execution time of all MC calls for that benchmark. We observe that the maximum execution time is always smaller with the use of the AI phase due to the simplification of program models. Using AI results, there are fewer MC calls and many of the suppressed MC calls are "cheap" calls: this explains why the average may be larger with AI phase. Some benchmarks are missing the "without AI phase" result: this is the case for benchmarks for which the analysis did not terminate within one hour.

6.4 Efficiency of the Full Analysis

First, we compare our approach to that of the related work [4,5]. Both tools from the related work operate at C level, while our analysis operates at LLVM IR level. Thus it is hard to reasonably compare analysis precision. To compare scalability we focus on total tool execution time, as this is available. In the experimental evaluation of [4] we see that it takes 395 seconds to analyze `statemate` (they stop the analysis at 100 MC calls). With a similar configuration, 64 sets, 4 ways, 4 instructions per block (resp. 8 instructions per blocks) our analysis makes 3 calls (resp. 0) to the model checker (compared with 832 (resp. 259) MC calls without

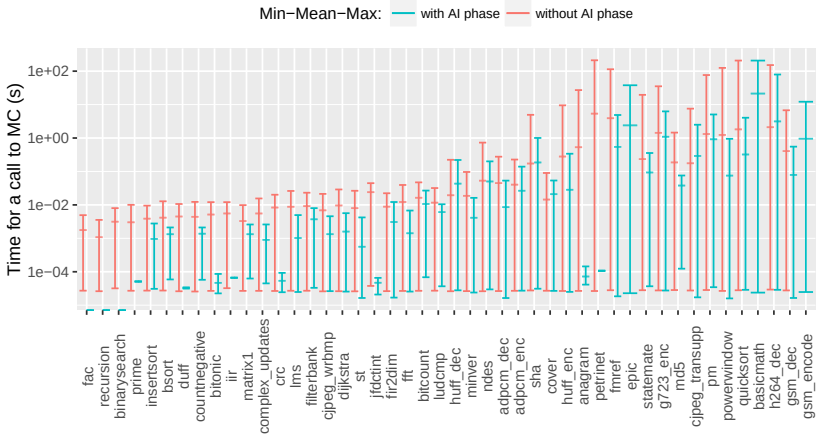
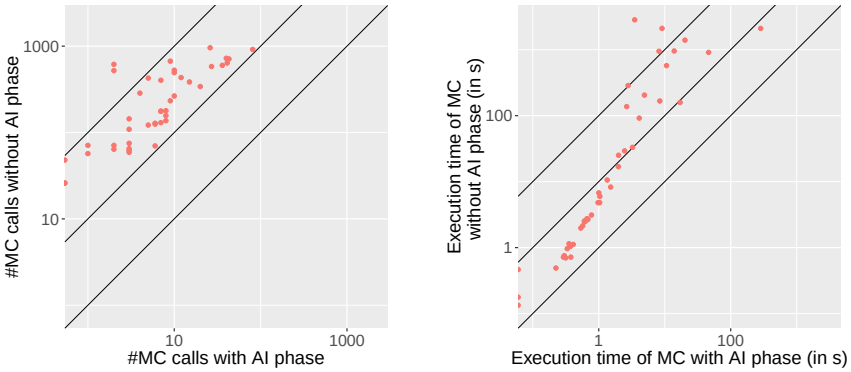


Fig. 8. MC execution time for individual call: min, mean, and max.

the AI phase) and spends less than 3 seconds (resp. 1.5s) on the entire analysis. Unfortunately, among all TACLeBench benchmarks [4] gives scalability results only for statemate, and thus no further comparison is possible. The analysis from [5] also spends more than 350 seconds to analyze statemate; for ndes it takes 38 seconds whereas our approach makes only 3 calls to the model checker and requires less than one second for the entire analysis. This shows that our analysis scales better than the two related approaches. However, a careful comparison of analysis precision remains to be done.

To see more generally how well our approach scales, we compare the total analysis time with and without the AI phase. The AI phase is composed of the may, must and definitely-unknown analyses: without the AI phase, the model checker is called for each memory access and the program model is not simplified. On all benchmarks the number of MC calls is reduced by a factor of at least 10, sometimes exceeding a factor of 100 (see Figure 9(a)). This is unsurprising given the strong effect of the definitely-unknown analysis, which we observed in the previous section. Additional reductions compared with those seen in Figure 7(a) result from the classical may and must analysis. Interestingly, the reduction in total MC time appears to increase with increasing benchmark sizes: see Figure 9(b). While the improvement is moderate for small benchmarks that can be handled in a few seconds with and without the AI phase, it increases to much larger factors for the larger benchmarks.

It is difficult to ascertain the influence our approach would have on a full WCET analysis, with respect to both execution time and precision. In particular, WCET analyses that precisely simulate the microarchitecture need to explore fewer pipeline states if fewer cache accesses are classified as “unknown”. Thus a costlier cache analysis does not necessarily translate into a costlier analysis



(a) Number of calls to the MC. (b) Total MC time.

Fig. 9. Analysis efficiency improvements due to the entire AI phase.

overall. We consider a tight integration with a state-of-the-art WCET analyzer as interesting future work, which is beyond the scope of this paper.

7 Conclusion and Perspectives

We have demonstrated that it is possible to precisely classify all accesses to an LRU cache at reasonable cost by a combination of abstract interpretation, which classifies most accesses, and model checking, which classifies the remaining ones.

Like all other abstraction-interpretation-based cache analyses, at least those known to us, ours considers all paths within a control-flow graph to be feasible regardless of functional semantics. Possible improvements over this include: i) encoding some of the functional semantics of the program into the model-checking problem [13,4] ii) using "trace partitioning" [18] or "path focusing" [14] in the abstract-interpretation phase.

References

1. Bernstein, D.J.: Cache-timing attacks on AES (2005), https://cr.yp.to/antiforgery/cachetiming-20050414.pdf
2. Canteaut, A., Lauradoux, C., Seznec, A.: Understanding cache attacks. Tech. Rep. 5881, INRIA (Apr 2006), https://hal.inria.fr/inria-00071387/en/
3. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: Computer-aided verification (CAV). LNCS, vol. 8559, pp. 334–342. Springer (2014)
4. Chattopadhyay, S., Roychoudhury, A.: Scalable and precise refinement of cache timing analysis via path-sensitive verification. Real-Time Systems 49(4), 517–562 (2013), http://dx.doi.org/10.1007/s11241-013-9178-0
5. Chu, D., Jaffar, J., Maghareh, R.: 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. pp. 293–304. IEEE Computer Society (2016), http://dx.doi.org/10.1109/RTAS.2016.7461358
6. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, 23-25 June 2008. pp. 51–65 (2008), http://dx.doi.org/10.1109/CSF.2008.7
7. Doychev, G., Köpf, B., Mauborgne, L., Reineke, J.: CacheAudit: A tool for the static analysis of cache side channels. ACM Trans. Inf. Syst. Secur. 18(1), 4:1–4:32 (Jun 2015), http://doi.acm.org/10.1145/2756550
8. Falk, H., Altmeyer, S., Hellinckx, P., Lisper, B., Puffitsch, W., Rochange, C., Schoeberl, M., Sorensen, R.B., Wägemann, P., Wegener, S.: 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. pp. 2:1–2:10 (2016), http://dx.doi.org/10.4230/OASIcs.WCET.2016.2
9. Ferdinand, C., Wilhelm, R.: Efficient and precise cache behavior prediction for real-time systems. Real-Time Systems 17(2–3), 131–181 (Dec 1999)
10. Lundqvist, T., Stenström, P.: Timing anomalies in dynamically scheduled microprocessors. In: 20th IEEE Real-Time Systems Symposium (RTSS) (1999)
11. Lv, M., Guan, N., Reineke, J., Wilhelm, R., Yi, W.: A survey on static cache analysis for real-time systems. Leibniz Transactions on Embedded Systems 3(1), 05–1–05:48 (2016), http://ojs.dagstuhl.de/index.php/lites/article/view/LITES-v003-i001-a005
12. Lv, M., Yi, W., Guan, N., Yu, G.: Combining abstract interpretation with model checking for timing analysis of multicore software. In: Proceedings of the 31st IEEE Real-Time Systems Symposium, RTSS 2010, San Diego, California, USA, November 30 - December 3, 2010. pp. 339–349. IEEE Computer Society (2010), http://dx.doi.org/10.1109/RTSS.2010.30
13. Metta, R., Becker, M., Bokil, P., Chakraborty, S., Venkatesh, R.: TIC: a scalable model checking based approach to WCET estimation. In: Kuo, T., Whalley, D.B. (eds.) Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems, LCTES 2016, Santa Barbara, CA, USA, June 13 - 14, 2016. pp. 72–81. ACM (2016), http://doi.acm.org/10.1145/2907950.2907961
14. Monniaux, D., Gonnord, L.: Using bounded model checking to focus fixpoint iterations. In: Yahav, E. (ed.) Static analysis (SAS). LNCS, vol. 6887, pp. 369–385. Springer (2011)

15. Mowery, K., Keelveedhi, S., Shacham, H.: Are AES x86 cache timing attacks still feasible? In: Cloud Computing Security Workshop. pp. 19–24. ACM, New York, NY, USA (2012)
16. Reineke, J.: Caches in WCET analysis: predictability, competitiveness, sensitivity. Ph.D. thesis, Universität des Saarlandes (2008)
17. Reineke, J., et al.: A definition and classification of timing anomalies. In: 6th International Workshop on Worst-Case Execution Time Analysis (WCET) (July 2006)
18. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Transactions on Programming Languages and Systems (TOPLAS) 29(5) (2007)
19. Touzeau, V., Maiza, C., Monniaux, D., Reineke, J.: Ascertaining uncertainty for efficient exact cache analysis. Tech. Rep. TR-2017-2, VERIMAG (2017)
20. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D.B., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P.P., Staschulat, J., Stenström, P.: The worst-case execution-time problem - overview of methods and survey of tools. ACM Trans. Embedded Comput. Syst. 7(3) (2008)
21. Wulf, W.A., McKee, S.A.: Hitting the memory wall: Implications of the obvious. SIGARCH Comput. Archit. News 23(1), 20–24 (Mar 1995), http://doi.acm.org/10.1145/216585.216588