Abstract Interpretation of FIFO Replacement

Daniel Grund and Jan Reineke

Saarland University, Saarbrücken, Germany

Abstract. In hard real-time systems, the execution time of programs must be bounded by static timing analysis. For today's embedded systems featuring caches, static analyses must predict cache hits and misses with high precision to obtain *useful* bounds. For caches with least-recently-used (LRU) replacement policy, efficient and precise cache analyses exist. However, for other widely-used policies like first-in first-out (FIFO), current cache analyses are much less precise.
This paper discusses challenges in FIFO cache analysis and advances the state of the art. We identify a generic framework for cache analysis that couples may- and must-analyses by means of domain cooperation. Our main contribution is a more precise may-analysis for FIFO. It not only increases the number of predicted misses, but also—due to the domain cooperation—the number of predicted hits. We instantiate the framework with a canonical must-analysis and three different may-analyses, including our new one, and compare the resulting three analyses to the collecting semantics. Our evaluation results characterize the progress achieved by our new may-analysis and reveal room for further improvement.

Key words: Cache Analysis, FIFO Replacement, Domain Cooperation, May/Must Reasoning

1 Introduction

In hard real-time systems, one needs to derive off-line guarantees for the timeliness of reactions. Thereby, one fundamental problem is to bound the worst-case execution time (WCET) of programs [1]. To obtain tight and thus useful bounds on the execution times, timing analyses *must* take into account the cache architecture of the employed CPUs. However, developing cache analyses—analyses that statically determine whether a memory access associated with an instruction will always be a hit or a miss—is a challenging problem.

Precise and efficient analyses have been developed for set-associative caches that employ the least-recently-used (LRU) replacement policy [2,3,4,5,6]. Generally, research in the field of embedded real-time systems assumes LRU replacement. In practice however, other policies like first-in first-out (FIFO) or pseudo-LRU (PLRU) are also commonly used, e.g., in the Intel XScale, some ARM9 and ARM11, and the PowerPC 75x series.

As Section 2.3 explains, two kinds of information can be naturally distinguished in cache analysis: must-information that allows for predicting hits, and may-information that allows for predicting misses. Previous work showed that it is inherently more difficult to obtain may-information for FIFO than for LRU; see [7] and Section 3. A first step towards the analysis of those policies was the general concept of relative competitiveness; see [8] and Section 5. Depending on the particular policy, however, a cache analysis based on relative competitiveness may be anything from very precise to ineffective.

In Section 4, we describe a generic policy-independent framework for cache analysis. It allows for cooperation of may- and must-analyses through a minimal interface, which improves their precision.

Then, we present our main contributions: a may- and a must-analysis for FIFO. The must-analysis borrows basic ideas from LRU-analysis [3]. To predict cache hits, it infers upper bounds on cache misses to prove containedness of memory blocks. To predict cache misses, the may-analysis infers lower bounds on cache misses to prove eviction. By taking into account the order in which hits and misses happen, we improve the may-analysis, thereby increasing the number of predicted cache misses. Through the cooperation of the two analyses in the generic framework, this also improves the precision of the must-analysis.

After describing related work in Section 5, we report on our evaluation in Section 6. Using the generic framework, we compare three may-analyses with each other and to the collecting semantics of FIFO. We show that our analysis yields better results than the generic approach using relative competitiveness. Additionally, using the collecting semantics, we illustrate the limits for any static analysis. This supplements analytical bounds derived in [7] and reveals opportunities of how to improve abstract domains for FIFO.

2 Foundations

2.1 Caches

Caches are fast but small memories that store a subset of the main memory's contents to bridge the latency gap between 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* $\mathcal{B}$ of size b bytes. Blocks are cached as a whole in cache lines of equal size. Usually, b is a power of two. This way, the block number is determined by the most significant bits of a memory address.

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"). To enable an efficient look-up, each block can only be stored in a small number of cache lines. For this purpose, caches are partitioned into equally-sized *cache sets* $\mathcal{S}$. The size of a cache set is called the *associativity* k of the cache. A cache with associativity k is often called k-*way* set-associative. It consists of k *ways*, each of which consists of one cache line in each cache set. In the context of a cache set, the term *way* thus refers to a single cache line. The number of such equally-sized cache sets s is usually a power of two such that the set number, also called *index*, is determined by the least significant bits of the block number. The remaining bits of an address are known as the *tag*. To finally decide whether and where a block is cached within a set, tags $t_i \in \mathcal{T}$ are stored along with the data.

Since the number of memory blocks that map to a set is usually far greater than the associativity of the cache, a so-called *replacement policy* must decide which memory block to replace upon a cache miss. Replacement policies try to exploit temporal locality and base their decisions on the history of memory accesses. Usually, cache sets are treated independently of each other such that accesses to one set do not influence replacement decisions in other sets.

Well-known policies for individual cache sets are least-recently used (LRU), a more cost-efficient variant of it (PLRU), and first-in first-out (FIFO). For details on the implementation of caches in hardware refer to Jacob [9].

2.2 Static Analysis

The goal of static analysis is to automatically determine properties of programs without actually executing the programs. Since the properties to determine are commonly incomputable, abstraction has to be employed. In general, there is a trade-off between analysis precision on the one hand and computability and analysis complexity on the other hand.

One formal method in static analysis, which our work is based on, is abstract interpretation. Instead of representing concrete semantic information in a concrete domain D, one represents more abstract information in an abstract domain $\widehat{D}$. The relation between concrete and abstract can be given by an abstraction function $\alpha : \mathcal{P}(D) \rightarrow \widehat{D}$ and a concretization function $\gamma : \widehat{D} \rightarrow \mathcal{P}(D)$.

To determine the properties, a data-flow analysis computes invariants for each program point, which are represented by values of $\widehat{D}$. A *transfer function* $\mathcal{U} : \widehat{D} \times I \rightarrow \widehat{D}$ models the effect of instructions I on abstract values. With the transfer function it is possible to set up a system of data-flow equations that correlates values before and after each instruction. If an instruction has multiple predecessors, a *join function* $\mathcal{J} : \widehat{D} \times \widehat{D} \rightarrow \widehat{D}$ combines all incoming values into a single one. If a data-flow framework meets certain conditions, the induced system of equations for a given program has a least solution, which can be obtained by a fixed-point computation. If the transfer- and the join-function satisfy certain conditions, the analysis is sound with respect to α and γ: True properties in the abstract map to true properties in the concrete. For an overview article with pointers to details on abstract interpretation refer to Cousot and Cousot [10].

2.3 Static Cache Analysis

Static cache analysis by abstract interpretation computes *may*- and *must*-cache information at program points: may- and must-cache information are used to derive upper and lower approximations, respectively, to the *contents* of all concrete cache states that will occur whenever program execution reaches a program point.

Must-cache information is used to derive safe information about cache hits. The more cache hits can be predicted, the better the upper bound on the execution times. May-cache information is used to safely predict cache misses.

Predicting more cache misses will result in a better lower bound on the execution times. Generally, the lower the number of unclassified accesses (neither hit nor miss can be predicted), the lower the runtime of a WCET analysis is because it has to consider fewer cases.

As most cache architectures manage their cache sets independently from each other, cache analyses can analyze them independently as well. Thus, we limit ourselves to the analysis of a single cache set. For details on (LRU-)cache analysis refer to Ferdinand et al. [11,3].

3 The FIFO Policy

The policy. Conceptually, a FIFO cache set maintains a fixed-size queue of tags $\mathcal{T}$. A concrete k-way FIFO cache set s can therefore be modeled as a k-tuple of cache tags, which are ordered from last-in to first-in from left to right:

$$s = [t_0, \dots, t_{k-1}] \in \mathcal{S} := \mathcal{T}^k$$

A cache hit does not change the cache set. A cache miss inserts the new tag at position 0, shifting the others to the right and evicting the one at the rightmost position. The update function $\mathcal{U}_{\mathcal{S}} : \mathcal{S} \times \mathcal{T} \to \mathcal{S}$ models the effect on a cache set when accessing a memory block with tag t:

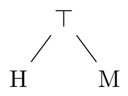
$$\mathcal{U}_{\mathcal{S}}([t_0, \dots, t_{k-1}], t) := \begin{cases} [t_0, \dots, t_{k-1}] & : \exists i : t = t_i \qquad \text{“cache hit”} \\ [t, t_0, \dots, t_{k-2}] & : \text{otherwise} \qquad \text{“cache miss”} \end{cases}$$

In hardware, the FIFO-update can be implemented more efficiently than the LRU-update and the resulting circuit has a lower latency.

Challenges for static analysis. For FIFO, it is difficult to obtain may-information. At the same time, may-information is necessary to obtain precise must-information. Consider a FIFO cache set with unknown contents. After observing a cache access to a block a, one knows that a must be cached—trivial must-information is available. If one cannot classify the access to a as a miss, another access to a different block b may immediately evict a. This is the case if the access to a is a hit on the first-in, i.e., right-most, position and the access to b is a miss. Thus, without (implicitly or explicitly) classifying some accesses as misses, it is not possible to infer that two or more blocks are cached.

Hence, may-information is important to obtain precise must-information, and thus to be able to classify a significant amount of accesses as hits. However, Reineke et al. [7] give the following bound for a k-way FIFO cache set. Assuming accesses to pairwise different blocks, it is impossible to classify an individual access as a miss before $2k - 1$ accesses have been observed.

One way to attenuate the lack of FIFO may-information is to invalidate the cache contents at the start of the program. This way, one can safely assume an empty cache, i.e., at program start one gets complete may- and must-information. However, cache information can be lost during the analysis, e.g., due to control-flow joins. Furthermore, an architecture might not support cache invalidation.



H : cache hit
M : cache miss
⊤ : unclassified

$\sqcup$	H	M	⊤
H	H	⊤	⊤
M	⊤	M	⊤
⊤	⊤	⊤	⊤

$\sqcap$	H	M	⊤
H	H		H
M		M	M
⊤	H	M	⊤

Fig. 1. Classification join semi-lattice *Class* and induced join ($\sqcup$) and meet ($\sqcap$).

4 The FIFO Cache Analysis

Our analysis is an instance of a generic framework for cache analysis that allows *several* cache analyses to cooperate by exchanging classifications of memory accesses. Instead of first describing the framework itself, we immediately describe our instance for FIFO that composes *two* analyses. Then, we present our main contribution: a may- and a must-analysis for FIFO used in this instance of the framework. In our case, the framework is instantiated with one must- and one may-analysis; its abstract domain is:

$$\mathit{Fifo} := \mathit{Must} \times \mathit{May}$$

Cache accesses are classified as hit (H), miss (M), or unclassified (⊤). See Figure 1 for the definition of the classification semi-lattice $\mathit{Class} := \{\mathrm{H}, \mathrm{M}\}^{\top}$. To classify an access to some block with tag $t \in \mathcal{T}$, the classification function of the framework combines the classifications of the may- and the must-analysis. Since these two classifications are sound, they cannot contradict each other. Thus, their meet ($\sqcap$) is always defined.

$$\mathcal{C}_{\mathit{Fifo}} : \mathit{Fifo} \times \mathcal{T} \to \mathit{Class}$$
$$\mathcal{C}_{\mathit{Fifo}}((\mathit{mst}, \mathit{may}), t) := \mathcal{C}_{\mathit{Must}}(\mathit{mst}, t) \sqcap \mathcal{C}_{\mathit{May}}(\mathit{may}, t)$$

The goal of our analysis is to gain better may-information and leverage it more than existing analyses. To enable these synergies, one has to introduce some information flow between the may- and the must-analysis. To this end, the update functions of may- and must-information are refined by an additional parameter that is used to pass the classification of the current access. This classification depends on both analyses, may and must. The main update function hence is defined as:

$$\mathcal{U}_{\mathit{Fifo}} : \mathit{Fifo} \times \mathcal{T} \to \mathit{Fifo}$$
$$\mathcal{U}_{\mathit{Fifo}}((\mathit{mst}, \mathit{may}), t) := (\mathcal{U}_{\mathit{Must}}(\mathit{mst}, t, \mathit{cl}), \mathcal{U}_{\mathit{May}}(\mathit{may}, t, \mathit{cl})),$$

where $\mathit{cl} = \mathcal{C}_{\mathit{Fifo}}((\mathit{mst}, \mathit{may}), t)$, and $\mathcal{U}_{\mathit{Must}}$ and $\mathcal{U}_{\mathit{May}}$ are the update functions of the must- and the may-analysis. This is a form of domain cooperation as described in Cousot et al. [12]. In our case, the additional information allows to define more precise update functions for both analyses.

The main join function is simply defined component-wise:

$$\mathcal{J}_{\mathit{Fifo}} : \mathit{Fifo} \times \mathit{Fifo} \to \mathit{Fifo}$$
$$\mathcal{J}_{\mathit{Fifo}}((\mathit{mst}_1, \mathit{may}_1), (\mathit{mst}_2, \mathit{may}_2)) := (\mathcal{J}_{\mathit{Must}}(\mathit{mst}_1, \mathit{mst}_2), \mathcal{J}_{\mathit{May}}(\mathit{may}_1, \mathit{may}_2))$$

The remainder of this section details the must- and the may-analysis and defines their classification-, update-, and join-functions, which we have used above.

4.1 Must Analysis

In a concrete k-way cache set, k misses must happen to evict a newly inserted memory block. To predict hits, our must-analysis approximates this number from above, it counts potential misses. We define the abstract domain as follows:

$$\mathit{Must} := \mathit{Must}_{\mathit{Fifo}_k} := [T_0, \ldots, T_{k-1}],$$

where $T_i \subseteq \mathcal{T}, \forall i \neq j : T_i \cap T_j = \emptyset$, and $\forall j \leq k : \sum_{i=0}^{j-1} |T_i| \leq j$. The position of a tag in the tuple is an upper bound on the number of misses that happened since the insertion of the block with that tag. If a tag $t \in T_i$, there have been at most i misses since the block with t was inserted into the cache set. It will not be evicted before at least $k - i$ further misses have happened. One must allow for sets of tags because multiple tags may have the same upper bound. However, at most j tags may have an upper bound $\leq j - 1$. Since it is senseless to specify multiple bounds for one tag, all k sets are defined to be disjoint. Otherwise, all but the least bound would be redundant.

The set of concrete cache sets represented by an abstract must cache set is given by the concretization function:

$$\gamma_{\mathit{Must}} : \mathit{Must} \to \mathcal{P}(\mathcal{S})$$
$$\gamma_{\mathit{Must}}([T_0, \ldots, T_{k-1}]) := \{[t_0, \ldots, t_{k-1}] \in \mathcal{S} \mid \forall i \forall t \in T_i \, \exists j \leq i : t_j = t\}$$

In other words: If a tag t is contained in some T_i in an abstract must cache set, the block with tag t must be located at one of the first i positions in the concrete cache set. Consider $\mathit{mst}_1 := [\{f\}, \{\}, \{a, c\}, \{b\}]$ and $\mathit{mst}_2 := [\{\}, \{d\}, \{b, c\}, \{a\}]$ as an example. Their concretizations are $\gamma_{\mathit{Must}}(\mathit{mst}_1) = \{[f, c, a, b], [f, a, c, b]\}$ and $\gamma_{\mathit{Must}}(\mathit{mst}_2) = \{[c, d, b, a], [b, d, c, a], [d, c, b, a], [d, b, c, a]\}$.

The classification function is straightforward. If the accessed tag is contained in any of the T_i-sets, the analysis can predict a hit. If k tags must be cached, no other tag may be cached; in this case, the analysis can predict a miss.

$$\mathcal{C}_{\mathit{Must}} : \mathit{Must} \times \mathcal{T} \to \mathit{Class}$$
$$\mathcal{C}_{\mathit{Must}}([T_0, \ldots, T_{k-1}], t) := \begin{cases} \mathrm{H} : t \in \cup_i T_i \\ \mathrm{M} : t \notin \cup_i T_i =: C, |C| = k \\ \top : \text{otherwise} \end{cases}$$

The update function has three cases. If the analysis can predict a hit, the must-information remains unchanged as FIFO does not change its state upon a hit. If, with the help of may-information, the analysis can predict a miss, one can update the must-information similarly to the concrete semantics. If neither hit nor miss can be predicted, the analysis has to account for both possibilities: Since the access might be a miss, all sets are shifted to the right. Since it might be a hit on the first-in position, the tag can only be added to the rightmost position. This results in:

$$\mathcal{U}_{Must} : Must \times \mathcal{T} \times Class \rightarrow Must$$

$$\mathcal{U}_{Must}([T_0,\ldots,T_{k-1}], t, cl) := \begin{cases} [T_0,\ldots,T_{k-1}] & : cl = \mathrm{H} \\ [\{t\}, T_0,\ldots,T_{k-2}] & : cl = \mathrm{M} \\ [\emptyset, T_0,\ldots,T_{k-2} \cup \{t\}] & : \text{otherwise} \end{cases}$$

If a cache access is not a hit, either the second or the third case of the update applies. They are identical, except for the position where t is inserted. Predicting a miss on the block with tag t allows to predict hits for t until k further misses might have happened. In contrast, the third case only allows to predict hits for t until the next miss might have happened.

The join function has to be sound w.r.t. the concretization function. Therefore, a tag may only be contained in the result if it is present in both operands. The position of such a tag must be the maximum of the two positions in the operands. The best possible join function for our domain is:

$$\mathcal{J}_{Must} : Must \times Must \rightarrow Must$$

$$\mathcal{J}_{Must}([X_0,\ldots,X_{k-1}], [Y_0,\ldots,Y_{k-1}]) := [Z_0,\ldots,Z_{k-1}],$$

with $Z_l := \{t \in \mathcal{T} \mid \exists i,j : t \in X_i \cap Y_j, l = \max\{i,j\}\}$. As an example consider the join of the two must cache sets from above. $mst_3 := \mathcal{J}_{Must}(mst_1, mst_2) = [\{\},\{\},\{c\},\{a,b\}]$. The concretization of mst_3 is "infinite", i.e., $|\gamma_{Must}(mst_3)| = 18 * (|\mathcal{T}| - 3)$: if less than k tags are contained in $\bigcup_i T_i$, any of the other $|\mathcal{T}| - |\bigcup_i T_i|$ tags may also be contained in the cache set.

4.2 May Analysis

The goal of the may-analysis is to infer information that allows for classifying accesses as misses. Our may-analysis associates information with each cache tag. This results in the abstract domain:

$$May := May_{Fifo_k} := \mathcal{T} \rightarrow TInfo_k$$

In the following paragraphs, we will motivate and describe all parts of $TInfo_k$. After describing $TInfo_k$, we define the classification-, update-, and join-functions.

Consider a k-way associative FIFO cache set s and a block with tag t that has just been inserted into s. If k misses happen, the block with t is evicted from s and the next access to that block can be predicted to be a miss. Hence, the may-analysis approximates the number of misses from below.[1] Thus, one constituent of $TInfo_k$ is the number of *definite misses*:

$$DM_k := \{0,\ldots,k-1\}$$

[1] This is analogous to must-information: May-information gives a lower bound on the number of misses (definite misses) while must-information gives an upper bound (potential misses). Must-information can also be represented as a mapping $\mathcal{T} \rightarrow PM$.

Before an analysis can predict a miss for a block, it must predict its eviction, i.e., it must prove that k misses have happened since the insertion of that block. Hence, there is a "bootstrapping problem" if the analysis starts with the worst may-information (i.e., any block could be cached). Similar problems arise if may-information is (partially) lost during the analysis, e.g., due to joins. To solve this problem, a may-analysis *must* infer and maintain additional information.

The only solution to this bootstrapping problem are amortizing observations like "k of $a \geq k$ accesses must have been misses". Consider the following lemma, which holds independently of the replacement policy.

Lemma 1. *Let s be a k-way cache set. Furthermore, let (t_n) be an access sequence (finite series of tags). If (t_n) contains $p \geq k$ pairwise different tags, at least $p-k$ misses must happen if (t_n) is carried out on s.*

Proof. Initially, s can contain at most k pairwise different blocks. Since only accessed blocks are inserted into the set, at most k of the p pairwise different accesses may therefore be hits. $\square$

With FIFO, a block is replaced after k misses.[2] Together with Lemma 1, this means that after at most $2k-1$ accesses to pairwise different blocks, blocks that are not contained in this access sequence cannot be cached. Subsequent accesses to them can be predicted as misses.

To prove that k misses have happened, a FIFO may-analysis *must* be able to distinguish repeating accesses from pairwise different ones. For each tag t, our analysis maintains a set of tags that may have been accessed since the insertion of t. Hence, another constituent of $TInfo_k$ is the set of *possibly accessed tags*:

$$PAT := \mathcal{P}(\mathcal{T})$$

Note that the lower bound on the number of misses provided by Lemma 1 is implicitly based on an upper bound on the number of hits. If one could improve the upper bound on hits, one could predict misses earlier.

Example 1. Consider the FIFO cache set $s = [x, c, b, a]$ and the four access sequences $\langle a, b, c, e, f, g, h \rangle$, $\langle a, e, b, f, c, g, h \rangle$, $\langle e, f, g, h \rangle$, and $\langle a, e, f, c, g, h \rangle$. Although being of different length, carrying out any of the sequences results in the final cache set state $[h, g, f, e]$. In case of the first two sequences, it takes exactly $2k-1 = 7$ accesses to pairwise different blocks to evict all blocks not contained in the sequence. This is because all of the original contents of s, except x, are accessed before their eviction. The third sequence evicts the original contents without accessing them. Sequence four lies in between the two extremes. Note that after accessing a, e, f, it is not possible to access more than three pairwise different blocks without evicting x because a hit on b is impossible.

As Example 1 shows and Figure 2(a) depicts, the order in which hits and misses happen matters. "Early misses", as in $\langle a, e, f, c, g, h \rangle$, preclude hits and

[2] For other replacement policies, this does not necessarily hold, e.g., for PLRU.

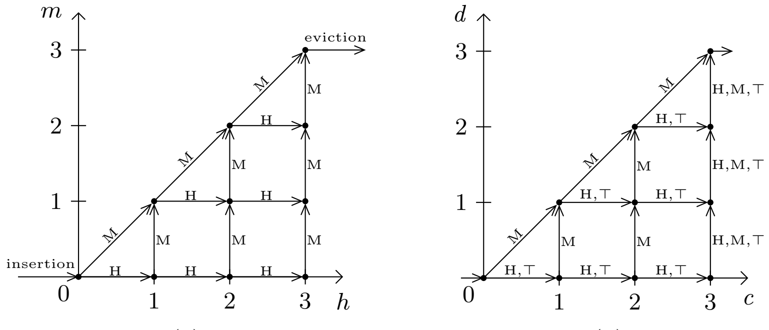


Fig. 2. (a) The paths illustrate all possible sequences of hits (H) and misses (M) between the insertion and eviction of a block in a 4-way associative FIFO cache set. Thereby, only accesses on pairwise different blocks trigger a transition. A block "enters" the cache set at (0, 0). At (h, m), m misses and at most h hits have happened. At (3, 3), the next accesses on a furthermore pairwise different block must be a miss; the block is evicted. (b) Evolution of the number of definite misses d and covered ways c depending on the classifications of accesses. $\top$ denotes the transition upon an unclassified access.

reduce the overall number of accesses to pairwise different blocks until x's eviction. Our analysis exploits that by maintaining a lower bound on the number of *covered ways*, which is the last constituent of TInfo_k:

$$CW_k := \{0, \ldots, k-1\}$$

A way is covered if it is occupied by a block whose tag is also contained in the set of potentially accessed tags $A \in PAT$. For each tag t, covered ways $c \in CW_k$ is a lower bound on the number of covered ways, *assuming* that all unclassified accesses were hits. Eventually, c reaches $k-1$, i.e., the cache set would have to be filled with blocks whose tags are in A. Then, there are two possibilities for further accesses to tags not contained in A. Either such an access is a miss. Or the access is a hit, which indicates that one of the previously unclassified accesses must have been a miss. In either case, the lower bound d on the number of definite misses can be incremented.

Figure 2(b) illustrates the evolution of the lower bounds $d \in DM_k$ and $c \in CW_k$. Accesses that are classified as hit (H) or as unclassified ($\top$) increase c (arrows from left to right). Although misses (M) insert a block into the cache set (cover a way), c is usually not incremented (upwards arrows). This is because the miss might evict a block whose tag is contained in A. Still, the number of definite misses is a lower bound on the number of covered ways. Thus, c can be incremented if $d = c$ (diagonal arrows). As explained above, the analysis can increase the number of definite misses if $c = k-1$, even if the current access cannot be classified as a miss (the upwards arrows at $c = 3 = 4-1$).

In summary, the domain for the information maintained per cache tag $t \in \mathcal{T}$ is $CW_k \times DM_k \times PAT$. Adding $\bot$ to indicate that a tag is definitely not cached yields $\mathit{TInfo}_k := (CW_k \times DM_k \times PAT)_\perp$. Substituting this in the definition of our abstract may cache sets results in:

$$\mathit{May} = \mathcal{T} \rightarrow (CW_k \times DM_k \times PAT)_\perp$$

$t \mapsto \perp$ indicates that t cannot be cached. Otherwise, $t \mapsto (c, d, A)$, where:

- A, potentially accessed tags, is an upper approximation of the set of tags that have been accessed since the last insertion of t.
- d, definite misses, is a lower bound on the number of misses that happened since the insertion of t into s.
- c, covered ways, is a lower bound on the number of ways that are occupied by tags in A, assuming that all unclassified accesses were hits.

The concretization function for *May* is:

$$\gamma_{\mathit{May}} : \mathit{May} \rightarrow \mathcal{P}(\mathcal{S})$$
$$\gamma_{\mathit{May}}(\mathit{may}) := \bigcap_{t \in \mathcal{T}} \gamma_{\mathit{TInfo}_k}(t, \mathit{may}(t))$$

The TInfo_k of each tag constrains the set of possible cache sets. $\gamma_{\mathit{TInfo}_k}$ defines such a constraint for one tag. Hence, the concretization function of *May* is the conjunction (intersection) of all these constraints (sets). Let $C_n([t_0, \ldots, t_{k-1}]) := \bigcup_{0 \leq i < n} \{t_i\}$, i.e., the cache contents of the n leftmost positions in a cache set. If $\mathit{may}(t) = \perp$, tag t is evicted and cannot be contained in a cache set represented by *may*. Hence,

$$\gamma^{\perp}_{\mathit{TInfo}_k}(t) := \{s \in \mathcal{S} \mid t \notin C_k(s)\}$$

Otherwise,

$$\gamma^{\not\perp}_{\mathit{TInfo}_k}(t, (c, d, A)) := \left\{s \in \mathcal{S} \mid o := |C_k(s) \cap A|, C_{\max\{d, d+c-o\}}(s) \subseteq A\right\}$$

Here, o is the number of ways *actually* covered by A in a particular concrete cache set state s. c was defined to be the number of ways covered by A, given that all unclassified accesses were hits. Hence, if $c > o$, $c - o$ unclassified accesses must have been misses. Together with the d definite misses, at least $\max\{d, d+c-o\}$ of the tags in A must occupy the leftmost positions of s ($C_{\max\{\ldots\}}(s) \subseteq A$).[3] As $t \notin A$ by construction, this also constrains t's position in s. The concretization function for TInfo_k is:

$$\begin{aligned} \gamma_{\mathit{TInfo}_k}(t, \perp) &:= \gamma^{\perp}_{\mathit{TInfo}_k}(t) \\ \gamma_{\mathit{TInfo}_k}(t, (c, d, A)) &:= \gamma^{\perp}_{\mathit{TInfo}_k}(t) \cup \gamma^{\not\perp}_{\mathit{TInfo}_k}(t, (c, d, A)) \end{aligned}$$

In the latter case, $\gamma^{\perp}_{\mathit{TInfo}_k}(t)$ reflects that t might not be cached, and the second part defines constraints for the case that t is cached.

[3] Recall that tags are inserted in the leftmost position upon a miss.

In the remaining part of this section, we will describe the classification-, update-, and join-functions for *May*. The classification function is straightforward:

$$\mathcal{C}_{May} : May \times \mathcal{T} \rightarrow Class$$
$$\mathcal{C}_{May}(may, t) := \begin{cases} \mathrm{M} : may(t) = \bot \\ \top \ : \text{otherwise} \end{cases}$$

The update function is defined separately for each of the three possible cases (H, M, $\top$) of the classification parameter.

$$\mathcal{U}_{May} : May \times \mathcal{T} \times Class \rightarrow May$$

$$\mathcal{U}_{May}(may, t, \mathrm{H}) := \lambda x.$$
$$\begin{cases} may(x) & : \begin{cases} x = t \\ x \neq t, may(x) = \bot \\ x \neq t, may(x) = (c, d, A), t \in A \end{cases} \\ (c+1, d, A \cup \{t\}) & : x \neq t, may(x) = (c, d, A), t \notin A, c < k-1 \\ (c, d+1, A \cup \{t\}) & : x \neq t, may(x) = (c, d, A), t \notin A, c = k-1, d < k-1 \\ \bot & : x \neq t, may(x) = (c, d, A), t \notin A, c = k-1, d = k-1 \end{cases}$$

FIFO does not change its state upon a hit. Furthermore, the $TInfo_k$ of a tag is only updated if the accessed tag is not contained in A. This explains the first case where nothing is changed. The remaining three cases update the $TInfo_k$ of tags different from the accessed one ($x \neq t$), that may be cached ($may(x) = (c, d, A)$), and t has definitely not been accessed since the insertion of x ($t \notin A$). If $c < k-1$, the number of covered ways is incremented. If $c = k-1$ and a hit happens, the number of definite misses is soundly incremented: Since no more than $k-1$ hits on pairwise different elements can happen, a previous access, which also incremented c, must have been a miss though the analysis could not classify it as a miss. In the last case, the number of definite misses reaches k; the block with tag x is definitely evicted.

$$\mathcal{U}_{May}(may, t, \mathrm{M}) := \lambda x.$$
$$\begin{cases} (0, 0, \emptyset) & : x = t \\ may(x) & : \begin{cases} x \neq t, may(x) = \bot \\ x \neq t, may(x) = (c, d, A), t \in A \end{cases} \\ (\max\{c, d+1\}, d+1, A \cup \{t\}) & : x \neq t, may(x) = (c, d, A), t \notin A, d < k-1 \\ \bot & : x \neq t, may(x) = (c, d, A), t \notin A, d = k-1 \end{cases}$$

The first case resets the information associated with a tag to $(0, 0, \emptyset)$ if a miss on this tag happens. The $TInfo_k$ of a tag is not updated if it is already evicted or if the accessed tag is contained in A (second case). In the third case, the number of definite misses is incremented. Furthermore, as explained above ("the diagonal"), c can be incremented if $d = c$. This can be abbreviated by the max expression. In the last case, the analysis can prove eviction of the tag.

The update for an unclassified access is defined as the join of the hit- and miss-update.

$$\mathcal{U}_{May}(may, t, \top) := \mathcal{J}_{May}(\mathcal{U}_{May}(may, t, \mathrm{H}), \mathcal{U}_{May}(may, t, \mathrm{M}))$$

The most interesting cases when spelling this out are

$$\begin{array}{ll} (c+1, d, A \cup \{t\}) & : x \neq t, may(x) = (c, d, A), t \notin A, c < k-1 \\ (c, d+1, A \cup \{t\}) & : x \neq t, may(x) = (c, d, A), t \notin A, c = k-1, d < k-1 \\ \perp & : x \neq t, may(x) = (c, d, A), t \notin A, c = k-1, d = k-1 \end{array}$$

This shows that the analysis can "bootstrap"; i.e., it can prove eviction of memory blocks without relying on explicit miss-classifications.

Finally, the join of may-information is defined as the component-wise join of the $TInfo_k$ for each tag:

$$\mathcal{J}_{May} : May \times May \rightarrow May$$

$$\mathcal{J}_{May}(may_1, may_2) := \lambda x. \begin{cases} may_1(x) & : may_2(x) = \perp \\ may_2(x) & : may_1(x) = \perp \\ (c', d', A_1 \cup A_2) & : may_i(x) = (c_i, d_i, A_i) \end{cases}$$

where $d' := \min\{d_1, d_2\}$ and $c' := \min\{c_1 + d_1, c_2 + d_2\} - d'$. In the first two cases, one of the operands maps $x \mapsto \perp$. Since $\forall I \in TInfo_k : \gamma_{TInfo_k}(t, \perp) \subseteq \gamma_{TInfo_k}(t, I)$, the other operand must be the least upper bound of the two. In the last case, the $TInfo_k$ of x of both operands is $\neq \perp$. As the set of potentially accessed tags must be an overapproximation, the join of A_1 and A_2 is the set union $A_1 \cup A_2$. The number of definite misses is an underapproximation. Hence, the join is the minimum $\min\{d_1, d_2\}$. For c one would also expect $\min\{c_1, c_2\}$. However, note that $\gamma_{TInfo_k}(t, (c, d, A)) \subseteq \gamma_{TInfo_k}(t, (c+\delta, d-\delta, A))$, i.e., one can "trade misses for hits". The join may result in a loss of precision for the definite misses, i.e., the difference $d_i - d'$. Due to the relation above, one can add $d_i - d'$ to c_i before taking the minimum, i.e., $c' = \min\{c_1 + (d_1 - d'), c_2 + (d_2 - d')\}$.

Theorem 1 (Soundness). *The abstract interpretation* $(Fifo, \mathcal{J}_{Fifo}, \mathcal{U}_{Fifo}, \gamma_{Fifo})$ *is a sound abstraction of the concrete* FIFO *semantics.*

Theorem 2 (Termination). *The update function* $\mathcal{U}_{Fifo}$ *is monotone and the abstract domain Fifo satisfies the ascending-chain condition.*

5 Related Work

There are different types of static cache analysis. Cache analyses directed at compiler optimizations compute bounds on the number of misses for larger program fragments, e.g., loop nests, whereas analyses directed at WCET analyses classify individual cache accesses as hits or misses.

Representatives of the first class are Ghosh et al. [13] and Chatterjee et al. [6]. Ghosh et al. [13] introduce *Cache Miss Equations* that characterize the cache behavior of loop-nests in direct-mapped caches by Diophantine equations. In subsequent work [5,14], they generalize their approach to set-associative caches with LRU replacement. Chatterjee et al. [6] propose an exact model of cache behavior of loop nests. It can handle imperfect loop nests and modest levels of associativity with LRU replacement.

Representatives of the second class include Mueller et al. [15], White et al. [4], Li et al. [16], and Ferdinand et al. [11,3]. Mueller et al. [15] present a *static cache simulation* for direct-mapped instruction caches. It classifies instructions as *always-miss*, *always-hit*, *first-miss*, or *conflict*. White et al. [4] extend this work to data caches, where the main challenges lie in the analysis of accessed addresses. Furthermore, an instruction cache analysis for set-associative LRU caches is sketched. Li et al. [16] present a timing analysis based on integer linear programming (ILP) formulations. It can handle set-associative caches by encoding their concrete semantics using linear constraints. However, since this approach integrates pipeline, cache, and path analysis into one ILP, it suffers from complexity problems. In practice it is limited to direct-mapped caches and simple pipelines. Ferdinand et al. [11,3] introduce the concepts of may- and must-caches and present an LRU analysis that is based on abstract interpretation.

Almost all cache analyses assume LRU replacement. As explained in Section 3 statically analyzing FIFO is inherently more difficult than LRU. In contrast to FIFO, it is possible to obtain precise must-information for LRU replacement without any may-analysis.

The concept of relative competitiveness [8] bounds the performance of one replacement policy relative to the performance of another one. This allows to use cache analyses for one policy as cache analyses for other policies. This implies that all of the existing analyses for LRU can be used as either may- or must-analyses for FIFO and PLRU. For instance, an LRU may-analysis for a $2k-1$-way associative cache can be reused as a may-analysis for a k-way FIFO. Due to the generic nature of this approach, however, the resulting analyses may be rather imprecise. In the case of FIFO, one would expect that a $2k-1$-way LRU performs much better than a k-way FIFO, i.e., the number of misses is much lower. Hence, the gap between actual and predicted number of misses might be large. The analysis presented in this work is tailored precisely to FIFO behavior and can therefore deliver more precise results.

Finally, our work is different from the analysis of so-called FIFO channels [17,18,19]. Such channels mostly model communication and have different characteristics than caches with FIFO replacement.

6 Evaluation

In the following, we compare three FIFO analyses with each other and to the collecting semantics. The collecting semantics is computed using a powerset domain of concrete cache set states and is denoted by CollSem. The three analyses are different instantiations of our framework. All analyses use the canonical must-analysis CM presented in this paper and only differ in their may-analysis.

– No+CM: An analysis that uses no may-analysis at all, i.e., the classification returned by the may-analysis is always $\top$.
– RC+CM: An analysis that uses a may-analysis based on relative competitiveness, as explained in Section 5, and the canonical must-analysis given in Section 4.1.
– EMX+CM: "Early Miss eXploitation"; the analysis proposed in this paper.

The analyses can be partially ordered according to their precision. Let $A \preccurlyeq B$ denote that analysis A is at least as precise as B for all programs. More precisely, A $\preccurlyeq$ B if for each access to be classified, the classification by A is equal or better than the respective classification of B. $A \prec B$ if $A \preccurlyeq B$ and $B \not\preccurlyeq A$.

Theorem 3 (Relative Precision).

$$\begin{tikzcd}[column sep=small, row sep=tiny]
& \text{EMX+CM} \arrow[dd, phantom, "\not\prec\ \not\succ"] \arrow[dr, phantom, "\prec"] & \\
\textsc{CollSem} \arrow[ur, phantom, "\prec"] \arrow[dr, phantom, "\prec"] & & \textsc{No+CM} \\
& \text{RC+CM} \arrow[ur, phantom, "\prec"] &
\end{tikzcd}$$

To quantify the precision of the analyses, we analyzed random access sequences and program fragments. In a first experiment, for each $1 \leq n \leq 31$, we generated 100 random access sequences that contain 500 accesses to n pairwise different tags. The parameter n controls the locality in the sequences: the greater the n, the lower is the locality.

Figure 3 shows the results; i.e., hit- and miss-rates guaranteed by the four analyses. The shape of the plot marks identify the analysis, e.g., circles for No+CM. The number of different tags (n) in the generated access sequences is plotted against the x-axis. The percentage of classifications (H, $\top$, M) is plotted against the y-axis. For each analysis there are two curves, which partition the 100%. The lower curve, with filled plot marks, shows the guaranteed hit-rate. The upper one, with empty plot marks, is plotted bottom-up (from 100% downwards) and shows the guaranteed miss-rate. The difference between the upper and the lower curve gives the percentage of unclassified accesses.

For example consider the squares at $n = 24$. For 100 access sequences, each 500 accesses long and containing $n = 24$ different tags, the average guaranteed hit-rate obtained by RC+CM was 20%. The average guaranteed miss-rate was 42%, and on average 38% could not be classified.

Discussion: No+CM cannot predict any misses, hence the upper curve is constantly at 100%. As explained in Section 3, without may-information, it is impossible to infer that more than one memory block must be cached. Thus, with increasing n, the lower curve decreases as it gets more unlikely to access the same tag twice in a row.

Both, RC+CM and EMX+CM cannot predict any misses until 15 pairwise different tags have been accessed. This is in line with the "evict" bound $2k-1 = 15$ determined in [7]. Hence, the curves of RC+CM and EMX+CM coincide with the one of No+CM up to $n = 15$. For larger n, both analyses predict misses, which in turn allows to predict more hits.

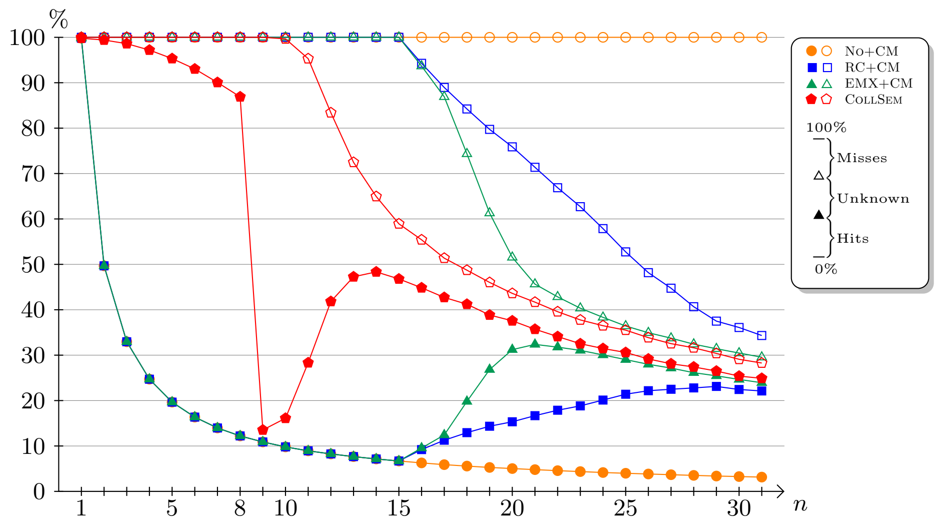


Fig. 3. Average guaranteed hit- and miss-rates for an 8-way cache set.

EMX+CM predicts more misses than RC+CM. For $n = 16, 17$ the difference is relatively small. This is because the benefit of predicting "early misses" is self-energizing. The more misses are predicted, the more does the "diagonal" in Figure 2(b) help, the more misses are predicted.... Due to the domain cooperation, the must-analysis also profits from the prediction of misses, i.e., more hits can be predicted. Put simply, RC+CM "takes the long way" and always takes the $\top$-transitions in Figure 2(b).

Interestingly, CollSem shows that one could statically predict misses with accesses to less than $2k - 1$ pairwise different blocks. EMX+CM and RC+CM require at least $2k - 1$ pairwise different blocks because they do not gain information from repeating accesses. For $n \leq k$, CollSem shows that one could predict a large fraction of the accesses as hits. This is due to the fact that in any concrete cache set at most n misses might occur if $n \leq k$. However, one cannot predict all but those n misses as hits since one has to account for all initial cache set states. Depending on the initial position of a tag within a cache set, what is a hit in one concrete cache set is a miss in another. Hence, the lower curve decreases super-linearly. At $n = 9 = k + 1$, the percentage of predictable hits drops extremely. Since $n > k$, the cache cannot hold all accessed blocks. At the same time, almost no may-information is available in the collecting semantics. Similarly to static analyses, which cannot gain precise must-information without may-information, there are not many guaranteed hits in the collecting semantics. For larger n, the number of predictable (and actually occurring) misses increases, causing initially different cache set states to converge more quickly. This allows to predict more accesses as hits.

In a second experiment, we generated program fragments with a large number of control-flow join points to evaluate the join functions of the analyses.

We generated recursively nested if-then-else patterns, i.e., $\circ \longrightarrow$ ◇. All nodes of the flow-graph contain 10 random memory accesses. This way, we tested the ability of the analyses to recover from (partially) lost information before the next join point was reached. Figure 4 shows the results. Generally, the relative evolution of the curves are the same as in Figure 3. The main difference is that all guarantees are worse since even in the collecting semantics the uncertainty is large.

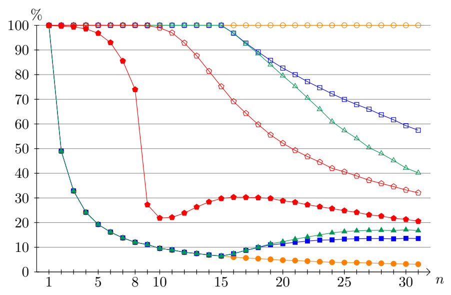


Fig. 4. Results of the join experiment.

7 Conclusions and Future Work

We presented the first abstract domain specifically tailored to the analysis of caches with FIFO replacement. With information about the order in which hits and misses have happened, our analysis can predict more misses than previous approaches. Due to an effective cooperation between our may- and must-analysis, this also improves the number of predicted hits.

Our evaluation clearly showed the characteristics of three different analyses, i.e., when and why an analysis is better than another one. Additionally, the illustration of the collecting semantics revealed characteristics of FIFO itself: While EMX+CM and RC+CM need to observe accesses to at least $2k-1$ pairwise different blocks to obtain may-information, may-information may be available after accessing fewer pairwise different blocks. How to exploit this in an abstract domain? Furthermore, there is room for a better must-analysis for $n \leq k$, which, however, would have to rely on implicit miss-classifications.

Acknowledgements We want to express our gratitude to Sebastian Hack for his helpful remarks on drafts of this paper. We also thank the anonymous reviewers for their fair and thorough reviews. The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n° 216008 (Predator). This work was supported by the DFG as part of the Transregional Collaborative Research Center SFB/TR 14 (AVACS).

References

1. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem—overview of methods and survey of tools. Transactions on Embedded Computing Systems **7**(3) (2008) 1–53
2. Alt, M., Ferdinand, C., Martin, F., Wilhelm, R.: Cache behavior prediction by abstract interpretation. In: SAS '96: Proceedings of the Third International Symposium on Static Analysis, London, UK, Springer-Verlag (1996) 52–66

3. Ferdinand, C., Wilhelm, R.: Efficient and precise cache behavior prediction for real-time systems. Real-Time Systems **17**(2-3) (1999) 131–181
4. White, R.T., Healy, C.A., Whalley, D.B., Mueller, F., Harmon, M.G.: Timing analysis for data caches and set-associative caches. In: RTAS '97: Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium, Washington, DC, USA, IEEE Computer Society (1997) 192
5. Ghosh, S., Martonosi, M., Malik, S.: Precise miss analysis for program transformations with caches of arbitrary associativity. In: ASPLOS-VIII: Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA, ACM Press (1998) 228–239
6. Chatterjee, S., Parker, E., Hanlon, P.J., Lebeck, A.R.: Exact analysis of the cache behavior of nested loops. In: PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, New York, NY, USA, ACM Press (2001) 286–297
7. Reineke, J., Grund, D., Berg, C., Wilhelm, R.: Timing predictability of cache replacement policies. Real-Time Systems **37**(2) (2007) 99–122
8. Reineke, J., Grund, D.: Relative competitive analysis of cache replacement policies. In: LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, New York, NY, USA, ACM Press (2008) 51–60
9. Jacob, B., Ng, S.W., Wang, D.T.: Memory Systems: Cache, DRAM, Disk. Morgan Kaufmann Publishers (2008)
10. Cousot, P., Cousot, R.: Basic Concepts of Abstract Interpretation. In: Building the Information Society. Kluwer Academic Publishers (2004) 359–366
11. Ferdinand, C.: Cache Behaviour Prediction for Real-Time Systems. PhD thesis, Saarland University (1997)
12. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Combination of abstractions in the ASTRÉE static analyzer. In: ASIAN'06: 11th Annual Asian Computing Science Conference. (2006) 272–300
13. Ghosh, S., Martonosi, M., Malik, S.: Cache miss equations: An analytical representation of cache misses. In: ICS '97: Proceedings of the 11th International Conference on Supercomputing, New York, NY, USA, ACM Press (1997) 317–324
14. Ghosh, S., Martonosi, M., Malik, S.: Cache miss equations: A compiler framework for analyzing and tuning memory behavior. ACM Transactions on Programming Languages and Systems **21**(4) (1999) 703–746
15. Mueller, F., Whalley, D.B., Harmon, M.: Predicting instruction cache behavior. In: LCTRTS '94: Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems. (1994)
16. Li, Y.T.S., Malik, S., Wolfe, A.: Cache modeling for real-time software: Beyond direct mapped instruction caches. In: RTSS '96: Proceedings of the 17th IEEE Real-Time Systems Symposium, Washington, DC, USA, IEEE Computer Society (1996) 254
17. Brand, D., Zafiropulo, P.: On communicating finite-state machines. Journal of the ACM **30**(2) (1983) 323–342
18. Peng, W., Iyer, S.P.: Data flow analysis of communicating finite-state machines. ACM Transactions on Programming Languages and Systems **13**(3) (1991) 399–442
19. Bouajjani, A., Habermehl, P.: Symbolic reachability analysis of FIFO-channel systems with nonregular sets of configurations. Theoretical Computer Science **221**(1-2) (1999) 211–250