Precise and Efficient FIFO-Replacement Analysis Based on Static Phase Detection

Daniel Grund
Saarland University, Saarbrücken, Germany

Jan Reineke
Saarland University, Saarbrücken, Germany
University of California, Berkeley, USA

***Abstract*—Schedulability analysis for hard real-time systems requires bounds on the execution times of its tasks. To obtain *useful* bounds in the presence of caches, static timing analyses must predict cache hits and misses with high precision. For caches with least-recently-used (LRU) replacement policy, precise and efficient cache analyses exist. However, other widely used policies like first-in first-out (FIFO) are inherently harder to analyze.**

The main contributions of this paper are precise and efficient must- and may-analyses of FIFO based on the novel concept of *static phase detection*. The analyses statically partition sequences of memory accesses as they will occur during program execution into phases. If subsequent phases contain accesses to the same (similar) set of memory blocks, each phase contributes a bit to the overall goal of predicting hits (misses). The new must-analysis is significantly more precise than prior analyses. Both analyses can be implemented space-efficiently by sharing information using abstract LRU-stacks.

I. Introduction

In hard real-time systems, timeliness of reactions must be guaranteed off-line. 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 processors. However, developing cache analyses—analyses that compute sound approximations to cache contents at program points—is a challenging problem.

At its heart, cache analysis is concerned with the analysis of the employed replacement policy. For LRU replacement, precise and efficient analyses have been developed [2], [3], [4], [5]. In practice however, other policies like FIFO or pseudo-LRU (PLRU) are more widely used, e.g. in the INTEL XSCALE, some ARM, and several POWERPC series. In addition, FIFO is the predominant replacement policy in other transparent buffers, like branch target buffers [6], where the requirement of low latency precludes expensive update computations as required by LRU.

Analyzing FIFO is harder than analyzing LRU since its behavior is more sensitive to its state [7]. As opposed to LRU, accessing a set of memory blocks that would entirely fit into the cache does not imply that all of those blocks are cached afterwards. Analogously, accessing a set of memory blocks that is larger than the cache does not necessarily evict all other previous cache contents.

However, if the same "fitting" set of blocks is accessed repeatedly, then eventually those blocks must be cached. Analogously, repeatedly accessing a "non-fitting" set will eventually evict all non-accessed blocks. As the basis of this paper, we introduce two theorems that show exactly how much information about the cache contents is available after how many of such *phases*. The two theorems respectively are concerned with the two kinds of information that can be naturally distinguished: Must-information [2] that allows for predicting hits, and may-information [2] that allows for predicting misses.

Our main contributions are a must- and a may-analysis of FIFO based on the two theorems. In contrast to the theorems, the two analyses operate on arbitrary control flow by joining analysis information. By *statically detecting phases*, the analyses are able to gradually build up precise must- and may-information. We also show how the analysis information can be encoded space-efficiently by employing the abstract LRU stacks of [2].

Section II introduces the notation used in this paper and presents the theoretical foundations we will use.

In Section III, we give motivational examples, introduce the theorems, and present formalizations of the analyses.

In Section IV, we cover related work and discuss qualitative differences and draw connections between the novel phase-detecting analyses and state-of-the-art analyses: All existing FIFO analyses [8], [6] first need to obtain may-information to obtain non-trivial must-information. This is not necessary for the phase-detecting must-analysis: It can obtain precise must-information even for programs where *no* static analysis can obtain may-information. The phase-detecting may-analysis is superior to prior analyses, too: It can predict misses on sequences that contain only accesses to $k+1$ pairwise different blocks. This is the theoretical minimum for predicting misses. Prior analyses require nearly twice the number of pairwise different blocks to predict misses.

In Section V, we evaluate the phase-detecting analyses at hand of synthetic benchmarks that provide detailed quantitative measures. We compare the phase-detecting analyses with prior analyses and to the collecting semantics of FIFO, which delimits the precision of *any* static analysis. The must-analysis closes most of the gap between prior analyses and the collecting semantics. In most cases, the may-analysis is also more precise than prior ones.

II. Foundations

A. Static Analysis

Static analysis determines properties of programs without actually executing the programs. Since the properties to determine are commonly incomputable, abstraction has to be employed. The level of abstraction governs the trade-off between analysis precision and analysis complexity.

One formal method in static analysis is abstract interpretation [9], which our work is based on. 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 $\alpha_{\widehat{D}} : \mathcal{P}(D) \to \widehat{D}$ and a concretization function $\gamma_{\widehat{D}} : \widehat{D} \to \mathcal{P}(D)$.

The result of an abstract interpretation are invariants for each program point, which are represented by values of a domain $\widehat{D}$. A program is analyzed by performing a fixed-point computation on a set of equations induced by that program. The equations are set up with the help of an *abstract transformer*, $\mathcal{U}_{\widehat{D}} : \widehat{D} \times I \to \widehat{D}$, that describes how abstract values before and after instructions I are correlated. If an instruction has multiple predecessors, a *join function* $\mathcal{J}_{\widehat{D}} : \widehat{D} \times \widehat{D} \to \widehat{D}$ combines all incoming values into a single one.

B. 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 equally-sized *memory blocks* $\mathcal{B}$. Blocks are cached as a whole in cache lines of equal 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"). To enable an efficient look-up, caches are partitioned into equally-sized *cache sets* $q \in \mathcal{Q}_k$ and each block can only be stored in one cache set. The size of a cache set is called the *associativity* 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 remainder of the paper, we will continue to use k for the associativity of a cache.

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), pseudo-LRU (PLRU) a cost-efficient variant of LRU, and first-in first-out (FIFO). For details on the implementation of caches in hardware refer to Jacob [10].

C. The FIFO Policy

A k-way associative FIFO cache set q can therefore be modeled as a k-tuple of memory blocks $b_i \in \mathcal{B}$, which are ordered from last-in to first-in from left to right:

$$q = [b_1, \ldots, b_k] \in \mathcal{Q}_k := \mathcal{B}^k$$

The update function $\mathcal{U}_{\mathcal{Q}_k} : \mathcal{Q}_k \times \mathcal{B} \to \mathcal{Q}_k$ models the effect on a cache set when accessing a memory block b:

$$\mathcal{U}_{\mathcal{Q}_k}([b_1, \ldots, b_k], b) := \begin{cases} [b_1, \ldots, b_k] & : \exists i : b_i = b \\ [b, b_1, \ldots, b_{k-1}] & : \text{otherwise} \end{cases}$$

A cache hit (first case) does not change the cache set. A cache miss (second case) inserts the new block at position 1, shifting the others to the right and evicting the block at the rightmost position. Finally, let $C_n : \mathcal{Q}_k \to \mathcal{P}(\mathcal{B})$ be the cache contents:

$$C_n([b_1, \ldots, b_k]) := \begin{cases} \emptyset & : n = 0 \\ \{b_1, \ldots, b_n\} & : n < k \\ \{b_1, \ldots, b_k\} & : n \geq k \end{cases}$$

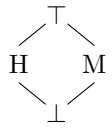
computes the set of memory blocks contained in the last n last-in positions of a cache set. For short, $C(q) := C_k(q)$.

D. Access Sequences

Let $\mathcal{S} := \mathcal{B}^*$ be the set of finite access sequences, e.g. $s_1 := \langle a, b, a, c \rangle, s_2 := \langle d \rangle \in \mathcal{S}$. Let $A : \mathcal{S} \to \mathcal{P}(\mathcal{B})$, where $A(s)$ is the set of memory blocks accessed by s, e.g. $A(s_1) = \{a, b, c\}$. Furthermore, let $\circ$ denote access sequence concatenation, e.g. $s_1 \circ s_2 = \langle a, b, a, c, d \rangle$. The update function $\mathcal{U}_{\mathcal{Q}_k}$ can be lifted from individual access to access sequences in the expected way.

E. Static Cache Analysis

The aim of static cache analysis is to classify individual memory accesses as hits (H) or misses (M). However, for some accesses an analysis might fail to classify them as hits or misses, i.e. they remain unclassified ($\top$). The classification lattice and its induced join ($\sqcup$) and meet ($\sqcap$) are defined as:



$\sqcup$	H	M	$\top$
H	H	$\top$	$\top$
M	$\top$	M	$\top$
$\top$	$\top$	$\top$	$\top$

$\sqcap$	H	M	$\top$
H	H	$\bot$	H
M	$\bot$	M	M
$\top$	H	M	$\top$

Cache analysis by abstract interpretation computes *must*- and *may*-cache information [2] at program points: Must- and may-cache information are under- and over-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 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.

In general, a cache analysis has to consider *any* possible cache state at program start: No memory block *must* be cached, any block *may* be cached. Assuming an empty cache is not safe because the program might incur less cache misses than starting with a partially filled cache [11].

One way to attenuate this lack of 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 would get complete must- and may-information. However, cache information can be partially lost during the analysis, e.g. due to control-flow joins, and then has to be regained.

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.

$$
\begin{aligned}
q_1 = [\bot,\bot,\bot,\bot] &\xrightarrow[\text{M}]{a} [a,\bot,\bot,\bot] \xrightarrow[\text{H}]{a} [a,\bot,\bot,\bot] \xrightarrow[\text{M}]{b} [b,a,\bot,\bot] \xrightarrow[\text{M}]{c} [c,b,a,\bot] = q_1' \\
q_2 = [a,x,b,c] &\xrightarrow[\text{H}]{a} [a,x,b,c] \xrightarrow[\text{H}]{a} [a,x,b,c] \xrightarrow[\text{H}]{b} [a,x,b,c] \xrightarrow[\text{H}]{c} [a,x,b,c] = q_2' \\
q_3 = [x,y,z,a] &\xrightarrow[\text{H}]{a} [x,y,z,a] \xrightarrow[\text{H}]{a} [x,y,z,a] \xrightarrow[\text{M}]{b} [b,x,y,z] \xrightarrow[\text{M}]{c} [c,b,x,y] = q_3' \\
q_4 = [x,y,b,z] &\xrightarrow[\text{M}]{a} [a,x,y,b] \xrightarrow[\text{H}]{a} [a,x,y,b] \xrightarrow[\text{H}]{b} [a,x,y,b] \xrightarrow[\text{M}]{c} [c,a,x,y] = q_4'
\end{aligned}
$$

Fig. 1: Dependency of FIFO cache set contents on the initial state.

III. FIFO Analysis by Phase-Detection

In this section we describe our ideas and formalize the analysis. Section III-A motivates why predicting hits for FIFO is difficult and presents *phase detection* as a solution. Section III-B then shows how these ideas can be translated into an efficient abstract domain for a FIFO must-analysis. Section III-C and Section III-D are structured analogously and treat the prediction of misses by a may-analysis.

A. Predicting Hits: Challenge & Idea

To see the difficulty inherent in FIFO, consider the examples in Figure 1. The access sequence $s = \langle a,a,b,c\rangle$ is carried out on different cache sets q_i of associativity 4. Although only 3 different memory blocks $\{a,b,c\}$ are accessed, some of the resulting cache sets q_i' do not contain all of the accessed blocks. In contrast, a k-way cache set with LRU replacement always consists of the k most-recently-used memory blocks, e.g. $\{a,b,c\}$ would be cached after carrying out s, independently of the initial state. This makes analysis of FIFO considerably harder than analysis of LRU.

To generalize, consider a FIFO cache set with unknown contents. After observing a memory access to a block a, trivial must-information is available: One knows that a must be cached but the position of a within the cache set is unknown. As the access to a could not be classified as a miss, another access to a different block b may actually 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 (as in case of q_3 in Figure 1). Thus, without implicitly or explicitly classifying some accesses as misses, it is hard to infer that two or more blocks are cached.

The following lemma shows how much information is available after accessing a set of memory blocks once:

Lemma 1. *Let $s \in \mathcal{S}, |A(s)| = n \leq k$, i.e. the access sequence s contains at most k pairwise different blocks. Then, for all cache sets $q \in \mathcal{Q}_k$ and $q' := \mathcal{U}_{\mathcal{Q}_k}(q,s)$:*

$$A(s) \subseteq C(q') \vee C_1(q') \subseteq A(s)$$

If no miss happened during s, all blocks were cached in q. Hence, the blocks are still cached after the update ($A(s) \subseteq C(q')$) since FIFO does not change its state upon a hit. Otherwise, at least one miss must have happened. In that case, the last-in position of q' must contain a block that caused a miss ($C_1(q') \subseteq A(s)$).

Lemma 1 is tight in the sense that $A(s) \subseteq C(q') \vee C_i(q') \subseteq A(s)$ does not hold for $i \geq 2$ in general. Hence, accessing a set of memory blocks does not imply that all of the accessed blocks will be cached, as can also be seen in Figure 1. However, if the same set of memory blocks is accessed multiple times subsequently, misses "accumulate" in the last-in positions. To profit from this, one has to partition access sequences into subsequences that each access the same set of memory blocks. We call such subsequences *phases*:

Definition 2 (Phase)**.** For a set of memory blocks B, a *B-phase* is an access sequence s such that $A(s) = B$. The *phase blocks* of a B-phase are the memory blocks in B. If $|B| = |A(s)| = n$ we say that s is of size n. For examples of phases and their inclusion consider:

$$\overbrace{\langle \underbrace{a}_{\{a,b\}\text{-phase}}, \underbrace{b}_{\{a,b\}\text{-phase}}, \underbrace{b}_{\{a,b\}\text{-phase}}, a, c \rangle}^{\{a,b,c\}\text{-phase}}$$

$$\langle a, b, b, \overbrace{a, \underbrace{c}_{\{c,d\}\text{-phase}}, \underbrace{d}_{\{c,d\}\text{-phase}}}^{\{a,c,d\}\text{-phase}}, \overbrace{\underbrace{d}_{\{c,d\}\text{-phase}}, \underbrace{d}_{\{c,d\}\text{-phase}}, \underbrace{c}_{\{c,d\}\text{-phase}}, \underbrace{d}_{\{c,d\}\text{-phase}}, \underbrace{c}_{\{c,d\}\text{-phase}}}^{\{a,c,d\}\text{-phase}}, a \rangle$$

The following theorem shows how much information is available after j subsequent B-phases:

Theorem 3. *Let $s \in \mathcal{S}$ be a B-phase of size $n \leq k$ that can be partitioned into $j \leq n$ B-phases, s_i, i.e. $s := s_1 \circ \ldots \circ s_j$. Then for each $q \in \mathcal{Q}_k$, and $q' := \mathcal{U}_{\mathcal{Q}_k}(q,s)$:*

$$A(s) \subseteq C(q') \vee C_j(q') \subseteq A(s)$$

To evict a newly inserted block from a FIFO cache set, it takes k misses. Since s contains at most k different blocks ($|A(s)| \leq k$), a block inserted by a miss in s cannot be evicted by s. Hence, there can be at most one miss to each block in B. Thus, if accessing a block results in a miss, the accessed block must be different from all previously accessed blocks that resulted in misses. After j B-phases, either (at least) one miss happened in each phase (which implies $C_j(q') \subseteq A(s)$) or there was a phase with no misses at all. In the latter case, all blocks in B were cached before that phase and remain cached throughout the remainder of s ($A(s) \subseteq C(q')$).

As a corollary, after at most $|B|$ B-phases all blocks in B must be cached: Since $|C_n(q')| = |A(s)| = n$, $C_n(q') \subseteq A(s)$ implies $C_n(q') = A(s)$. This result is tight: In general, $n-1$ phases are not sufficient to guarantee hits, i.e. a miss can happen in the last phase, s_n.

Sketch of the analysis: While the analysis processes memory accesses one-by-one, it virtually partitions the access se-

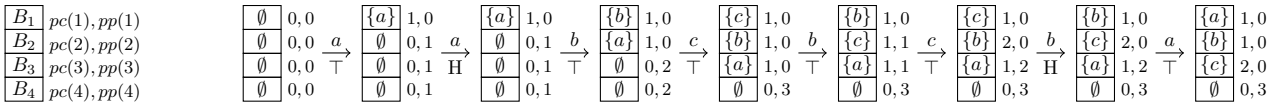


Fig. 2: Evolution of must-analysis information when processing $\langle a, a, b, c, b, c, b, a\rangle$. Each state consists of an abstract LRU-stack with annotated phase progress and phase counters $(pc(n), pp(n))$. The second access to a must be a hit since there was one $\{a\}$-phase before. The last access to b must be a hit since there were two $\{b, c\}$-phases before $(pc(2) = 2, L_2^{\cap}(lru^{\cap}) = \{b, c\})$. The whole sequence can be partitioned into two $\{a, b, c\}$-phases.

quence into phases. To predict hits using Theorem 3, the must-analysis proceeds in three stages: In the first two stages, it accumulates information by virtually partitioning the access sequence into phases. In the first stage, the phase blocks B are determined, i.e. the memory accesses encountered in the program define the first B-phase. In the second stage, the analysis tries to detect another $|B|-1$ B-phases. If this succeeds the analysis has detected $|B|$ B-phases in total and can proceed with the third stage. In the third stage, the analysis can exploit the accumulated information. It can predict hits for all accesses to blocks in B, until the first access to a block that is not contained in B.

If, at any time after the first stage, a block is accessed that does not belong to the phase blocks B, the analysis has to restart in stage one. This is because an access such a block may evict *any* block contained in B from the cache. Even after $|B|$ B-phases one "only" knows that all blocks in B must be cached. There is no information about the positions of the blocks within the cache set. Hence, *any* block may reside in the first-in position.

To arrive at a viable analysis, one has to overcome a last obstacle: In stage one, it is not apparent how to choose the phase blocks B, i.e. when to end the first phase. Note, however, that for a given access sequence, the phase size uniquely determines the phase blocks B: For phase size n, the phase blocks are the n most-recently-used blocks. If the phase size is chosen too large, completing stage two might take long or may never happen. If it is chosen too small, accesses to blocks not in B might happen frequently, which forces the analysis to restart in stage one. To solve this problem, our must-analysis actually performs k analyses in parallel, one for each phase size from 1 to k.

B. Efficient Must-analysis Implementation

Conceptually, each of the k analyses has to maintain the following information:

- *phase blocks* $B \subseteq \mathcal{B}$, which are defined by stage one.
- *phase progress* $P \subseteq B \subseteq \mathcal{B}$, which contains blocks that have already been accessed in the current phase. This is used in stage two to determine when a B-phase ends.
- *phase counter* $pc \in \mathbb{N}$, which counts the number of completed B-phases. Used to decide when stage two ends.

Representing this information naively would result in an inefficient implementation. In this section we show how the k analyses can be implemented space-efficiently by sharing information between them.

Each analysis has to maintain its phase blocks B. As soon as a block $b \notin B$ is accessed, the analysis has to start over. As noted above, the phase blocks for phase size n, B_n, are always the n most-recently-used blocks. This implies that the blocks of a phase with larger size always subsume the blocks of a phase of smaller size, i.e. $B_n \supseteq B_{n-1}$. The data structure perfectly suitable for this kind of "suffix-sharing" is an LRU-stack: In an LRU-stack the n most-recently-used blocks are contained in the n topmost positions. Thus, a single LRU-stack of size k is sufficient to represent the phase blocks for all phase sizes. Figure 2 shows how an LRU-stack changes upon accesses.

To generalize from access sequences to arbitrary control-flow, one needs an abstraction for LRU-stacks. The abstraction must allow for conservatively concluding when a phase ends, i.e. it must be able to tell when all phase blocks *must* have been accessed. This condition is fulfilled, for instance, by the LRU must-analysis of Ferdinand [2], which is the first constituent of our domain:

$$lru^{\cap} \in \mathrm{Lru}_k^{\cap} := \mathcal{P}(\mathcal{B})^k$$

The LRU must-analysis maintains an "abstract" LRU must-stack with a set of blocks for each position. To "extract" the (approximation of) phase blocks from the must-stack we define the function $L_n^{\cap} : \mathrm{Lru}_k^{\cap} \to \mathcal{P}(\mathcal{B})$:

$$L_n^{\cap}([B_1, \ldots, B_k]) := \bigcup_{i=1}^{n} B_i$$

For short, $L^{\cap}(lru^{\cap}) := L_k^{\cap}(lru^{\cap})$. Our analysis builds upon the important invariant of the LRU must-analysis that $L_n^{\cap}(lru^{\cap})$ is an *under*approximation of the set of the n most-recently-used blocks. Hence, the phase blocks for phase size n are approximated by $L_n^{\cap}(lru^{\cap})$.

Each of the k analyses needs to count the number of detected phases. We represent this as a mapping from phase size to phase counter:

$$PC_k := \{f : \{1, \ldots, k\} \to \{0, \ldots, k\} \mid \forall i : f(i) \leq i\}$$

For a $pc \in PC_k$, $pc(n)$ is a lower bound on the number of detected $L_{pp(n)}^{\cap}(lru)$-phases. The first column next to the stacks in Figure 2 shows the value of the phase counters.

To determine when a phase ends, each analysis has to maintain its phase progress P, i.e. the blocks that have already been accessed in the current phase. These blocks are always a subset of the corresponding phase blocks, $P_n \subseteq B_n$. More precisely, if i pairwise different blocks have already been accessed in a phase, these blocks are contained in the i topmost positions of

the LRU-stack, i.e. $P_i = L_i^{\cap}(lru^{\cap})$. Hence, the phase progress for a single phase size can be represented by a "pointer" into the LRU-stack. For the k phase sizes, there are k pointers, which we represent as a mapping from phase size to phase progress:

$$PP_k := \{f : \{1, \ldots, k\} \to \{0, \ldots, k-1\} \mid \forall i : f(i) < i\}$$

For a $pp \in PP_k$, $pp(n)$ indicates the phase progress for phase size n. The blocks contained in the $pp(n)$ topmost positions in the LRU-stack, $L^{\cap}_{pp(n)}(lru^{\cap})$, are an *under*approximation of the set of blocks that have been accessed in the current phase. The second column next to each stack in Figure 2 shows the value of the phase progress for each phase size.

In summary, our must-analysis consists of k analyses. The phase blocks of all k analyses are managed collectively by a single LRU must-analysis $\mathrm{Lru}^{\cap}_k$. The k phase progress are represented by pointers PP_k into the LRU must-stack, and there are k phase counters PC_k. The domain of our must-analysis is:

$$PMust_k := \mathrm{Lru}^{\cap}_k \times PP_k \times PC_k$$

Concretization: The set of concrete cache sets represented by our must-information is given by the concretization function. Since the must analysis comprises k analyses for different sets of phase blocks, the concretization is the intersection of the k respective concretizations:

$$\gamma_{PMust_k} : PMust_k \to \mathcal{P}(\mathcal{Q}_k)$$
$$\gamma_{PMust_k}((lru^{\cap}, pp, pc)) := \bigcap_{1 \le n \le k} \gamma_n(L^{\cap}_n(lru^{\cap}), L^{\cap}_{pp(n)}(lru^{\cap}), pc(n))$$
$$\gamma_n : \mathcal{P}(\mathcal{B}) \times \mathcal{P}(\mathcal{B}) \times \mathbb{N} \to \mathcal{P}(\mathcal{Q}_k)$$
$$\gamma_n(B, P, pc) := \{q \in \mathcal{Q}_k \mid B \subseteq C(q) \vee C_{pc+am}(q) \subseteq B \wedge am := |P \setminus C(q)|\}$$

Due to Theorem 3 either all phase blocks are cached ($B \subseteq C(q)$) or at least pc misses must have happened. For each block that was accessed in the current phase but is not cached ($P \setminus C(q)$) an additional miss must have happened. Hence, the last $pc+am$ last-in positions must contain phase blocks ($C_{pc+am}(q) \subseteq B$).

Classification: The function $\mathcal{C} : PMust_k \times \mathcal{B} \to Class$ shows how our must-analysis classifies memory accesses.

$$\mathcal{C}((lru^{\cap}, pp, pc), b) := \begin{cases} \mathrm{H} & : \exists i : pc(i) = i, b \in L^{\cap}_i(lru^{\cap}) \\ \mathrm{M} & : pc(k) = k, b \notin L^{\cap}(lru^{\cap}) \\ \top & : \text{otherwise} \end{cases}$$

The analysis can classify a hit for an access to a block b if it detected i phases of size i and b belongs to the corresponding phase blocks (case 1). See Figure 2 for examples. If k blocks are cached, no other block may be cached (case 2). In this special case the analysis can predict misses for all blocks not belonging to the phase blocks for phases size k. Otherwise, the analysis cannot classify the access (case 3).

Update: The update function is defined component-wise:

$$\mathcal{U}_{PMust_k} : PMust_k \times \mathcal{B} \to PMust_k$$
$$\mathcal{U}_{PMust_k}((lru^{\cap}, pp, pc), b) := (lru^{\cap\prime}, pp'', pc'')$$
$$lru^{\cap\prime} := \mathcal{U}_{\mathrm{Lru}^{\cap}_k}(lru^{\cap}, b)$$
$$pp' := \lambda n. \begin{cases} pp(n) & : b \in L^{\cap}_{pp(n)}(lru^{\cap}) \\ pp(n)+1 & : b \in L^{\cap}_n(lru^{\cap}) \setminus L^{\cap}_{pp(n)}(lru^{\cap}) \\ \mathrm{mcs}(lru^{\cap}, n) & : b \notin L^{\cap}_n(lru^{\cap}) \end{cases}$$
$$pc' := \lambda n. \begin{cases} 0 & : b \notin L^{\cap}_n(lru^{\cap}) \\ pc(n) & : \text{otherwise} \end{cases}$$
$$pc'' := \lambda n. \begin{cases} pc'(n)+1 & : pp'(n) = n, pc'(n) < n \\ pc'(n) & : \text{otherwise} \end{cases}$$
$$pp'' := \lambda n. \begin{cases} 0 & : pp'(n) = n \\ pp'(n) & : \text{otherwise} \end{cases}$$

For the LRU must-stack, the normal update of the LRU-analysis can be reused, which maintains the invariant described above. For an explanation on *how* the "abstract" LRU-stack is updated the interested reader is referred to [2]. pp and pc are updated in two steps. The first step (pp' and pc') has three cases:

1) In the first case, b is a phase block but has already been accessed in the current phase ($b \in L^{\cap}_{pp(n)}(lru^{\cap})$); no phase progress. See the second access to a in Figure 2.

2) In the second case, b is a phase block and has not been accessed in the current phase; the phase progress can be incremented. E.g. the second access to b in Figure 2 increments $pp(2)$ and $pp(3)$.

3) In the last case, the accessed block b is not a phase block ($b \notin L^{\cap}_n(lru^{\cap})$); the phase blocks change and the analysis has to restart in stage one. In Figure 2, the last access to a enforces a restart for phase size 2 and 1. Although the phase counter is reset first ($pc'(n) = 0$), the analysis does not have to start from scratch. Instead of starting a new phase with only b as phase progress, the (up to) n most-recently-used blocks become the new phase progress. These might be enough to complete a new first phase.

The analysis completes a phase of size n if $pp(n) = n$. Hence, at that point $|L^{\cap}_n(lru^{\cap})| = n$ must hold. However, after a join it may be that $|L^{\cap}_n(lru^{\cap})| < n$. To prevent the case $|L^{\cap}_{pp(n)}(lru^{\cap})| < pp(n) = n$ we make sure that $pp(n) \le |L^{\cap}_{pp(n)}(lru^{\cap})|$ holds at any time. To do so, the function $\mathrm{mcs} : \mathrm{Lru}^{\cap}_k \times \mathbb{N} \to \mathbb{N}$ computes a "maximal concrete sub-stack". For a given position p it computes the maximal position n in the stack such that the sub-stack up to position n contains exactly n blocks: $\mathrm{mcs}(lru^{\cap}, p) := \max\{n \mid |L^{\cap}_n(lru^{\cap})| = n \le p\}$.

The second step of the update (pp'' and pc'') handles phase completion. If a phase is completed ($pp'(n) = n$), the phase counter is incremented and the phase progress is reset to 0.

Join: For the LRU-stack, the join function of the $\mathrm{Lru}^{\cap}_k$ domain can be reused.

$$\mathcal{J}_{PMust_k} : PMust_k \times PMust_k \to PMust_k$$
$$\mathcal{J}_{PMust_k}((lru_1^\cap, pp_1, pc_1), (lru_2^\cap, pp_2, pc_2)) := (lru^{\cap\prime}, pp'', pc')$$
$$lru^{\cap\prime} := \mathcal{J}_{\mathrm{Lru}_k^\cap}(lru_1^\cap, lru_2^\cap)$$
$$pc' := \lambda n. \begin{cases} \min\{pc_1(n), pc_2(n)\} & : \left|L_n^\cap(lru^{\cap\prime})\right| = n \\ 0 & : \text{otherwise} \end{cases}$$
$$pp' := \lambda n. \begin{cases} \min\{pp_1(n), pp_2(n)\} & : pc_1(n) = pc_2(n) \\ pp_1(n) & : pc_1(n) < pc_2(n) \\ pp_2(n) & : pc_1(n) > pc_2(n) \end{cases}$$
$$pp'' := \lambda n. \begin{cases} \mathrm{mcs}(lru^{\cap\prime}, pp'(n)) & : \left|L_n^\cap(lru^{\cap\prime})\right| = n \\ \mathrm{mcs}(lru^{\cap\prime}, n) & : \text{otherwise} \end{cases}$$

The updates of pp and pc have two main cases: If the sub-stack still contains n elements after the join ($\left|L_n^\cap(lru^{\cap\prime})\right| = n$), the minimum of the respective values are the join result. Otherwise, the analysis has to restart with $pc' = 0, pp'' = \mathrm{mcs}(lru^{\cap\prime}, n)$ as in one case of the update. The three cases in pp' stem from the fact that one actually takes the minimum of the overall progress, pp and pc combined, and not the minimum of the individual values. E.g. if $pc_1(n) > pc_2(n)$ then pc_1, pp_1 is the greater overall progress. Hence pp_2 is the "smaller" phase progress, regardless of pp_1.

C. Predicting Misses: Challenge & Idea

Predicting misses for FIFO is also more difficult than predicting misses for LRU: Consider the access sequence $s = \langle a, b, c, d, e\rangle$. If s is conducted on a 4-way associative cache set with LRU replacement, the resulting cache set will consist of the 4 most-recently-used memory blocks $\{b, c, d, e\}$ independently of the set's initial state. Subsequent accesses to other memory blocks can be classified as cache misses. For FIFO, this is not necessarily the case: If s is conducted on the initial state $[f, g, h, i]$ with FIFO replacement, the resulting cache set will be $\mathcal{U}_{\mathcal{Q}_k}([f, g, h, i], \langle a, b, c, d, e\rangle) = [e, d, c, b]$ which has the same contents as in the LRU case. On the other hand, for another initial state, $\mathcal{U}_{\mathcal{Q}_k}([f, a, b, c], s) = [e, d, f, a]$. In the latter case, the memory block f, which is not contained in the sequence s, has "survived" in the cache set. As *any* memory block could have survived in place of f, no may-information is available, i.e. it is not possible to classify accesses as misses at this point.

However, *some* knowledge can be inferred about the state of the FIFO cache set after conducting s: Since s contains $l = 5 \geq k = 4$ different memory blocks, there must have been at least $l - k = 5 - 4 = 1$ cache miss. So the $l - k$ last-in positions must contain blocks of the set $\{a, b, c, d, e\}$. We say that the positions are *covered* by the set. In fact, one does know a bit more about the state of the cache than that. Assume that there were exactly $l - k$ misses on the sequence. Then, there must have also been hits to k different memory blocks. In that case, the contents of the cache set would be completely covered by the contents of the sequence. Otherwise, there must have been at least $l - k + 1$ misses. So, in any case, the $l - k + 1$ last-in positions must be covered by the contents of s:

Lemma 4. *Let $s \in \mathcal{S}, |A(s)| = n \geq k$. Then, for each $q \in \mathcal{Q}_k$ and $q' := \mathcal{U}_{\mathcal{Q}_k}(q, s)$:*

$$C_{n-k+1}(q') \subseteq A(s)$$

As in the must-analysis, the idea for the may-analysis is to split the access sequences into several phases. Each phase will contribute a bit to the overall goal of predicting misses. Lemma 4 shows the progress that can be achieved in a single phase. In contrast to the must-analysis, different phases can contribute differently. A single, long phase s with $n = |A(s)| = 2k - 1$ blocks alone can provide may-information: $n - k + 1 = k$. Therefore $C_k(q') \subseteq A(s)$. A short phase with $|A(s)| = k$ on the other hand, provides only little progress towards predicting misses.

The following theorem shows that the progress of separate phases adds up:

Theorem 5. *Let s be an access sequence that can be partitioned into phases $s := s_1 \circ \ldots \circ s_j$ such that each s_i contains accesses to at least k pairwise different blocks: $|A(s_i)| = n_i \geq k$. Then for each $q \in \mathcal{Q}_k$ and $q' := \mathcal{U}_{\mathcal{Q}_k}(q, s)$:*

$$C_{\sum_{i=1}^{j}(n_i - k + 1)}(q') \subseteq A(s) = \bigcup_i A(s_i)$$

This is similar to Theorem 3 for the must-analysis. However, there is one important difference: For the may-analysis it is not necessary that all phases access the same blocks and hence may be of different size. Those relaxed conditions for the may-analysis entail an additional degree of freedom: The may-analysis can finish each phase profitably as soon as it contains at least k blocks ($|A(s_i)| = n_i \geq k$). Depending on the following accesses, it may be beneficial or detrimental to the precision of the analysis to do so. For an example, consider a 4-way associative cache set q and the sequences $s_1 = \langle a, b, c, d, e, a, b, c, d, e\rangle$ and $s_2 = \langle a, b, c, d, e, f, g, a, b, c\rangle$, which have a common prefix. For s_1 it is best to partition it into two phases $\langle a, b, c, d, e\rangle \circ \langle a, b, c, d, e\rangle$ of size 5 each, which results in $C_{2+2}(q') \subseteq \{a, b, c, d, e\}$. For s_2 it is better to partition into $\langle a, b, c, d, e, f, g\rangle \circ \langle a, b, c\rangle$. This way, Theorem 5 already guarantees $C_4(q') \subseteq \{a, b, c, d, e, f, g\}$ after the first phase of size 7. Again, we resolve this dependency on future accesses by efficiently performing multiple analyses in parallel.

D. Efficient May-analysis Implementation

As in the must-analysis, one can use an LRU-stack to collectively represent the phase blocks for all phase sizes. However, the abstraction of the stack has to be different for the may-analysis: The respective (sub-)stack contents must be a superset of the accumulated phase blocks of all phases, i.e. of the set $A(s) = \bigcup_i A(s_i)$ in Theorem 5. This way, the analysis can soundly predict misses for all blocks not contained in the substack. Consequently, the first constituent of the may-domain is the domain of the LRU may-analysis of [2]:

$$lru^\cup \in \mathrm{Lru}_k^\cup := \mathcal{P}(\mathcal{B})^k$$

To "extract" the (approximation of) accumulated phase blocks from the may-stack we define the function $L_n^{\cup} : \mathrm{Lru}_k^{\cup} \to \mathcal{P}(\mathcal{B})$:

$$L_n^{\cup}([B_1, \ldots, B_k]) := \bigcup_{i=1}^{n} B_i$$

For short, $L^{\cup}(lru^{\cup}) := L_k^{\cup}(lru^{\cup})$. The important invariant of the LRU may-analysis is that $L_n^{\cup}(lru^{\cup})$ is an *over*approximation of the set of the n most-recently-used blocks.

Although Theorem 5 only requires $n_i \geq k$, i.e. the phase size n_i is not bounded, one can limit the stack size. After one phase of size $2k-1$, Theorem 5 guarantees $C_{2k-1-k+1}(q) = C(q) \subseteq A(s)$, i.e. no blocks other than those of the sequence s may be cached. Hence, it would be redundant to consider phase sizes larger than $2k-1$. So we will use $\mathrm{Lru}_{2k-1}^{\cup}$.

There is no direct analogy to the phase counter of the must-analysis: Since in the may-analysis phases may be of different size, and different sizes induce different progress in analysis information, it would not be useful to simply count the number of phases. Instead, the may-analysis keeps track of its progress by counting the number of *covered ways*: The term $C_{\sum_{i=1}^{j}(n_i-k+1)}(q') \subseteq A(s)$ of Theorem 5 reflects that. The first $\sum(n_i - k + 1)$ ways of the cache set q' are covered by the set $A(s)$. $A(s)$ is in turn over-approximated using $L_n^{\cup}(lru^{\cup})$, where $n = |A(s)|$. If the number of covered ways reaches k, the analysis can predict misses for blocks not contained in the accumulated phase blocks $L_n^{\cup}(lru^{\cup})$.

Due to the additional degree of freedom described in Section III-C, it is beneficial to allow for *multiple* phase progress for each set of accumulated phase blocks. This way, the analysis can always follow both options:

- finishing a phase and starting a new one, which is beneficial if a small set of memory blocks is repeatedly accessed as in $s_1 = \langle a, b, c, d, e, a, b, c, d, e \rangle$.
- continuing a phase, which is beneficial if a large number of different memory blocks are accessed in a short period as in $s_2 = \langle a, b, c, d, e, f, g, a, b, c \rangle$.

Once a phase is finished, the partitioning of the access sequence up to this point becomes irrelevant. What matters is the number of covered ways $\sum(n_i - k + 1)$ provided by the partitioning. When different partitions finish their current phase after the same access, we keep only the greatest number of covered ways.

The information about covered ways can be represented as a mapping from accumulated phase blocks and phase progress to covered ways:

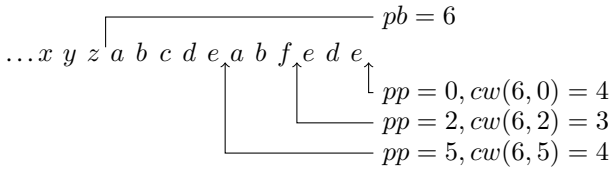
$$CW_k := \{1, \ldots, 2k-1\} \times \{0, \ldots, 2k-1\} \to \{0, \ldots, k\}$$

For a $cw \in CW_k$, $cw(pb, pp)$ is a lower bound on the number of last-in cache set ways that are covered by blocks contained in the LRU substack $L_{pb}^{\cup}(lru^{\cup})$. For a given phase progress pp, $L_{pp}^{\cup}(lru^{\cup})$ is an *over*approximation of the blocks that have been accessed in the current phase. If a block $b \notin L_{pp}^{\cup}(lru^{\cup})$ is accessed, the phase progress can safely be incremented.

Altogether, the domain of the may-analysis is:

$$PMay_k := \mathrm{Lru}_{2k-1}^{\cup} \times CW_k$$

Example: The analysis maintains the number of covered ways $cw(pb, pp)$ for each set of accumulated phase blocks, over-approximated by $L_{pb}^{\cup}(lru^{\cup})$, *and* phase progress, overapproximated by $L_{pp}^{\cup}(lru^{\cup})$. The following shows an access sequence with examples of may-analysis information of a 4-way cache set. The example only considers the blocks $\{a, b, c, d, e, f\}$, which are contained in the top $pb = 6$ LRU-stack positions, and different partitions of those into phases.



If the phase progress *at the end of the sequence* is $pp = 5$, the current phase started with the second access to a. Up to that point, the best partitioning provided 2 covered ways. The phase progress $pp = 5$ of the current phase provides an additional 2 covered ways. In total $cw(6, 5) = 4$ at the end of the sequence. If the previous phase was finished after the access to f, the best cw-value was 3. In this case, only e and d have been accessed in the current phase, which provides no additional guarantee. Hence, $cw(6, 2) = 3$. If a phase was finished after the last access, the best partition up to this point provides $cw(6, 0) = 4$. Note that phases can always be finished, i.e. $\forall i : cw(pb, 0) \geq cw(pb, i)$.

Concretization: As the may-analysis performs multiple sub-analyses in parallel, an abstract element *may* can also be interpreted as a conjunction of constraints, one constraint per sub-analysis. The information maintained in each sub-analysis is the blocks of the phase, the blocks of the phase progress, the phase progress, and the number of covered ways. The function CS computes the constraint set of an abstract element:

$$CS((lru^{\cup}, cw)) := \{(L_{pb}^{\cup}(lru^{\cup}), L_{pp}^{\cup}(lru^{\cup}), pp, cw(pb, pp)) \mid 1 \leq pb < 2k, 0 \leq pp \leq pb\}$$

In the example above, $(\{a, b, c, d, e, f\}, \{d, e\}, 2, 3)$ is contained for $pb = 6, pp = 2$. As *each* constraint in a constraint set holds true, the concretization of an abstract element is the intersection of all concretizations of the individual constraints:

$$\gamma_{PMay_k} : PMay_k \to \mathcal{P}(\mathcal{Q}_k)$$
$$\gamma_{PMay_k}((lru^{\cup}, cw)) := \bigcap_{c \in CS((lru^{\cup}, cw))} \gamma_{cw}(c)$$

The concretization of a single constraint is given by:

$$\gamma_{cw} : \mathcal{P}(\mathcal{B}) \times \mathcal{P}(\mathcal{B}) \times \mathbb{N} \times \mathbb{N} \to \mathcal{P}(\mathcal{Q}_k)$$
$$\gamma_{cw}(B, P, pp, cw) := \{q \in \mathcal{Q}_k \mid C_{cw+ac}(q) \subseteq B \wedge ac := \max\{0, \min\{pp, k-1\} - |P \cap C(q)|\}\}$$

where ac are *additionally covered ways* that are not already accounted for by cw. pp blocks have been accessed in the current phase, at most $|P \cap C(q)|$ are still contained in the cache. Hence,

$pp - |P \cap C(q)|$ is a lower bound on the number of additional misses that have happened. For each miss, a last-in position is covered. However, the value of cw already reflects the number of covered ways due to pp being larger than $k-1$. The min ensures that no miss is counted twice.

Classification: The analysis can classify a miss for b if b is not contained in a set of accumulated phase blocks ($b \notin L^{\cup}_{pb}(lru^{\cup})$) and those blocks cover all k ways of the cache set ($cw(pb, 0) = k$). As $cw(pb, 0) \geq cw(pb, i)$, it is sufficient to only check this value in the condition of the classification function $\mathcal{C} : PMay_k \times \mathcal{B} \to Class$:

$$\mathcal{C}((lru^{\cup}, cw), b) := \begin{cases} \mathrm{M} & : \exists pb : b \notin L^{\cup}_{pb}(lru^{\cup}), cw(pb, 0) = k \\ \top & : \text{otherwise} \end{cases}$$

Join: To define the join and update operations it is necessary to define a partial order on the constraints. Let c_1, c_2 be constraints of the form (B, P, pp, cw). The partial order on these constraints is defined by set inclusion of their respective concretizations: $c_1 \sqsubseteq c_2 :\iff \gamma_{cw}(c_1) \subseteq \gamma_{cw}(c_2)$. With the concretization function one can find that:

$$\begin{aligned}(B, P, pp, cw) \sqsubseteq (B', P', pp', cw') \iff \\ B \subseteq B' \wedge (pp' = 0 \vee P \subseteq P') \\ \wedge\, cw' \leq cw + \min\{0, \min\{k-1, pp\} - \min\{k-1, pp'\}\}.\end{aligned}$$

A guarantee is weaker, if less ways ($cw' \leq cw$) are covered, or if more memory blocks ($B' \supseteq B$) cover the same number of ways, or the same B covers the same number of ways but with higher phase progress ($P' \supseteq P$). As one can always end a phase, the phase progress does not matter for $pp' = 0$.

The join function first computes the joined LRU-stack $lru^{\cup\prime}$. For each pair of sub-stacks induced by pb' and pp', each operand provides a best guarantee on the number of covered ways (bcw_i). To be sound, the join has to take the minimum of those two best guarantees.

$$\begin{aligned}&\mathcal{J}_{PMay_k} : PMay_k \times PMay_k \to PMay_k \\ &\mathcal{J}_{PMay_k}((lru^{\cup}_1, cw_1), (lru^{\cup}_2, cw_2)) := (lru^{\cup\prime}, cw') \\ &lru^{\cup\prime} := \mathcal{J}_{\mathrm{Lru}^{\cup}_k}(lru^{\cup}_1, lru^{\cup}_2) \\ &\quad cw' := \lambda pb'.\lambda pp'.\, \min\{bcw_1, bcw_2\}\end{aligned}$$

To be optimal, one has to maximize cw' over all pairs of sub-stacks an operand provides, which can be further simplified.

$$bcw_i := \max_{c \,\in\, CS(lru^{\cup}_i, cw_i)} \{n \mid c \sqsubseteq (L^{\cup}_{pb'}(lru^{\cup\prime}), L^{\cup}_{pp'}(lru^{\cup\prime}), pp', n)\}$$

Update: Like the join, which determines the best guarantees provided by the operands, the update can also be explained in terms of guarantees. The proceeding is the same: One has to compute guarantees for an updated LRU-stack. The difference to the join is that old guarantees provided by the operand need to be updated first. As the may-update can profit from it, we distinguish between different classifications cl.

$$\begin{aligned}&\mathcal{U}_{PMay_k} : PMay_k \times \mathcal{B} \to PMay_k \\ &\mathcal{U}_{PMay_k}((lru^{\cup}, cw), b, cl) := (lru^{\cup\prime}, cw') \\ &lru^{\cup\prime} := \mathcal{U}_{\mathrm{Lru}^{\cup}_k}(lru^{\cup}, b) \\ &\quad cw' := \lambda pb'.\lambda pp'. \max_{c \in CS(lru^{\cup}, cw)} \{n \mid c' \in \mathcal{U}_{cw}(c, cl), \\ &\qquad\qquad c' \sqsubseteq (L^{\cup}_{pb'}(lru^{\cup\prime}), L^{\cup}_{pp'}(lru^{\cup\prime}), pp', n)\}\end{aligned}$$

The update of a constraint in case of a miss is:

$$\begin{aligned}&\mathcal{U}_{cw}((B, P, pp, cw), a, \mathrm{M}) := \{(\{a\}, \{a\}, 1, 1), \\ &\quad (B \cup \{a\}, \{a\}, 1, cw + 1), (B \cup \{a\}, P \cup \{a\}, pp, cw + 1)\}\end{aligned}$$

The three resulting constraints correspond to a complete restart, starting a new phase, and continuing the phase. In each case, cw can be incremented when a is added to B.

The update of a constraint in case of a hit is:

$$\begin{aligned}&\mathcal{U}_{cw}((B, P, pp, cw), a, \mathrm{H}) := \{(\emptyset, \{a\}, 1, 0), (B, \{a\}, 1, cw)\} \cup \\ &\begin{cases} (B, P, pp, cw) & : a \in P \\ (B, P \cup \{a\}, pp + 1, cw) & : a \notin P, pp < k - 1 \\ (B \cup P \cup \{a\}, P \cup \{a\}, pp + 1, cw + 1) & : a \notin P, pp \geq k - 1 \end{cases}\end{aligned}$$

The first two resulting constraints correspond to a complete restart and starting a new phase. The third constraint, continuing the phase, depends on further conditions: If the accessed block has already been accessed ($a \in P$) there is no progress. If $a \notin P$, a is added to the phase and the phase progress is incremented ($pp + 1$). If furthermore $pp \geq k - 1$, one more way must be covered by B together with one of the blocks accessed in this phase ($P \cup \{a\}$).

IV. Related Work and Qualitative Comparison

We limit our general discussion about cache analyses to those directed at WCET analysis, which include [12], [3], [13], [2].

Mueller et al. [12] 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. [3] 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. [13] 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. [2] introduce the concepts of may- and must-caches and present a precise, context-sensitive LRU analysis based on abstract interpretation.

As explained in Section III, statically analyzing FIFO is inherently more difficult than analyzing LRU. LRU-analysis can be seen as a special case of phase-detecting analyses: For LRU it is sufficient to detect *one phase* to prove that the associated phase blocks are cached, instead of n *phases* for phase size n.

The concept of relative competitiveness (RC) [8] bounds the performance of one replacement policy relative to the performance of another one. Under certain conditions, this allows for using cache analyses for one policy as cache analyses for other policies. 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. Likewise, an LRU must-analysis for 1-way associative cache (direct mapped) can be used as a must-analysis for a k-way FIFO. Due to the generic nature of this approach, however, the resulting analyses may be rather imprecise.

There is a striking relation between those RC-analyses and the analyses based on phases. PMUST and PMAY consist of a spectrum of sub-analyses for phase sizes $1 \ldots k$ and respectively $1 \ldots 2k-1$. The analyses based on relative competitiveness mark the extremal points of this spectrum. The PMAY sub-analysis for phase size $2k-1$ coincides with the RC may-analysis and the PMUST sub-analysis for phase size 1 coincides with the RC must-analysis.

The analyses PMUST and PMAY presented in this paper gain information by "global" observations in the following sense: They observe several accesses (a phase) and deduce that some property holds for a *subset* of those accesses, e.g. at least one of the accessed blocks must be cached but its position within the cache set is unknown. In contrast, both analyses presented in [6] are aimed at exploiting "local" miss classifications: If an *individual* access is predicted as a miss, its position is known (last-in position) and it will take k misses to evict it. In [6], the *canonical must-analysis*, CM, exploits a miss classification for a block b by predicting hits for b until k further misses might have happened. CM is complemented by the *early-miss exploiting may-analysis*, EMX, which provides miss classifications.

However, before EMX can predict any misses, it needs to observe accesses to $2k$ pairwise different blocks. If no misses are predicted by EMX, CM can only predict "trivial" hits, i.e. for subsequent accesses to the same block. As a consequence, CM can only predict non-trivial hits if more than $2k$ pairwise different blocks are accessed. In contrast, PMUST, can infer precise must-information for programs where *no* static analysis could classify an access as a miss. Additionally, PMAY, can classify misses without having to observe accesses to $2k$ pairwise different blocks. Accesses to $k+1$ can be sufficient.

In the related field of memory management, Madison and Batson [14] propose *bounded locality intervals* to capture the intuitive notion of "program phases". To determine the bounded locality intervals, they introduce a dynamic analysis that uses an *extended* LRU *stack*. Besides the referenced elements, this stack additionally keeps track of reference times of elements in substacks. Their stack and its update are similar to ours, however the stacks are "annotated" differently for different purposes.

V. Quantitative Evaluation

We compare three FIFO analyses with each other and to the collecting semantics:

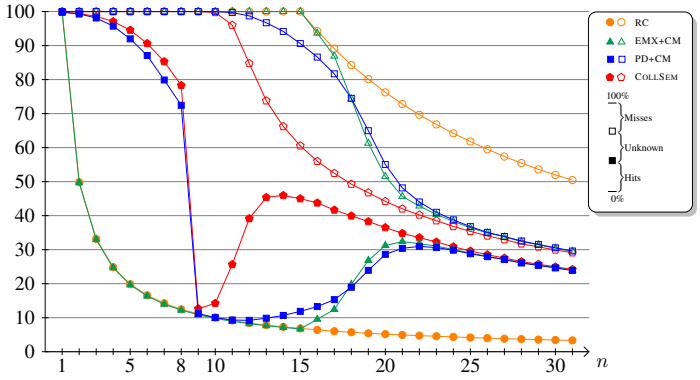


Fig. 3: Average hit- and miss-rates guaranteed by the analyses for an 8-way cache set.

- RC: a combination of a must- and a may-analysis, both solely based on relative competitiveness [8] as explained in Section IV.
- EMX+CM: the analysis presented in [6], which consists of EMX and CM as explained in Section IV.
- PD+CM: the phase-detecting analyses proposed in this paper, PMUST and PMAY, combined with CM.
- COLLSEM: the collecting semantics, which determines the set of cache set states that may reach a program point. If a memory access cannot be classified as hit or miss in the collecting semantics, no sound static analysis can do so. We computed this using an expensive analysis based on a powerset domain of concrete cache-set states.

To quantify the precision of the analyses, we analyzed random access sequences; the same as in [6]. For each $1 \leq n \leq 31$, we generated 100 random access sequences that contain 500 accesses to n pairwise different blocks. Hence, the greater 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 RC. The number of different blocks (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 top-down (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 $n = 15$: For 100 access sequences, each 500 accesses long and containing $n = 15$ distinct blocks, the average guaranteed hit-rate obtained by PD+CM was 12%, the average guaranteed miss-rate was 9%, and on average 79% could not be classified.

For a discussion of the collecting semantics, we refer the reader to [6]. Using analyses that are only based on relative competitiveness considerably overapproximates the collecting semantics, as the gap between RC and COLLSEM shows.

Both, RC and EMX+CM cannot predict any misses with less than $2k$ pairwise different blocks. Hence, the curves of RC and EMX+CM coincide up to $n = 15$. For larger n, EMX predicts more misses than RC, and due to cooperation with CM more hits can be predicted. This is the main contribution of [6] regarding analysis precision.

PMUST predicts a large fraction of the hits for $n \leq k$ and closes the gap to the collecting semantics. This is particularly important for loops in which different blocks are reused (temporal locality). Consider a loop that iterates i times and accesses $n \leq k$ different blocks, i.e. $s = \langle b_1, \ldots, b_n \rangle^i$. An analysis based on PMUST will result in a guaranteed hit-rate of $100(1 - \frac{n}{i})$ (e.g. 87.5% for $i = 32$ and $n = 4$). In contrast, EMX+CM cannot predict a single hit. Subsequent accesses to the same block (spatial locality), the "trivial hits", can be predicted equally well by CM and PMUST. For $1 \leq n \leq k$, the predicted hits are solely due to PMUST. Starting with $n = k + 1$, both, PMUST and CM predict hits.

PMAY is incomparable to EMX, i.e. it is not better for all inputs. However, it can predict misses with less pairwise different blocks; the upper curve already starts decreasing at $n = k+1$. For larger n, misses happen relatively often and EMX performs slightly better. However, PMAY can be implemented more efficiently than EMX: While EMX needs to maintain independent sets of memory block, PMAY uses sets that can be encoded as substacks of a single LRU stack. This roughly corresponds to a reduction in space from $O(k^2)$ to $O(k)$.

VI. Application in WCET Analysis

As in the case of abstract LRU domains, employing the phase-detecting FIFO domains in WCET analysis requires special attention regarding the analysis of loops. Even for simple loops, like `for (1..n) {a b}`, the bare analysis as described above will not be able to classify any hits. This is because of the join in the loop header: The number of observed $\{a, b\}$-phases is 0 before the loop and 1 at the end of the loop body. The join results in $pc(2) = 0$: No hits can be classified.

However, context-sensitive analyses can solve this problem. For simple loops, virtual loop peeling [15] is sufficient. A loop containing n accesses to a cache set needs to be virtually peeled n times. This way, the context-sensitive analysis of the above loop mimics a context-insensitive analysis of the program `a b a b for (3..n) {a b}`. Then, $pc(2) = 2$ before the loop and thus after the join, too: The hits to a and b get classified.

Although this topic deserves closer attention, the full particulars for different kinds of loops cannot be discussed here.

VII. Conclusions

With Theorem 3 and 5 we provide tight bounds on the number of phases that need to happen until all blocks accessed in these phases must be cached in a FIFO cache (until non-accessed blocks must be evicted, respectively). Furthermore, we generalized dynamic phase detection to *static phase detection* by employing abstract LRU stacks. Both together, static phase detection and the theorems, allow for designing precise FIFO analyses and answering questions left open by [6].

Especially PMUST has a much higher precision than prior analyses and is the first analysis that can predict a significant amount of hits in simple loops ($n \leq k$). For $k < n < 2k$, PMAY is also always more precise than prior analyses. For $n \geq 2k$, it generally depends on the input, but it is mostly more precise.

PMUST and PMAY actually perform a spectrum of sub-analyses in parallel to resolve dependencies on future accesses. Nevertheless, the analysis information that needs to be maintained can be efficiently encoded: We store sets of sets of memory blocks in abstract LRU stacks and annotate these stacks with the remaining analysis information, e.g. the phase progress and phase counters.

VIII. Acknowledgments

We want to thank Sebastian Hack and the anonymous reviewers for their helpful comments on this paper. 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).

References

[1] R. Wilhelm *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 1–53, 2008.

[2] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems," *Real-Time Systems*, vol. 17, no. 2-3, pp. 131–181, 1999.

[3] R. T. White, C. A. Healy, D. B. Whalley, F. Mueller, and M. G. Harmon, "Timing analysis for data caches and set-associative caches," in *RTAS*. IEEE, 1997, p. 192.

[4] S. Ghosh, M. Martonosi, and S. Malik, "Precise miss analysis for program transformations with caches of arbitrary associativity," in *ASPLOS*. ACM, 1998, pp. 228–239.

[5] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck, "Exact analysis of the cache behavior of nested loops," in *PLDI*. ACM, 2001, pp. 286–297.

[6] D. Grund and J. Reineke, "Abstract interpretation of FIFO replacement," in *SAS*. Springer, 2009, pp. 120–136.

[7] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Timing predictability of cache replacement policies," *Real-Time Systems*, vol. 37, no. 2, pp. 99–122, 2007.

[8] J. Reineke and D. Grund, "Relative competitive analysis of cache replacement policies," in *LCTES*. ACM, 2008, pp. 51–60.

[9] P. Cousot and R. Cousot, *Building the Information Society*. Kluwer, 2004, ch. Basic Concepts of Abstract Interpretation.

[10] B. Jacob, S. W. Ng, and D. T. Wang, *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008.

[11] J. Reineke, "Caches in WCET analysis," Ph.D. dissertation, Saarland University, 2008.

[12] F. Mueller, D. B. Whalley, and M. Harmon, "Predicting instruction cache behavior," in *LCTRTS*. ACM, 1994.

[13] Y.-T. S. Li, S. Malik, and A. Wolfe, "Cache modeling for real-time software: Beyond direct mapped instruction caches," in *RTSS*. IEEE, 1996, p. 254.

[14] A. W. Madison and A. P. Batson, "Characteristics of program localities," *Comm. of the ACM*, vol. 19, no. 5, pp. 285–294, 1976.

[15] F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand, "Analysis of loops," in *CC*. Springer, 1998, pp. 80–94.