Toward Precise PLRU Cache Analysis*

Daniel Grund[1] and Jan Reineke[2]

1 **Saarland University, Saarbrücken, Germany.**
`grund@cs.uni-saarland.de`
2 **University of California, Berkeley, USA.**
`reineke@eecs.berkeley.edu`

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, cache analysis is mandatory.
The subject-matter of this article is the static analysis of the tree-based PLRU cache replacement policy (pseudo least-recently used), for which the precision of analyses lags behind those of other policies. We introduce the term *subtree distance*, which is important for the update behavior of PLRU and closely linked to the peculiarity of PLRU that allows cache contents to be evicted in "logarithmic time". Based on an abstraction of subtree distance, we define a must-analysis that is more precise than prior ones by excluding spurious logarithmic-time eviction.

Keywords and phrases Cache Analysis, PLRU Replacement, PLRU Tree

Digital Object Identifier 10.4230/OASIcs.WCET.2010.23

1 Introduction

In hard real-time systems, one needs to derive offline guarantees for the timeliness of reactions. Thereto, one must determine bounds on the worst-case execution time (WCET) of programs [12]. 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 statically classify memory accesses as cache hits or cache misses—is a challenging problem.

Besides the determination of addresses that are being accessed, cache analysis is concerned with the analysis of the employed replacement policy. Precise and efficient analyses have been developed early on for LRU [3, 11] and more recently also for FIFO [4, 5]. However, there is a third major policy, PLRU (pseudo least-recently used), which is for instance employed in the TriCore 1798 and several PowerPC variants (MPC603e, MPC755, MPC7448). Compared to analyses of LRU or FIFO, no analyses of similar precision exist for PLRU. The best known PLRU analysis was introduced in [6] for associativity 8 and later categorized as an instance of relative-competitiveness-based analyses [8]. For PLRU, such analyses can at most classify $\log_2(k) + 1$ out of k cached memory blocks as hits.

* The research leading to these results has received funding from or was supported by the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement no 216008 (Predator), the DFG as part of the Transregional Collaborative Research Center SFB/TR 14 (AVACS), and the Center for Hybrid and Embedded Software Systems at UC Berkeley (CHESS), the latter of which receives support from the National Science Foundation (NSF awards #CCR-0225610 (ITR), #0720882 (CSR-EHS: PRET) and #0931843 (ActionWebs)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312 and AF-TRUST #FA9550-06-1-0244), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC) and the following companies: Bosch, National Instruments, Thales, and Toyota.

10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).
Editor: Björn Lisper; pp. 23–35

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In Section 3, we describe three properties of PLRU that make its analysis challenging and coin terms for them: *non-trivial logical states*, *logarithmic-time eviction* and *arbitrary survival*. In Section 4, we address the first one: we adapt knowledge about PLRU [9] and show how to represent the logical state of PLRU cache sets by abstracting from cache set states that are physically different but exhibit the same replacement behavior. Section 5, presents our main contributions. We identify two new sizes of PLRU that are relevant to logarithmic-time eviction: *number of leading zeros* and *subtree distance*. Subsequently, we define a must-analysis that is based on abstractions of those two sizes and can exclude spurious logarithmic-time eviction.

In Section 6 we cover closely-related work including relative-competitiveness-based analyses, against which we compare in Section 7. The introduced analysis is more precise than prior ones and has strong advantages in the analysis of loops, at the cost of an acceptable loss in analysis performance.

2 Foundations

Memory Blocks, Caches, and Access Sequences

Caches store a subset of the main memory's contents to bridge the latency gap between CPU and main memory. To reduce management overhead, main memory is logically partitioned into a set of equally-sized *memory blocks* $\mathcal{B}$. Blocks are cached as a whole in cache lines of equal size. To enable an efficient cache look-up of blocks, each block can only be stored in a small number of cache lines. For this purpose, caches are partitioned into equally-sized *cache sets* $\mathcal{Q}_{P_k}$. The size of a cache set is called the *associativity* k of the cache. As the cache is smaller than the main memory, the number of memory blocks that map to a particular cache set is greater than the size of the cache set. Upon cache misses, a *replacement policy* must decide which memory block to replace. Well-known policies for individual cache sets are least-recently used (LRU), first-in first-out (FIFO), and pseudo-LRU (PLRU) a cost-efficient variant of LRU. For an introduction to caches refer to [7]. A cache set can be formalized by:

- Its domain $\mathcal{Q}_{P_k}$, where the subscript denotes policy and associativity. E.g., $\mathcal{Q}_{\mathrm{FIFO}_k}$ is the set of all FIFO-controlled cache sets of associativity k.
- An update function $U_{P_k} : \mathcal{Q}_{P_k} \times \mathcal{B} \to \mathcal{Q}_{P_k}$, which computes the state of a cache set $q \in \mathcal{Q}_{P_k}$ after a memory block $b \in \mathcal{B}$ has been accessed.

Let $\mathcal{S} := \mathcal{B}^*$ be the set of finite access sequences to memory blocks, e.g. $s_1 := \langle a, b, a, c \rangle$. The update function U_{P_k} can be lifted from a single access to access sequences in the expected way.

Static Analyis

Our work is based on static analysis by abstract interpretation, which abstracts from the concrete program semantics and its respective concrete domain D. Instead, it represents more abstract information in an abstract domain A. The relation between D and A can be given by an abstraction function $\alpha_A : D \to A$ and a concretization function $\gamma_A : A \to D$. For safety properties, one often abstracts from a *collecting semantics*. In that case, D is a powerset domain.

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*, $U_A : A \times I \to A$, that describes how abstract values before and after instructions I are

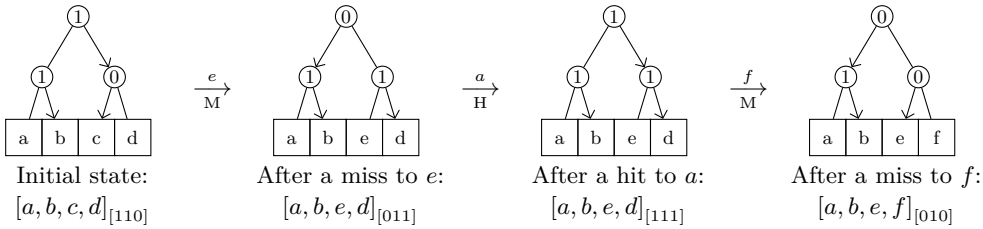


Figure 1 Updates of a PLRU cache set for the access sequence $\langle e, a, f \rangle$.

correlated. If an instruction has multiple predecessors, a *join function* $J_A : A \times A \to A$ combines all incoming values into a single one. For an introduction to abstract interpretation refer to [2].

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 domain is given by $Class := \{\mathrm{H}, \mathrm{M}\}^\top$.

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 might 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.

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 [3].

3 PLRU: Semantics and Analysis Challenges

Pseudo-LRU (PLRU) is a tree-based approximation of the LRU policy. It arranges the k cache lines at the leaves of a tree with $k-1$ "tree bits" pointing to the line to be replaced/filled next; a 0 indicating the left subtree, a 1 indicating the right. After every access, all tree bits on the path from the accessed line to the root are set to point away from the line. Other tree bits are left untouched.

There are at least two variants of PLRU that differ in their handling of invalid cache lines:

Sequential-fill If there are invalid lines upon a cache miss, the least of them (w.r.t. an ordering) is filled. Only if all lines are valid the tree-bits determine which line to replace.

Tree-fill Regardless of invalid lines, the line to be filled or replaced is always determined by the tree-bits.

In the following, we only consider the tree-fill variant. Examine Figure 1: In the initial state, the tree bits point to the line containing memory block c. We textually represent a PLRU-state by the contents of its cache lines and the pre-order traversal of its tree bits. The initial state in the example is thus written $[a, b, c, d]_{[110]}$. A miss to e evicts the memory block c which was pointed to by the tree bits. To protect e from eviction, all tree bits on the

path to the root of the tree are made to point away from it. Similarly, upon the following hit to a, the bits on the path from a to the root of the tree are made to point away from a. Note that they are not necessarily flipped. Another access to a would not change the tree bits at all as they already point away from a. Finally, a miss to f eliminates d from the cache set. So, one can represent PLRU cache sets as a k-tuple of memory blocks and $k-1$ bits:

$$q \in \mathcal{Q}_{\mathrm{PLRU}_k} := \mathcal{B}_{\perp}^{k} \times \mathbb{B}^{k-1} \tag{1}$$

Non-trivial logical cache states

Caches implemented in hardware have to satisfy several contradictory optimization goals. For instance, they should provide a low hit latency at a lower power consumption and implementation cost, i.e., area consumption. To satisfy these goals, a cache implementation cannot arbitrarily rearrange the contents of its cache lines upon every access to reflect an access's effect on its logical state. Instead, as in the implementation of PLRU described above, a small number of additional status bits is maintained and updated upon accesses. Due to these status bits, several physical states of the cache represent the same logical state. For static cache analyses it would be inconvenient and inefficient to distinguish such states, as they exhibit the same observable behavior in terms of hits and misses. *The first step in the design of a cache analysis should therefore be to abstract from physical cache states to logical cache states.*

For caches employing LRU or FIFO it is easy to abstract from the physical positions of memory blocks in cache sets. For LRU one can abstract from physical cache set positions by ordering the memory blocks from most-recently to least-recently used, i.e. by their age [3]. For FIFO one can abstract by ordering the blocks from last-in to first-in, i.e. according to their distance to the FIFO pointer [4]. In Section 4, we introduce a sound and complete abstraction from physical cache positions for PLRU, which is more involved due to its "non-linear" tree structure. This abstraction is a coarsest that is still complete: It distinguishes two concrete states if and only if there are access sequences that will result in a different hit/miss behavior.

Logarithmic-time eviction

Consider a cache set of size k: If LRU is employed, it takes at least k accesses to evict a block that has just been accessed [9]. If PLRU is employed, a block might already be evicted after only $\log_2(k)+1$ accesses [9]. Although this is usually not the case, it is challenging for a must-analysis to prove containedness of more than $\log_2(k)+1$ blocks. In Section 5, we introduce a must-analysis that keeps track of correlations between memory blocks in order to exclude spurious logarithmic-time eviction.

Arbitrary survival

May-analysis of PLRU is also more difficult than may-analysis of LRU or FIFO: as opposed to LRU and FIFO, a block b may still be cached after an arbitrary number of accesses to other memory blocks, even if arbitrarily many different blocks are accessed [1]. This makes it challenging for a may-analysis to prove eviction of blocks. The only may-analysis we currently see would be based on the "evict"-metric introduced in [9]. It would have to observe $e(k) = \frac{k}{2}\log_2(k)+1$ *successive* accesses to *pairwise different* blocks in order to be able to then predict a miss. Such an analysis would likely be of little or no practical use.

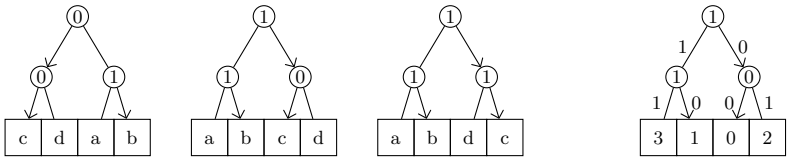


Figure 2 Three equivalent states and a state annotated with edge bits and logical cache positions.

4 Coarsest Complete Abstraction: from Physical to Logical Cache States

Ordering blocks from last-in to first-in in logical states of FIFO and from most- to least-recently used in logical states of LRU is similar. In both cases, blocks are ordered by decreasing miss replacement distance:

▶ **Definition 1** (Miss Replacement Distance). The *miss replacement distance*, $mrd(q, b)$, of a block b is the minimum number of successive misses that evict b from the cache set q. If $b \notin q$, $mrd(q, b) := 0$.

For PLRU, blocks can also be ordered by their miss replacement distance. However, this is more involved. Consider the cache set states in Figure 2. All these states are equivalent with respect to their replacement behavior: if one carries out an arbitrary but fixed access sequence on all states, the same blocks will be evicted in the same order. Given that only misses happen, the blocks will be evicted in the order c, b, d, a. The relation between the physical position of a block and its miss replacement distance is established in four steps:

1. For replacement it does not matter whether a block b is contained in a left or a right subtree. What matters is whether or not a tree bit points to the subtree containing b or not. Hence, we associate an *edge bit* with each edge in the tree. It is 0 if the corresponding tree bit points along this edge and 1 otherwise.
2. Subsequently, we associate an *access path* with each block b in a cache set q. An access path, $ap(q, b)$, is the sequence of edge bits encountered on the path from b to the root of q. If $b \notin q$, then $ap(q, b) = \bot$. In (all of) the above examples, the access path of d is 10 and the one of c is 00.
3. The *logical position* of a block in a cache set is its access path interpreted as a binary number. In the above examples, the logical position of d is 2 and the one of c is 0.
4. The miss replacement distance of a block is its logical position plus one.

For an example, see the rightmost PLRU tree in Figure 2: edges are annotated with edge bits and leaves are annotated with logical cache positions.

▶ **Observation 2** (Access Path Update). *Consider two cached blocks $a \neq b$ with access paths p_a and p_b. Let $p_a = pre_a \circ p_1 \circ suff$ and $p_b = pre_b \circ \overline{p_1} \circ suff$, where $|p_1| = 1$: the paths p_a and p_b start with different prefixes pre_a respectively pre_b, join after the last different bit p_1, and finish with the (possibly empty) shared suffix $suff$. After accessing b, all tree bits on the path of b point away from b, i.e. all edge bits are 1 and the new access path of b is $p'_b = 1 \ldots 1$. Since a and b share a suffix, setting the tree bits on the path to b also affects a's suffix: its new access path is $p'_a = pre_a \circ \underbrace{0}_{p_1} \circ \underbrace{1 \ldots 1}_{suff}$.*

▶ **Theorem 3** (Miss Replacement Distance [9]). *A block b with access path $ap(q, b) = p_1 \ldots p_n$ has miss replacement distance $mrd(q, b) = p_1 \ldots p_n + 1$.*

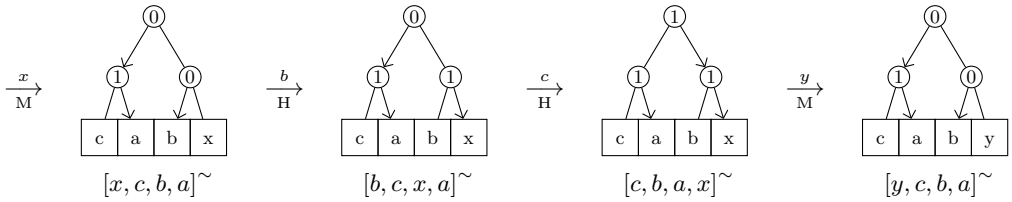


Figure 3 Eviction of x in $\log_2(4) + 1 = 3$ steps by the access sequence $\langle b, c, y \rangle$.

Essentially, the above proceeding defines equivalence classes on PLRU cache sets, i.e. the quotient structure $\mathcal{Q}_{\mathrm{PLRU}_k^{\sim}} := \mathcal{Q}_{\mathrm{PLRU}_k}/\!\sim$, where the equivalence relation is based on access paths and abstracts from the tree bits. $q_1 \sim q_2 \iff \forall b \in \mathcal{B} : ap(q_1, b) = ap(q_2, b)$. Hence, logical cache set states $\tilde{q} \in \mathcal{Q}_{\mathrm{PLRU}_k^{\sim}}$ can be represented as a function that maps blocks $b \in \mathcal{B}$ to their logical position $\tilde{q}(b)$:

$$\tilde{q} \in \mathcal{Q}_{\mathrm{PLRU}_k^{\sim}} = \mathcal{B} \to \{\bot, 0, \ldots, k-1\} \tag{2}$$

In an isomorphic representation as k-tuples of blocks, one can order blocks decreasingly by their logical position (miss replacement distance). For instance, $\tilde{q} = [a, d, b, c]^{\sim}$ represents the three equivalent states in Figure 2: $\tilde{q}(a) = 3, \tilde{q}(d) = 2, \tilde{q}(b) = 1, \tilde{q}(c) = 0$, and $\tilde{q}(x) = \bot$ for all other blocks x.

5 More Precise Must-Analysis Based on Subtree Distances

Logarithmic-time eviction

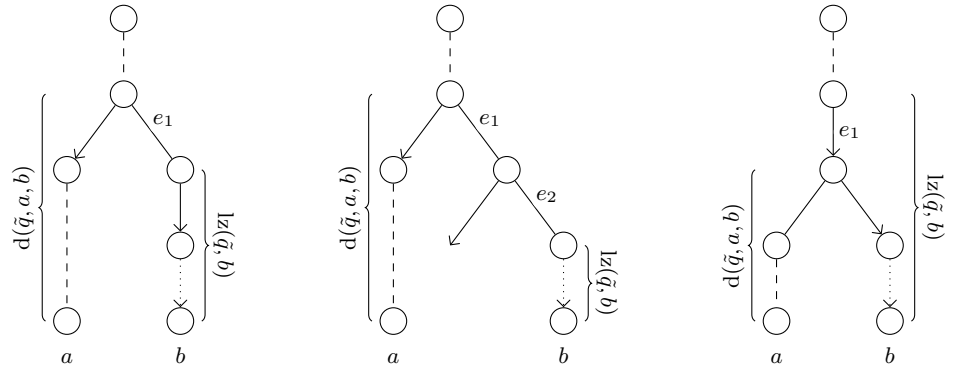
Consider the succession of states in Figure 3. After x has been inserted into the cache set, it only takes 3 accesses to evict it—although the associativity is 4.

Generalized to a k-way PLRU, $\log_2(k) + 1$ is a tight lower bound on the number of accesses that are necessary to evict a just inserted block [9]: After the access to a block x, its access path is $1 \ldots 1$. To replace x, all edge bits on its access path must be flipped to $0 \ldots 0$. By Observation 2, an access to another block a flips at most one of x's edge bits to 0. Also, the shared suffix of a and x is set to $1 \ldots 1$. Hence, to evict x with as few accesses as possible, one set the edge bits to 0 *from left to right* to avoid flipping bits back to 1.

For instance consider Figure 3: The access to b, which is the "direct neighbor" of x, sets the first edge bit of x to 0. The access to c, which is contained in a subtree "that is one step further away" than b, sets the second edge bit of x to 0. Note that accessing a instead of c has the same effect on x. If the cache set was 8-way associative, the third edge bit could be set to 0 by accessing one of the 4 blocks in the "next" subtree. Below we will formally define the notions in double quotes as *subtree distance*.

Sketch of the Analysis

As edge bits in an access path must be set to 0 from left to right to evict a block, the number of leading zeros in access paths is an interesting size. To predict hits, our must-analysis maintains an *upper bound on the number of leading zeros*: as long as this bound is less than $\log_2(k)$ for a block, that block can not be evicted. To improve analysis precision, we additionally maintain *approximations on subtree distances*. With information about subtree distances, the analysis can model the flipping of tree-bits more precisely and is able to exclude more spurious behavior, e.g. the logarithmic-time eviction of blocks.



(a) $\mathrm{d}(\tilde{q}, a, b) = \mathrm{lz}(\tilde{q}, b) + 1$, $e_1 = 1, e_1' = 0$, $\mathrm{lz}(\tilde{q}', b) = \mathrm{lz}(\tilde{q}, b) + 1$

(b) $\mathrm{d}(\tilde{q}, a, b) > \mathrm{lz}(\tilde{q}, b) + 1$, Although $e_1' = 0$: $e_2 = e_2' = 1$, $\mathrm{lz}(\tilde{q}', b) = \mathrm{lz}(\tilde{q}, b)$

(c) $\mathrm{d}(\tilde{q}, a, b) < \mathrm{lz}(\tilde{q}, b) + 1$, $e_1 = 0, e_1' = 1$, $\mathrm{lz}(\tilde{q}', b) = \mathrm{d}(\tilde{q}, a, b)$

Figure 4 Update of the number of leading zeros $\mathrm{lz}(\tilde{q}, b)$ upon a cache hit to block a.

Leading Zeros and Subtree Distance

▶ **Definition 4** (Number of Leading Zeros)**.** The *number of leading zeros* of a block, $\mathrm{lz}(b)$, is the number of leading zeros ($\mathrm{nlz} : \mathbb{N} \to \mathbb{N}$)[1] in the access path of that block ($\tilde{q}(b)$):

$$\mathrm{lz} : \mathcal{Q}_{\mathrm{PLRU}_k^{\sim}} \times \mathcal{B} \to \{\bot, 0, \ldots, \log_2(k)\} \tag{3}$$

$$\mathrm{lz}(\tilde{q}, b) := \begin{cases} \mathrm{nlz}(\tilde{q}(b)) & : \tilde{q}(b) \neq \bot \\ \bot & : \text{otherwise} \end{cases} \tag{4}$$

For example, in the state $\tilde{q} = [x, c, b, a]^{\sim}$ of Figure 3, $\mathrm{lz}(\tilde{q}, a) = 2, \mathrm{lz}(\tilde{q}, b) = 1, \mathrm{lz}(\tilde{q}, c) = \mathrm{lz}(\tilde{q}, x) = 0$, and $\mathrm{lz}(\tilde{q}, y) = \bot$ for all other blocks y.

▶ **Definition 5** (Subtree Distance)**.** The *subtree distance* between two cached blocks, $\mathrm{d}(\tilde{q}, a, b)$, is the distance to their least common ancestor in the tree.

$$\mathrm{d} : \mathcal{Q}_{\mathrm{PLRU}_k^{\sim}} \times \mathcal{B} \times \mathcal{B} \to \{\bot, 0, \ldots, \log_2(k)\} \tag{5}$$

$$\mathrm{d}(\tilde{q}, a, b) := \begin{cases} \log_2(k) - \mathrm{ntz}(\tilde{q}(a) \oplus \tilde{q}(b)) & : \tilde{q}(a) \neq \bot, \tilde{q}(b) \neq \bot \\ \bot & : \text{otherwise} \end{cases} \tag{6}$$

If both blocks are cached ($\tilde{q}(\cdot) \neq \bot$) the subtree distance between them is the height of the tree ($\log_2(k)$) minus the length of their shared suffix ($\mathrm{ntz}(\cdot)$). Otherwise, the distance is undefined ($\bot$). Assuming two's complement binary encoding, the length of the shared suffix can be computed by: First, a bitwise `xor` ($\oplus$) of the logical positions, which produces a 0 bit if two bits are equal. Then, the number of trailing zeros[1] in the result is the length of the shared suffix. For example, in the first tree of Figure 3, $\mathrm{d}(\tilde{q}, c, c) = 0, \mathrm{d}(\tilde{q}, c, a) = 1, \mathrm{d}(\tilde{q}, c, b) = \mathrm{d}(\tilde{q}, c, x) = 2$, and $\mathrm{d}(\tilde{q}, a, b) = \mathrm{d}(\tilde{q}, a, x) = 2$, and so on.

To see how the number of leading zeros is updated upon a cache hit to a block a, consider Figure 4. In the successor state $\tilde{q}'$ of $\tilde{q}$, the number of leading zeros after a hit to a ($\tilde{q}(a) \neq \bot$)

[1] $\mathrm{nlz}, \mathrm{ntz} : \mathbb{N} \to \mathbb{N}$ compute the number of leading/trailing zeros of two's-complement numbers. For definitions and efficient implementations see [10].

is

$$\mathrm{lz}(\tilde{q}', b) = \begin{cases} \bot & : \mathrm{lz}(\tilde{q}, b) = \bot \\ \mathrm{lz}(\tilde{q}, b) + 1 & : \mathrm{d}(\tilde{q}, b, a) = \mathrm{lz}(\tilde{q}, b) + 1 \quad \text{Figure 4a} \\ \min\{\mathrm{lz}(\tilde{q}, b), \mathrm{d}(\tilde{q}, b, a)\} & : \text{otherwise} \quad \text{Figures 4b, 4c} \end{cases} \tag{7}$$

Upon a cache miss to block a ($\tilde{q}(a) = \bot$) the logical position of each cached block is decremented ($\tilde{q}(b) - 1$), see Definition 1 and Theorem 3. Furthermore, the block at logical position 0 is evicted and trivially $\mathrm{lz}(\tilde{q}', a) = 0$:

$$\mathrm{lz}(\tilde{q}', b) = \begin{cases} 0 & : b = a \\ \mathrm{lz}(\tilde{q}(b) - 1) & : b \neq a, \tilde{q}(b) \neq \bot, \tilde{q}(b) > 0 \\ \bot & : \text{otherwise} \end{cases} \tag{8}$$

Abstraction

As explained above, the number of leading zeros is decisive for the question whether a block can be evicted. Hence, the first constituent of our abstract domain is the number of *potential leading zeros*:

$$plz \in PLZ_k := \mathcal{B} \to \{0, \ldots, \log_2(k), \top\} \tag{9}$$

As the prefix *potential* suggests, $plz(b)$ is an upper bound on the number of leading zeros in the access path of b. If $plz(b) = \top$, then b may not be cached (anymore). Formally, the meaning of $plz \in PLZ_k$ is given by the concretization function:

$$\gamma_{PLZ_k} : PLZ_k \to \mathcal{Q}_{\mathrm{PLRU}_k^{\sim}} \tag{10}$$

$$\gamma_{PLZ_k}(plz) := \{\tilde{q} \in \mathcal{Q}_{\mathrm{PLRU}_k^{\sim}} \mid plz(b) \neq \top \Rightarrow 0 \leq \mathrm{lz}(\tilde{q}, b) \leq plz(b)\} \tag{11}$$

For the analysis to be able to exclude the possibility of logarithmic-time eviction, it needs information about subtree distances. To see this, consider Equation 7, specifically that the second case depends on $\mathrm{d}(\tilde{q}, b, a)$: If the analysis had *no* information about the subtree distance $\mathrm{d}(\tilde{q}, b, a)$, it would have to conservatively take into account the case that $\mathrm{d}(\tilde{q}, b, a) = \mathrm{lz}(\tilde{q}, b) + 1$. This would mean that upon an access to block a, the upper bound $plz(b)$ would have to be incremented for all $b \neq a$. Ultimately, *an analysis that abstracts completely from subtree distances can never exclude logarithmic-time eviction.* Consequently, to increase analysis precision, the second constituent of our abstract domain maintains some information about subtree distances.

There are several ways to approximate subtree distances in an abstract domain. We chose an abstraction such that abstract elements can be represented efficiently. For each pair of blocks we distinguish between four classes of distances; zero, non-maximal, maximal, and unknown:

$$AD_k := \mathcal{B} \times \mathcal{B} \to \{\{0\}, [1, \log_2(k)), \{\log_2(k)\}, \top\} \tag{12}$$

Formally, the meaning of $ad \in AD_k$ is given by:

$$\gamma_{AD_k} : AD_k \to \mathcal{Q}_{\mathrm{PLRU}_k^{\sim}} \tag{13}$$

$$\gamma_{AD_k}(ad) := \{\tilde{q} \in \mathcal{Q}_{\mathrm{PLRU}_k^{\sim}} \mid ad(a, b) \neq \top \Rightarrow \mathrm{d}(\tilde{q}, a, b) \in ad(a, b)\} \tag{14}$$

Although the domain $\mathcal{B} \times \mathcal{B}$ of an $ad \in AD_k$ is of size $O(|\mathcal{B}|^2)$, each $ad \in AD_k$ can be stored in size $O(|\mathcal{B}|)$ by using a set of two disjoint sets of blocks $\{B_0, B_1\}$:

$$\begin{array}{lll} ad(a,b) = \{0\} & \iff & a = b, \exists i : a \in B_i \\ ad(a,b) = [1, \log_2(k)) & \iff & a \neq b, \exists i : a, b \in B_i \\ ad(a,b) = \{\log_2(k)\} & \iff & a \neq b, \exists i : a \in B_i, b \in B_{1-i} \\ ad(a,b) = \top & \iff & \{a,b\} \not\subseteq B_0 \cup B_1 \end{array} \tag{15}$$

Blocks with maximal distance are contained in different sets, blocks with non-maximal distance are contained in the same set. For instance $(\{a,b\},\{c\})$ corresponds to $ad(a,b) = [1,\log_2(k))$, $ad(a,c) = ad(b,c) = \{\log_2(k)\}$, and $ad(x,y) = \top$ for all other $x \neq y$.

Now that PLZ_k and AD_k are introduced, we define the abstract domain of the must-analysis, which is a partial function that associates bounds on leading zeros with approximations of subtree distances:

$$Plru_k^{AD} := AD_k \hookrightarrow PLZ_k \tag{16}$$

The set of concrete cache set states represented by $\hat{q} \in Plru_k^{AD}$ is determined by:

$$\gamma_{Plru_k}^{AD} : Plru_k^{AD} \to \mathcal{P}(\mathcal{Q}_{\mathrm{PLRU}_k^\sim}) \tag{17}$$

$$\gamma_{Plru_k}^{AD}(\hat{q}) := \bigcup_{ad \in dom(\hat{q})} \gamma_{AD_k}(ad) \cap \gamma_{PLZ_k}(\hat{q}(ad)) \tag{18}$$

A concrete state must satisfy any of the distance constraints $\gamma_{AD_k}(ad)$ and the associated constraints on leading zeros $\gamma_{PLZ_k}(\hat{q}(ad))$.

Classification

An access to a block b can be classified as a hit if its number of leading zeros is at most $\log_2(k)$ for all approximations of subtree distances. Otherwise, the access might be a miss.

$$C_{Plru_k}^{AD} : Plru_k^{AD} \times \mathcal{B} \to Class \tag{19}$$

$$C_{Plru_k}^{AD}(\hat{q}, b) := \begin{cases} \mathrm{H} & : \forall ad \in dom(\hat{q}) : \hat{q}(ad)(b) \neq \top \\ \top & : \text{otherwise} \end{cases} \tag{20}$$

Join

For coinciding approximations of subtree distances, the associated approximation of leading zeros is joined by taking the maximum bound for each block:

$$J_{Plru_k}^{AD}(\hat{q}_1, \hat{q}_2) := \lambda ad. \begin{cases} \hat{q}_1(ad) & : ad \in dom(\hat{q}_1) \setminus dom(\hat{q}_2) \\ \hat{q}_2(ad) & : ad \in dom(\hat{q}_2) \setminus dom(\hat{q}_1) \\ J_{PLZ_k}(\hat{q}_1(ad), \hat{q}_2(ad)) & : ad \in dom(\hat{q}_1) \cap dom(\hat{q}_2) \end{cases} \tag{21}$$

$$J_{PLZ_k}(plz_1, plz_2) := \begin{cases} \lambda b. \max\{plz_1(b), plz_2(b)\} & : plz_1(b) \neq \top, plz_2(b) \neq \top \\ \top & : \text{otherwise} \end{cases} \tag{22}$$

Update

The update of $Plru_k^{AD}$ is based on the updates of AD_k and PLZ_k. First, consider $U_{AD_k} : AD_k \times PLZ_k \times \mathcal{B} \times Class \to 2^{AD_k}$:

$$U_{AD_k}(\{B_0, B_1\}, plz, a, \mathrm{H}) := \{\{B_0', B_1\} \mid B_0' := B_0 \cup \{a\}, a \notin B_1, |B_0'| \leq k/2\} \tag{23}$$

$$U_{AD_k}(\{B_0, B_1\}, plz, a, \mathrm{M}) := \{\{B_0', B_1\} \mid B_0' := B_0 \backslash \{x\} \cup \{a\}, plz(x) \in \{\log_2(k), \top\}, |B_0'| \leq k/2\} \tag{24}$$

$$U_{AD_k}(ad, plz, a, \top) := U_{AD_k}(ad, plz, a, \mathrm{H}) \cup U_{AD_k}(ad, plz, a, \mathrm{M}) \tag{25}$$

Upon a hit to block a, a must be cached. Hence, we can make assumptions about its distances by adding it to a set B_i. To maintain consistency, a must not be contained in both sets simultaneously ($a \notin B_{1-i}$). Furthermore, at most $k/2$ blocks can have non-maximal subtree disctance to each other ($|B_i'| \leq k/2$). (Note that B_0 and B_1 are interchangeable since $\{B_0, B_1\}$ is a set.)

Upon a cache miss, the accessed block a inherits its subtree distances from the replaced block x. Due to the abstraction, several blocks might come into consideration for eviction, namely all blocks with $plz(x) \in \{\log_2(k), \top\}$.

If the access is unclassified, one has to take the union of the results of the hit- and miss-update.

The update of the potential leading zeros closely resembles the three cases in Figure 4:

$$U_{PLZ_k}(plz, ad, a) := \lambda b. \begin{cases} 0 & \text{if } a = b \\ \top & \text{else if } ad(a,b) = \top \\ plz(b) & \text{else if } plz(b) + 1 < L \\ plz(b) + 1 & \text{else if } L \leq plz(b) + 1 \leq U \\ U & \text{else if } plz(b) + 1 > U \end{cases} \tag{26}$$

Since the subtree distances are approximated, one has to rely on lower and upper bounds ($L \equiv \min\{n \in ad(a,b)\}, U \equiv \max\{n \in ad(a,b)\}$) of the interval $ad(a,b) \neq \top$. Consider the fourth case for instance: since $plz(b) + 1 = \mathrm{d}(\tilde{q}, a, b)$ *might* be possible, one has to increment $plz(b)$.

The update on $Plru_k^{AD}$ assigns each subtree distance approximation ad' an updated approximation of leading zeros. Different approximations ad might be updated to the same ad'. Hence, one must join ($\sqcup$) all updated approximations of leading zeros ($U_{PLZ_k}()$) of all ad for which $ad' \in U_{AD_k}(ad, \ldots)$.

$$U^{AD}_{Plru_k}(\hat{q}, a, cl) := \lambda ad'. \bigsqcup \{U_{PLZ_k}(\hat{q}(ad), ad', a) \mid ad \in dom(\hat{q}), ad' \in U_{AD_k}(ad, \hat{q}(ad), a, cl)\} \tag{27}$$

Uncertainty about accessed addresses

Cache analysis comprises *value analysis* and *replacement analysis*. Value analysis determines approximations to accessed addresses, which are the inputs to the cache. Given the accessed addresses, replacement analysis determines approximations to cache contents. Therefore a replacement analysis is generally applicable to instruction, data, and unified caches.

There are cases where the value analysis cannot precisely determine the address of a memory access. Nonetheless, the replacement analysis can *always* handle such uncertainty in a *sound* way: a sound successor state can be computed by performing updates of the

current state for all addresses that might be accessed and then joining all those states into a single one. However, this way, the uncertainty about accessed addresses translates into additional uncertainty about cache set states, which might translate into less classified accesses.

6 Closely Related Work

The related works most relevant for this paper are cache analyses of LRU [3], FIFO [4, 5] and PLRU [8]. [3] introduces the concepts of may- and must-caches and present may- and must-analyses for LRU that are based on abstract interpretation. [4, 5] introduce several must- and a may-analyses for FIFO and show how to combine the corresponding abstract domains in order to improve analysis precision. For pointers to earlier work on cache analysis directed at WCET analysis and other cache analyses, we kindly refer the reader to [4].

The only prior analysis of PLRU, $Plru_k^{RC}$, is a must-analysis based on relative competitiveness [8]. Under certain conditions, relative competitiveness allows one to use cache analyses for one policy as cache analyses for other policies. For instance, an LRU must-analysis for a $\log_2(k) + 1$-way associative cache can be employed as a must-analysis for a k-way PLRU.

7 Evaluation

In the following, we compare to each other
(a) the analysis $Plru_k^{AD}$ presented in this paper,
(b) the analysis $Plru_k^{RC}$ based on relative competitiveness [8] as explained in Section 6, and
(c) the collecting semantics $Plru_k^{CS}$ of PLRU.
The collecting semantics is the exact set of cache set states that may reach a program point. It delimits the precision of *any* static analysis. If a memory access cannot be classified as hit or miss in the collecting semantics, no sound static analysis can do so. We computed it using an analysis based on a powerset domain of symbolically-represented concrete cache-set states.

To quantify the precision of the analyses, we applied the analyses to two parametrizable classes of synthetic benchmarks, where the parameter n controls the level of temporal locality: *Loop(n)* is a loop that iterates 16 times and accesses n different blocks, i.e. $(1\ 2 \ldots n)^{16}$. *Rand(n)* is a set of 100 sequences, each containing 100 randomly distributed accesses to n different blocks, i.e. $(1|2|\ldots|n)^{100}$.

The results for associativities $k = 4$ and $k = 8$ are shown in Table 1 (for $k = 2$, PLRU is identical to LRU). Except for one negligible exception, $Plru_k^{AD}$ can guarantee higher hit

Table 1 Guaranteed hit rates [%] provided by the two analyses and the collecting semantics.

		Associativity $k = 4$				Associativity $k = 8$						
	n	2	3	4	5	2	3	4	5	6	7	8
Loop	$Plru_k^{RC}$	93.8	93.8	0.0	0.0	93.8	93.8	93.8	0.0	0.0	0.0	0.0
	$Plru_k^{AD}$	93.8	93.8	92.2	0.0	93.8	93.8	93.8	92.5	90.6	0.0	0.0
	$Plru_k^{CS}$	93.8	93.8	92.2	0.0	93.8	93.8	93.8	92.5	91.7	90.2	86.7
Rand	$Plru_k^{RC}$	98.0	97.0	73.1	58.8	98.0	97.0	96.0	77.7	64.4	55.7	48.1
	$Plru_k^{AD}$	98.0	97.0	94.7	75.4	98.0	97.0	95.8	93.0	84.3	63.5	52.0
	$Plru_k^{CS}$	98.0	97.0	94.7	75.4	98.0	97.0	96.0	93.9	91.0	84.0	68.4

rates than $Plru_k^{RC}$.

For the *Loop* benchmarks, $Plru_k^{RC}$ cannot classify any hits if $n > \log_2(k) + 1$, wheras $Plru_k^{AD}$ can do so for up to $n = 2\log_2(k)$. Hence, $Plru_k^{AD}$ is preferable for loops containing more than $\log_2(k) + 1$ different accesses. If the analyses predict hits, the amount is close to the limit given by $Plru_k^{CS}$.

The *Rand* benchmarks are stress tests for the abstract domains. For smaller n, both analyses perfom equally well. For $n > \log_2(k) + 1$, the gap between the collecting semantics and the analysis results grows. With increasing n, $Plru_k^{RC}$ falls behind $Plru_k^{AD}$.

Regarding the efficiency of the analyses, please note that $Plru_k^{AD}$ and $Plru_k^{CS}$ are implemented as prototypes whereas $Plru_k^{RC}$ is tuned. For each analysis we measured the overall time needed to complete all benchmarks. For $k = 4$, all analyses took around 3.3s. For $k = 8$, $Plru_k^{RC}$ completed after 3.5s, $Plru_k^{AD}$ after 5.5s, and $Plru_k^{CS}$ after 12.5s. Compared to $Plru_k^{RC}$, the disadvantage of $Plru_k^{AD}$ is that it is a disjunctive domain, which entails higher memory consumption and lower performance.

8 Conclusions and Further Work

Our first contribution, the $Plru_k^{AD}$ analysis, has pros and cons: It is more precise than its sole competitor $Plru_k^{RC}$. Most importantly, it can classify hits in "larger" loops, where $Plru_k^{RC}$ cannot. On the other hand, its higher memory consumption might hamper scalability. However, tradeoffs are possible by changing the approximation of subtree distances, i.e. plugging-in different domains AD_k.

Our second contribution, the understanding of the *subtree distance* and its relation to other sizes, is perhaps more valuable than the analysis itself: For FIFO, it was shown to be beneficial for precision to refine the abstract transformer by discriminating between hits and misses [4]. For PLRU, this is not sufficient as a hit can both, accelerate or defer the eviction of other blocks. Instead, we consider the subtree distance as an important size in the design of future PLRU (may-)analyses, which will possibly degrade $Plru_k^{AD}$ to a proof-of-concept analysis.

References

1 Christoph Berg. PLRU cache domino effects. In *WCET*, 2006.

2 Patrick Cousot and Radhia Cousot. *Building the Information Society*, chapter Basic Concepts of Abstract Interpretation, pages 359–366. Kluwer Academic Publishers, 2004.

3 Christian Ferdinand. *Cache Behaviour Prediction for Real-Time Systems.* PhD thesis, Saarland University, 1997.

4 Daniel Grund and Jan Reineke. Abstract interpretation of FIFO replacement. In Jens Palsberg and Zhendong Su, editors, *Static Analysis, 16th International Symposium, SAS 2009*, volume 5673 of *LNCS*, pages 120–136. Springer-Verlag, August 2009.

5 Daniel Grund and Jan Reineke. Precise and efficient FIFO-replacement analysis based on static phase detection. In *ECRTS*, 2010.

6 Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003.

7 John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 3rd Edition.* Morgan Kaufmann, 2003.

8 Jan Reineke and Daniel Grund. 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, pages 51–60, New York, NY, USA, 2008. ACM Press.

9 Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.

10 Henry S. Warren. *Hacker's Delight*. Addison-Wesley, 2003.

11 Randall T. White, Christopher A. Healy, David B. Whalley, Frank Mueller, and Marion G. Harmon. Timing analysis for data caches and set-associative caches. In *RTAS '97: Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*, page 192, Washington, DC, USA, 1997. IEEE Computer Society.

12 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *Transactions on Embedded Computing Systems*, 7(3):1–53, 2008.