On the complexity of cache analysis for different replacement policies

David Monniaux, Valentin Touzeau

▶ **To cite this version:**

David Monniaux, Valentin Touzeau. On the complexity of cache analysis for different replacement policies. Journal of the ACM (JACM), 2019, 66 (6), pp.41. 10.1145/3366018 . hal-01910216v3

HAL Id: hal-01910216
https://hal.science/hal-01910216v3

Submitted on 18 Nov 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

On the complexity of cache analysis for different replacement policies

David Monniaux and Valentin Touzeau

Univ. Grenoble Alpes, CNRS, Grenoble INP[*], VERIMAG, 38000 Grenoble, France

November 18, 2019

Abstract

Modern processors use cache memory: a memory access that "hits" the cache returns early, while a "miss" takes more time. Given a memory access in a program, cache analysis consists in deciding whether this access is always a hit, always a miss, or is a hit or a miss depending on execution. Such an analysis is of high importance for bounding the worst-case execution time of safety-critical real-time programs.

There exist multiple possible policies for evicting old data from the cache when new data are brought in, and different policies, though apparently similar in goals and performance, may be very different from the analysis point of view. In this paper, we explore these differences from a complexity-theoretical point of view. Specifically, we show that, among the common replacement policies, LRU (Least Recently Used) is the only one whose analysis is NP-complete, whereas the analysis problems for the other policies are PSPACE-complete.

1 Introduction

Most high performance processors implement some form of *caching*: frequently used instructions and data are retained in fast memory close to the processing unit, to avoid costly requests from the main memory. While the intuition is that a cache retains the most recently accessed memory words, up to its size, reality is far more complex: what happens depends on the number of cache levels, the size of each level, the "number of ways" (also known as the *associativity*) of the cache and the *cache replacement policy*, that is, the algorithm used for choosing which memory block to evict from the cache to make room for a new block.

Reading a memory block not in cache can take 10 times to 100 times the time needed to access it if it is cache. Thus, static analysis approaches for

[*]Institute of Engineering Univ. Grenoble Alpes

bounding the worst-case execution time (WCET) of programs have to take into account whether or not data are cached. Such analyses are used, for instance, for proving that the execution time of software in a critical control loop (e.g. in avionics) can never exceed the period of the loop. Not only does caching, or lack of caching, directly influence execution time, it also complicates analysis itself, as different microarchitectural execution paths may be taken inside the processor depending on whether or not data are cached, and analysis has to take all these paths into account.

For these reasons, all static analyses for bounding execution time include a cache analysis, which determines which of the memory accesses made by the program are hits, which are misses, and which cannot be classified. Analyses depend on the cache replacement policy, and, in the literature, there is a clear preference for the LRU (*Least Recently Used*) policy, from the well-known age-based abstract analysis of Ferdinand [5] to recent work proved to be optimally precise in a certain sense [14, 15].

In contrast, other policies such as PLRU (pseudo-LRU), NMRU and FIFO (*First-In, First-Out*) have a reputation for being very hard to analyze [7] and for having poor predictability [13]. A legitimate question is whether these problems are intrinsically difficult, or is it just that research has not so far yielded efficient analyses.

Issues of static analysis of programs under different cache policies are not necessarily correlated with the practical efficiency of cache policies. Static analysis is concerned with worst-case behavior, and policies with approximately equal "average"[1] practical performance may be very different from the analysis point of view. Even though PLRU and NMRU were designed as "cheap" (easier to implement in hardware) alternatives to LRU and have comparable practical efficiency [1], they are very different from the worst-case analysis point of view.

In this paper, we explore these questions as decision problems:

Definition 1 (Exist-Hit)**.** The *exist-hit* problem is, for a given replacement policy:

Inputs	a control flow graph $G = (V, E)$ with edges adorned with memory block names
	a starting node S in the graph
	a final node F in the graph
	the cache *associativity* (*number of ways* N), in unary
	a memory block name a
Outputs	a Boolean: is there an execution trace from S to F, starting with an empty initial cache and ending with a cache containing a?

Definition 2 (Exist-Miss)**.** The *exist-miss* problem is defined as above but with an ending state *not* containing a.

[1] By "average" we do not imply any probabilistic distribution, but rather an informal meaning over industrially relevant workloads, as opposed to examples concocted for exhibiting very good or very bad behavior.

We shall also study the variant of this problem where the initial cache contents are arbitrary:

Definition 3. The exist-hit (respectively, exist-miss) problem with arbitrary initial state contents is defined as above, except that the output is "are there a legal initial cache state σ and an execution trace from S to F, starting in σ and ending with a cache containing (respectively, not containing) a?".

We shall here prove that

- for policies LRU, FIFO, pseudo-RR, PLRU, and NMRU, the exist-hit and exist-miss problems are NP-complete for acyclic control flow graphs;
- for LRU, these problems are still NP-complete for cyclic control flow graphs;
- for PLRU, FIFO, pseudo-RR, PLRU, and NMRU, these problems are PSPACE-complete for cyclic control flow graphs
- for LRU, FIFO, pseudo-RR, and PLRU, the above results extend to exist-miss and exist-hit problems from an arbitrary starting state

Under the usual conjecture that PSPACE-complete problems are not in NP, this may justify why analyzing properties of FIFO, PLRU and NMRU caches is harder than for LRU.

Real-life CPU cache systems are generally complex (multiple levels of caches) and poorly documented (often, the only information about replacement policies is by reverse engineering). For our complexity-theoretical analyses we need simple models with clear mathematical definitions; thus we consider only one level of cache, and only one "cache set" per cache.[2]

In this paper, we consider that the control-flow graph carries only identifiers of memory blocks to be accessed, abstracting away the data that are read or written, as well as arithmetic operations and guards. Therefore, we take into account executions that cannot take place on the real system. This is the same setting used by many static analyses for cache properties. Some more precise static analyses attempt to discard some infeasible executions — e.g. an execution with guards $x < 0$ and $x > 0$ with no intervening write to x is infeasible. In general, however, this entails deciding the reachability of program locations, a problem that is undecidable if the program operates over unbounded integers, and already PSPACE-complete if the program operates on a finite vector of bits [4];. Clearly we cannot use such a setting to isolate the contribution of the cache analysis itself.

[2]A real cache system is composed of a large number of "cache sets": a memory block may fit in only one cache set depending on its address, and the replacement policy applies only within a given cache set. For all commonly found cache replacement policies except pseudo-round-robin, and disregarding complex CPU pipelines, this means that the cache sets operate completely independently, each seeing only memory blocks that map to it; each can be analyzed independently. It is therefore very natural to consider the complexity of analysis over one single cache set, as we do in this paper.

2 Fixed associativity

In a given hardware cache, the associativity is fixed, typically $N = 2$, 4, 8, 12 or 16. It thus makes sense to study cache analysis complexity for fixed associativity. However, such analysis can always be done by explicit-state model-checking (enumeration of reachable states) in polynomial time:

Theorem 4. *Let us assume here that the associativity N is fixed, as well as the replacement policy (among those cited in this article). Then exist-hit and exist-miss properties can be checked in polynomial time, more precisely in $O(|G|^{N+1})$ where $|G|$ is the size of the control-flow graph.*

Proof. Let (V, E) be the control-flow graph; its size is $|G| = |V| + |E|$. Let B the set of possible cache blocks. Without loss of generality, for all policies discussed in this article, the only blocks that matter in B are those that are initially in the cache (at most N) and those that are found on the control edges. Let us call the set of those blocks B'; $|B'| \leq |E| + N$.

The state of the cache then consists in N blocks chosen among $|B'|$ possible ones, plus possibly some additional information that depends on the replacement policy (e.g. the indication that a line is empty); say b bits per way. The number of possible cache states is thus $(2^b|B'|)^N$.

Let us now consider the finite automaton whose states are pairs (p, σ) where p is a node in the control-flow graph and σ is the cache state, with the transition relation $(p, \sigma) \to (p', \sigma')$ meaning that the processor moves in one step from control node p with cache state σ to control node p' with cache state σ'. The number of states of this automaton is $|V|.(2^b|B'|)^N$, which is bounded by $|G|.(|G| + N)^N.2^{bN}$, that is, $O(|G|^{N+1})$.

Exist-miss and exist-hit properties amount to checking that certain states are reachable in this automaton. This can be achieved by enumerating all reachable states of the automaton, which can be done in linear time in the size of the automaton. □

It is an open question whether it is possible to find algorithms that are provably substantially better in the worst-case than this brute-force enumeration. Also, would it be possible to separate replacement policies according to their growth with respect to associativity? It is however unlikely that strong results of the kind "PLRU analysis needs at least $K.|G|^N$ operations in the worst case" will appear soon, because they imply P $\neq$ NP or P $\neq$ PSPACE.

Theorem 5. *Consider a policy among PLRU, FIFO, pseudo-RR (with known or unknown initial state) or NMRU with known initial state (respectively, LRU), and a problem among exist-miss and exist-hit. Assume (H): for this policy, for any algorithm A that decides this problem on this policy, and any associativity N, there exist $K(N)$ and $e(N)$ such that for all g_0 there exists $g(N, g_0) \geq g_0$ such that the worst-case complexity of A on graphs of size g is at least $K(N).g(N, g_0)^{e(N)}$. Assume also $e(N) \to \infty$ as $N \to \infty$, then P is strictly included in PSPACE (respectively, NP).*

Proof. Suppose (H'): there exists a polynomial-time algorithm A solving the analysis problem for arbitrary associativity, meaning that there exist a constant K' and an exponent e' such that A' takes time at most $K'.(N+g)^{e'}$ on a graph of size g for associativity N.

Let N be an associativity. From (H) there is a strictly ascending sequence g_m such that the worst-case complexity of A on graphs of size g_m is at least $K(N).g_m^{e(N)}$. From (H'), $K(N).g_m^{e(N)} \leq K'.(N+g_m)^{e'}$. When $g_m \to \infty$ this is possible only if $e(N) \leq e'$.

Since $e(N) \to \infty$ as $N \to \infty$, the above is absurd. Thus there is no polynomial-time algorithm A for solving the analysis problem for the given policy. We prove later in this paper that these analysis problems are PSPACE-complete for PLRU, FIFO, NMRU, pseudo-RR, and NP-complete for LRU; the result follows. □

3 LRU

The "Least Recently Used" (LRU) replacement policy is simple and intuitive: the data block least recently used is evicted when a cache miss occurs. The cache is thus a queue ordered by age: on a miss, the oldest block is discarded to make room for a new one, which has age 0; the ages of all other blocks are incremented. If a block is already in the cache, it is "rejuvenated": its age is set to zero, and the ages of the blocks before it in the queue are incremented.

In other words, the state of an LRU cache with associativity N is a word of length at most N over the alphabet of cache blocks, composed of pairwise distinct letters; an empty cache is defined by the empty word. When an access is made to a block a, if it belongs to the word (*hit*), then this letter is removed from the word and appended to the word. If it does not belong to the word (*miss*), and the length of the word is less than N, then a is appended to the word; otherwise, the length of the word is exactly N — the first letter of the word is discarded and a is appended.

LRU has been used in Intel Pentium I, MIPS 24K/34K [12, p.21], among others. Notably, for Kalray processors K1a and K1b, LRU caches are advertised as advancing "timing predictability".

In this section, we shall extend our recent NP-hardness results [15] to NP-completeness, and also prove NP-hardness for exist-hit on a restricted class of control-flow graphs.

3.1 Motivation and fundamental properties

LRU caches are appreciated by designers of static analysis tools that bound the worst-case execution time of the program, since an analysis based on abstract interpretation by Ferdinand and Wilhelm [5] (basically, an interval for the age of each possible block) has long been known. The analysis classifies each access in the program as "always hit" (all execution traces leading to that access produce a hit there), "always miss" (all execution traces leading to that access produce

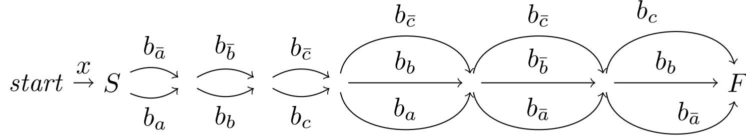


Figure 1: There is a path from q_0 to F with at most 3 different labels if and only if the formula $(\bar{c} \vee b \vee a) \wedge (\bar{c} \vee \bar{b} \vee \bar{a}) \wedge (c \vee b \vee \bar{a})$ has a model. Thus, for a LRU cache with associativity $N = 4$, there is an execution from S to F ending in a cache state containing x if and only if this formula has a model.

a miss there), or "unknown". When it answers "unknown", it may be that it is in fact "always hit" or "always miss", but the analysis is too weak to come to a conclusion about it, or that there is at least one execution leading to a hit there and one leading to a miss there ("definitely unknown").

In recent work [14, 15], we closed that loophole and proposed an analysis that completely decides whether a given access is "always hit", "always miss", or "definitely unknown": the classical abstract interpretation is applied, along with another age-based abstract interpretation capable of concluding, in some cases, that an access is "definitely unknown"; the remaining cases are decided by an exact but expensive (exponential worst-case) analysis. These analyses solve both the exist-hit and the exist-miss problems; was such an exponential cost unavoidable? This motivated the studies in this paper.

Our analyses, as well as all our results on LRU in this paper, are based on the following easy, but fundamental, property of LRU caches:

Proposition 6. *After an execution path starting from an empty cache, a block a is in the cache if and only if there has been at least one access to a along that path and the number of distinct blocks accessed since the last access to a is at most* $N - 1$.

Example 7. Assume a 4-way cache, initially empty. After the sequence of accesses *bcabdcdb*, a is in the cache because *bdcdb* contains only 3 distinct blocks b, c, d. In contrast, after the sequence *bcabdceb*, a is no longer in the cache because *bdceb* contains 4 distinct blocks b, c, d, e.

Remark 8. In definitions 1 and 2, it does not matter if the associativity is specified in unary or binary. An associativity larger than the number of different blocks always produces hits, thus the problems become trivial. This also applies to FIFO, PLRU, NMRU caches and, more generally, to any cache analysis problem starting from an empty cache with a replacement policy that never evicts cache blocks as long as there is a free cache line.

3.2 Exist-Hit

Theorem 9. *The exist-hit problem is NP-complete for LRU and acyclic control-flow graphs.*

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

Now consider the following reduction from CNF-SAT (see Figure 1 for an example). Let n_V be the number of variables in the SAT problem. With each variable v in the SAT problem we associate two cache block labels b_v and $b_{\bar{v}}$. The idea is to represent a variable assignment as a cache state. We store b_v in the cache when v is set to true, and $b_{\bar{v}}$ when v is set to false. Let x be a fresh name. We first load x into the cache. The control-flow graph is then a sequence of switches:

- For each variable v in the SAT problem, a switch between two edges labeled with b_v and $b_{\bar{v}}$ respectively. Executing this sequence of switches then loads into the cache a set of blocks representing a variable assignment. In addition, the next block that will be evicted (if any) is x.
- For each clause in the SAT problem, a switch between edges labeled with the blocks associated to the literals present in the clause. If at least one of these blocks has been accessed before, then the corresponding access can be performed without evicting x. Otherwise, x is guaranteed to be evicted.

Each path through the sequence of switches with at most n_V different labels corresponds to a SAT valid assignment, and conversely. Then there exists an execution such that at F the cache contains x if and only if there exists a SAT valid assignment. □

The objection can be made that the reduction in this proof produces control-flow graphs in which the same label occurs an arbitrary number of times — the number of times the corresponding literal occurs in the CNF-SAT problem, plus one. This is realistic for a data cache, since in a given program the same cache block may be accessed an arbitrary number of times. It is however unrealistic for an instruction cache:[3] an instruction cache block has a fixed size, contains a maximum number of instructions, and thus cannot be accessed from an arbitrary number of control edges.[4] However, we can refine the preceding result to account for this criticism.

Theorem 10. *The exist-hit problem is NP-complete for LRU for acyclic control-flow graphs, even when the same cache block is accessed no more than thrice.*

Proof. We use the same reduction as in Th. 10, but from a CNF-SAT problem where each literal occurs at most twice, as per the following lemma. □

[3]Unless procedure calls are "inlined" in the graph, because then the cache blocks corresponding to the inline procedures appear as many times as the number of locations it is called from.

[4]Consider a cache with 64-byte cache lines, as typical in x86 processors. In order for several basic blocks of instructions to overlap with that cache line, each, except perhaps the last one, must end with a branch instruction, which, in the shortest case, takes 2 bytes. No more than 32 basic blocks can overlap this cache line, and this upper bound is achieved by highly unrealistic programs.

Lemma 11. *CNF-SAT is NP-hard even when restricted to sets of clauses where the same literal occurs at most twice, the same variable exactly thrice.*[5]

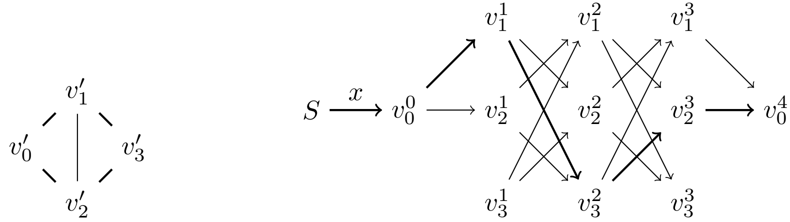
Proof. In the set of clauses, rename each occurrence of the same variable v_i as a different variable name $v_{i,j}$, then add clauses $v_{i,1} \Rightarrow v_{i,2}$, $v_{i,2} \Rightarrow v_{i,3}$, …, $v_{i,n-1} \Rightarrow v_{i,n}$, $v_{i,n} \Rightarrow v_{i,1}$ to establish logical equivalence between all renamings. Each literal now occurs once or twice, each variable thrice. Each model of the original formula corresponds to a model of the renamed formula, and conversely. □

Remark 12. The exist-hit problem is easy when the same cache block is accessed only once in the graph. Assume that the aim is to test whether there exists an execution leading to a cache containing x at the final node F. Either there exists one reachable access R to x in the control-flow graph, or there is none (in the latter case, x cannot be in the cache at node F) . Then there exists an execution leading to a cache state containing x at node F if and only if there exists a path of length at most $N - 1$ between R and F (see Proposition 6), which may be tested for instance by breadth-first traversal. The complexity question remains opens when cache blocks are accessed at most twice.

Theorem 13. *The exist-hit problem is still in NP for LRU when the graph may be cyclic.*

Proof. To prove that the problem is still in NP in case of cyclic graph, we show that the non-deterministic search of a exist-hit witness can be restricted to "short" paths (i.e. path of polynomial size). Consider a path π from the starting node S to the final node F such that the final cache content contains x (i.e. π is a witness for the exist-hit problem). The idea of the proof is to remove accesses from π to build a new witness π' which can be found in polynomial time.

The initial cache state being empty, there must be at least one access to x in π. Let i be the index of the last access to x: the edge (π_i, π_{i+1}) is labeled with block x, and for any $j > i$, (π_j, π_{j+1}) does not access x. We now split π at index i into two paths π^1 and π^2: $\pi^1 = \pi_1 \dots \pi_i$ and $\pi^2 = \pi_{i+1} \dots \pi_{|\pi|}$. $\pi^{1\prime}$ and $\pi^{2\prime}$ denote the paths obtained from π^1 and π^2 by removing cycles (subpaths beginning and ending in the same node). By construction, $\pi^{2\prime}$ does not evict x: the set of distinct memory blocks accessed along $\pi^{2\prime}$ is included in the set of distinct memory blocks accessed along π^2, which is insufficient to evict x. Thus, x is guaranteed to be cached at the end of $\pi^{2\prime}$ (due to Property 6). Because $\pi^{1\prime}$ and $\pi^{2\prime}$ are cycle free, we have $|\pi^{1\prime}| \leq |V|$ and $|\pi^{2\prime}| \leq |V|$. The path $\pi' = \pi^{1\prime}\pi^{2\prime}$ obtained as the concatenation of $\pi^{1\prime}$ and $\pi^{2\prime}$ has thus length at most $2|V|$. In addition π' goes from S to F and leads to a cache state containing x. Thus, the nondeterministic search for a witness hit path may be restricted to paths of length at most $2|V|$, which ensures membership in NP. □



(a) Graph with (thick) Hamiltonian cycle

(b) Acyclic control-flow graph obtained by the reduction. Edge labels are not shown; the path corresponding to the Hamiltonian cycle is shown in a thick line.

Figure 2: Reduction from Theorem 14 from the Hamiltonian cycle problem to the exist-miss problem for LRU caches.

3.3 Exist-Miss

Theorem 14. *The exist-miss problem is NP-complete for LRU for acyclic control-flow graphs.*

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

We reduce the Hamiltonian circuit problem to the exist-miss problem (see Figure 2 for an example). Let $G' = (V', E')$ be a graph, let $n = |V'|$, $V' = \{v'_0, \dots, v'_{n-1}\}$ (the ordering is arbitrary). We check the existence of Hamiltonian circuit in G'. Let us construct an acyclic control-flow graph G suitable for cache analysis as follows:

- two copies v_0^0 and v_0^n of v'_0
- for each v'_i, $i \geq 1$, $|V'| - 1$ copies v_i^j, $1 \leq j < n$ (this arranges these nodes in layers indexed by j)
- for each pair v_i^j, $v_{i'}^{j+1}$ of nodes in consecutive layers, an edge, labeled by the address i', if and only if there is an edge (i, i') in E'.

There is a Hamiltonian circuit in G' if and only if there is a path in G from v_0^0 to v_0^n such that no edge label is repeated, thus if and only if there exists a path from v_0^0 to v_0^n with at least n distinct edge labels. Prepend an edge accessing a fresh block x from the start node to v_0^0, then there exists a trace such that x is not in the cache at v_0^n if and only if this Hamiltonian circuit exists. □

The proof of Theorem 13 (exist-hit problem is still in NP for cyclic CFG) does not carry over to the exist-miss case. Indeed, this proof shows that if there is a path leading to a hit, then there is a "short" path that also lead to hit and can be discovered in polynomial time. This "short" path might contain

[5]We thank Pálvölgyi Dömötör for pointing out to us that this restriction is still NP-hard.

fewer blocks between the last access to x and the final node F than the original witness. Due to the fundamental property of LRU (see Property 6) this is not a problem and the "short" path is still guaranteed to lead to a hit. However, in the case of the exist-miss problem, the short witness must be built carefully, as cutting cycles might remove accesses needed to evict x. To show that the exist-miss problem for LRU is still in NP for cyclic control-flow graphs, one must exhibits short paths with the same blocks accessed (after x) than the original witness. The following lemma ensures that this is always possible.

Lemma 15. *Let B the set of memory blocks and $G = (V, E)$ a control-flow graph with edges decorated with elements of B on edges. From any node v_1 and v_2 in V, and any path from $v_1 \in V$ to $v_2 \in V$ we can extract a path from v_1 to v_2 with the same contents (same memory blocks accessed) and length at most $|V| \cdot |B|$ (and thus at most $|V| \cdot |E|$).*

Proof. Consider a path π from v_1 to v_2. π can be segmented into sub-paths $\pi_1, \ldots, \pi_m$, each beginning with the first occurrence of a new label not present in previous sub-paths.

Each sub-path π_i consists of an initial edge e_i followed by π'_i. From π'_i one can extract a simple path π''_i — that is, π''_i has no repeated node — of length at most $|V| - 1$. By definition, there are no new edge label between e_i and e_{i+1}. Thus removing cycles from π'_i does not change the set of blocks accessed. Then, the concatenated path $e_1\pi''_1 \cdots e_m\pi''_m$ has the same contents as π, starts and ends with the same nodes, and has at most $|V| \cdot |B|$ edges. □

Theorem 16. *The exist-miss problem is still in NP for cyclic control-flow graphs.*

Proof. Given a witness of miss existence π, one can split π into π^1 and π^2 at the last access to x (if any). If π does not contain any access to x, then simply consider $\pi^1 = \varepsilon$ and $\pi^2 = \pi$.

From the preceding lemma, one can extract from π^1 and π^2 two paths $\pi^{1\prime}$ and $\pi^{2\prime}$ of length at most $|V| \cdot |B|$ accessing the same sets of blocks. There are thus enough blocks accessed along $\pi^{2\prime}$ to evict x, and the path π' obtained by chaining $\pi^{1\prime}$ and $\pi^{2\prime}$ is guaranteed to lead to a miss. One can thus search for a witness path of length at most $2 \cdot |V| \cdot |E|$ to check the existence of a miss.. □

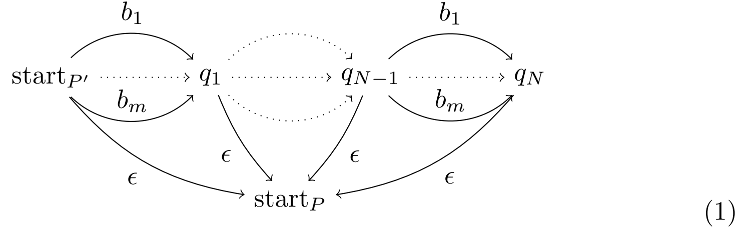
3.4 Extensions

Theorem 17. *The above theorems hold even if the starting cache state is unspecified: a problem with arbitrary starting cache state can be reduced to a problem of the same kind with empty starting cache state, and vice-versa.*

Proof. Consider a problem P with empty initial cache state. Prepend to the control-flow graph of P a sequence $f_1 \ldots f_N$ accesses to N pairwise distinct accesses to fresh blocks (blocks not appearing in P), where N is the associativity, and call the resulting problem P'. After executing this sequence, the cache only contains blocks from $f_1 \ldots f_N$ (not necessarily in that order) and none of the

blocks of P. It is thus equivalent to check exist-hit or exist-miss properties on P' from an arbitrary initial state and on P from an empty initial state.

Consider a problem P with arbitrary initial cache state. Prepend to the control-flow graph of P the following gadget, where ϵ denotes ϵ-transitions (no memory access) and $b_1, \ldots, b_m$ denote the alphabet of memory blocks:



(1)

This gadget loads into the cache any combination from zero to N blocks from $b_1, \ldots, b_m$, in all possible orders. Thus analyzing P' with a empty initial cache state is equivalent to analyzing P with an arbitrary initial cache state. □

Remark 18. The proofs of NP-hardness for exist-hit and exist-miss on acyclic graphs for LRU carry over to FIFO (section 5).

4 Boolean register machine

In the next sections, we shall prove that the exist-hit and the exist-miss problems for a variety of replacement policies are NP-hard for acyclic control-flow graphs and PSPACE-hard for general control-flow graphs. All proofs will be by reduction from the reachability problem on a class of very simple machines, which we describe in this section: this problem is NP-complete if the control-flow graph of the machine is assumed to be acyclic, and PSPACE-complete in general.

Definition 19. A *Boolean register machine* is defined by a number r of registers and a directed (multi)graph with an initial node and a final node, with edges adorned by instructions of the form:

Guard $v_i = b$ where $1 \leq i \leq r$ and $b \in \{\mathbf{f}, \mathbf{t}\}$,

Assignment $v_i := b$ where $1 \leq i \leq r$ and $b \in \{\mathbf{f}, \mathbf{t}\}$.

The *register state* is a vector of r Booleans. An edge with a guard $v_i = b$ may be taken only if the i-th register contains b; the register state is unchanged. The register state after the execution of an edge with an assignment $v_i := b$ is the same as the preceding register state except that the i-th register now contains b.

The *reachability problem* for such a system is the existence of a valid execution starting in the initial node with all registers equal to $\mathbf{f}$, and leading to the final node.

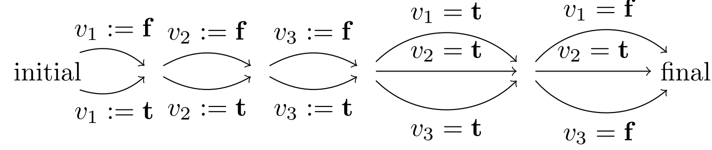


Figure 3: Reduction of CNF-SAT over 3 unknowns with clauses $\{v_1 \vee v_2 \vee v_3, \bar{v_1} \vee v_2 \vee \bar{v_3}\}$ to a Boolean 3-register machine

Lemma 20. *The reachability problem for Boolean register machines is PSPACE-complete.*

Proof. Such a machine is easily simulated by a polynomial-space nondeterministic Turing machine; based on Savitch's theorem, the reachability problem is thus in PSPACE.

Any Turing machine using space $P(|x|)$ on input x can be simulated by a Boolean register machine with $O(P(|x|))$ registers, encoding the state of the tape of the Turing machine, and a number of transitions in $O(P(|x|).|D|)$ where D is the description of the Turing machine. □

Lemma 21. *The reachability problem for Boolean register machines with acyclic control-flow is NP-complete.*

Proof. A path from initial to final nodes, along with register values, may be guessed nondeterministically, then checked in polynomial time, thus reachability is in NP.

Any CNF-SAT problem with r Boolean unknowns may be encoded as a Boolean r-register machine as follows: a sequence of r disjunctions between $v_i := \mathbf{t}$ and $v_i := \mathbf{f}$ for each variable i, and then for each clause $v_{i_1^+} \vee \cdots \vee v_{i_{n^+}^+} \vee v_{i_1^-} \vee \cdots \vee v_{i_{n^-}^-}$, a disjunction between edges $v_{i_i^+} = \mathbf{t}$ for all $1 \leq i \leq n^+$ and edges $v_{i_i^-} = \mathbf{f}$ for all $1 \leq i \leq n^-$ (Figure 3). □

All forthcoming reductions will replace each instruction edge ($v_i = b$ guard edges, $v_i := b$ assignment edge) by a "gadget", a small acyclic piece of control-flow graph adorned with accesses to memory blocks; they will also add a prologue and an epilogue. The idea is to simulate executions of the Boolean register machine by executions of the cache system.

However, there is one problem: in Boolean register machines, an execution aborts when a guard is not satisfied, whereas in our cache analysis problems, executions never abort except when reaching a control node with no outgoing edge. We overcome this limitation by arranging the cache analysis problems so that the cache states following the simulation of an *invalid* guard (e.g. the simulated state encodes $v_i = \mathbf{t}$ but the guard is $v_i = \mathbf{f}$) are irremediably marked as incorrectly formed; thus the reachability problem for the Boolean register

machine is reduced to the reachability of a correctly formed cache state at a certain node, which, in the epilogue, is encoded into the reachability of a cache hit (or a cache miss, respectively).

5 FIFO

FIFO (First-In, First-Out), also known as "round-robin", caches follow the same mechanism as LRU (a bounded queue ordered by age in the cache), except that a block is not rejuvenated on a hit. They are used in Motorola PowerPC 56x, Intel XScale, ARM9, ARM11 [12, p.21], among others.

5.1 Fundamental properties

The state of a FIFO cache with associativity N is a word of length at most N over the alphabet of cache blocks, composed of pairwise distinct blocks; an empty cache is defined by the empty word. When an access is made to a block a, if it belongs to the word (*hit*) then the cache state does not change. If it does not belong to the word (*miss*), and the length of the word is less than N, then a is appended to the word; otherwise, the length of the word is exactly N — the first block of the word is discarded and a is appended.

Lemma 22. *The exist-hit and the exist-miss problems are in NP for acyclic control flow graphs.*

Proof. Guess a path nondeterministically and execute the policy along it. □

Lemma 23. *The exist-hit and the exist-miss problems are in PSPACE for general graphs.*

Proof. Simulate the execution of the policy using a polynomial-space nondeterministic Turing machine. Based on Savitch's theorem, both problems are in PSPACE. □

5.2 Reduction to Exist-Hit

We reduce the reachability problem for the Boolean register machine to the exist-hit problem for the FIFO cache as follows. The associativity of the cache is chosen as $N = 2r - 1$. The alphabet of cache blocks is $\{(a_{i,b})_{1 \le i \le r, b \in \{\mathbf{f},\mathbf{t}\}}\} \cup \{(e_i)_{1 \le i \le r}\} \cup \{(f_i)_{1 \le i \le r-1}\} \cup \{(g_i)_{1 \le i \le r-1}\}$.

The main idea is to encode the value of registers by loading the blocks $a_{i,b}$ into the cache ($a_{i,\mathbf{t}}$ is used when the register i contains value true, and $a_{i,\mathbf{f}}$ is used for false). The blocks e_i are used to distinguished valid boolean machine executions from executions where the machine should have halt. Finally, blocks f_i and g_i are used in epilogue to turn valid states into cache hits and invalid states into cache misses.

The register state $v_1, \dots, v_r$ of the register machine is to be encoded as the FIFO state

$$a_{1,v_1} e_2 a_{2,v_2} \dots e_r a_{r,v_r}. \tag{2}$$

We use the FIFO state essentially as a delay-line memory.[6]

Definition 24. We say that a FIFO state of that form is *well-formed at shift* 1, or *well-formed* for short if it is of the form

$$a_{1,v_1} e_2 a_{2,v_2} \dots e_r a_{r,v_r} \tag{3}$$

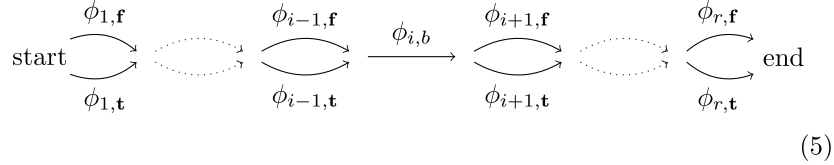
We say that a FIFO state is well-formed at shift i ($2 \leq i \leq r$) if it is of the form

$$a_{i,v_i} e_{i+1} a_{i+1,v_{i+1}} \dots a_{r,v_r} e_1 a_{1,v_1} e_2 \dots a_{i-1,v_{i-1}} \tag{4}$$

In both cases, we say that the FIFO state *corresponds* to the state $v_1, \dots, v_r$. This formalizes the notion of valid and invalid states.

Definition 25. We turn the register machine graph into a cache analysis graph as follows.

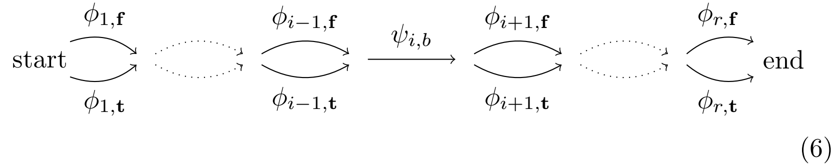
- From the cache analysis initial node I_f to the register machine former initial node I_r there is a prologue, a sequence of accesses $a_{1,\mathbf{f}} e_2 \dots a_{r-1,\mathbf{f}} e_r a_{r,\mathbf{f}}$.
- Each guard edge $v_i = b$ is replaced by the gadget



(5)

  where $\phi_{i,b}$ denotes the sequence of accesses $a_{i,b} e_i a_{i,b}$.

- Each assignment edge $v_i := b$ is replaced by the gadget



(6)

  where $\psi_{i,b}$ denotes the sequence of accesses $e_i a_{i,b} e_i$.

- From the register machine former final node F_r to a node F_a there is a sequence of accesses $\psi_{1,\mathbf{f}} \dots \psi_{r,\mathbf{f}}$, constituting the first part of the epilogue.
- From F_a to a node F_h there is a sequence of accesses $a_{1,\mathbf{f}} g_1 e_2 f_2 a_{2,\mathbf{f}} g_2 \dots e_{r-1} f_{r-1} a_{r-1,\mathbf{f}} g_{r-1} e_r f_r$, constituting the second part of the epilogue.

[6] We thank Ken McMillan for this remark.

- The final node is $F_f = F_h$.

The main difficulty in this reduction is that the Boolean register machines may terminate traces if a guard is not satisfied, whereas the cache problem has no guards and no way to terminate traces. Our workaround is that cache states that do not correspond to traces from the Boolean machine are irremediably marked as incorrect (formally: *well-phased but not well-formed*, per the following definition).

Definition 26. We say that a FIFO state of that form is *well-phased at shift* 1, or *well-phased* for short if it is of the form

$$\beta_1 \alpha_2 \beta_2 \dots \alpha_r \beta_r \tag{7}$$

where, for each i:

- either $\alpha_i = e_i$ and $\beta_i = a_{i,b_i}$ for some b_i,
- or $\beta_i = e_i$ and $\alpha_i = a_{i,b_i}$ for some b_i.

We say that a FIFO state is well-phased at shift i ($2 \leq i \leq r$) if it is of the form

$$\beta_i \alpha_{i+1} \beta_{i+1} \dots \alpha_r \beta_r \alpha_1 \beta_1 \dots \alpha_{i-1} \beta_{i-1} \tag{8}$$

Lemma 27. *Assume w is well-formed at shift i, corresponding to state $\sigma = (\sigma_1, \dots, \sigma_r)$. If $\sigma_i = b$, then executing $\phi_{i,b}$ over FIFO state w leads to a state well-formed at shift $i+1$ (1 if $i = r$), corresponding to σ too. If $\sigma_i = \neg b$, then executing $\phi_{i,b}$ over FIFO state w leads to a state well-phased, but not well-formed, at shift $i+1$ (1 if $i = r$).*

Proof. Without loss of generality we prove this for $i = 1$ and $b = \mathbf{f}$. Assume $w = a_{1,\mathbf{f}} e_2 a_{2,v_2} \dots e_r a_{r,v_r}$; then the sequence $\phi_{1,\mathbf{f}} = a_{1,\mathbf{f}} e_1 a_{1,\mathbf{f}}$ yields $a_{2,v_2} \dots e_r a_{r,v_r} e_1 a_{1,\mathbf{f}}$. Assume now $w = a_{1,\mathbf{t}} e_2 a_{2,v_2} \dots e_r a_{r,v_r}$; then $\phi_{1,\mathbf{f}}$ yields $a_{2,v_2} \dots e_r a_{r,v_r} a_{1,\mathbf{f}} e_1$. ☐

Lemma 28. *Assume w is well-formed at shift i, corresponding to state $\sigma = (\sigma_1, \dots, \sigma_r)$. Executing $\psi_{i,b}$ over FIFO state w leads to a state well-formed at shift $i+1$ (1 if $i = r$), corresponding to σ where σ_i has been replaced by b.*

Proof. Without loss of generality we prove it for $i = 1$ and $b = \mathbf{f}$. Assume $w = a_{1,v_1} e_2 a_{2,v_2} \dots e_r a_{r,v_r}$; then the sequence $\psi_{1,\mathbf{f}} = e_1 a_{1,\mathbf{f}} e_1$ yields $a_{2,v_2} \dots e_r a_{r,v_r} e_1 a_{1,\mathbf{f}}$. ☐

Corollary 29. *Assume starting in a well-formed FIFO state, corresponding to state σ, then any path through the gadget encoding an assignment or a guard*

- *either leads to a well-formed FIFO state, corresponding to the state σ' obtained by executing the assignment, or $\sigma' = \sigma$ for a valid guard;*
- *or leads to a well-phased but not well-formed state.*

Lemma 30. *Assume w is well-phased, but not well-formed, at shift i, then executing $\psi_{i,b}$ or $\phi_{i,b}$ over FIFO state w leads to a state well-phased, but not well-formed, at shift $i+1$.*

Proof. Without loss of generality, we shall prove this for $i = 1$. Let $w = \beta_1\alpha_2\beta_2\ldots\alpha_r\beta_r$.

First case: $\beta_1 = e_1$. $\psi_{1,b} = e_1 a_{1,b} e_1$ then leads to $\beta_2\alpha_3\beta_3\ldots\alpha_r\beta_r a_{1,b} e_1$, which is well-phased, but not well-formed due to the last two blocks, at shift 2. $\phi_{1,b} = a_{1,b} e_1 a_{1,b}$ also leads to the same state.

Second case: β_1 is either $a_{1,\mathbf{f}}$ or $a_{1,\mathbf{t}}$; assume the former without loss of generality. Then there exists $j > 1$ such that $\alpha_j = a_{j,v_j}$ and $\beta_j = e_j$. $\psi_{1,b}$ then leads to $\beta_2\alpha_3\beta_3\ldots\alpha_r\beta_r e_1 a_{1,b}$, which is well-phased, but not well-formed due to the α_j, β_j, at shift 2.

$\phi_{1,\mathbf{f}}$ leads to $\beta_2\alpha_3\beta_3\ldots\alpha_r\beta_r e_1 a_{1,\mathbf{f}}$, which is well-phased, but not well-formed due to the α_j, β_j, at shift 2.

$\phi_{1,\mathbf{t}}$ leads to $\beta_2\alpha_3\beta_3\ldots\alpha_r\beta_r a_{1,\mathbf{t}} e_1$, which is well-phased, but not well-formed due to the last two blocks, at shift 2. □

Corollary 31. *Assume starting in a well-phased but not well-formed FIFO state, then any path through the gadget encoding an assignment or a guard leads to a well-phased but not well-formed FIFO state.*

Corollary 32. *Any path from a well-formed FIFO state in I_r to F_r in the FIFO graph from Theorem 25*

- *either corresponds to a valid sequence of assignments and guards from the register machine from I_r to F_r, and leads to a well-formed FIFO state corresponding to the final state of that sequence*
- *or corresponds to an invalid sequence of assignments and guards from the register machine, and leads to a well-phased but not well-formed FIFO state.*

Conversely, any valid sequence of assignments and guards from the register machine maps from I_r to F_r transforms a well-formed FIFO state into a well-formed FIFO state, corresponding respectively to the initial and final states of that sequence.

Corollary 33. *The path from F_r to F_a (as in Definition 25):*

- *transforms a well-phased but not well-formed FIFO state into a well-phased but not well-formed FIFO state*
- *transforms any well-formed FIFO state into a well-formed FIFO state w_0 corresponding to the initial register state (all registers zero).*

Lemma 34. *The path $a_{1,\mathbf{f}} g_1 e_2 f_2 a_{2,\mathbf{f}} g_2 \ldots e_{r-1} f_{r-1} a_{r-1,\mathbf{f}} g_{r-1} e_r f_r$ (from F_a to F_h in Definition 25):*

- *transforms w_0 into $a_{r,\mathbf{f}} g_1 f_2 g_2 \ldots f_{r-1} g_{r-1} f_r$*

- *transforms any other word w consisting of a's and e's into a word not containing $a_{r,\mathbf{f}}$.*

Proof. The first item is trivial. We shall now prove that it is necessary for the input word to be exactly w_0 in order for the final word to contain $a_{r,\mathbf{f}}$. In order for that, there must have been at most $2r-2$ misses along the path. The accesses to $g_1, f_2, g_2, \ldots, f_{r-1}, g_{r-1}, f_r$ are always misses. As there are $2r-2$ of them, there must have been exactly those misses and no others. This implies that $a_{r,\mathbf{f}}$ was in the last position in w.

When e_r is processed, similarly there were exactly $2r-3$ misses, and e_r must be a hit. This implies that e_r was in the last or the penultimate position in w, but since the last position was occupied by $a_{r,\mathbf{f}}$, e_r must have been in the penultimate position.

The same reasoning holds for all preceding locations, down to the first one, and thus the lemma holds. □

From all these lemmas, the main result follows:

Corollary 35. *There is an execution of the FIFO cache from I_f to F_f such that $a_{r,\mathbf{f}}$ is in the final cache state if and only if there is an execution of the Boolean register machine from I_r to F_r.*

Theorem 36. *The exist-hit problem for FIFO caches is NP-complete for acyclic graphs and PSPACE-complete for general graphs.*

Proof. As seen above, a register machine reachability problem can be reduced in polynomial time to a exist-hit FIFO problem, preserving acyclicity if applicable. □

Remark 37. We have described a reduction from a r-register machine to a FIFO cache problem with an odd $2r-1$ number of ways. This reduction may be altered to yield an even number of ways as follows. Two special padding blocks p and p' are added. A well-formed state is now $pa_{1,b_1}e_2a_{2,b_2}\ldots a_{r,b_r}$; the definition of well-phased states is similarly modified. Each gadget G for assignment or guard is replaced by $p'GpG$. The first p' turns padding p into p', G is applied. The second p' turns p' into p and G is applied again.

This remark also applies to the exist-miss problem.

5.3 Reduction to Exist-Miss

We modify the reduction for exist-hit in order to exhibit a miss on $a_{r,\mathbf{f}}$ later on if and only if it is in the cache at the end of the graph defined above.

Definition 38. We transform the register machine graph into a cache analysis graph as in Theorem 25, with the following modification: in between F_h and F_f we insert a sequence $a_{r,\mathbf{f}}e_1a_{1,\mathbf{f}}\ldots e_{r-1}a_{r-1,\mathbf{f}}e_r$, constituting the third part of the epilogue.

Lemma 39. *The path from F_h to F_f transforms $a_{r,\mathbf{f}} g_1 f_2 g_2 \ldots f_{r-1} g_{r-1} f_r$ into a word not containing $a_{r,\mathbf{f}}$. It transforms any word composed of f's and g's only into a word containing $a_{r,\mathbf{f}}$.*

Theorem 40. *The exist-miss problem for FIFO caches is NP-complete for acyclic graphs and PSPACE-complete for general graphs.*

5.4 Extension to arbitrary starting cache

Lemma 41. *The exist-hit and exist-miss problems for an empty starting FIFO cache state are reduced, in linear time, to the same kind of problem for an arbitrary starting cache state, with the same associativity.*

Proof. Let Σ be the alphabet of blocks in the problem and N its associativity. Let $e_1, \ldots, e_{2N-1}$ be new blocks not in Σ; after accessing them in sequence, the cache contains only elements from these accesses [13, Th. 1]. Prepend this sequence as a prologue to the cache problem; then the rest of the execution of the cache problem will behave as though it started from an empty cache. □

Corollary 42. *The exist-hit and exist-miss problems for FIFO caches with arbitrary starting state is NP-complete for acyclic graphs and PSPACE-complete for general graphs.*

5.5 Extension to Pseudo-RR caches

Recall how a FIFO cache with multiple cache sets — the usual approach in hardware caches — operates. A memory block of address x is stored in the cache set number $H(x)$ where H is a suitable function, normally a simple combination of the bits of x. In typical situations, this is as though the address x were specified as a pair (s, a) where s is the number of the cache set and a is the block name to be used by the FIFO in cache set number s.

In a FIFO cache, each cache set, being a FIFO, can be implemented as a circular buffer: an array of cache blocks and a "next to be evicted" index. In contrast, in a pseudo-RR cache, the "next to be evicted" index is global to all cache sets.

A FIFO cache exist-hit or exist-miss problem with cache blocks $a_1, \ldots, a_n$ can be turned into an equivalent pseudo-RR problem simply by using $(s, a_1), \ldots, (s, a_n)$ as addresses for a constant distinguished cache set s. Thus, both exist-hit and exist-miss are NP-hard for acyclic control-flow graphs on pseudo-RR caches, and PSPACE-hard for general control-flow graphs.

The same simulation arguments used for FIFO (subsection 5.1) hold for establishing membership in NP and PSPACE respectively.

6 PLRU

Because LRU caches were considered too difficult to implement efficiently in hardware, various schemes for heuristically approximating the behavior of a

LRU cache (keeping the most recently used data) have been proposed. By "heuristically approximating" we mean that these schemes are assumed, on "typical" workloads, to perform close to LRU, even though worst-case performance may be different.[7] Some authors lump all such schemes as "pseudo-LRU" or "PLRU", and call the scheme in the present section "tree-based PLRU" or "PLRU-t" [1], while some others [12, p. 26] call "PLRU" only the scheme discussed here.

PLRU has been used in i486 [8] (4-way), Intel Pentium II-IV, PowerPC 75x [12, p. 21]; an 8-way PLRU is used in NXP/Freescale MPC745x [10, p. 3-41] and MPC75x [11, p. 3-19], e6500, MPC8540.

6.1 PLRU caches

The cache lines of a PLRU cache, which may contain cached blocks, are arranged as the leaves of a full binary tree — thus the number of ways N is a power of 2, often 4 or 8. Two lines may not contain the same block. Each internal node of the tree has a tag bit, which is represented as an arrow pointing to the left or right branch. The state of the cache is thus the content of the lines and the $N - 1$ tag bits.

There is always a unique line such that there is a sequence of arrows from the root of the tree to the line; this is the line *pointed at by the tags.* Tags are said to be *adjusted away* from a line as follows: on the path from the root of the tree to the line, tag bits are adjusted so that the arrows all point away from that path.

When a block a is accessed:

- If the block is already in the cache, tags are adjusted away from this line.
- If the block is not already in the cache and one or more cache lines are empty, the leftmost empty line is filled with a, and tags are adjusted away from this block.
- If the block is not already in the cache and no cache line is empty, the block pointed at by the tags is evicted and replaced with a, and tags are adjusted away from this block.

[7] Experimentally, on typical workloads, the tree-based PLRU scheme described in this section is said to produce 5% more misses on a level-1 data cache compared to LRU [1]. However, that scheme may, under specific concocted workloads, indefinitely keep data that are actually never used except once— a misperformance that cannot occur with LRU [7]. This can produce *domino effects*: the cache behavior of a loop body may be indefinitely affected by the cache contents before the loop [2].

Because of the difficulties in obtaining justifiable bounds on the worst-case execution times of programs running on a PLRU cache, some designers of safety-critical real-time systems lock all cache ways except for two, exploiting the fact that a 2-way PLRU cache is the same as a 2-way LRU cache and thus recovering predictability [2, §3].

6.2 Exist-Hit Problem

We reduce the reachability problem of a Boolean r-register machine to the PLRU exist-hit problem for a $(2r+2)$-way cache — without loss of generality, we can always add useless registers so that $2r+2$ is a power of two. The alphabet of cache blocks is $\{(a_{i,b})_{1\le i\le r, b\in\{\mathbf{f},\mathbf{t}\}}\} \cup \{(e_i)_{0\le i\le r}\} \cup \{c\}$.

Definition 43. We say that a PLRU cache state is *well-formed* and *corresponds* to a Boolean state $(b_i)_{1\le i\le r}$ if its leaves are, from left to right: $c, e_0, a_{1,b_1}, e_1, \dots, a_{r,b_r}, e_r$.

Definition 44. We say that a PLRU cache state is *well-phased* if its leaves are, from left to right: $x_0, e_0, a_{1,b_1}, e_1, \dots, a_{r,b_r}, e_r$ where x_0 can be c or any $a_{i,b}$ block.

We use the PLRU state as a random access memory. Appropriate sequence of accesses define the memory location to be read or written.

Lemma 45. *Let $0 \le i \le r$, there exists a sequence π_i of accesses, of length logarithmic in r, such that, when run on a well-phased cache state $x_0, e_0, \dots, x_r, e_r$, that sequence makes tags point at x_i without changing the contents of the cache lines.*

Proof. Moving from x_i to the root of the tree, at every node one e_i block is accessed from the other branch at that node (see Figure 4 for an example of sequence π_1). ☐

Let $1 \le i \le r$, $b \in \{\mathbf{t}, \mathbf{f}\}$. Let $\phi_{i,b}$ be the sequence $\pi_0 a_{i,b}$, and $\psi_{i,b}$ the sequence $\pi_i a_{i,b}$.

Definition 46. We turn the register machine graph into a cache analysis graph as follows.

- From the cache analysis initial node I_p to the register machine former initial node I_r there is a sequence of accesses $c, e_0, a_{1,\mathbf{f}}, e_1, \dots, a_{r,\mathbf{f}}, e_r$.
- Each guard edge $v_i = b$ is replaced by the sequence $\phi_{i,b}$, and each assignment edge $v_i := b$ by the sequence $\psi_{i,b}$.
- The cache final node F_p is the same as the register machine final node F_r.

The idea is that any missed guard irremediably removes c from the cache. The following lemmas are easily proved by symbolically simulating the execution of the gadgets over the cache states:

Lemma 47. *$\phi_{i,b}$ and $\psi_{i,b}$ map any well-phased but not well-formed state to a well-phased but not well-formed state.*

Lemma 48. *$\psi_{i,b}$ maps a well-formed state to a well-formed state corresponding to the same Boolean state where register i has been replaced by b.*

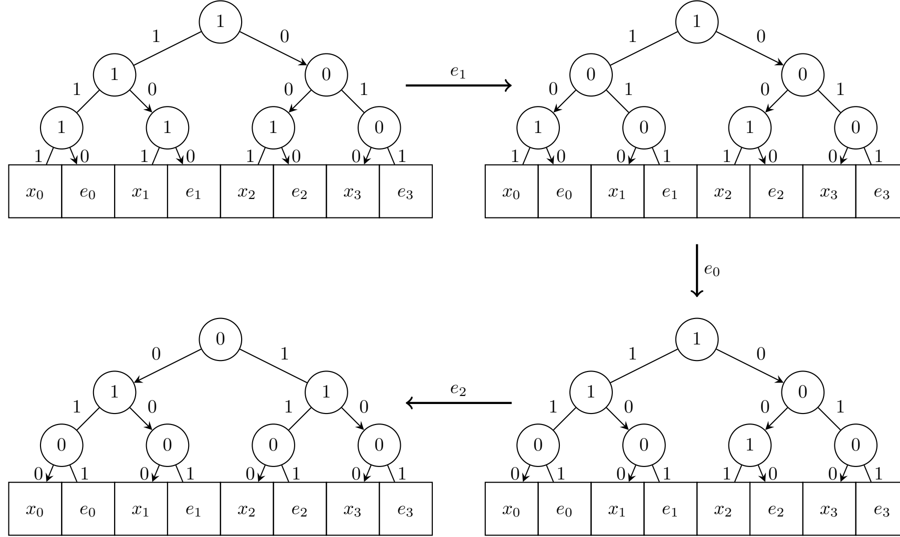


Figure 4: Sequence $\pi_1 = e_1 e_0 e_2$ makes tags point at x_1 without changing cache content

Lemma 49. *$\phi_{i,b}$ maps a well-formed state corresponding to a Boolean state $(b_i)_{1 \leq i \leq r}$ to*

- *if $b_i = b$, a well-formed state corresponding to the same Boolean state;*
- *otherwise, a well-phased but not well-formed state.*

Corollary 50. *There is an execution of the PLRU cache from I_p to F_p such that c is in the final cache state if and only if there is an execution of the Boolean register machine from I_r to F_r.*

Proof. A well-phased state is well-formed if and only if it contains c. □

Theorem 51. *The exist-hit problem for PLRU caches is NP-complete for acyclic graphs and PSPACE-complete for general graphs.*

6.3 Exist-Miss Problem

We use two extra blocks d and f.

Definition 52. Let Z be the sequence $\pi_r d \pi_r c \pi_0 f$.

Lemma 53. *Z turns any well-formed state into a state not containing c. Z turns any well-phased but not well-formed state into a state containing c.*

Proof. Consider a well-formed state $c e_0 a_{1,b_1} e_1 \dots a_{r,b_r} e_r$. Z replaces a_{r,b_r} by d; then the access to c does not change the line contents since c is in the cache, and c is replaced by f.

Consider a well-phased but not well-formed state $x_0 e_0 a_{1,b_1} e_1 \dots a_{r,b_r} e_r$ where $x_0 \neq c$. Z replaces a_{r,b_r} by d; then the access to c replaces d by c, and x_0 is replaced by f. □

Definition 54. We turn the register machine graph into a cache analysis graph in the same manner as in Definition 46, but between F_r and F_p we insert Z as epilogue.

Lemma 55. *There is an execution of the PLRU cache from I_p to F_p such that c is not in the final cache state if and only if there is an execution of the Boolean register machine from I_r to F_r.*

Theorem 56. *The exist-miss problem for PLRU caches is NP-complete for acyclic graphs and PSPACE-complete for general graphs.*

6.4 Extension to an arbitrary starting cache

Reineke et al. [13, Th. 12] proved the following result on PLRU caches:

Theorem 57. *It takes at most $\frac{k}{2}\log_2 k + 1$ pairwise different accesses to evict all entries from a k-way set-associative PLRU cache set. Again, this is a tight bound.*

More precisely, they prove that there is a sequence of accesses of length $\frac{k}{2}\log_2 k + 1$ such that after executing this sequence over a cache with arbitrary initial state, the cache contains only elements from the sequence.

Lemma 58. *The exist-hit and exist-miss problems for an empty starting PLRU cache state are reduced, in linear time, to the same kind of problem for an arbitrary starting cache state, with the same associativity.*

Proof. Same proof as Theorem 41, except we need a sequence of $M = \frac{N}{2}\log_2 N + 1$ new blocks.

Let Σ be the alphabet of blocks in the problem and N its associativity. Let $e_1, \dots, e_M$ be new blocks not in Σ; after accessing them in a sequence as constructed in Theorem 57, there is no longer any block from Σ in the cache. Prepend this sequence as a prologue to the cache problem; then the rest of the execution of the cache problem will behave as though it started from an empty cache. □

Corollary 59. *The exist-hit and exist-miss problems for FIFO caches with arbitrary starting state is NP-complete for acyclic graphs and PSPACE-complete for general graphs.*

7 NMRU

Other forms of "pseudo-LRU" schemes have been proposed than the one discussed in section 6. One of them, due to Malamy, Patel, and Hayes [9] is based

on the use of "most recently used" bits. It is thus sometimes referred to as the "not most recently used" (NMRU) policy, or "PLRU-m" [1]. Confusingly, some literature [12] also refers to this policy as "MRU" despite the fact that in this policy, it is *not* the most recently used data block that is evicted first.

NMRU is used in the Intel Nehalem architecture, among others.

7.1 NMRU caches

Definition 60. The state of an N-way NMRU cache is a sequence of at most N memory blocks α_i, each tagged by a 0/1 "MRU-bit" r_i saying whether the associated block is to be considered not recently used (0) or recently used (1), denoted by $\alpha_1^{r_1} \dots \alpha_N^{r_N}$.

An access to a block in the cache, a *hit*, results in the associated MRU-bit being set to 1. If there were already $N-1$ MRU-bits equal to 1, then all the other MRU-bits are set to 0.

An access to a block a not in the cache, a *miss*, results in:

- if the cache is not full (number of blocks less than N), then a^1 is appended to the sequence
- if the cache is full (number of blocks equal to N), then the leftmost (least index i) block with associated MRU-bit 0 is replaced by a^1. If there were already $N-1$ MRU-bits equal to 1, then all the other MRU-bits are set to 0.

Remark 61. This definition is correct because the following invariant is maintained: either the cache is not full, or it is full but at least one MRU-bit is zero.

Example 62. Assume $N = 4$. If the cache contains $a^0b^0c^0$, then an access to d yields $a^0b^0c^0d^1$ since the cache was not full. If a is then accessed, the state becomes $a^1b^0c^0d^1$. If e is then accessed, the state becomes $a^1e^1c^0d^1$ since b was the leftmost block with a zero MRU-bit. If f is then accessed, then the state becomes $a^0e^0f^1d^0$.

7.2 Reduction to Exist-Hit

We reduce the reachability problem for the register machine to the exist-hit problem for the NMRU cache as follows. The associativity of the cache is chosen as $N = 2r+3$. The alphabet of the cache blocks is $\{(a_{i,b})_{1\leq i\leq r, b\in\{\mathbf{f},\mathbf{t}\}}\} \cup \{(e_i)_{1\leq i\leq r}\} \cup \{(c_i)_{1\leq i\leq r}\} \cup \{d\} \cup \{g_0, g_1\}$.

The register state $v_1, \dots, v_r$ of the register machine is to be encoded as the NMRU state

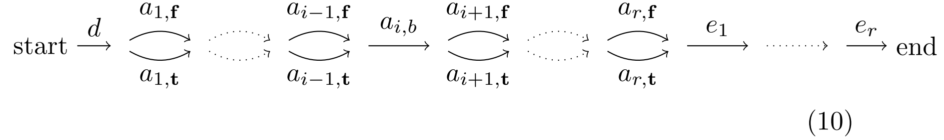
$$e_1^0 \dots e_r^0 d^0 a_{1,v_1} \dots a_{r,v_r}^0 g_0^0 g_1^1 \tag{9}$$

where the exponent (0 or 1) is the MRU-bit associated with the block.

Definition 63. We turn the register machine graph into a cache analysis graph as follows.

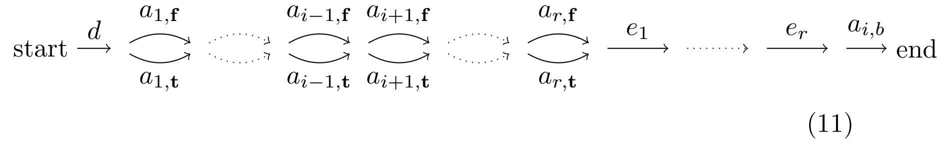
- From the cache analysis initial node I_f to the register machine former initial node I_r there is the prologue: the sequence of accesses $e_1 \dots e_r d a_{1,\mathbf{f}} \dots a_{r,\mathbf{f}} g_0 g_1$.

- Each guard edge $v_i = b$ is replaced by the gadget $\phi_{i,b} g_0 \phi_{i,b} g_1$, where $\phi_{i,b}$ is



(10)

- Each assignment edge $v_i := b$ is replaced by the gadget $\psi_{i,b} g_0 \psi_{i,b} g_1$, where $\psi_{i,b}$ is



(11)

- From the register machine former final node F_r to a node F_a there is a sequence of gadgets for the assignments $v_1 := \mathbf{f} \dots v_r := \mathbf{f}$, the first part of the epilogue.

- From F_a to a node F_h there is a sequence of accesses $a_{1,\mathbf{f}} \dots a_{r,\mathbf{f}} c_1 \dots c_r$, the second part of the epilogue.

- The final node is $F_f = F_h$.

Definition 64. We say that an NMRU state is well-formed at step $s \in \{0, 1\}$ if it is of the form

$$\beta_1^0 \dots \beta_r^0 d^0 \alpha_1^0 \dots \alpha_r^0 g_0^s g_1^{1-s} \tag{12}$$

where $\forall i, 1 \le i \le r, \alpha_i \in \{a_{\sigma(i),\mathbf{f}}, a_{\sigma(i),\mathbf{t}}\}, \beta_i = e_{\sigma'(i)}$ and σ and σ' are two permutations of $[1, r]$. In other words, a well-formed state contains r distinct blocks e_i placed before d, and r blocks $a_{i,b}$, with distinct i's, placed between d and g_0. We say "well-formed" for short if $s = 0$.

Definition 65. We say that an NMRU state is well-phased at step $s \in \{0, 1\}$ if it is of the form

$$\gamma^0_{\sigma(1)} \cdots \gamma^0_{\sigma(r)} d^0 \gamma^0_{\sigma(r+1)} \cdots \gamma^0_{\sigma(2r)} g_0^s g_1^{1-s} \tag{13}$$

where $\gamma_1 = e_1, \dots, \gamma_r = e_r, \gamma_{r+1} \in \{a_{1,\mathbf{f}}, a_{1,\mathbf{t}}\}, \dots, \gamma_{2r} \in \{a_{r,\mathbf{f}}, a_{r,\mathbf{t}}\}$ and σ is a permutation of $[1, 2r]$. We say "well-phased" for short if $s = 0$.

Lemma 66. *Executing a path through $\phi_{i,b} g_s$ over an NMRU state w well-phased at step s always leads to a state well-phased at step $1-s$. Furthermore that state*

- *either is not well-formed at step $1 - s$*

- *or is identical to w except for the g_0 and g_1 blocks, and this may occur only if $a_{i,b}$ belongs to w.*

Proof. The input state w is $x_1^0, \dots, x_r^0, d^0, x_{r+1}^0, \dots, x_{2r}^0, g_0^0, g_1^1$ where the $(x_i)_{1 \le i \le 2r}$ are a permutation of $\{e_i \mid 1 \le i \le r\} \cup \{a_{i,\beta_i} \mid 1 \le i \le r\}$ for some sequence of Booleans $(\beta_i)_{1 \le i \le r}$.

Consider a path through $\phi_{i,b}$: it consists of d, followed by a sequence of r a's, then r e's. Each of these accesses either freshens, or overwrites, one of the x positions. After the sequence of a's, there are either no a's to the left of d, or at least one. The former case is possible only if all a's are hits, freshening positions to the right of d — this means all these positions are left untouched except that their MRU bits are flipped to 1. Then the sequence of e's just flips to 1 the MRU-bits of the e's, all located to the left of d. The resulting state is thus identical to w except that all MRU-bits to the left of g_0 have been flipped to 1; thus after accessing g_0, the state is identical to the initial state except that it ends with $g_0^{1-s} g_1^s$ instead of $g_0^s g_1^{1-s}$.

Now consider the latter case: after the sequence of a's there is at least one position of the form $a_{i,\beta}^1$ to the left of d. This position cannot be overwritten by the e's. After the path through $\phi_{i,b}$, the state is thus of the form $x_1^1, \dots, x_r^1, d^1, x_{r+1}^1, \dots, x_{2r}^1, g_0^0, g_1^1$, and one of the x_i for $1 \le i \le r$ is an a. The access to g_0 yields $x_1^0, \dots, x_r^0, d^0, x_{r+1}^0, \dots, x_{2r}^0, g_0^{1-s}, g_1^s$. This state is well-phased but not well-formed. ☐

Lemma 67. *Executing a path through $\psi_{i,b} g_s$ over an NMRU state well-phased at step s always leads to a state well-phased at step $1-s$. Furthermore that state*

- *either is not well-formed at step $1-s$*
- *or is identical to the initial state except for the g_0 and g_1 blocks, and, possibly, the a_{i,β_i} block replaced by $a_{i,b}$.*

Proof. Again, the initial state is $x_1^0, \dots, x_r^0, d^0, x_{r+1}^0, \dots, x_{2r}^0, g_0^0, g_1^1$ where the $(x_i)_{1 \le i \le 2r}$ are a permutation of $\{e_i \mid 1 \le i \le r\} \cup \{a_{i,\beta_i} \mid 1 \le i \le r\}$ for some sequence of Booleans $(\beta_i)_{1 \le i \le r}$.

Consider a path through $\psi_{i,b}$: it consists of d, followed by a sequence of $r-1$ a's, then r e's, then $a_{i,b}$. Each of these accesses either freshens, or overwrites, one of the x positions. After the sequence of a's, there are either no a's to the left of d, or at least one. The former case is possible only if all these a's are hits, freshening positions to the right of d. Then the sequence of e's freshens the e's to the left of d. There is one remaining x position with a zero MRU-bit: it is to the right of d and carries a block a_{i,β_i}. This block is then updated or freshened by the $a_{i,b}$ access. Then the access to g_0 flips all MRU-bits to 0 except the one for g_0, which is flipped to 1. Since all of the accesses before the $a_{i,b}$ access were hits, the permutation of the positions has not changed: the state is the same as the initial state except that a_{i,β_i}^0 is replaced by $a_{i,b}^0$ and $g_0^s g_1^{1-s}$ is replaced by $g_0^{1-s} g_1^s$.

Now consider the latter case: after the sequence of a's there is at least one position of the form $a^1_{i,\beta}$ to the left of d. Then, as in the proof of the previous lemma, there is still $a^0_{i,\beta}$ to the left of d at the end of the path through $\psi_{i,b} g_s$. Thus the final state cannot be well-formed. □

Corollary 68. *Assume starting in a well-formed NMRU state, corresponding to Boolean state σ, then any path through the gadget encoding an assignment or a guard*

- *either leads to a well-formed NMRU state, corresponding to the state σ' obtained by executing the assignment, or $\sigma' = \sigma$ for a valid guard;*
- *or leads to a well-phased but not well-formed state.*

Lemma 69. *Executing the sequence $a_{1,\mathbf{f}} \dots a_{r,\mathbf{f}} c_1 \dots c_r$ from F_a to F_h over a well-formed NMRU state corresponding to a zero Boolean state leads to a state containing d — more specifically, a state of the form $c^1_1, \dots, c^1_r, d^0, a^1_{\pi(1),\mathbf{f}}, \dots, a^1_{\pi(r),\mathbf{f}} g^0_0 g^1_1$ where π is a permutation.*

Proof. The $a_{1,\mathbf{f}} \dots a_{r,\mathbf{f}}$ just freshen the corresponding blocks (MRU-bit set to 1), and then the $c_1 \dots c_r$ overwrite the e's. □

Lemma 70. *Executing the sequence $a_{1,\mathbf{f}} \dots a_{r,\mathbf{f}} c_1 \dots c_r$ from F_a to F_h over a well-phased but not well-formed NMRU state leads to a state not containing d — where the $2r$ first MRU bits are set to 1, the next one to 0, and then $g^0_0 g^1_1$.*

Proof. The well-phased but not well-formed NMRU state contains at least one b to the right of d. When applying $a_{1,\mathbf{f}} \dots a_{r,\mathbf{f}}$, at least one of the a's must thus freshen or replace a block to the left of d. Then when applying $c_1 \dots c_r$, d gets erased. □

Corollary 71. *There is an execution sequence from I_f, with empty cache, to F_f, such that the final cache contains d if and only if there is an execution trace from I_r to F_r.*

Theorem 72. *The exist-hit problem for NMRU caches is NP-complete for acyclic graphs and PSPACE-complete for general graphs.*

7.3 Reduction to Exist-Miss

Definition 73. We modify the reduction of Theorem 63 as follows. Between F_h and F_f we insert the sequence $d g_0 c_1 \dots c_r a_{1,\mathbf{t}}$, as the third part of the epilogue.

Lemma 74. *Executing $d g_0 c_1 \dots c_r a_{1,\mathbf{t}}$ over a state of the form $c^1_1, \dots, c^1_r, d^0, a^1_{\pi(1),\mathbf{f}}, \dots, a^1_{\pi(r),\mathbf{f}} g^0_0 g^1_1$, where π is a permutation, leads to a state without d.*

Proof. d gets freshened, then g_0 is the sole block with a zero MRU-bit. Thus, when it is freshened, all other MRU bits are set to zero. Then $c_1 \dots c_r$ freshen the first r blocks, and $a_{1,\mathbf{t}}$ erases d. □

Lemma 75. *Executing $dg_0c_1 \dots c_r a_{1,\mathbf{t}}$ over a state not containing d, where the $2r$ first MRU bits are set to 1, the next one to 0, and then $g_0^0 g_1^1$ leads to a state containing d.*

Proof. d overwrites the $2r+1$-th position, then g_0 is the sole block with a zero MRU-bit. Thus, when it is freshened, all other MRU bits are set to zero. Then possibly some blocks get overwritten among the $r+1$ first blocks, and d is still in the cache. □

Corollary 76. *There is an execution sequence from I_f, with empty cache, to F_f, such that the final cache does not contain d if and only if there is an execution of the Boolean register machine from I_r to F_r.*

Theorem 77. *The exist-miss problem for NMRU caches is NP-complete for acyclic graphs and PSPACE-complete for general graphs.*

We have no results for reductions to cache analysis problems with arbitrary starting state. The proof method that we used for FIFO and PLRU — prepend a sufficiently long sequence of accesses that will bring the cache to a sufficiently known state — does not seem to easily carry over to NMRU caches. Even though it is known that $2N-2$ pairwise distinct accesses are sufficient to remove all previous content from an NMRU cache [13, Th. 4], it can be shown that there is no sequence guaranteed to yield a completely known cache state [13, Th. 5].

8 Conclusion and future work

We have shown complexity-theoretical properties that separate LRU from other policies such as PLRU, NMRU, FIFO, pseudo-RR and that give a more rigorous meaning to the intuition that static analysis for LRU is "easier" than for other policies, due to its forgetful tendencies. Reachability problems for PLRU, NMRU, FIFO, pseudo-RR caches in fact are of the same complexity class (PSPACE-complete) as generic reachability problems with arbitrary transitions over bits [4], which again justifies why they are hard.

Our assumption that all paths through the control-flow graph are feasible is not realistic. As explained in the introduction, this assumption is made because arbitrary arithmetic and tests on lead to all cache analysis problems being equivalent to Turing's halting problem. Another option is restricting the program to a finite number of bits and a transition relation represented by a Boolean circuit, formula, or OBDD, linking the previous values of the bits to the next values; but as we have explained, this would not help distinguishing between policies since all problems then become PSPACE-complete [4]. We thus would need some kind of weaker execution model, but it is difficult to find one that would still be realistic enough to have an interest.

One possibility, suggested by a reviewer, is to add a call stack. In Section 2, we used the fact that the combination of a finite control automaton and the exact state of the cache is just a bigger finite automaton, thus exist-miss and exist-hit can be decided by explicit-state model checking. Adding a call stack,

for modeling procedure calls, would turn the problem into model-checking reachability on a stack automaton, in other words a pushdown system. Despite these systems having infinite state, reachability is decidable [3, 6]. Investigating the complexity of the combination of the call stack with our cache models is left to future work.

Another direction is to distinguish the analysis difficulty for different policies for a fixed associativity. As explained in Section 2, all problems become polynomial with respect to the program size; but one could still want to distinguish asymptotic growths. However, as we have shown, too strong results in this area would entail answers to very hard conjectures in complexity theory.

It is difficult to draw practical implications from worst-case complexity-theoretic asymptotic results. We however think that our results are yet another indication, in addition to the existence of efficient and precise analyses for LRU [5, 14, 15] and their lack for other policies, that the LRU policy is to be preferred for ease of analysis and thus for hard real time critical applications where a worst case execution time bound must be established.[8]

References

[1] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. "Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite". In: *Proceedings of the 42Nd Annual Southeast Regional Conference*. ACM-SE 42. Huntsville, Alabama: ACM, 2004, pp. 267–272. ISBN: 1-58113-870-9. DOI: 10.1145/986537.986601.

[2] Christoph Berg. "PLRU Cache Domino Effects". In: *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*. Ed. by Frank Mueller. Vol. 4. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2006, pp. 69–71. ISBN: 978-3-939897-03-3. DOI: 10.4230/OASIcs.WCET.2006.672.

[3] Ahmed Bouajjani, Javier Esparza, and Oded Maler. "Reachability Analysis of Pushdown Automata: Application to Model-Checking". In: *Concurrency Theory (CONCUR)*. Ed. by Antoni W. Mazurkiewicz and Józef Winkowski. Vol. 1243. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Verlag, 1997, pp. 135–150. DOI: 10.1007/3-540-63141-0_10.

[4] Joan Feigenbaum et al. "Complexity of Problems on Graphs Represented as OBDDs (Extended Abstract)". In: *Symposium on Theoretical Aspects of Computer (STACS)*. Ed. by Michel Morvan, Christoph Meinel, and Daniel Krob. Vol. 1373. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Verlag, 1998, pp. 216–226. DOI: 10.1007/BFb0028563.

[8] We have anecdotal evidence that certain designers of safety critical systems lock 6 ways out of 8 PLRU ways in the MPC755 processor [11, Table C.3] so that the remaining 2 ways are equivalent to a 2-way LRU cache, amenable to analysis. This illustrates the cost of using a policy that performs well "on average" but that is not easy to statically predict.

[5] Christian Ferdinand and Reinhard Wilhelm. "Efficient and Precise Cache Behavior Prediction for Real-Time Systems". In: *Real-Time Systems* 17.2–3 (Dec. 1999), pp. 131–181. ISSN: 0922-6443. DOI: 10.1023/A:1008186323068.

[6] Alain Finkel, Bernard Willems, and Pierre Wolper. "A direct symbolic approach to model checking pushdown systems". In: *Electr. Notes Theor. Comput. Sci.* 9 (1997), pp. 27–37. DOI: 10.1016/S1571-0661(05)80426-8.

[7] Reinhold Heckmann et al. "The influence of processor architecture on the design and the results of WCET tools". In: *Proceedings of the IEEE* 91.7 (2003), pp. 1038–1054. DOI: 10.1109/JPROC.2003.814618.

[8] *i486 Microprocessor data sheet*. Intel. 1989. URL: https://archive.org/stream/bitsavers_intel80486ataSheetApr89_12763574/i486_Microprocess

[9] Adam Malamy, Rajiv N. Patel, and Norman M. Hayes. *Methods and apparatus for implementing a pseudo-LRU cache memory replacement scheme with a locking feature*. US patent 5,353,425. US Patent Office, Oct. 1994. URL: https://patents.google.com/patent/US5353425.

[10] *MPC7450 RISC Microprocessor Family Reference Manual*. 5th ed. NXP / Freescale Semiconductor. Jan. 2005. URL: https://www.nxp.com/docs/en/reference-manual/MPC7450UM.pdf.

[11] *MPC750 RISC Microprocessor Family Reference Manual*. 1st ed. NXP / Freescale Semiconductor. Dec. 2001. URL: https://www.nxp.com/docs/en/reference-manual/MPC750UM.pdf.

[12] Jan Reineke. "Caches in WCET analysis: predictability, competitiveness, sensitivity". PhD thesis. Universität des Saarlandes, 2008. URL: http://www.rw.cdl.uni-saarland.de/~reineke/publications/DissertationCachesInWCETAnalys

[13] Jan Reineke et al. "Timing predictability of cache replacement policies". In: *Real-Time Systems* 37.2 (2007), pp. 99–122. DOI: 10.1007/s11241-007-9032-3. URL: http://www.rw.cdl.uni-saarland.de/~grund/papers/rts07-predictability.pdf.

[14] Valentin Touzeau et al. "Ascertaining Uncertainty for Efficient Exact Cache Analysis". In: *Computer-aided verification (CAV)*. Ed. by Viktor Kuncak and Rupak Majumdar. Vol. 10427. Cham: Springer Verlag, 2017, pp. 22–17. DOI: 10.1007/3-540-63141-0_10. arXiv: 1709.10008.

[15] Valentin Touzeau et al. "Fast and exact analysis for LRU caches". In: *Proceedings of the ACM on Programming Languages (PACMPL)* 3.POPL (2019), 54:1–54:29. DOI: 10.1145/3290367. arXiv: 1811.01670.