Measurement-based Modeling of the Cache Replacement Policy

Andreas Abel and Jan Reineke
Department of Computer Science
Saarland University
Saarbrücken, Germany
Email: abeand@studcs.uni-saarland.de, reineke@cs.uni-saarland.de

***Abstract*—Modern microarchitectures employ memory hierarchies involving one or more levels of cache memory to hide the large latency gap between the processor and main memory. Cycle-accurate simulators, self-optimizing software systems, and platform-aware compilers need accurate models of the memory hierarchy to produce useful results. Similarly, worst-case execution time analyzers require faithful models, both for soundness and precision. Unfortunately, sufficiently precise documentation of the logical organization of the memory hierarchy is seldom available publicly.**

In this paper, we propose an algorithm to automatically model the cache replacement policy by measurements on the actual hardware. We have implemented and applied this algorithm to various popular microarchitectures, uncovering a previously undocumented cache replacement policy in the Intel Atom D525.

I. Introduction

Detailed models of the microarchitecture are at the heart of static worst-case execution time (WCET) analyzers [1]. To develop such detailed models, engineers need a deep understanding of the microarchitecture. Unfortunately, documentation at the level of detail required for sound and precise static WCET analysis is hard to come by. Processor manuals are often ambiguous as they are written in natural language. Sometimes they do not provide any information about a particular architectural feature at all.

As a consequence, engineers are forced to either create "conservative" models, which yield imprecise WCET estimates, sacrifice soundness, or to obtain a better understanding of the microarchitecture by other means. An engineer might contact the processor manufacturer to inquire more details, which the manufacturer is often not willing or able to provide to protect his intellectual property. As a last resort, the engineer is often found tinkering with evaluation boards, performing measurements on microbenchmarks, iteratively refining his models of architectural features, until he is sufficiently confident in his conjectured model. This process is both costly and error-prone.

We believe that the process of inferring a model of the microarchitecture by measurements can—at least partially—be automated. Automation promises to reduce costs and to increase confidence in the results.

In this paper, we focus on the memory hierarchy. The memory hierarchy is an attractive target, because it has a strong influence on execution times. At the same time, most memory hierarchies are structured similarly, and offer a simple interface: loads and stores. Further, modern microarchitectures feature hardware performance counters that can be used to count the number of cache accesses and misses a program generates. We have developed algorithms to determine the capacities, associativities, block sizes, and replacement policies of first- and non-inclusive second-level instruction and data caches.

On the algorithmic side, the main contribution of our work is an algorithm to automatically infer the replacement policy used in a cache. To this end, we introduce *permutation policies*, a class of replacement policies that includes well-known policies, such as least recently used (LRU), first-in first-out (FIFO), and pseudo-LRU (PLRU) in addition to a large set of so far undocumented policies. Permutation policies admit efficient inference through measurements on the actual hardware.

We have developed **chi**[1], a portable and robust implementation of the algorithm that can be run on standard Linux distributions without any modifications. This implementation includes a novel approach to handle instruction caches. We have successfully applied it to a number of x86 processors. On the Intel Atom D525, we discovered a—to our knowledge—previously undocumented approximation of LRU.

In addition to WCET analysis, automatic microarchitectural modeling as performed by **chi** may be of interest in other domains: It can aid in the development of cycle-accurate simulators, such as PTLsim [2]. It can also be employed within self-optimizing software systems like ATLAS [3], PHiPAC [4], or FFTW [5], platform-aware compilers, such as PACE [6], as well as system-level protection mechanisms against side-channel attacks, such as STEALTHMEM [7], all of which require detailed knowledge of cache characteristics.

A. Outline

In Section II, we provide the necessary background regarding caches and the formalization of a cache template. Based on this cache template, we present a concise problem statement in Section III. We introduce permutation policies in Section IV. In Section V, we then describe our algorithm and its implementation. The experimental evaluation, illustrating strengths and weaknesses of our approach, is provided in Section VI. We discuss related work in Section VII and conclude the paper with an overview of potential future work.

[1] **chi** is available for download: http://embedded.cs.uni-saarland.de/chi.php

A	$\in$	Associativity $= \mathbb{N}$	The associativity of the cache.
B	$\in$	BlockSize $= \mathbb{N}$	The block size in bytes.
N	$\in$	NumberOfSets $= \mathbb{N}$	The number of cache sets.
C	$=$	$A \cdot B \cdot N$	The cache capacity in bytes.
P	$\in$	Policy	The set of replacement policies.
$addr$	$\in$	$Address \subseteq \mathbb{N}$	Set of memory addresses.
tag	$\in$	$Tag \subseteq \mathbb{N}$	Set of tags.
v, w	$\in$	$Way = \{0, \ldots, A-1\}$	Set of cache ways.
i	$\in$	$Index = \{0, \ldots, N-1\}$	Set of indices.

Fig. 1: Parameters and basic domains.

II. Cache Organization

In this section we introduce caches and their parameters intuitively and formally. The formalization allows us to precisely state the problem in the following section.

A. Intuition and Parameters

Caches are fast but small memories that store a subset of the main memory's contents to bridge the latency gap between CPU and main memory. To profit from spatial locality and to reduce management overhead, main memory is logically partitioned into a set of *memory blocks* of *block size* B. Blocks are cached as a whole in cache lines of the same size. Usually, the block size is a power of two. This way, the block number is determined by the most significant bits of a memory address, more generally: $\mathrm{block}_B(address) = \lfloor address/B \rfloor$.

When accessing a memory block, the cache logic has to determine whether the block is stored in the cache ("cache hit") or not ("cache miss"). To enable an efficient look-up, each block can only be stored in a small number of cache lines. For this purpose, caches are partitioned into N equally-sized *cache sets*. The size of a cache set is called the *associativity* A of the cache. A cache with associativity A is often called A-*way* set-associative. It consists of A *ways*, each of which consists of one cache line in each cache set. In the context of a cache set, the term *way* thus refers to a single cache line. Usually, also the number of cache sets N is a power of two such that the set number, also called *index*, is determined by the least significant bits of the block number. More generally: $\mathrm{index}_{B,N}(address) = \mathrm{block}_B(address) \bmod N$. The remaining bits of an address are known as the *tag*: $\mathrm{tag}_{B,N}(address) = \lfloor \mathrm{block}_B(address)/N \rfloor$. To decide whether and where a block is cached within a set, tags are stored along with the data.

Since the number of memory blocks that map to a set is usually far greater than the associativity of the cache, a so-called *replacement policy* must decide which memory block to replace upon a cache miss. Replacement policies try to exploit temporal locality and base their decisions on the history of memory accesses. Usually, cache sets are treated independently of each other such that accesses to one set do not influence replacement decisions in other sets. Well-known replacement policies in this class are least recently used (LRU); pseudo-LRU (PLRU), a cost-efficient variant of LRU; first-in first-out (FIFO), also known as Round Robin; and not-most-recently used (NMRU) [8]. An exception to the rule of treating cache sets independently, used in several ARM and Motorola processors [9], is Pseudo Round Robin, which maintains a single "FIFO pointer" for all cache sets.

B. Formalization of Caches and a Cache Template

A cache can be modeled as a 4-tuple $C = (\mathrm{CacheState}_C, s^0_C, \mathrm{up}_C, \mathrm{hit}_C)$, where $\mathrm{CacheState}_C$ is the cache's set of states, s^0_C is its initial state, $\mathrm{up}_C : \mathrm{CacheState}_C \times Address \to \mathrm{CacheState}_C$ models the change in state upon a memory access, and $\mathrm{hit}_C : \mathrm{CacheState}_C \times Address \to \mathbb{B}$, determines whether a memory access results in a cache hit or a cache miss.

Similarly, a replacement policy can be modeled as a tuple $P = (\mathrm{PolState}_P, s^0_P, \mathrm{evict}_P, \mathrm{up}_P)$, where $\mathrm{PolState}_P$ is the set of states of the policy, $s^0_P \in \mathrm{PolState}_P$ is the initial state of the policy, $\mathrm{evict}_P : \mathrm{PolState}_P \to Way$ determines which memory block to evict upon a cache miss, and $\mathrm{up}_P : \mathrm{PolState}_P \times (Way \cup \{miss\}) \to \mathrm{PolState}_P$ computes the change in state upon a cache hit to a particular cache way or upon a miss. Parameters and basic domains used here and in the following are introduced in Figure 1.

Caches considered in this paper are fully determined by the four parameters *associativity* A, *block size* B, *number of cache sets* N, and *replacement policy* P introduced above. Specifically, they are instances of the *cache template* T, which maps each parameter combination to a particular *cache*:

$$T(A, B, N, P) := (\mathrm{CacheState}_T, s^0_T, \mathrm{up}_T, \mathrm{hit}_T),$$

whose components are introduced in a top-down fashion below. $\mathrm{CacheState}_T$ consists of a cache set state (defined in more detail later) for each index:

$$\mathrm{CacheState}_T := Index \to \mathrm{SetState}.$$

The initial state s^0_T maps each index to the initial cache set state: $s^0_T := \lambda i.s^0_{set}$.

The cache update delegates an access to the appropriate cache set; similarly, whether an access is a hit or a miss is determined by querying the appropriate cache set:

$$\begin{aligned} \mathrm{up}_T(cs, addr) &:= cs[index \mapsto \mathrm{up}_{set}(cs(index), tag)], \\ \mathrm{hit}_T(cs, addr) &:= \mathrm{hit}_{set}(cs(index), tag), \end{aligned}$$

where $index = \mathrm{index}_{B,N}(addr)$ and $tag = \mathrm{tag}_{B,N}(addr)$. We denote by $f[i \mapsto j]$ the function that maps i to j and agrees with the function f elsewhere.

States of *cache sets* consist of the blocks that are being cached, modeled by a mapping assigning to each *way* of the cache set the *tag* of the block being cached (or $\perp$, if the cache line is invalid), and additional state required by the *replacement policy* P to decide which block to evict upon a cache miss:

$$\mathrm{SetState} := (Way \to (Tag \cup \{\perp\})) \times \mathrm{PolState}_P.$$

Initially, cache sets are empty and the replacement policy is in its initial state, thus $s^0_{set} := \langle \lambda w.\perp, s^0_P \rangle$. An access to a set constitutes a hit, if the requested tag is contained in the cache: $\mathrm{hit}_{set}(\langle bs, ps \rangle, tag) := \exists i : bs(i) = tag$.

The cache set update needs to handle two cases: cache misses, where the replacement policy determines which line to evict, and cache hits, where the accessed way determines how to update the state of the replacement policy:

$$\mathrm{up}_{set}(\langle bs, ps \rangle, tag) := \begin{cases} \langle bs[w \mapsto tag], \mathrm{up}_P(ps, miss) \rangle & : \text{if } \forall v : bs(v) \neq tag \\ \langle bs, \mathrm{up}_P(ps, v) \rangle & : \text{else if } bs(v) = tag \end{cases}$$

where $w = \mathrm{evict}_P(ps)$.

Clearly, the above model is not exhaustive, but just fine-grained enough for our purpose. One of the limitations is that reads are not distinguished from writes.

III. Problem Statement: The Cache Parameter Inference Problem

In order to state our goal more precisely, we first need to discuss by which means inference algorithms can gain information about the cache.

A. What can be measured?

Inference algorithms can neither observe nor directly control the internal state or operation of the cache. They can, however, use hardware performance counters to determine the number of cache misses incurred by a sequence of memory operations.

Let $\mathrm{misses}_C(s, \vec{a})$ denote the number of misses that the cache C incurs performing the sequence of memory accesses $\vec{a}$ starting in cache state s. For brevity, we omit the precise definition of $\mathrm{misses}_C(s, \vec{a})$ in terms of up_C and hit_C, which could easily be given.

An inference algorithm cannot control the cache state at the beginning of its execution. This introduces nondeterminism when measuring the number of misses incurred by a sequence of memory accesses:

$$\textbf{measure}_C(\vec{a}) = \{\mathrm{misses}_C(s, \vec{a}) \mid s \in \mathrm{CacheState}_C\}.$$

By performing "preparatory" memory accesses $\vec{p}$, aimed at establishing a particular cache state, before starting to measure the number of misses, an inference algorithm can often extract more information about the cache:

$$\textbf{measure}_C(\vec{p}, \vec{m}) = \{\mathrm{misses}_C(\mathrm{up}_C(s, \vec{p}), \vec{m}) \mid s \in \mathrm{CacheState}_C\},$$

where up_C is lifted from individual memory accesses to sequences. The preparatory memory accesses may reduce or even eliminate the nondeterminism from the measurement.

Later on, in pseudo code, we will use $\textbf{measure}_C(\vec{p}, \vec{m})$ as an expression that will nondeterministically evaluate to one of the values of the set defined above.

B. What can be inferred?

Two caches may exhibit the same hit/miss behavior but differ internally, e.g. in how they implement a replacement policy. For instance, there are many different realizations of LRU replacement.

Since only a cache's hit/miss behavior can be observed, it is impossible to infer anything about how it is realized internally. For the purpose of modeling a cache's behavior such aspects are irrelevant anyway. The following definition captures when we consider two caches to be *observationally equivalent*:

Definition 1 (Observational Equivalence of Caches)**.** *Caches C and D are observationally equivalent, denoted $C \equiv D$, iff $\mathrm{misses}_C(s^0_C, \vec{a}) = \mathrm{misses}_D(s^0_D, \vec{a})$ for all memory access sequences $\vec{a} \in Address^*$. Otherwise, we call two caches observationally different.*

Now we are ready to more precisely state the problem tackled in this paper:

Definition 2 (Cache Parameter Inference Problem)**.** *The cache parameter inference problem is to derive parameter values $A, B, N,$ and P through measurements on an implementation of a cache C, such that $T(A, B, N, P) \equiv C$.*

IV. A Class of Replacement Policies: Permutation Policies

Parameter inference algorithms work as follows. They perform a predetermined set of measurements[2]. Each measurement result excludes some of the parameter values, e.g., a measurement result may imply that the associativity of the cache is greater than 3, excluding associativities 1, 2, and 3. The set of measurements needs to be designed in such a way that for any possible outcome at most one parameter value will be compatible with all measurement results. For this to be possible, we need to a priori restrict the set of parameter values to a finite set. For the first three parameters, associativity, block size, and number of sets, this can be done easily by fixing a maximal value for each of the parameters.

As replacement policies are implemented in hardware, a simple and realistic assumption is that their set of states is finite. Putting a bound b on the number of states of the policy yields a finite set of observationally different replacement policies. Unfortunately, however, b needs to be at least the factorial of the associativity, which is the minimal number of states to implement LRU. For such a b, there are too many observationally different replacement policies for inference to be practically feasible.

[2]Adaptive inference algorithms that base later measurements on earlier measurement results are also conceivable. The core of the following argument applies to them as well.

$\Pi_0^{\text{LRU}} = (0,1,2,3,4,5,6,7)$
$\Pi_1^{\text{LRU}} = (1,0,2,3,4,5,6,7)$
$\Pi_2^{\text{LRU}} = (2,0,1,3,4,5,6,7)$
$\Pi_3^{\text{LRU}} = (3,0,1,2,4,5,6,7)$
$\Pi_4^{\text{LRU}} = (4,0,1,2,3,5,6,7)$
$\Pi_5^{\text{LRU}} = (5,0,1,2,3,4,6,7)$
$\Pi_6^{\text{LRU}} = (6,0,1,2,3,4,5,7)$
$\Pi_7^{\text{LRU}} = (7,0,1,2,3,4,5,6)$

(a) LRU

$\Pi_0^{\text{PLRU}} = (0,1,2,3,4,5,6,7)$
$\Pi_1^{\text{PLRU}} = (1,0,3,2,5,4,7,6)$
$\Pi_2^{\text{PLRU}} = (2,1,0,3,6,5,4,7)$
$\Pi_3^{\text{PLRU}} = (3,0,1,2,7,4,5,6)$
$\Pi_4^{\text{PLRU}} = (4,1,2,3,0,5,6,7)$
$\Pi_5^{\text{PLRU}} = (5,0,3,2,1,4,7,6)$
$\Pi_6^{\text{PLRU}} = (6,1,0,3,2,5,4,7)$
$\Pi_7^{\text{PLRU}} = (7,0,1,2,3,4,5,6)$

(b) PLRU

$\Pi_0^{\text{FIFO}} = (0,1,2,3,4,5,6,7)$
$\Pi_1^{\text{FIFO}} = (0,1,2,3,4,5,6,7)$
$\Pi_2^{\text{FIFO}} = (0,1,2,3,4,5,6,7)$
$\Pi_3^{\text{FIFO}} = (0,1,2,3,4,5,6,7)$
$\Pi_4^{\text{FIFO}} = (0,1,2,3,4,5,6,7)$
$\Pi_5^{\text{FIFO}} = (0,1,2,3,4,5,6,7)$
$\Pi_6^{\text{FIFO}} = (0,1,2,3,4,5,6,7)$
$\Pi_7^{\text{FIFO}} = (0,1,2,3,4,5,6,7)$

(c) FIFO

Fig. 2: Permutation vectors for LRU, PLRU, and FIFO at associativity 8.

A. Permutation Policies

We have identified what we call *permutation policies* as an interesting finite (for a fixed associativity) set of replacement policies for which inference is practically feasible.

Permutation policies maintain an order on the blocks stored in a cache set. Such an order can be represented by a permutation π of the set of cache ways. Then, $\pi(i)$ determines which cache way stores the ith element in this order. Upon a miss, the last element in the order, i.e., the block in cache way $\pi(A-1)$ is evicted. In LRU, for example, $\pi(0)$ and $\pi(A-1)$ determine the cache ways that store the most- and the least-recently-used blocks, respectively.

Permutation policies differ in how they update this order upon cache hits and misses. In LRU, upon a cache hit, the accessed block is brought to the top of the order, whereas in FIFO, cache hits do not alter the order at all. More complex updates happen in case of, e.g., PLRU. The update behavior upon hits and misses of a policy can be specified using a permutation vector $\Pi = \langle \Pi_0, \ldots, \Pi_{A-1}, \Pi_{\text{miss}} \rangle$. Here, permutation Π_i determines how to update the order upon an access to the i^{th} element of the order, and Π_{miss} determines how to update the order upon a cache miss. Then, the new order π' is $\pi \circ \Pi_i$ or $\pi \circ \Pi_{\text{miss}}$, respectively. By $\pi \circ \Pi_i$ we denote the functional composition of π and Π_i defined by $(\pi \circ \Pi_i)(x) = \pi(\Pi_i(x))$ for all x.

Upon a cache miss, all policies we have come across move the accessed block to the top of the order and shift all other blocks one position down. In the following, we thus fix Π_{miss} to be $(A-1, 0, 1, \ldots, A-2)$. Essentially, this implies that upon consecutive cache misses a policy fills all of the ways of a set and evicts blocks in the order they were accessed.

The following *permutation policy template* PT formalizes how a permutation vector Π defines a replacement policy:

$$PT(\Pi) = (\text{PolState}_\Pi, s^0_\Pi, \text{evict}_\Pi, \text{up}_\Pi),$$

where

$$\begin{aligned}
\text{PolState}_\Pi &:= \{\pi : Way \to Way \mid \pi \text{ is a permutation}\},\\
s^0_\Pi &:= id_{Way} = \lambda i \in Way.i,\\
\text{evict}_\Pi(\pi) &:= \pi(A-1),\\
\text{up}_\Pi(\pi, w) &:= \begin{cases} \pi \circ \Pi_{\text{miss}} & : \text{if } w = \text{miss},\\ \pi \circ \Pi_{\pi^{-1}(w)} & : \text{otherwise.} \end{cases}
\end{aligned}$$

Fixing the miss permutations as described above has an additional benefit: two permutation policies $PT(\Pi)$ and $PT(\Psi)$ with $\Pi_{\text{miss}} = \Psi_{\text{miss}}$ are *observationally equivalent*[3] if and only if $\Pi_i = \Psi_i$ for all $i \in \{0, \ldots, A-1\}$. That is, once the miss permutation is fixed there is a unique representation of each permutation policy. Without this restriction, there would be observationally equivalent policies defined by different permutation vectors.

Thus, the set of permutation policies has the following three beneficial properties:

1) It includes well-known replacement policies, such as LRU, FIFO, and PLRU (as demonstrated in Figure 2 for associativity 8), in addition to a large set of so far undocumented policies.
2) For a fixed miss permutation, each permutation policy has a unique representation, which enables easy identification of known policies.
3) For all "realistic" associativities the set is sufficiently small for efficient inference.

B. Logical Cache Set States

Cache set states as defined in Section II-B model the contents of the physical ways of the cache in addition to the state of the replacement policy. Several such cache set states may exhibit the same replacement behavior on any possible future access sequence. As an example, consider the two cache set states $\langle [0 \mapsto a, 1 \mapsto b], (0,1) \rangle$ and $\langle [0 \mapsto b, 1 \mapsto a], (1,0) \rangle$ of a two-way set-associative cache managed by a permutation policy. In both states, the order on the blocks maintained by the policy is a then b, and this order will change in the same way upon future memory accesses.

Logical cache set states abstract from the physical location of cache contents, distinguishing two states if and only if they differ in their possible future replacement behavior. Permutation policies give rise to a particularly simple domain of logical cache set states, ordering cache contents according to the state of the permutation policy:

$$\text{LogicalSetState} := Way \to (Tag \cup \{\bot\}).$$

A cache set state of a permutation policy can be transformed into a logical cache set state as follows:

[3]The notion of observational equivalence is transferred from caches to replacement policies in the expected way.

$logical(\langle bs, \pi \rangle) := bs \circ \pi$. Both of the above example cache set states map to the same logical state $[0 \mapsto a, 1 \mapsto b]$. For brevity, in the following, we will denote such a state by $[a, b]$. We say that a is in position 0, and b is in position 1.

V. Replacement Policy Inference Algorithm

We focus our presentation on the replacement policy inference algorithm. We have also developed algorithms to determine associativity, way size, and block size that are similar to those presented by Yotov et al. [10]. More details on these algorithms can be found in Andreas Abel's Master's thesis [11] and a workshop contribution [12].

Algorithm 1: Naive implementation of replacement policy inference algorithm.

```
Input: A ← Associativity
       B ← Block Size
       N ← Number of Sets
       W ← B · N (= Way Size)

procedure initializeBasePointers()
    emptyBase ← allocate(3 · A · W)
    initBase ← emptyBase + A · W
    evictBase ← initBase + A · W
seq function emptyCacheSet(int set)
    return ⟪emptyBase + set · B, W, A⟫
seq function initializeSet(int set)
    return ⟪initBase + set · B + (A − 1) · W, −W, A⟫
seq function accessBlockInSet(int block, int set)
    return ⟨initBase + set · B+block·W⟩
seq function evictKBlocksInSet(int k, int set)
    return ⟪evictBase + set · B, W,k⟫

int function
      newPosOfBlockInPerm(int block, int perm)
    initializeBasePointers()
    empty ← emptyCacheSet(0)
    init ← initializeSet(0)
    accPerm ← accessBlockInSet(perm, 0)
    accBlock ← accessBlockInSet(block, 0)
    for newPos ← A − 1 downto 0 do
        evictk ←
            evictKBlocksInSet(A−newPos, 0)
        prep ← empty∘init∘accPerm∘evictk
        if measure_C( prep, accBlock) = 1 then
            return newPos
```

A. Intuitive Description of Algorithm

Our algorithm determines the permutation vector defining a permutation policy one permutation at at time. To determine permutation i, we execute the following three steps:

1) Establish a known logical cache set state.
2) Trigger permutation i by accessing the i^{th} element of the logical cache set state.
3) Read out the resulting logical cache set state.

Then, relating the state established in step 1 with the state determined in step 3 yields permutation i.

Given a known logical cache set state $[a_0, \ldots, a_{A-1}]$, step 2 simply amounts to accessing address a_i. But how do we establish such a state?

a) Establishing a known logical cache set state: Given the fixed miss permutation, as described in the previous section, we can establish a desired logical state, simply by causing a sequence of cache misses to the particular cache set. Accessing the sequence $\langle a_{A-1}, \ldots, a_0 \rangle$ consisting of addresses $a_0, \ldots, a_{A-1}$ mapping to the same cache set, results in the logical cache set state $[a_0, \ldots, a_{A-1}]$, provided that all of the accesses result in cache misses.

b) Reading out a logical cache set state: We need to determine the position of each block a_k of the original logical state in the resulting logical state. We do so by a sequence of checks that determine whether the block's new position is greater than j, for $j \in \{0, \ldots, A-1\}$.

By accessing block a_k and comparing the performance counters before and after the access, we can—at least in theory—determine whether the access caused a miss, and thus whether a_k was cached or not. By causing $A - j$ additional cache misses before, we can determine whether the block's position was greater than j or not.

Each such check destroys the cache state. Thus, before each check, the state before the measurement needs to be reestablished by going through steps 1 and 2 again.

B. A Naive Implementation

Algorithm 1 shows a naive implementation of the function to determine a block's new position after triggering a particular permutation. To compactly represent strided access sequences, we use $\langle\langle base, stride, count \rangle\rangle$ to denote the access sequence $\langle base, base + 1 \cdot stride, \ldots, base + (count - 1) \cdot stride \rangle$. We first introduce five helper functions:

- `initializeBasePointers()` initializes the three pointers, *emptyBase*, *initBase*, *evictBase*, which are maintained in global variables. *emptyBase*, *initBase*, and *evictBase* are used to access disjoint memory areas of the size of the cache. Note that the three pointers map to the same cache set.
- `emptyCacheSet(`**int** *set*`)` generates an access sequence used to evict previous contents from a given cache set, where cache set 0 is defined to be the set that *emptyBase* happens to map to.
- `initializeSet(`**int** *set*`)` generates an access sequence to establish a known logical state in set *set*. As the addresses in this sequence are disjoint from those in the sequence produced by `emptyCacheSet(`**int** *set*`)`, and both sequences entirely fill the cache, both sequences will never produce any cache hits.
- `accessBlockInSet(`**int** *block,* **int** *set*`)` generates a singleton access sequence to the $block^{th}$ element of the logical state created by `initializeSet(`*set*`)`.
- `evictKBlocksInSet(`**int** *k,* **int** *set*`)` generates a sequence of k memory accesses to set *set* from the memory area pointed to by *evictBase*. Accessing this sequence will cause k misses in the desired set.

The loop in `newPosOfBlockInPerm` performs successive measurements to determine the position of the $block^{th}$ element of the logical state after triggering permutation $perm$. It uses the above helper functions to generate a sequence *prep*, which empties the cache set, establishes a known logical state, triggers permutation $perm$, and finally evicts a number of blocks from the set. Here $\circ$ denotes the concatenation operator. This sequence is used to check whether or not the new position (after triggering the permutation) of the $block^{th}$ element of the logical cache state was greater than $newPos$. In principle, one could perform a binary search of the new position, slightly improving the algorithm's complexity, but for simplicity we stick to the linear algorithm as typical associativities are small.

In an ideal world with perfect measurement ability and no interference on the cache the above algorithm would work. However, as in physics, measurements through the performance counter library PAPI [13] (discussed in Section V-D1) disturb the state of what is being measured. In particular, PAPI methods need to be invoked between the preparation and the measurement phase, which may alter the cache contents, and cause cache misses of its own. Further, while performance counters count events on a per-process basis, other processes running concurrently may alter the cache contents and thus increase the measured number of cache misses. As a result, we cannot reliably determine for a single access whether it results in a cache hit or a miss. We thus need to increase its robustness to such disturbance in the measurements.

C. A More Robust Implementation

The first idea to improve robustness is to perform memory accesses in all N cache sets instead of just one. This way, the measurement needs to distinguish between 0 or N misses rather than between 0 or 1 misses.

A simple way of realizing this is to create an interleaving of N copies of the original access sequence, replacing every access in the original sequences by equivalent accesses to all cache sets, e.g., $\langle a, b\rangle$ would be replaced by $\langle a, a+B, a+2\cdot B, \ldots, a+(N-1)\cdot B, b, b+B, b+2\cdot B, \ldots, b+(N-1)\cdot B\rangle$. Let *prep*′ and *accBlock*′ be the result of interleaving several copies of *prep* and *accBlock*, respectively. Then, replacing the condition $\textbf{measure}_C(prep, accBlock) = 1$ by $\textbf{measure}_C(prep', accBlock') \geq N$, already yields a much more robust algorithm.

Incidentally, the above approach also reduces the effect of hardware-based prefetching. Prefetchers try to detect and exploit some form of regularity in the access pattern, and may thus introduce memory accesses that are not present in the program, as well as changing the order of memory accesses that are. Clearly such additional accesses may affect our measurement results. The interleaving of access sequences introduced above results in a very regular access pattern easy to correctly predict by common prefetching mechanisms.

As discussed earlier, the execution of PAPI methods between the preparatory and the measurement phase may disturb the results. To avoid this disturbance, we replace $\textbf{measure}_C(empty' \circ init' \circ accPerm' \circ evictk', accBlock')$ by $\textbf{measure}_C(empty', init' \circ accPerm' \circ evictk' \circ accBlock') - \textbf{measure}_C(empty', init' \circ accPerm' \circ evictk')$. In an ideal setting, this does not change the outcome, as the execution of *empty*′ results in an empty cache, and the following execution of *init*′∘*accPerm*′∘*evictk*′ causes the same number of misses in both measurements. The advantage is, that now, the measurement routines are called when the cache contains data that will not be accessed in the following. The disturbance caused by the measurement routines does not invalidate this invariant. In addition, taking the difference between two measurements immediately eliminates any constant overhead incurred by the measurements.

To further reduce the likelihood of interference affecting the result, we repeat the measurements several times and only take the minimum of the measured values into account.

D. Implementing $\textbf{measure}_C(\vec{p}, \vec{m})$

How do we implement $\textbf{measure}_C(\vec{p}, \vec{m})$? This really amounts to two questions: 1. How to measure the number of incurred cache misses? 2. How to implement the access sequences $\vec{p}$ and $\vec{m}$?

1) Measuring the Number of Cache Misses:

a) Performance Counters: Hardware Performance Counters are special-purpose registers that are used to count the number of various hardware-related events, including the number of hits and misses in different levels of the memory hierarchy. They are available on many modern processors. The "Performance Application Programming Interface" (PAPI) [13] provides a common interface to access these counters on a number of different platforms, including all recent Intel and AMD x86 processors.

b) Measuring Execution Time: Another approach is to instead measure the execution time of an access sequence. Since a cache miss takes more time than a cache hit, this can be used to estimate the number of hits/misses. This approach has the advantage of being available on virtually all platforms. However, there are a number of caveats:

- A processor might not offer a timer that is precise enough to capture the time differences when executing very short code fragments.
- Modern CPUs feature non-blocking caches that can serve other requests while fetching the data for a miss.
- Misses in different levels of the memory hierarchy have different latencies making it challenging to analyze a specific part of the memory hierarchy.

In our implementation, we leverage hardware performance counters using PAPI, whenever they are available, because of the higher accuracy they offer. If performance counters are unavailable, to support as many platforms as possible, we fall back to variations of our algorithms that use execution-time measurements. Unfortunately, due to space limitations we cannot discuss these in detail here.

2) Implementing Access Sequences:

a) Data Caches: To minimize the effect of non-blocking caches and out-of-order execution, we serialize memory accesses by using a form of "pointer chasing" where each memory location contains the address of the next access.

The following code snippet gives an idea of our implementation. We assume that $base$ is an array that contains such a sequence of memory addresses, starting at the offset $start$.

```
register char* cur=(char*)(base+start);
while (cur!=0)
  cur=*(char**)cur;
```

b) Instruction Caches: To analyze instruction caches of x86 processors, we created a function, that, given an array of memory addresses, allocates memory, declares it executable, and populates it with a sequence of jump instructions that will cause the desired instruction fetches. This approach could easily be adapted to other instruction sets such as ARM instruction sets, which are more prevalent in the embedded world.

In particular, an **unsigned char** array is filled with the machine instructions for a function call, a sequence of jumps and finally a return instruction, as illustrated in the following excerpt from our code:

```
code[0]=0x55; //push %rbp
    ⋮
code[40]=0xc9; //leaveq
code[41]=0xc3; //retq
```

The code can then be called with the statement ((**void**(∗)(**void**))code)(); Special care needs to be taken when a memory address is to be accessed twice.

c) Second-level Caches: We assume a *non-inclusive* cache hierarchy, so that the second-level cache can be analyzed independently of the first-level cache. In such a cache hierarchy, an access that misses in the first-level cache is passed to the second-level cache. Exclusive and strictly-inclusive caches feature more complicated interactions between the first- and the second-level cache and are thus be more difficult to analyze. Further, our implementation is based on the assumption that the way size of the second-level cache is larger than the size of the first-level cache, which was the case for all second-level caches under consideration. Then, between two accesses to the same set of the second-level cache, our algorithm accesses all other cache sets. So all memory accesses lead to misses in the first-level cache and are thus passed to the second-level cache.

d) Virtual Memory: Most current platforms have physically-indexed L2 caches, whereas L1 caches are virtually-indexed. As the way size of these caches is usually larger than the page size, consecutive virtual addresses need not map to consecutive cache sets, and virtual addresses that are way size-apart need not map to the same cache set. We deal with this problem by allocating huge pages[4]. This allows us to allocate physically-contiguous memory areas that are significantly larger than the standard page size of 4 kB and usually a multiple of the way size of large caches. As a consequence, we can deal with both physically- and virtually-indexed L1 and L2 caches.

e) Eliminating Interference Constructively: The reader might wonder why we have put so much emphasis on making our implementation robust to interference. Many of the sources of interference our implementation needs to deal with could be eliminated:

[4]For information on huges pages see http://en.wikipedia.org/wiki/Page_size.

$$\Pi_0^{\text{ATOM}} = (0,1,2,3,4,5)$$
$$\Pi_1^{\text{ATOM}} = (1,0,2,4,3,5)$$
$$\Pi_2^{\text{ATOM}} = (2,0,1,5,3,4)$$
$$\Pi_3^{\text{ATOM}} = (3,1,2,0,4,5)$$
$$\Pi_4^{\text{ATOM}} = (4,0,2,1,3,5)$$
$$\Pi_5^{\text{ATOM}} = (5,0,1,2,3,4)$$

Fig. 3: Permutation vectors for Intel Atom D525 policy.

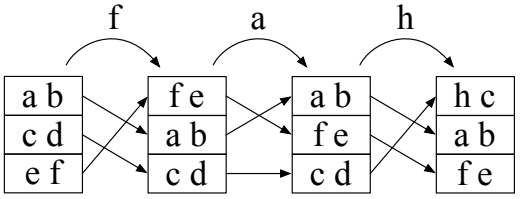

Fig. 4: Intuitive description of Intel Atom D525 policy.

- Disabling caching for PAPI methods would eliminate interference in the cache from the execution of these methods.
- Flushing the cache with a special cache flush instruction prior to our experiments would eliminate variability due to the initial cache state.
- Running our tool in single-user mode would reduce, but not eliminate, interference from processes running concurrently.
- Prefetching can often be disabled.

These measures are all valid and *should* be applied if possible. We refrained from implementing these measures as part of our implementation, however, as they would reduce its portability, which was one of the main design goals. As our experimental evaluation shows, portability did not come at the cost of a decrease in inference precision.

VI. Experimental Evaluation

Using our algorithms, we were able to successfully determine the replacement policies of a number of different x86 CPUs that were introduced in the last twelve years. Table I shows the results for first-level instruction and data caches as well as for unified second-level caches. We could confirm the inferred sizes and associativities from publicly available documents. For the replacement policies, less precise documentation is available. Intel characterizes their replacement policies as pseudo-LRU. Our inference is consistent with this characterization, while providing an *exact* specification of the policies' logical behaviors.

A. Intel Atom D525 Replacement Policy

The Intel Atom D525 CPU features a 24 kB L1 data cache with associativity 6. Using our approach, we obtained the permutation vector shown in Figure 3 for its L1 replacement policy. We were not able to find any detailed information about the replacement policies used in Intel Atom CPUs in the

TABLE I: Results of replacement policy inference.

Architecture	L1 Data			L1 Instruction			L2 Unified		
	Size	Assoc.	Policy	Size	Assoc.	Policy	Size	Assoc.	Policy
Intel Atom D525	24kB	6	see Section VI-A	32kB	8	PLRU	512kB	8	PLRU
Intel Pentium 3 900	16kB	4	PLRU	16kB	4	PLRU	256kB	8	PLRU
Intel Core 2 Duo E6300	32kB	8	PLRU	32kB	8	PLRU	2048kB	8	PLRU
Intel Core 2 Duo E6750	32kB	8	PLRU	32kB	8	PLRU	4096kB	16	see Section VI-B1
Intel Core 2 Duo E8400	32kB	8	PLRU	32kB	8	PLRU	6144kB	24	see Section VI-B1
Intel Core i5 460M	32kB	8	PLRU	32kB	4	see Section VI-B3	256kB	8	PLRU
Intel Xeon W3550	32kB	8	PLRU	32kB	4	see Section VI-B3	256kB	8	PLRU
AMD Athlon 64 X2 4850e	64kB	2	LRU	64kB	2	LRU	512kB	16	see Section VI-B2
AMD Opteron 8360SE	64kB	2	LRU	64kB	2	LRU	512kB	16	see Section VI-B2

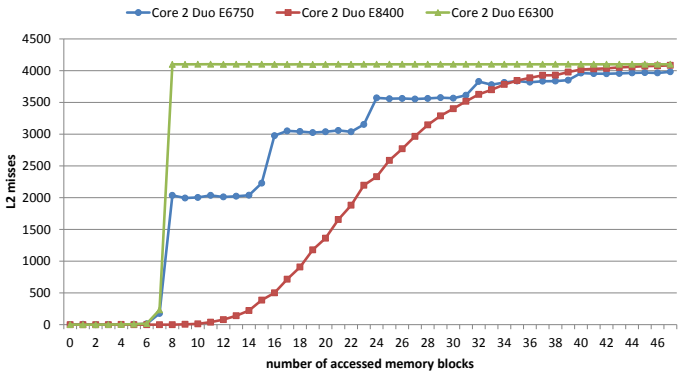

Fig. 5: Experimental analysis of L2 cache behavior of the Intel Core 2 Duo E6750, E6300 and E8400.

documentation or elsewhere, so to the best of our knowledge, this is the first publicly-available description of this policy.

Obviously, the policy is not a strict LRU policy but it approximates LRU. Previously described implementations of pseudo-LRU [8] were based on perfect binary trees and thus required the associativity to be a power of two. We have found the following intuitive description of its behavior, which is illustrated in Figure 4: The six ways of the cache are organized into three groups of two ways each. The policy maintains the following two invariants:

1) Within each group, the two ways are ordered by the recency of the last access to each of the two ways.
2) The three groups are ordered by the recency of the last access to one of their two ways.

Upon a cache hit, the order of the groups and within each group is updated to maintain the above invariants. Upon a cache miss, the least-recently-used block in the least-recently-used group is replaced.

Using RELACS[5], we have determined that this replacement policy is $(1, 0)$-competitive relative to LRU at associativity 4 [14]. This means that all static cache analyses previously developed for LRU can be immediately applied in WCET analyses of the Intel Atom D525.

[5]See http://rw4.cs.uni-saarland.de/~reineke/relacs for details on RELACS.

B. Difficulties

1) Intel Core 2 Duo E6750 and E8400: The L2 replacement policy could not be inferred by our algorithm on both an Intel Core 2 Duo E6750 (4 MB, 16-way set-associative) and an Intel Core 2 Duo E8400 (6 MB, 24-way set-associative), i.e., the new positions determined by `newPosOfBlockInPerm` did not form a permutation. However, on an Intel Core 2 Duo E6300 (2 MB, 8-way set-associative), the PLRU replacement policy was inferred. According to Intel, all of these CPUs "use some variation of a pseudo LRU replacement algorithm" [15]. To further investigate why our algorithm could not infer the policy of the two processors mentioned above, we designed an experiment, which:

1) Clears the L2 cache.
2) For each cache set, accesses one memory block that maps to this set.
3) Accesses n other memory blocks in each cache set.
4) Counts the L2 cache misses when accessing the memory blocks from step 2 again.

Under the PLRU policy, and all other permutation policies, we would expect to get zero misses if n is smaller than the associativity of the L2 cache and as many misses as there are cache sets otherwise. Figure 5 shows that this is indeed almost the case on the E6300. The slight jump at $n = 7$ is likely due to interfering memory accesses. However, on the other two Core 2 Duo machines, the results look different. On the E6750, the curve can roughly be modeled by the function $4096 \cdot \left(1 - \left(\frac{1}{2}\right)^{\lfloor n/8 \rfloor}\right)$, where 4096 is the number of cache sets. So far, we have not been able to find a conclusive explanation for this behavior.

2) AMD Athlon 64 X2 4850e and Opteron 8360SE: Both AMD CPUs have *exclusive* L2 caches with the same way size as their first-level caches. Thus, their replacement policy cannot be inferred by our current algorithm.

3) Intel Core i5 460M and Xeon W3550: Our implementation could not infer the replacement policies of the instruction caches of these two Nehalem-based processors. This might be due to the Micro-Op buffer first introduced in this architecture[6].

[6]For information about the Micro-Op buffer, see http://www.bit-tech.net/hardware/cpus/2008/11/03/intel-core-i7-nehalem-architecture-dive/5.

C. Experimental Setup

The experiments were performed on different versions of Ubuntu (with kernels $\geq$ 2.6.32), depending on what was already installed on the machines we examined. Aside from enabling support for huge pages, we did not perform any modifications to the operating system. In particular, we did not stop background processes or disable interrupts, which may be useful to reduce interference. However, our goal is for our implementation to be as robust as possible, so that it can be easily applied in any context. To get access to performance counter data, we used PAPI in version 4.2.1. The code was compiled with GCC at optimization level 0, to avoid compiler optimizations that might influence measurement results. The execution time of our algorithm was usually less than one minute.

VII. Related Work

A. Measurement of Cache Parameters

Several publications have presented approaches to determine parameters like the cache size, the associativity and the block size of data caches through measurements. Some of these approaches use hardware performance counters to perform the measurements [16], [17], [18] while the others use timing information [10], [19], [20], [21], [22], [23], [24].

Only few [19], [18], [23] have also analyzed these parameters for instruction caches. However, in contrast to our implementation, the approach described by Yotov et al. [19] generates and compiles C source code dynamically, thus requiring access to the compiler at runtime. On the other hand, [18] relies on specific features of the GCC compiler. Blanquer and Chalmers [23] do not give a detailed description of their implementation. A compiler in the loop may not be available on resource-constrained embedded systems, and is an additional source of potential problems, which is eliminated in our approach.

Some of these approaches make assumptions as to the underlying replacement policy (e.g. [25] and [22] assume that LRU replacement is used). However, only a few publications have tried to determine the cache replacement policies as well. The approaches described in [18] and [23] are able to detect LRU-based policies but treat all other policies as random. John and Baumgartl [16] use performance counters to distinguish between LRU and several of its derivatives. However, the implementation requires a special real-time operating system environment. Further, it does not rely on PAPI for the performance counters, and thus needs to be adapted to every different processor architecture. Additionally, they only consider data caches and assume the other cache parameters to be known and to be powers of two, which is not the case on several of the processors we used in our evaluation. While the above restrictions can be lifted rather easily, the main advantage of our work over John and Baumgartl's is its ability to discover previously *unknown* replacement policies, such as the one found in the Intel Atom D525. In John and Baumgartl's approach the set of replacement policies needs to be provided beforehand by the user, and the measurements need to be adapted *manually* to this set of policies and for *each* associativity.

B. Template-based Synthesis

Our work can be seen as a form of template-based synthesis [26], [27]. In the terms of Godefroid and Taly [27], our algorithm is based on *smart sampling*, as the set of measurements is independent of intermediate measurement results.

C. Machine Learning

Caches as defined in Section II-B define a formal language: a sequence of memory accesses is a member of the language if the final memory access in the sequence results in a cache hit, if the sequence is fed to the cache starting in its initial state. Thus it would be interesting to apply methods to learn formal languages.

Given an infinite set of memory addresses, caches are infinite-state systems. However, replacement policies as defined in Section II-B are finite-state systems. It thus might be possible to adapt Angluin's algorithm to learn regular languages [28] to our problem. Angluin's algorithm is based on membership and equivalence queries. It is conceivable but not immediately obvious that membership queries can be realized through measurements. Equivalence queries can be realized—at least probabilistically—using membership queries as described in [29].

The connection between caches and canonical register automata [30] is more immediate. We are thus exploring the use of Howar et al.'s technique to learn such automata [31]. The question is whether general automata-learning methods scale to the size of the state space of a replacement policy such as LRU.

VIII. Conclusions and Future Work

We have presented a novel algorithm and its implementation **chi** to infer the cache replacement policy of a processor by a series of measurements. Application to the Intel Atom D525 has revealed a previously undocumented approximation of LRU. It should be straightforward to port our implementation to ARM or PowerPC instruction sets, which are more prevalent in embedded systems. Further, our work has raised a number of challenges and questions to tackle in the future:

- Can we find a more general class of replacement policies that still admits efficient inference?
- What are the replacement policies employed in the Intel Core 2 Duo E6750 and E8400?
- With some effort it should be possible to extend our approach to inclusive and exclusive L2 caches such as the one in the AMD Opteron 8360SE. After the L1 cache has been inferred, knowledge about its behavior can be exploited to systematically direct memory accesses at the L2 cache.
- What about trace and victim caches?
- Shared caches in multi-core processors feature various coherency protocols. It would be interesting to automatically characterize these precisely.

We also plan to extend our work to other architectural features, such as translation lookaside buffers, branch predictors and prefetchers.

WCET analysis based on abstract interpretation requires good abstractions of the memory hierarchy. Instead of inferring a concrete model of a cache it might be possible to instead directly infer a valid abstraction of the cache's behavior. This may be possible for classes of replacement policies for which efficient inference of concrete models is impossible.

ACKNOWLEDGMENTS

We would like to thank Peter Backes, Daniel Grund, Claire Maiza, Christoph Mallon, and Pascal Raymond for insightful discussions on this paper and for assistance in deploying our implementation on a variety of machines. This work was supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See http://www.avacs.org/ for more information.

REFERENCES

[1] R. Wilhelm *et al.*, "The worst-case execution time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, 2008. [Online]. Available: http://dx.doi.org/10.1145/1347375.1347389

[2] M. T. Yourst, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator," in *ISPASS*, 2007, pp. 23–34. [Online]. Available: http://dx.doi.org/10.1109/ISPASS.2007.363733

[3] R. Clint Whaley, A. Petitet, and J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1, pp. 3–35, 2001. [Online]. Available: http://dx.doi.org/10.1016/S0167-8191(00)00087-9

[4] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel, "Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology," in *Proceedings of the 11th international conference on Supercomputing*. ACM, 1997, pp. 340–347. [Online]. Available: http://doi.acm.org/10.1145/263580.263662

[5] M. Frigo and S. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005. [Online]. Available: http://dx.doi.org/10.1109/JPROC.2004.840301

[6] K. Cooper *et al.*, "The platform-aware compilation environment, preliminary design document," Department of Computer Science, Rice University, Tech. Rep., September 2010. [Online]. Available: http://pace.rice.edu/uploadedFiles/Publications/PACEDesignDocument.pdf

[7] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud," in *Proceedings of the 21st USENIX Security Symposium*. USENIX Association, 2012, pp. 11–26. [Online]. Available: http://dl.acm.org/citation.cfm?id=2362793.2362804

[8] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic, "Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite," in *Proceedings of the 42nd annual Southeast regional conference*, New York, NY, USA, 2004, pp. 267–272. [Online]. Available: http://doi.acm.org/10.1145/986537.986601

[9] D. Grund, "Static cache analysis for real-time systems – LRU, FIFO, PLRU," Ph.D. dissertation, Saarland University, 2012.

[10] K. Yotov, K. Pingali, and P. Stodghill, "Automatic measurement of memory hierarchy parameters," in *SIGMETRICS*. New York, NY, USA: ACM, 2005, pp. 181–192. [Online]. Available: http://doi.acm.org/10.1145/1064212.1064233

[11] A. Abel, "Measurement-based inference of the cache hierarchy," Master's thesis, Saarland University, 2012. [Online]. Available: http://embedded.cs.uni-saarland.de/literature/AndreasAbelMastersThesis.pdf

[12] A. Abel and J. Reineke, "Automatic cache modeling by measurements," in *6th Junior Researcher Workshop on Real-Time Computing (in conjunction with RTNS)*, November 2012. [Online]. Available: http://embedded.cs.uni-saarland.de/literature/CacheModelingJRWRTC.pdf

[13] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," in *Proceedings of the DoD HPCMP Users Group Conference*, 1999, pp. 7–10. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.117.6801

[14] J. Reineke and D. Grund, "Relative competitive analysis of cache replacement policies," in *LCTES*. ACM, 2008, pp. 51–60. [Online]. Available: http://doi.acm.org/10.1145/1375657.1375665

[15] R. Singhal, Personal communication, Intel, August 2012.

[16] T. John and R. Baumgartl, "Exact cache characterization by experimental parameter extraction," in *Proceedings of the 15th international conference on Real-Time and Network Systems*, Nancy, France, 2007, pp. 65–74. [Online]. Available: http://hal.inria.fr/docs/00/16/85/30/PDF/actes.pdf

[17] J. Dongarra, S. Moore, P. Mucci, K. Seymour, and H. You, "Accurate cache and TLB characterization using hardware counters," in *Proceedings of the international conference on Computational Science*. New York, NY, USA: Springer, 2004, pp. 432–439. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24688-6_57

[18] C. Coleman and J. Davidson, "Automatic memory hierarchy characterization," in *ISPASS*, 2001, pp. 103–110. [Online]. Available: http://dx.doi.org/10.1109/ISPASS.2001.990684

[19] K. Yotov *et al.*, "Automatic measurement of instruction cache capacity," in *Proceedings of the 18th international workshop on Languages and Compilers for Parallel Computing*. Springer, 2006, pp. 230–243. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69330-7_16

[20] S. Manegold, "The calibrator (v0.9e), a cache-memory and TLB calibration tool," http://homepages.cwi.nl/~manegold/Calibrator/, June 2004.

[21] V. Babka and P. Tůma, "Investigating cache parameters of x86 family processors," in *Proceedings of the 2009 SPEC benchmark workshop*. Springer, 2009, pp. 77–96. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-93799-9_5

[22] C. Thomborson and Y. Yu, "Measuring data cache and TLB parameters under Linux," in *Proceedings of the symposium on Performance Evaluation of Computer and Telecommunication Systems*, Jul. 2000, pp. 383–390. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.1427

[23] J. M. Blanquer and R. C. Chalmers, "MOB: A memory organization benchmark," Department of Computer Science, University of California at Santa Barbara, Tech. Rep., 2000. [Online]. Available: http://www.gnu-darwin.org/www001/ports-1.5a-CURRENT/devel/mob/work/mob-0.1.0/doc/mob.ps

[24] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *ISPASS*, 2010, pp. 235–246. [Online]. Available: http://dx.doi.org/10.1109/ISPASS.2010.5452013

[25] A. J. Smith and R. H. Saavedra, "Measuring cache and TLB performance and their effect on benchmark runtimes," *IEEE Trans. Comput.*, vol. 44, no. 10, pp. 1223–1235, Oct. 1995. [Online]. Available: http://dx.doi.org/10.1109/12.467697

[26] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu, "Programming by sketching for bit-streaming programs," in *PLDI*. New York, NY, USA: ACM, 2005, pp. 281–294. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065045

[27] P. Godefroid and A. Taly, "Automated synthesis of symbolic instruction encodings from I/O samples," in *PLDI*. New York, NY, USA: ACM, 2012, pp. 441–452. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254116

[28] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and computation*, vol. 75, no. 2, pp. 87–106, 1987. [Online]. Available: http://dx.doi.org/10.1016/0890-5401(87)90052-6

[29] ——, "Queries and concept learning," *Machine learning*, vol. 2, no. 4, pp. 319–342, 1988. [Online]. Available: http://dx.doi.org/10.1023/A:1022821128753

[30] S. Cassel *et al.*, "A succinct canonical register automaton model," in *Proceedings of the 9th international conference on Automated Technology for Verification and Analysis*. Springer, 2011, pp. 366–380. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24372-1_26

[31] F. Howar, B. Steffen, B. Jonsson, and S. Cassel, "Inferring canonical register automata," in *VMCAI*. Springer, 2012, pp. 251–266. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-27940-9_17