Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Master's Thesis

Measurement-based Inference of the Cache Hierarchy

submitted by

Andreas Abel

submitted

December 28, 2012

Supervisor

Prof. Dr. Jan Reineke

Reviewers

Prof. Dr. Jan Reineke
Prof. Dr. Dr. h.c. Reinhard Wilhelm

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,……………………………….. ………………………………………….
(Datum / Date) (Unterschrift / Signature)

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. Detailed models of such memory hierarchies are required in a number of areas, including static worst-case execution time analysis, cycle-accurate simulation, self-optimizing software, and platform-aware compilation.

Unfortunately, sufficiently precise documentation of the logical organization of the memory hierarchy is seldom available publicly. Thus, engineers are forced to obtain a better understanding of the microarchitecture by other means. This often includes performing measurements on microbenchmarks in order to iteratively refine a conjectured model of the architecture; a process that is both costly and error-prone.

In this thesis, we investigate ways to automate this process. We propose algorithms to automatically infer the cache size, the associativity, the block size, and the replacement policy. We have implemented and applied these algorithms to various popular microarchitectures, uncovering a previously undocumented cache replacement policy in the Intel Atom D525.

Acknowledgements

First and foremost, I would like to thank Prof. Jan Reineke for his excellent supervision. Parts of the work presented in this thesis, in particular the formalization of caches in Chapter 2, and the description of the replacement policy algorithm in Section 4.3, were developed in close collaboration with him. This collaboration has also lead to two publications: The paper "Automatic Cache Modeling by Measurements" [AR12] was presented at the Junior Researcher Workshop on Real-Time Computing (JRWRTC 2012) in Pont-à-Mousson, France, and the paper "Measurement-based Modeling of the Cache Replacement Policy" [AR13] will be presented at RTAS 2013 in Philadelphia, USA.

I would also like to thank the people at the Compiler Design Lab for their support and helpful advice, in particular Jörg Herter, who helped me find a topic for my Master's thesis and put me into contact with Jan Reineke.

Finally, I would like to thank Prof. Reinhard Wilhelm for agreeing to review this thesis.

Contents

1 Introduction

In recent years, processor speeds have improved significantly faster than access times to main memory. To bridge the resulting performance gap, modern microarchitectures employ memory hierarchies consisting of one or more levels of cache. These caches are small but fast pieces of memory that store copies of recently accessed code or data blocks.

A number of areas require detailed models of such cache hierarchies. On the one hand, such models are an essential part of static worst-case execution time (WCET) analyzers like aiT [FH04]; they are necessary both for soundness and precision. Similarly, cycle-accurate simulators, such as PTLsim [You07], need accurate cache models to produce useful results. On the other hand, self-optimizing software systems like ATLAS [CWPD01], PHiPAC [BACD97], or FFTW [FJ05], as well as platform-aware compilers, such as PACE [C^+10], require detailed knowledge of cache characteristics. The same is true for the performance modeling technique described by Snavely et al. [SCW^+02].

Unfortunately, documentation at the required level of detail is often hard to come by. Processor manuals can be ambiguous as they are written in natural language. Sometimes, a single manual describes a whole family of related processors and just gives a range of possible values for the parameters of interest. Moreover, information in vendor documents can be incorrect; Coleman and Davidson [CD01] described such a case. Finally, processor manuals might not provide any information about a particular architectural feature at all. In particular, vendor documents often do not contain information on the replacement policies used in different levels of the cache hierarchy.

As a consequence, engineers are forced to obtain a better understanding of the microarchitecture by other means. On x86 processors, the CPUID instruction [Int12b] provides information on some cache parameters. However, it does not characterize the replacement policy, and similar instructions are not available on many other architectures. Another possibility for obtaining more information would be to contact the manufacturer. However, the manufacturer is often not able or willing to provide more information, as he wants to protect his intellectual property. So, as a last resort, the engineer often has to perform measurements on microbenchmarks using evaluation boards. This way, he can iteratively refine his understanding of the architectural features until he is sufficiently confident in his conjectured model.

However, as this process of inferring a model of the cache hierarchy by measurements is both costly and error-prone, the question arises whether it can be automated. In this thesis, we investigate this question with respect to replacement policies used in first- and second-level instruction and data caches.

We introduce a class of replacement policies, we call them permutation policies, that includes widely used 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. We then present an algorithm that can automatically infer the replacement policy used in a cache, provided that it is part of this class. Since this algorithm requires knowledge of a number of other cache parameters, namely the cache size, the associativity and the block size, we develop and improve algorithms for these parameters as well.

Furthermore, we describe two implementations of our algorithms. The first implementation uses hardware performance counters to measure the number of cache misses, while the second one measures the execution time. We call these implementations *chi-PC* and *chi-T* (*chi* stands for "cache hierarchy inference", *PC* for "performance counters", and *T* for "time").

1.1 Outline

The thesis is structured as follows. In Chapter 2, we provide the necessary background regarding caches and replacement policies. In Chapter 3, we review and discuss related work. Chapter 4 begins with a concise problem statement. We then present and analyze different algorithms to solve our problem; details regarding their implementation are provided in Chapter 5. In Chapter 6, we evaluate our approach on a number of different platforms. Finally, Chapter 7 concludes the thesis and presents some ideas for future work.

2
Caches

While processor speeds have been increasing by about 60% per year, access times to main memory have improved by less than 10% per year [LGBT05]. Although faster memory techniques are available, their capacities are too small to use them as main memory. This is due to both their high cost and technological constraints. Modern systems therefore use memory hierarchies consisting of one or more levels or cache memory, small but fast pieces of memory that store a subset of the main memory's content.

The rationale behind this design is known as the *"principle of locality"*, which is comprised of *temporal* and *spatial locality*. *Temporal locality* means that memory locations that have been accessed recently tend to be accessed again in the near future. *Spatial locality* refers to the observation that elements with addresses that are nearby are likely to be accessed close together in time. In fact, for a typical program, about 90 percent of its memory accesses are to only 10 percent of its data. Caches utilize both types of locality by storing small contiguous memory areas containing recently accessed elements.

Figure 2.1 shows a typical memory hierarchy for a server computer, including the sizes and speeds of its components.

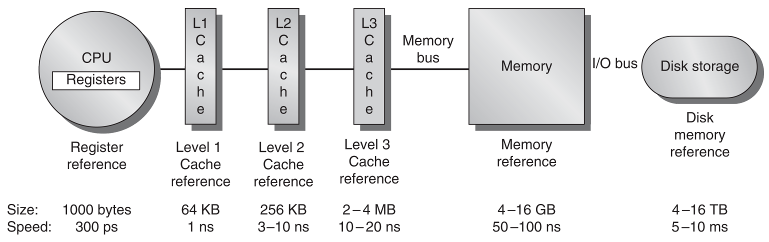


Figure 2.1: Example of a memory hierarchy. *(taken from [HP11b])*

A	$\in$	$\text{Associativity} = \mathbb{N}$	The associativity of the cache.
B	$\in$	$\text{BlockSize} = \mathbb{N}$	The block size in bytes.
N	$\in$	$\text{NumberOfSets} = \mathbb{N}$	The number of cache sets.
S	$=$	$A \cdot B \cdot N$	The cache size 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.

Figure 2.2: Parameters and basic domains.

2.1 Cache Organization

In this section, we describe how caches are typically structured internally and how memory accesses are managed. Here and in the following, we use the parameters and basic domains depicted in Figure 2.2. Figure 2.3 illustrates the concepts introduced in the following paragraphs.

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, which are are cached as a whole. 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: $\text{block}_B(address) = \lfloor address/B \rfloor$.

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: $\text{index}_{B,N}(address) = \text{block}_B(address) \mod N$. The remaining bits of an address are known as the *tag*: $\text{tag}_{B,N}(address) = \lfloor \text{block}_B(address)/N \rfloor$. To decide whether and where a block is cached within a set, tags are stored along with the data.

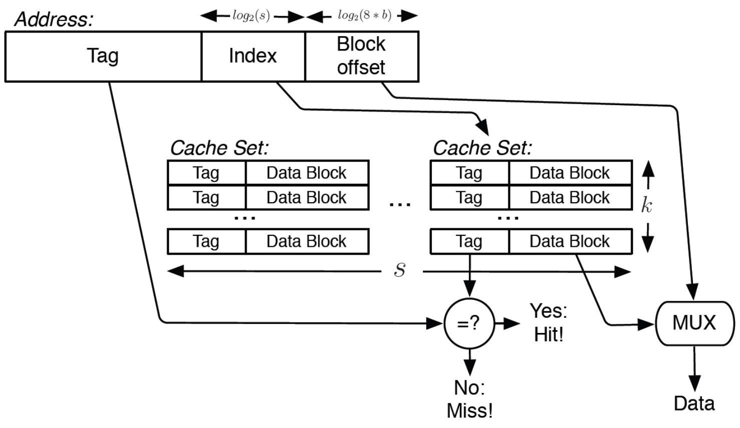


Figure 2.3: Logical organization of a k-way set-associative cache.
(taken from [Rei08])

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. Usually, cache sets are treated independently of each other, such that accesses to one set do not influence replacement decisions in other sets. We describe a number of commonly used replacement policies in detail in Section 2.3.

2.2 Formalization and a Cache Template

In this section, we first define caches and replacement policies formally and describe a cache template. Based on these definitions, we then discuss by which means inference algorithms can gain information about the cache.

Definition 1 (Cache)**.** A cache is a 4-tuple $C = (\text{CacheState}_C, s_C^0, \text{up}_C, \text{hit}_C)$, where

- CacheState_C is the cache's set of states
- $s_C^0 \in \text{CacheState}_C$ is its initial state
- $\text{up}_C : \text{CacheState}_C \times \mathit{Address} \to \text{CacheState}_C$ models the change in state upon a memory access
- $\text{hit}_C : \text{CacheState}_C \times \mathit{Address} \to \mathbb{B}$ determines whether a memory access results in a cache hit or a cache miss.

Definition 2 (Replacement policy)**.** A replacement policy is a 4-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 its initial state
- $\mathrm{evict}_P : \mathrm{PolState}_P \to \mathit{Way}$ determines which memory block to evict upon a cache miss
- $\mathrm{up}_P : \mathrm{PolState}_P \times (\mathit{Way} \cup \{\mathit{miss}\}) \to \mathrm{PolState}_P$ computes the change in state upon a cache hit to a particular cache way or upon a miss.

Caches as described in Section 2.1 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:

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

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:

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

where $index = \mathrm{index}_{B,N}(addr)$ and $tag = \mathrm{tag}_{B,N}(addr)$. Here, 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:

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

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

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:

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

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

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.

2.2.1 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. (Alternatively, for higher portability, inference algorithms may instead measure the execution times of memory access sequences.)

Let $\text{misses}_C(s, \vec{a})$ denote the number of misses that the cache C incurs performing the sequence of memory accesses $\vec{a}$ starting in cache state s. We define $\text{misses}_C(s, \vec{a})$ by structural recursion on the length of the sequence $\vec{a}$:

$$\begin{array}{rcl} \text{misses}_C(s, \langle\rangle) & := & 0 \\ \text{misses}_C(s, \langle a_0, a_1, \ldots, a_n \rangle) & := & (1 - \text{hit}_C(s, a_0)) \\ & & + \text{misses}_C(\text{up}_C(s, a_0), \langle a_1, \ldots, a_n \rangle). \end{array}$$

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:

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

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:

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

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 $\mathbf{measure}_C(\vec{p}, \vec{m})$ as an expression that will nondeterministically evaluate to one of the values of the set defined above.

2.2.2 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 [S$^+$04].

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

2.3 Replacement Policies

In order to exploit temporal locality, most replacement policies use information about the access history to decide which block to replace. Usually, they treat cache sets independently of each other, such that accesses to one set do not influence replacement decisions in other sets. Well-known policies in this class include:

- Least-recently used (LRU): replaces the element that was not used for the longest time.
- Pseudo-LRU (PLRU), a cost-efficient variant of LRU.

- First-in first-out (FIFO), also known as Round Robin.
- Not-most-recently used (NMRU) [GLYY12].

An exception to the rule of treating cache sets independently is Pseudo Round Robin, which maintains a single "FIFO pointer" for all cache sets. This policy is used in a number of ARM and Motorola processors [Gru12]. A policy that does not consider the access history is Pseudo Random, which is used in several recent ARM CPUs [ARM10].

2.3.1 Permutation Policies

There are infinitely many observationally different replacement policies. So, for inference to be possible, one needs to restrict the set of states PolState_P of a policy P. 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, 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.

However, we have identified a finite (for a fixed associativity) set of replacement policies that contains many widely-used policies, and for which inference is feasible. We call these policies *permutation policies.*

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*[1] 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 2.4 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.

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

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

(a) LRU

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

(b) PLRU

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

(c) FIFO

Figure 2.4: Permutation vectors for LRU, PLRU & FIFO at associativity 8.

2.3.2 Logical Cache Set States

Cache set states as defined in 2.2 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 state as follows: $logical(\langle bs, \pi\rangle) := bs \circ \pi$. 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 $[a, b]$.

2.4 Cache Optimizations

Processor manufacturers have developed and implemented a number of techniques to optimize the memory hierarchy. While some of these optimizations only influence the performance or the power consumption of a cache, others also modify its logical structure and violate some of the assumptions we made in the previous sections. Prefetching, for instance, can violate the assumption that the cache state is only modified upon a memory access. For an extensive treatment of cache optimizations we refer to [HP11b]. In the following paragraphs, we briefly introduce some of the optimizations that were relevant to us when developing and implementing our inference algorithms.

Hardware Prefetching Hardware prefetchers try to predict future data accesses, and load memory blocks before they are actually needed. The prefetched blocks can either be placed in a dedicated buffer or directly in the cache, which leads to the eviction of other blocks in the cache. There are several types of hardware prefetchers:

- An *adjacent cache line prefetcher* fetches, upon a cache miss, one or more cache lines that are next to the accessed address.
- A *stride prefetcher* tries to detect patterns in the memory accesses in order to predict the next addresses.
- A *streamer prefetcher* detects multiple accesses to a single cache line or accesses to a sequence of consecutive cache lines, and loads the following line.

Often, prefetchers do not prefetch accross page boundaries [Int12a].

Cache Indexing Most modern computer systems use *virtual memory*: they divide the physical memory into pages and allocate them to different processes, giving each process a contiguous *virtual address space*. There are different possibilities as to which address to use for the index and the tag of a cache:

- *Virtually indexed, virtually tagged* caches use the virtual address for both the index and the tag, which can lead to faster caches than other approaches because no address translation is required. However, there are a number of drawbacks. After every process switch, the virtual addresses refer to different physical addresses, which requires the cache to be flushed. Apart from that, two different virtual addresses can refer to the same physical address; this is also called "aliasing". In such a case, the same data can be cached twice in different locations. Moreover, virtual-to-physical address mappings can change, which requires the cache to be flushed as well.
- *Physically indexed, physically tagged* caches use the physical address for both the index and the tag. While this approach avoids the problems just mentioned, it is much slower as every cache access requires expensive address translations.
- *Virtually indexed, physically tagged* caches combine the advantages of the two other approaches. They use the part of the address that is identical for both the virtual and physical address (i.e. a part of the page offset) to index the cache. While checking the cache for this index, the remaining bits of the virtual address can be translated, and thus the physical address can be used as tag. This approach is, however, limited to smaller caches.

Modern processors often use virtually indexed, physically tagged first-level caches and physically indexed, physically tagged higher-level caches.

Cache Hierarchy Management Policies Multi-level caches use different approaches regarding the question whether the data stored in lower-level caches must also be present in higher-level caches. (Note: The terms *lower-level cache* and *higher-level cache* are used inconsistently in the literature. The more common variant, however, seems to refer to caches that are closer to the processor as *lower-level caches* (i.e. the first-level cache is the lowest-level cache). We will use this definition in the following). In *(strictly) inclusive* cache hierarchies, every level of the cache hierarchy is a subset of the next-higher level. In *exclusive* cache hierarchies, on the other hand, data is in at most one level of the cache hierarchy. Most recent AMD x86 processors use this design. In *non-inclusive* (sometimes also called *non-inclusive/non-exclusive (NI/NE)* or *mainly inclusive*) caches, data in lower-level caches may often also be in higher-level caches but is not required to. This design is used by many Intel x86 CPUs [HP11a].

Nonblocking caches A nonblocking cache can serve other access requests while processing a cache miss. This reduces the effective miss penalty [FJ94].

Way Prediction Way prediction is a technique to reduce the hit time of a cache. It keeps extra bits for each set to predict the way of the next access. If the prediction is right, only a single tag comparison needs to be performed [HP11b].

Victim Caches A victim cache stores recently evicted blocks. In case of a cache miss, the processor first checks whether the requested element is in the victim cache before forwarding the request to the next level of memory. Victim caches are typically fully-associative and can hold 4 to 16 cache blocks [ZV04].

Trace Caches A trace cache is an advanced form of a first-level instruction cache. It sits between the instruction decode logic and the execution core and stores already decoded instructions. Trace caches are for example used in the Intel Pentium 4 processor [HSU^{+}01].

Hash Functions While "traditional" caches use a part of the memory address to index the cache, Intel uses instead a hash function for the third-level cache of its *Sandy Bridge* architecture[2].

[2]`http://www.realworldtech.com/sandy-bridge/8/`

3 Related Work

3.1 Measurement of Cache Hierarchy Parameters

Several publications have presented approaches for determining 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 [CD01, D$^+$04, JB07, BT09], while the others use timing information [SS95, MS96, LT98, TY00, BC00, Y$^+$04, Y$^+$05b, Y$^+$05c, Y$^+$05a, Y$^+$06, Man04, GD$^+$10, Cha11, CS11].

Only few [Y$^+$06, CD01, BC00, BT09] have also analyzed these parameters for instruction caches. However, in contrast to our implementation, the approach described in [Y$^+$06] generates and compiles C source code dynamically, thus requiring access to the compiler at runtime. On the other hand, [CD01] relies on specific features of the GCC compiler. Blanquer and Chalmers [BC00] require the user to supply a number of primitives in assembly for a given architecture. Babka and Tůma [BT09] state that they use "chains of jump instruction" but do not describe their implementation in detail.

Some of these approaches make assumptions as to the underlying replacement policy (e.g. [SS95] and [TY00] assume LRU replacement is used). However, only a few publications have tried to determine the cache replacement policies as well. The approaches described in [CD01] and [BC00] are able to detect LRU-based policies but treat all other policies as random. John and Baumgartl [JB07] use performance counters to distinguish between LRU and several of its derivatives.

Several papers have used measurement-based approaches to analyze other parts of a computer system: [SS95, TY00, CS11] have analyzed TLB parameters, [UM09] branch predictors, [DSGP08] multi-core specific features, and [W$^+$10] parameters of the GPU.

Conducting a comprehensive literature review on the topics of our interest was rather difficult. This is, on the one hand, because the literature is scattered over a number of different research areas, including self-optimizing software [Y^+05c], compiler optimizations [CS11], performance-analysis [CD01], benchmarking [MS96], and WCET analysis [JB07]. On the other hand, the publications use different terminology for describing similar goals, e.g., "automatic measurement of hardware parameters" [Y^+05c], "automatic memory hierarchy characterization" [CD01], "memory organization benchmark" [BC00], "cache-memory and TLB calibration tool" [Man04], or "experimental parameter extraction" [JB07]. In fact, several papers we found claimed to be the first in their area, although we found earlier papers describing similar work: [Y^+06] claims that "neither well known benchmarks [...], nor existing tools" attempt to measure the instruction cache capacity, and [JB07] believes that "techniques to discover the replacement strategies by experiments have not been described before", although both [BC00] and [CD01] were published at least four years earlier; [D^+04] claims to be the first to use hardware performance counters to analyze cache parameters, although [CD01] was published more than two years earlier.

In the following paragraphs we describe the most relevant publications more in detail.

Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes [SS95] Most publications on memory characterization identify the paper by Saavedra and Smith as the first paper in this area. Using a Fortran benchmark, they create a set of curves that has to be interpreted manually to determine the cache size, the associativity, and the block size of first and second level data caches, as well as the size, the associativity and the page size of the TLB.

Data Cache Parameter Measurements [LT98] *and* Measuring Data Cache and TLB Parameters under Linux [TY00] Thomborson et al. extended Saavedra and Smith's approach by separating read from write accesses, by developing a different algorithm to infer the associativity, and by analyzing a larger set of cache parameters (e.g. cache write policies).

lmbench: Portable Tools for Performance Analysis [MS96] `lmbench` is a comprehensive benchmark suite that can detect the size and the block size of data caches, besides a number of other parameters like the bandwidths of memory accesses, or the latencies of disk accesses or context switches. Similar to our approach, it uses a form of pointer chasing to analyze caches. Though the paper dates back to 1996, the tools seems to be still maintained (as of November 2012, the tool was last updated in 2010).

MOB: A Memory Organization Benchmark [BC00] This paper was the first to automatically analyze instruction cache parameters and replacement policies. However, Blanquer and Chalmers only distinguish between LRU and random replacement. They try to determine the replacement policy by performing three tests:

1. Access repeatedly associativity many elements that map to the same cache set.
2. Access repeatedly 2*associativity many elements that map to the same cache set.
3. Access repeatedly 1.5*associativity many elements that map to the same cache set.

The decision between LRU and random is then based on the assumption that in a LRU system, the tests 2 and 3 should lead to the same miss rate, whereas under random replacement, test 3 should cause about half as many misses as test 2. The replacement policy is then considered to be an LRU policy if the miss rate of test 3 is greater than the middle point between test 1 and 2, and random otherwise.

Automatic Memory Hierarchy Characterization [CD01] This paper by Coleman and Davidson seems to be the first to use hardware performance counters to infer memory hierarchy parameters such as the cache size, the associativity and the block size of data and instruction caches. Moreover, the authors describe an approach to detect LRU-based replacement policies, and, similar to Blanquer and Chalmers, they treat all other policies as random. Their algorithm works as follows:

- Access associativity many elements that map to the same cache set.
- Access the first associativity-1 elements again.
- Access a new element that maps to the same cache set.
- Start the performance counters and measure the number of misses when accessing element i again to determine whether element i was replaced.

This experiment is performed for each i, s.t. $0 \leq i \leq associativity - 1$. If the same element is replaced every time, the replacement policy is considered to be LRU-based, otherwise it is considered to be random.

X-Ray [Y$^+$04, Y$^+$05b, Y$^+$05c, Y$^+$05a, Y$^+$06] X-Ray is a tool that can detect the cache sizes, associativities and block sizes of data and instruction caches, in addition to a number of other parameters like latencies or the number of available registers. X-Ray uses an approach that generates, compiles and executes C Code at runtime.

Exact Cache Characterization by Experimental Parameter Extraction [JB07] John and Baumgartl describe an algorithm that can be used to detect LRU and some of its derivatives. The authors claim that "the presented algorithms are easy adapt and thus allow the identification of other strategies, too". Their algorithm works as follows (it assumes the cache size S, the associativity A, and the block size B to be known and to be powers of two):

1. Invalidate the cache.
2. Fill the cache by reading associativity many ways $w_1, \ldots, w_A$ (*reading a way* means performing a read access in all blocks of that way).
3. Read a subset of the ways from step 2 again.
4. Read a new way, which replaces way w_i for some $i \in \{1, \ldots, A\}$.
5. n_{w_1} := count misses when reading way w_1 again
   $\vdots$
   n_{w_A} := count misses when reading way w_A again.

The authors then claim that exactly one of $\{n_{w_1}, \ldots, n_{w_A}\}$ "has a significantly larger value than the other [...] counters. The associated way has been replaced [...]." The sequences for step 3 are chosen such that different replacement policies can be distinguished from the results. There is, however, a major flaw in this description of the algorithm. Assume a cache with an LRU replacement policy. When implementing the algorithm like this, then in step 4, way w_1 is replaced, and in step 5, reading way w_i replaces way w_{i+1}. Thus, all accesses in step 5 will be misses, and the values of $n_{w_1}, \ldots, n_{w_A}$ will be roughly the same. The problem could for example be fixed by repeating step 1 to 4 before every part of step 5. However, the authors do not mention how they implemented this algorithm.

Apart from that, there are a number of major differences to our work:

- John and Baumgartl's approach is not able to determine the replacement policy automatically. The access sequences in step 3 need to be generated by hand for every replacement policy and every associativity. Moreover, the corresponding code for these sequences has to be written manually.
- The authors do not formally analyze which class of replacement policies their approach is able to detect. We have identified several policies that would only be detected by our approach. Consider for example the class of permutation policies for which $\Pi_i(0) = 0$ for all $i \in \{0, \ldots, A-1\}$. This class contains the FIFO policy. But it also contains, for example, the following (hypothetical) approximation to FIFO which can be described (at associativity 4) by the permutation vector $\Pi_i = (0, 2, 1, 3)$ for all $i \in \{0, \ldots, 3\}$. John and Baumgartl's approach is not able to distinguish these policies from FIFO as the way replaced in step 4 is always w_1.
- John and Baumgartl do not attempt to analyze instruction caches.
- Their implementation requires a special real-time operating system environment and needs to be adapted to every different processor architecture.

Servet: A benchmark suite for autotuning on multicore clusters [GD$^+$10] González-Domínguez et al. present a tool that can detect parameters of cache hierarchies used in multi-core systems, including cache sizes. To this end, they describe a variation of Saavedra and Smith's algorithm that uses a probabilistic approach for dealing with physically indexed lower level caches. This approach is based on the assumption that the virtual pages are uniformly distributed over the physical address space. They further assume that the cache size is a power of two if the size is less than 1MB and a multiple of 1MB otherwise.

Robust Method to Determine Cache and TLB Characteristics [Cha11] In his master's thesis, Chandran claims that all previous approaches "fail when encoutering [sic] multiple levels of cache and TLB present in current generation of processors." He then presents a modified version of the CALIBRATOR framework [Man04] that uses a "staggered and robust approach to first detecting the hardware entities and their sizes and then disambiguating between the entities by deducing their block sizes." He evaluates his approach on four recent Intel and one AMD CPU and claims that his "approach produces more accurate and complete results than existing tools". However, the presented results are not particularly convincing. Although Chandran's tool returned the correct values for all reported results, select results are omitted (the results for the L2 cache parameters are for example only reported for one CPU, and for the AMD CPU, only the results for the L1 data cache are presented). This leaves the impression that his tool did not return the correct values in these cases.

Moreover, several parts of Chandran's thesis (including the introduction) consist mainly of copy-and-paste plagiarism from both [D^{+}04] and [Man04]. Both papers were cited in the thesis but not anywhere near the plagiarized sections.

Portable Techniques to Find Effective Memory Hierarchy Parameters [CS11] This paper analyzes memory parameters in order to build compilers that can perform model-specific tuning automatically. It presents algorithms to infer the cache size, the associativity and the block size of first level data caches and the size and the block size of lower level caches, as well as latencies and TLB parameters. The authors try to build a portable solution that does not depend on system-specific features like huge pages or performance counters. However, in contrast to our approach, they do not try to achieve exact results, they rather try to detect "effective values" for which they claim to provide better optimization results than would be obtained using the actual values. Because they do not consider the effect of physically indexed caches, their results for the size of lower levels caches is usually significantly smaller than the actual size (they argue that this is the "effective size").

3.2 Machine Learning

Caches as defined in Section 2.2 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 Section2.2 are finite-state systems. It thus might be possible to adapt Angluin's algorithm to learn regular languages [Ang87] 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 [Ang88].

The connection between caches and canonical register automata [C$^+$11] is more immediate. We are thus exploring the use of Howar et al.'s technique to learn such automata [H$^+$12]. The question is whether general automata-learning methods scale to the size of the state space of a replacement policy such as LRU.

4 Algorithms

In this chapter, we describe our approach for inferring the different cache parameters. More formally, we present algorithms that derive parameter values A, B, N, and P through measurements on an implementation of a cache C, such that $T(A, B, N, P) \equiv C$. Instead of deriving N directly, we first derive the cache size S. N can then be obtained by $N = \frac{S}{A \cdot B}$.

In general, our algorithms are all based on the following scheme:

1. Generate multiple sequences of memory accesses.
2. Measure the number of cache misses (using hardware performance counters), or alternatively the execution time, on each of the sequences.
3. Deduce the property of interest from the measurement results and some structural assumptions. This step usually involves finding the point at which a "jump" in the measurement results occurs.

Notations In the following, we will use the notation $\vec{a}^n$ to denote the n-fold concatenation of the sequence $\vec{a}$, i.e.,

$$\begin{aligned} \vec{a}^0 &:= \langle\rangle \\ \vec{a}^n &:= \vec{a} \circ \vec{a}^{n-1}, \end{aligned}$$

where the operator $\circ$ is used for concatenating sequences.

To compactly represent strided access sequences, we use

$$\langle\langle base, stride, count \rangle\rangle$$

to denote the access sequence

$$\langle base, base + 1 \cdot stride, \ldots, base + (count - 1) \cdot stride \rangle.$$

During the development of our tool, we usually experimented with a number of different algorithms for each of the cache parameters we were interested in. We usually started with a simple algorithm and then we modified and improved this algorithm based on the problems we encountered. The structure of the following sections reflects this development process.

4.1 Cache Size/Associativity

In this section, we present different algorithms to infer the cache size S and the associativity A, and we analyze their strengths and weaknesses.

4.1.1 Simple Algorithms

Cache Size Algorithm 1 illustrates our initial approach for determining the cache size.

Algorithm 1: Cache Size (Simple Algorithm)

```
Output: Cache Size
$measurements[maxSize]$
$curSize \leftarrow 1$
while $curSize < maxSize$ do
    $\vec{p} \leftarrow \langle 0, 16, ..., curSize * 1024 \rangle$
    $\vec{a} \leftarrow \langle 0, 16, ..., curSize * 1024 \rangle^{100}$
    $measurements[curSize] \leftarrow \mathbf{measure}_C(\vec{p}, \vec{a})$
    if "jump occurred" then
        return $curSize - 1$
    $curSize \leftarrow curSize + 1$
```

The algorithm accesses a contiguous memory area of *curSize* bytes repeatedly with a stride of 16 bytes (which we assume to be a lower bound on the block size). As long as *curSize* is less than or equal to the actual cache size, all accessed bytes fit into the cache and thus, no misses should occur. If *curSize* exceeds the cache size, we expect to see a sharp increase in the number of misses.

Figure 4.1 shows the results of running this algorithm on an Intel Core 2 Duo E6750 (32kB 8-way associative L1 Cache).

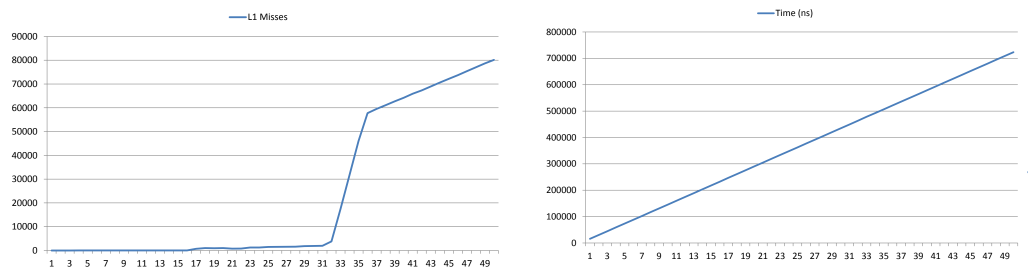


Figure 4.1: Result of running the simple algorithm for determining the cache size on an Intel Core 2 Duo E6750 (32kB L1 Cache).

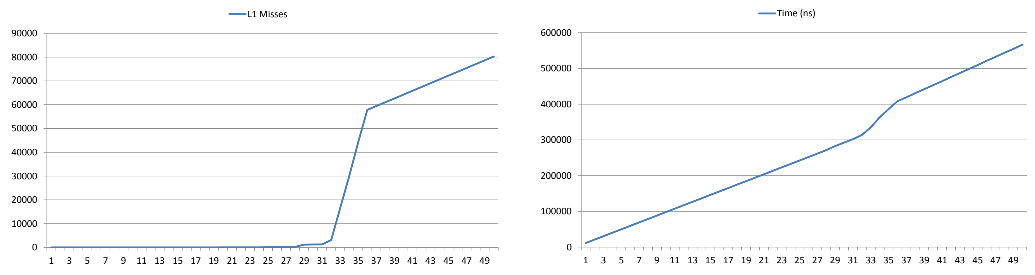


Figure 4.2: Result of running the simple algorithm with pointer chasing on an Intel Core 2 Duo E6750 (32kB L1 Cache).

The left diagram was created using hardware performance counters. Here, the "jump" at 32kB is clearly visible (at this stage of the development process, we analyzed the graphs manually to detect the jumps; we will describe how to do this automatically when we describe our final algorithms). The diagram also shows a second jump at 36kB (i.e. 32kB + 4kB, where 4kB is the way size of the cache). The reason for this behavior is that between 32kB and 36kB some cache accesses lead to cache hits and other accesses to cache misses; if the memory area is larger than 36kB, all memory accesses lead to cache misses (this behavior, however, depends on the replacement policy; thus we will not use it to detect the way size).

The right diagram shows the result of the time-based algorithm. Here, the jump is hardly visible. A number of factors can contribute to this behavior, e.g., non-blocking caches, out-of-order execution, prefetching, and other cache optimizations like way-prediction.

To minimize the effects of non-blocking caches and out-of-order execution, we can we serialize memory accesses by using a form of "pointer chasing" where each memory location contains the address of the next access (we will describe this technique in more detail in Section 5.3).

Figure 4.2 shows the result of this modification. Now, a slight jump in the diagram for the time based approach is visible. But it is hard to find the exact location of the jump. Moreover, there is also already a small jump between 31kB and 32kB in the left diagram. (Interestingly, the overall times are lower when using this approach. This is probably because of the lower overhead for address computations.)

Associativity Algorithm 2 shows our first approach at inferring the associativity.

Algorithm 2: Associativity (Simple Algorithm)

```
Input: $S \leftarrow$ Cache Size
Output: Associativity
$measurements[maxAssoc]$
$curAssoc \leftarrow 1$
while $curAssoc < maxAssoc$ do
    $\vec{p} \leftarrow \langle\langle 0, S, curAssoc \rangle\rangle$
    $\vec{a} \leftarrow \langle\langle 0, S, curAssoc \rangle\rangle^{100}$
    $measurements[curAssoc] \leftarrow \mathbf{measure}_C(\vec{p}, \vec{a})$
    if "jump occurred" then
        return $curAssoc - 1$
    $curAssoc \leftarrow curAssoc + 1$
```

The algorithm uses the fact that the cache size is always a multiple of the way size. Thus, when accessing the memory with a stride of cache size many bytes, all accesses map to the same cache set. If *curAssoc* exceeds the actual associativity, the cache can no longer store all accessed memory locations, and so we expect to see a jump in the number of misses.

Figures 4.3 and 4.4 show the result of running this algorithm on the same platform as above, both with and without pointer chasing.

4.1.2 A More Robust Algorithm

Based on the observation that the algorithm to infer the associativity worked quite well, we developed a new algorithm (Algorithm 3) that uses a similar approach and that can determine both the associativity and the cache size. This algorithm uses the fact that the way size is always a power of two on modern processors.

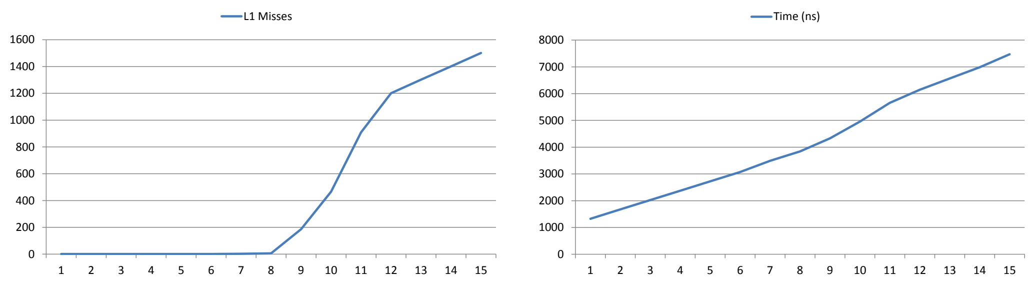


Figure 4.3: Result of running the simple algorithm for the associativity without pointer chasing on an Intel Core 2 Duo E6750 (8-way associative).

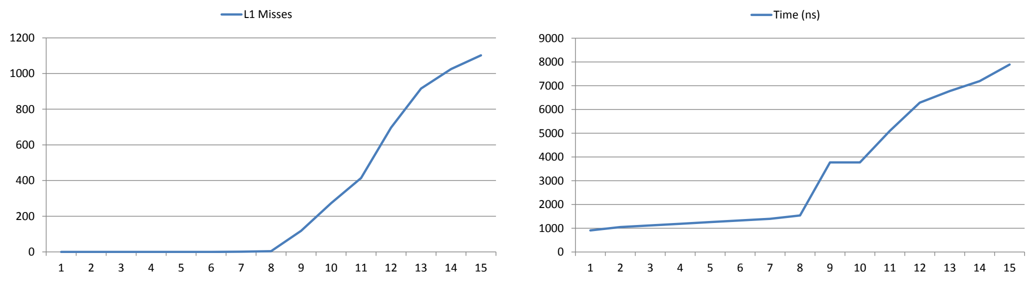


Figure 4.4: Result of running the simple algorithm for the associativity with pointer chasing on an Intel Core 2 Duo E6750 (8-way associative).

Algorithm 3: Cache Size/Associativity

```
Output: (Cache Size, Associativity)
$ws \leftarrow maxWaysize$
$assoc_{old} \leftarrow 0$
while $true$ do
    for $(assoc \leftarrow 1; ; assoc \leftarrow assoc + 1)$ do
        $\vec{p} \leftarrow \langle\langle 0, ws, assoc \rangle\rangle$
        $\vec{a} \leftarrow \langle\langle 0, ws, assoc \rangle\rangle^{100}$
        $measurements[assoc] \leftarrow \mathbf{measure}_C(\vec{p}, \vec{a})$
        if "jump occurred" then
            if $assoc - 1 = 2 * assoc_{old}$ then
                return $(ws * 2 * assoc_{old}, assoc_{old})$
            else
                $assoc_{old} = assoc - 1$
                break
    $ws = ws/2$
```

The algorithm iterates over different possible way sizes ws, and determines for every ws how many elements can be accessed with a stride of ws until the accessed elements don't fit in the cache any more, which means that a jump in the number of misses occurs. As long as ws is greater than or equal to the actual way size, this value stays the same (in fact, it corresponds to the actual associativity of the cache). If ws gets smaller than the way size, the value doubles.

Figure 4.5 shows the result of running this algorithm. The upper diagram was created using performance counters. Note that the curves for $ws \geq 4096$ coincide. The lower diagram uses the execution time. The second jump in the curve for $ws = 65536$ is caused by misses in the second-level cache.

4.1.3 An Algorithm Supporting Physically Indexed Caches

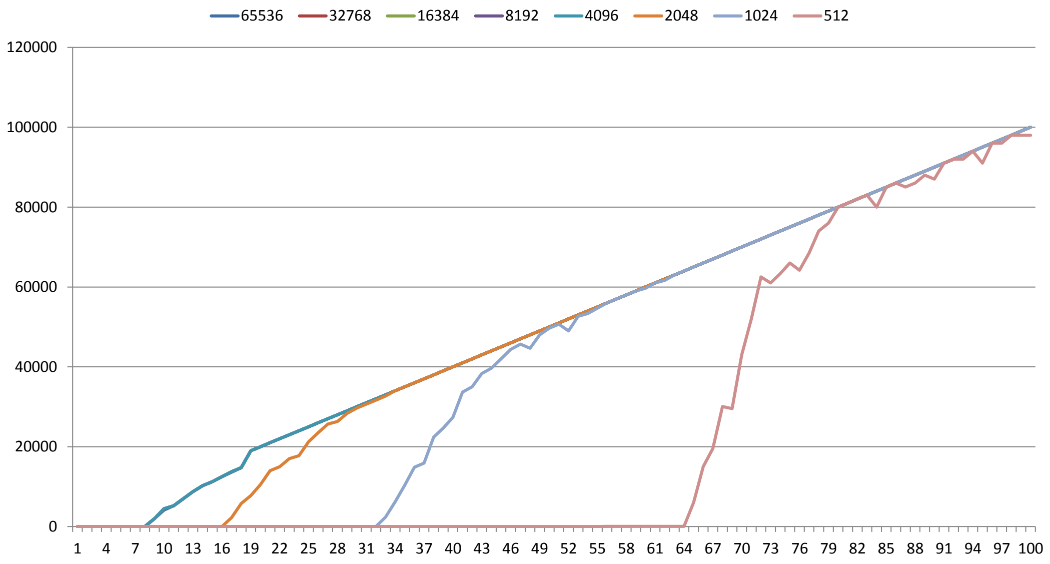
All algorithms we have considered so far are based on the assumption that the memory area they use is physically contiguous or that the caches are virtually indexed. However, most recent processors use physically indexed second-level caches, and some recent ARM CPUs even physically indexed first-level data caches [ARM10]. 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.

One way to deal with this problem is to allocate *huge pages*[1]. 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.

Unfortunately, however, huge pages are not activated by default on common Linux-based systems, and superuser privileges are required to activate them. Moreover, on some platforms, huge pages are not available at all; in particular older processors or CPUs for embedded systems might not support them. They are also not available on Android-based devices.

We therefore developed an alternative approach for dealing with physically indexed caches that does not require huge pages. Algorithm 4 illustrates this approach.

[1]`http://en.wikipedia.org/wiki/Page_size.`



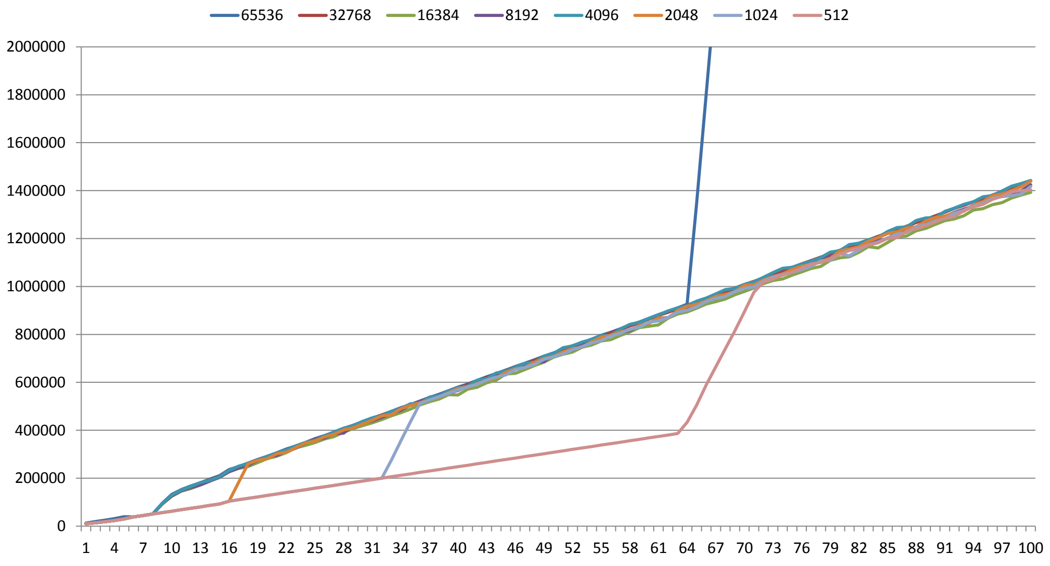


Figure 4.5: Result of running Algorithm 3 on an Intel Core 2 Duo E6750 (32kB 8-way associative L1 cache).

Algorithm 4: Cache Size/Associativity (Supporting Physically Indexed Caches)

```
Input: -
Output: (Cache Size, Associativity)

List $pages$
addMaxNumNoncollidingPages($pages$)
$nNonCollidingPages \leftarrow pages.size - 1$
removeNonCollidingPages($pages$)
$associativity \leftarrow pages.size - 1$
for ($ws \leftarrow pagesize/2; ws > minWaysize; ws \leftarrow ws/2$) do
    $pages.last \leftarrow pages.last + ws$
    $\vec{p} \leftarrow \langle pages.first, ..., pages.last \rangle$
    $\vec{a} \leftarrow \langle pages.first, ..., pages.last \rangle^{100}$
    $measurements[pages.size] \leftarrow \textbf{measure}_C(\vec{p}, \vec{a})$
    if "jump occurred" then
        continue
    else
        $ws \leftarrow 2 * ws$
        break
if $ws \geq pagesize$ then
    $size \leftarrow nNonCollidingPages * pagesize$
else
    $size \leftarrow ws * associativity$
return ($size, associativity$)
```

Algorithm 5: Helper Functions for Algorithm 4

```
procedure addMaxNumNoncollidingPages(List pages)
    $curPage \leftarrow 0$
    $nCollisions \leftarrow 0$
    while $nCollisions < 100$ do
        $\vec{p} \leftarrow \langle pages.first, ..., pages.last, curPage \rangle$
        $\vec{a} \leftarrow \langle pages.first, ..., pages.last, curPage \rangle^{100}$
        $measurements[pages.size] \leftarrow \mathbf{measure}_C(\vec{p}, \vec{a})$
        if "jump occurred" then
            $nCollisions \leftarrow nCollisions + 1$
        else
            $pages.add(curPage)$
            $nCollisions \leftarrow 0$
        $curPage \leftarrow curPage + pagesize$
    $pages.add(curPage)$

procedure removeNonCollidingPages(List pages)
    foreach page in pages do
        $pages.remove(page)$
        $\vec{p} \leftarrow \langle pages.first, ..., pages.last \rangle$
        $\vec{a} \leftarrow \langle pages.first, ..., pages.last \rangle^{100}$
        $measurements[pages.size] \leftarrow \mathbf{measure}_C(\vec{p}, \vec{a})$
        if "jump occurred" then
            continue
        else
            $pages.add(page)$
```

The algorithm first calls the procedure `addMaxNumNoncollidingPages`. This procedure creates a maximal list of pages (more specifically, addresses that are a multiple of the page size) that does not contain a collision (by *collision* we mean a set of more than associativity many pages that map to the same cache set). The algorithm stops when the previous 100 pages that were examined all resulted in collisions (we found this threshold to work well for first- and second-level caches). Before the procedure returns it adds the last page it examined to the list. So at this point, the list contains exactly one collision. Furthermore, the number of non-colliding pages found by the procedure (i.e. the current size of the list minus one) is stored in a variable.

In the next step, the algorithm removes all pages from the list that are not part of this collision. This step thus determines the associativity of the cache, which corresponds to the new size of the list minus one. (Note that this step could be optimized by modifying `addMaxNumNoncollidingPages` such that it additionally returns the list after the first collision was found, and using this list for the current step. This would, however, still require removing pages from the list, as not all pages in the list are necessarily part of the collision).

After this, the algorithm checks whether the way size is a smaller than the page size (note that we assume both to be powers of two). This is done by adding different offsets to the last page in the list. As long as this offset is greater than or equal to the actual way size, all accesses still map to the same cache set and we can observe a collision. If the offset is smaller than the way size, the last access maps to a different set, and hence, no collision occurs.

If the way size is indeed smaller than the page size, the cache size can be obtained by multiplying the way size with the associativity. If the way size is not smaller than the page size, the cache size can be computed by multiplying the maximum number of non-colliding pages (that was determined in the first step of the algorithm) with the page size.

Detecting the "jumps" At several points, the algorithm tries to detect jumps in the measurement results in order to determine whether more than associativity many elements were accessed in the same cache set.

If performance counters are used for the measurements, one simple method is to use a fixed threshold. Ideally, we would expect no misses if all accessed elements fit into the cache. However, due to measurement overhead and interference (see Section 5.2), the actual measurement results will typically be larger than zero. On the other hand, as the access sequences are executed 100 times, there must be at least 100 misses if one element does not fit into the cache (for common replacement policies such as LRU there are significantly more misses). We found that using 100 misses as a threshold for detecting the jumps worked well on all systems we tested.

If timing information is used for the measurements, detecting the jumps is more difficult, as one has to deal with relative thresholds instead of fixed values.

One possibility is to consider the time differences when adding additional pages. This difference should be significantly higher when the new page causes a collision. Another possibility is to use *simple linear regression*[2], i.e. fitting a line through the previous measurement results that minimizes the sum of squared residuals. If the current measurement result is significantly above that curve, we would assume that a miss occurred.

However, a problem with both of these approaches is that other factors can influence the execution time. In particular, some optimization techniques such as *way prediction* (see Section 2.4) can lead to jumps in the execution time when accessing more elements, even if all these accesses are cache hits.

We therefore developed another approach, that worked quite well in most cases: We perform an additional measurement in which we replace the current element by one that is guaranteed to be in the cache, such that the same number of elements is accessed in both cases. Such an element is for example the current element with an offset that is larger than the block size but smaller than the way size. We then compare the two measurements; if the original result is significantly higher then the result of the additional measurement, we assume that a cache miss occurred. This approach could be further improved by performing yet another measurement in which the last access is guaranteed to be a cache miss; one could then take the midpoint between the two additional measurements as a threshold. To guarantee that accessing the last element results in a miss, one could use different elements for each access. This can, however, be problematic as misses in higher-level caches might increase the execution in this case.

4.2 Block Size

In the following, we assume, for reasons of simplicity, that the memory is space is physically contiguous. It would, however, also be possible to extend our approach from the last section to these algorithms.

[2]`http://en.wikipedia.org/wiki/Simple_linear_regression`

Algorithm 6: Block Size (Simple Algorithm)

```
Input: $S \leftarrow$ Cache Size, $ws \leftarrow$ Way Size
Output: Block Size
$measurements[maxSize]$
$curSize \leftarrow 32$
while $curSize < maxSize$ do
    $\vec{p} \leftarrow \langle 0, curSize, 2 * curSize, ..., S \rangle$
    $\vec{a} \leftarrow \langle 0, 16, 32, ..., S \rangle$
    $measurements[curSize] \leftarrow \mathbf{measure}_C(\vec{p}, \vec{a})$
    if "jump occurred" then
        return $curSize/2$
    $curSize \leftarrow curSize * 2$
```

A straight-forward approach to detecting the block size of a cache is illustrated in Algorithm 6. The algorithm first accesses the cache with a stride of *curSize* bytes. Then it measures the misses when accessing the cache again with a stride of 16 bytes (we assume that 16 bytes is a lower bound on the block size). As long as *curSize* is less than or equal to the actual block size, we would expect to get zero misses. If *curSize* is twice the actual block size, we would expect to get a miss for every second block.

However, if the cache uses an *adjacent cache line prefetcher* (see Section 2.4), this approach does not work. A more sophisticated algorithm, that is robust against this form of prefetching, is shown in Algorithm 7.

Algorithm 7: Block Size

```
Input: $A \leftarrow$ Associativity, $ws \leftarrow$ Way Size
Output: Block Size
$measurements[maxSize]$
$curSize \leftarrow 32$
while $curSize < maxSize$ do
    $\vec{a_1} \leftarrow \langle\langle 0, ws, A/2 \rangle\rangle$
    $\vec{a_2} \leftarrow \langle\langle ws * (A/2) + curSize, ws, A/2 + 1 \rangle\rangle$
    $\vec{a} \leftarrow (\vec{a_1} \circ \vec{a_2})^{100}$
    $measurements[curSize] \leftarrow \mathbf{measure}_C(\vec{0}, \vec{a})$
    if "jump occurred" then
        return $curSize$
    $curSize \leftarrow 2 * curSize$
```

The algorithm first accesses $\lceil associativity/2 \rceil$ many elements that map to the same cache set. Then $\lfloor associativity/2 + 1 \rfloor$ many elements are accessed with an offset of $curSize$ such that all accesses map to the same cache set as long as $curSize$ is less than the actual block size, and if $curSize$ exceeds the actual block size, the accesses map to two different cache sets. Thus, in the first case, the cache is not large enough to store all accessed locations, while in the second case, all elements fit into the cache.

4.3 Replacement Policy

4.3.1 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.

Establishing a known logical cache set state Given the fixed miss permutation, as described in Section 2.3, 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. Cache misses can be assured by first accessing a *sufficiently* large number of other memory blocks evicting all previous cache contents.

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

Algorithm 8: Naive Implementation of the Replacement Policy Inference Algorithm.

```
Input: $A \leftarrow$ Associativity
       $B \leftarrow$ Block Size
       $N \leftarrow$ Number of Sets
       $W \leftarrow B \cdot N$ (= Way Size)

procedure initializeBasePointers()
    $emptyBase \leftarrow 0$
    $initBase \leftarrow emptyBase + A \cdot W$
    $evictBase \leftarrow initBase + A \cdot W$
seq function emptyCacheSet(int $set$)
    return $\langle\langle emptyBase + set \cdot B, W, A\rangle\rangle$
seq function initializeSet(int $set$)
    return $\langle\langle initBase + set \cdot B + (A-1) \cdot W, -W, A\rangle\rangle$
seq function accessBlockInSet(int $block$, int $set$)
    return $\langle initBase + set \cdot B + block \cdot W\rangle$
seq function evictKBlocksInSet(int $k$, int $set$)
    return $\langle\langle evictBase + set \cdot B, W, k\rangle\rangle$

int function newPosOfBlockInPerm(int $block$, int $perm$)
    initializeBasePointers()
    $empty \leftarrow$ emptyCacheSet($0$)
    $init \leftarrow$ initializeSet($0$)
    $accPerm \leftarrow$ accessBlockInSet($perm$, $0$)
    $accBlock \leftarrow$ accessBlockInSet($block$, $0$)
    for $newPos \leftarrow A-1$ downto 0 do
        $evictk \leftarrow$ evictKBlocksInSet($A - newPos$, $0$)
        $prep \leftarrow empty \circ init \circ accPerm \circ evictk$
        if measure$_C$( $prep$, $accBlock$) $= 1$ then
            return $newPos$
```

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.

4.3.2 A Naive Implementation

Algorithm 8 shows a naive implementation of the function to determine a block's new position after triggering a particular permutation. We first introduce five helper functions:

- `initializeBasePointers()` initializes the three pointers *emptyBase*, *initBase*, and *evictBase*, which are maintained in global variables. These pointers are used to access disjoint memory areas of the size of the cache. Note that they 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* maps 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 the function `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. 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 discussed in Section 5.2, the measurement process itself, as well as concurrently running tasks, can disturb the state of what is being measured. 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.

4.3.3 A More Robust Implementation

The basic 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 $\mathit{accBlock}'$ be the result of interleaving several copies of *prep* and *accBlock*, respectively. Then, replacing the condition $\mathbf{measure}_C(\mathit{prep}, \mathit{accBlock}) = 1$ by $\mathbf{measure}_C(\mathit{prep}', \mathit{accBlock}') \geq N$, already yields a much more robust algorithm.

Incidentally, the above approach also reduces the effect of hardware-based prefetching (see Section 2.4). 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 change 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 in Section 5.2, starting and stopping the measurement process between the preparatory and the measurement phase may disturb the results. To avoid this disturbance, we replace

$$\mathbf{measure}_C(\mathit{empty}' \circ \mathit{init}' \circ \mathit{accPerm}' \circ \mathit{evictk}', \mathit{accBlock}')$$

by

$$\begin{aligned} &\mathbf{measure}_C(\mathit{empty}', \mathit{init}' \circ \mathit{accPerm}' \circ \mathit{evictk}' \circ \mathit{accBlock}') \\ -&\mathbf{measure}_C(\mathit{empty}', \mathit{init}' \circ \mathit{accPerm}' \circ \mathit{evictk}'). \end{aligned}$$

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 the sequence $\mathit{init}' \circ \mathit{accPerm}' \circ \mathit{evictk}'$ 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.

4.4 Second-level Caches

We assume a *non-inclusive* cache hierarchy, so that the second-level caches can be analyzed independently of first-level caches. 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 caches and would thus be more difficult to analyze. In particular, in strictly-inclusive caches, misses in the (unified) second-level cache that were caused by data accesses could lead to evictions in the first-level instruction cache. Then the code of our algorithms might need to be fetched again, which could again lead to evictions in the second-level cache.

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 is usually the case. Then, between two accesses to the same set of the second-level cache, our algorithms perform accesses in all cache sets of the first-level cache. So all memory accesses lead to misses in the first-level cache and are thus passed to the second-level cache.

5 Implementation

In this chapter, we describe the most important aspects of our implementation. First, we describe how cache misses can be measured. Then, we analyze the impact of measurement errors, and present several countermeasures. After that, we explain how we implemented access sequences both for data and instruction caches. Finally, we briefly describe the implementation of our Android app.

5.1 Measuring Cache Misses

All of our algorithms use the function $\mathbf{measure}_C(\vec{p}, \vec{m})$ to quantify the number of cache misses that occur when running the access sequences $\vec{p}$ and $\vec{m}$. There are two ways to implement this function on an actual system: one can either use performance counters or measure the execution time. We will explain both approaches in the following subsections. Furthermore, we will present a way to measure cache misses when running our implementation in the cache hierarchy simulator *Cachegrind*.

5.1.1 Hardware Performance Counters

Hardware performance counters are special-purpose registers that are used to count the occurrence of various hardware-related events, including the number of committed instructions, branch mispredictions, and the number of hits and misses in different levels of the cache hierarchy. Performance counters are available on many modern processors; one of the first processors to implement them was the Intel Pentium [Spr02].

The "Performance Application Programming Interface" (PAPI) [M+99] provides a common interface to access these counters on a number of different platforms, including all recent Intel and AMD x86 processors, as well as a number of ARM, PowerPC, MIPS, Cray and UltraSparc CPUs. On recent Linux kernels ($\geq$ 2.6.32), PAPI can be used without an additional setup, while older kernels require special drivers or kernel patches.

PAPI comes with two APIs to the underlying counter hardware: a simple high-level API and a low-level API. In our implementation, we use the low-level API because of its higher efficiency and lower overhead.

5.1.2 Measuring the Execution Time

Unfortunately, performance counters are not available on all processors. To support as many platforms as possible, we have also implememented variations of our algorithms that measure the execution time of access sequences instead. Since a cache miss takes more time than a cache hit, the execution time can be used to estimate the number of hits and misses. To perform these measurements, we use the `clock_gettime` function[1].

However, there are a number of caveats to this approach:

- 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 cache have different latencies, which makes it challenging to analyze a specific level.
- Cache optimizations like way prediction (see Section 2.4) can influence the execution time.

5.1.3 Simulation

When performing experiments with early versions of our algorithms, we sometimes obtained unexpected results. However, we were not sure whether these were caused by hardware optimizations like prefetching, or whether there was a flaw in our algorithms or implementations. One idea to analyze these

[1] `pubs.opengroup.org/onlinepubs/9699919799/functions/clock_gettime.html`

kinds of problems was to run our implementation using a simulator that uses a simple memory hierarchy.

One widely used tool to simulate how a program interacts with a machine's cache hierarchy is Cachegrind, which is part of the Valgrind framework [NS07, NS03]. It simulates a machine with separate first-level instruction and data caches and a unified second-level cache.

Unfortunately, Cachegrind only outputs the number of cache misses after running the whole program. For our algorithms, however, we need to have access to this information at runtime. We therefore extended Cachegrind to support this kind of access. To do this, we used Valgrind's "trapdoor mechanism", which allows a program to pass requests to Cachegrind at runtime. This is achieved by macros that insert "into the client program a short sequence of instructions that are effectively a no-op (six highly improbable, value-preserving rotations of register %eax). When Valgrind spots this sequence of instructions during x86 disassembly, the resulting translation causes control to drop into its code for handling client requests." [NS03]

5.2 Dealing with Measurement Errors

Unfortunately, the results of the measurements are (except for the simulation-based approach) often not completely accurate and they usually vary from run to run. There are a number of reasons for this behavior:

- Overhead of the measurement process itself can perturb the results.
- Concurrently running tasks can cause interference.
- The result reported by the performance counters might not represent the actual number of cache misses that occurred, because "chip designers remain reluctant to make guarantees about the accuracy of the counts" [WD10], and "their hardware circuits are not validated like the rest of the CPU" [U^+08].
- The resolution of the clock might be too low (for the time based approach).

In the following paragraphs, we analyze the influence of the measurement overhead and of interfering tasks. All experiments were performed on an Intel Atom D525 (24kB 6-way associative L1 cache, 512kB 8-way associative L2 cache, block size: 64 bytes) running Ubuntu 12.04.1.

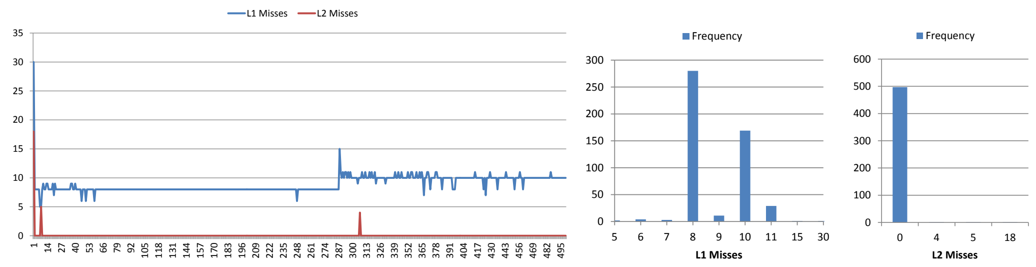


Figure 5.1: Result of measuring an empty sequence using PAPI.

Overhead To analyze the measurement overhead, we first performed an experiment in which we started and stopped the measurement process, without executing any code in between. We repeated this experiment 500 times.

The left diagram of Figure 5.1 shows the number of cache misses for each run; the right diagram shows how often a particular number of misses was measured. We can see that we usually get around 8 to 10 L1 and 0 L2 misses on most runs; on the first run, however, we get significantly more misses. This indicates that the functions to start and stop the performance counters access a number of memory locations; most of these seem to be cached after the first run.

Figure 5.2 shows how this changes, if we access more than cache size many elements before each run of the experiment, i.e. we evict all memory blocks used by the measurement functions. Now, we get in most runs at least as many misses as we got previously in the first run. However, we are not sure why there are two maxima in the distribution plots at around 24 and 55 for the L1 cache, and 17 and 43 for the L2 cache. One possible explanation could be that, because the functions to start and stop the performance counters need more time if there are more cache misses, it is more likely that concurrently running processes are scheduled in between. The memory accesses performed by these can then again lead to more cache misses of the PAPI functions.

Figure 5.3 shows the results of performing the same experiments using the time based approach.

Interference In this paragraph, we analyze how concurrently running tasks affect our measurement results. Although PAPI counts cache misses on a per-process basis, other tasks may alter the cache contents and thus increase the number of cache misses.

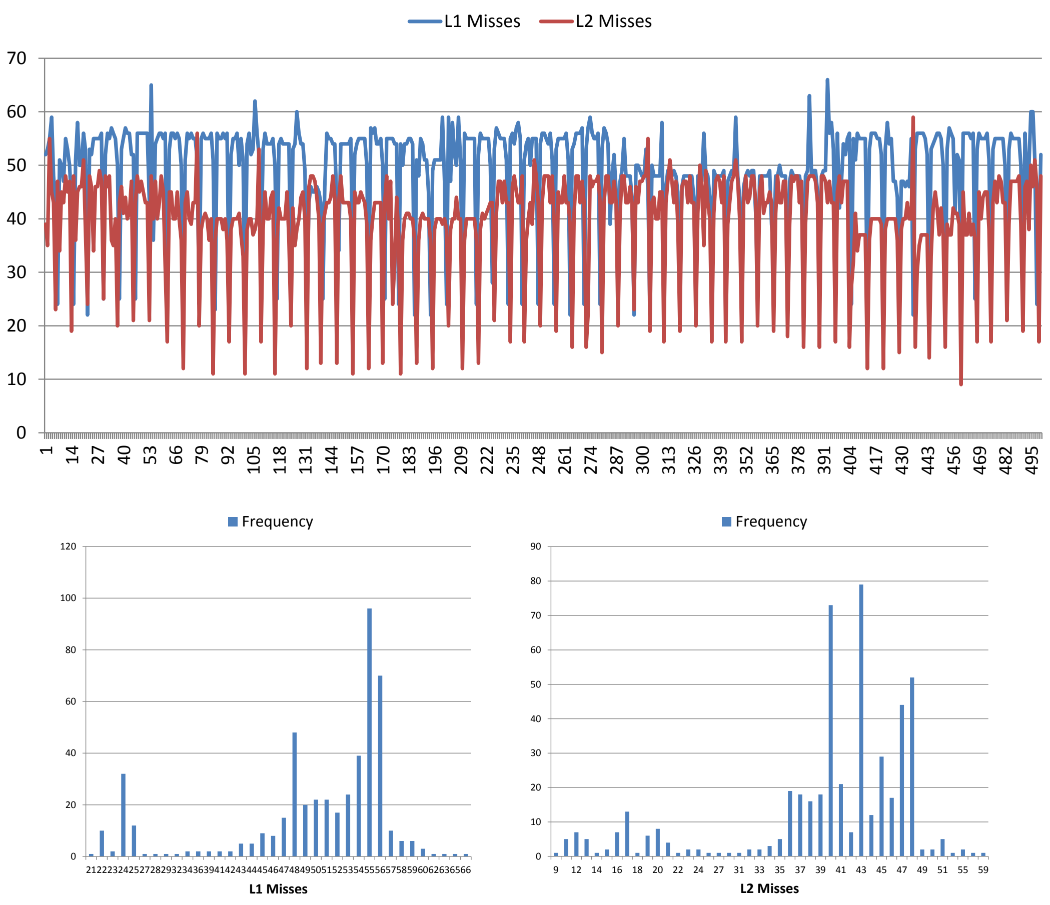


Figure 5.2: Result of measuring an empty sequence using PAPI (with evicting all cache blocks before each measurement).

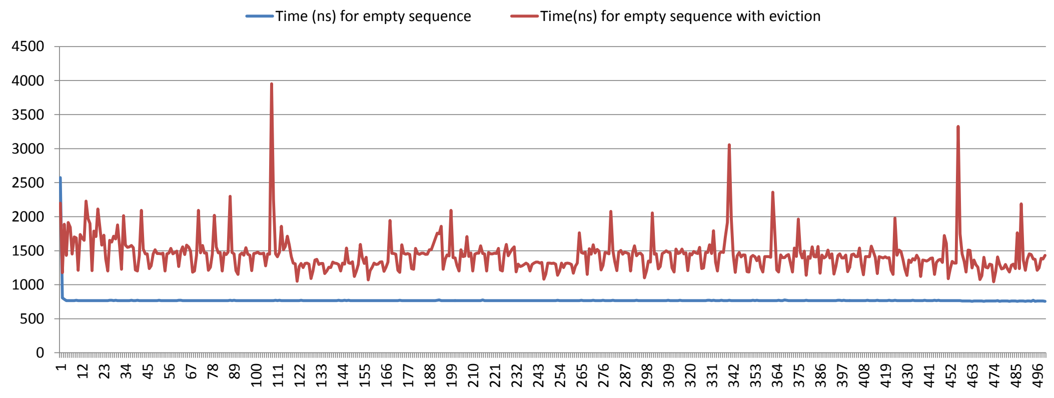


Figure 5.3: Execution time for an empty sequence.

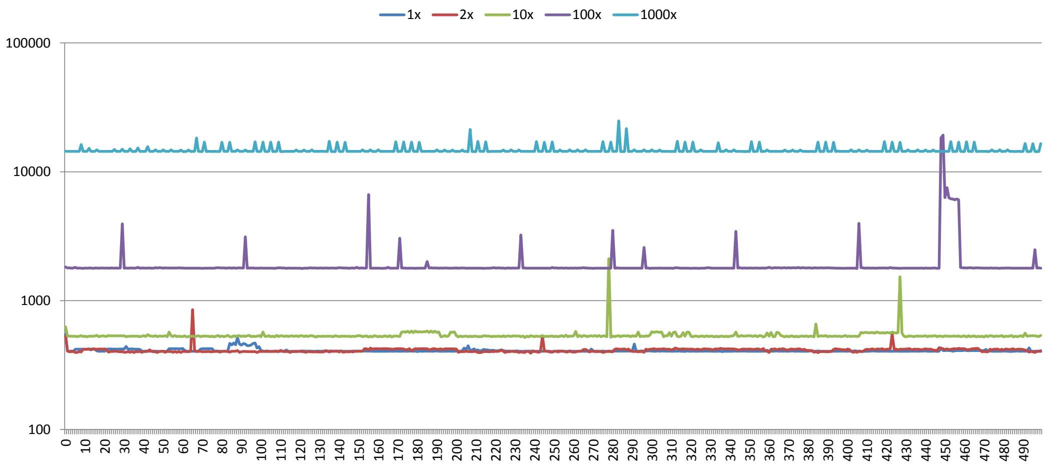


Figure 5.4: Cache misses when accessing all blocks in the cache n times.

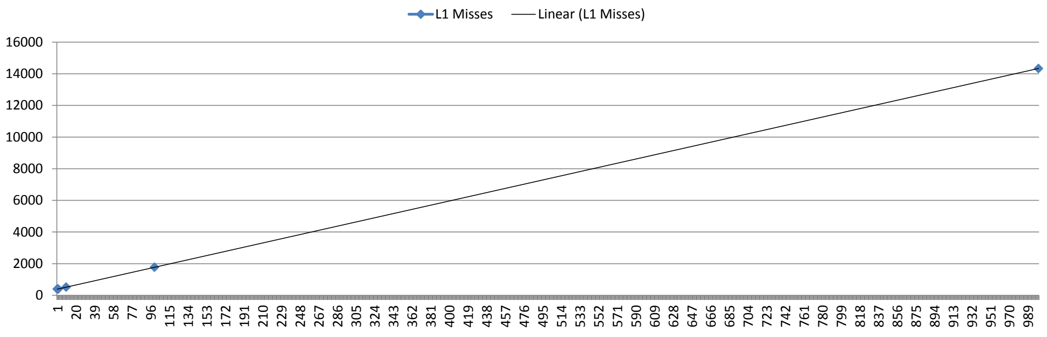


Figure 5.5: Linear interpolation between the minima of the curves from Figure 5.4.

Figure 5.4 shows the result of the following experiment (note the logarithmic scale): We first access a memory area of cache size many bytes with a stride of block size, i.e. every block in the cache is accessed once and should be in the cache afterwards. Then we start the performance counters for the L1 data cache and access the same memory area again n times, where $n \in \{1, 2, 10, 100, 1000\}$. A couple of things are interesting here:

- For $n = 1$, we get about 400 caches misses (the L1 data cache of the Intel Atom D525 CPU has 384 blocks); for $n = 2$, this value does not increase much. So it seems that starting the performance counters leads to memory accesses in all cache sets.
- If we interpolate linearly between the minima for $n = 10$, $n = 100$, and $n = 1000$ (see Figure 5.5), we can see that the slope in the two areas is about the same (around 14). These additional misses might be caused

by tasks that are scheduled periodically and that have a small memory footprint.

- Most of the points on all curves are near the minimum of that curve. However, sometimes a number of successive point are significantly above that value.

A worst-case scenario for our tool would be a concurrently running program that makes as many memory accesses as possible. Since the program we just described makes a lot of memory accesses, especially if n is large, we decided to perform the previous experiment again, this time with a second copy of the program running in the background. Both processes were assigned to the same core using the TASKSET command[2].

Figure 5.6 shows the result of this experiment. At this scale, the curves look rather similar to those of the previous experiment, although the maxima are higher. If we compare the individual curves of this experiment with the previous experiment (Figure 5.7 shows a part of both curves for $n = 10$), we see that the minimum of both curves is roughly the same. However, in the second experiment, quite a number of runs caused about 384 (i.e. the number blocks in the cache) more misses than the minimum. This is probably caused by interference from the concurrently running program.

Figure 5.8 and 5.9 show the result of the same experiment for the time-based approach.

5.2.1 Countermeasures

Based on these observations, we implemented our algorithms such that we perform all measurements repeatedly and keep only the minimum. As to the number of repetitions necessary for a robust implementation, we found that 10 repetitions usually provide a reasonable trade-off between run-time and accuracy, at least for algorithms relying on big jumps. For the replacement policy algorithm, higher values (e.g. 100 or 1000) might be necessary, as this algorithm requires exact values. An alternative approach was proposed by Cooper and Sandoval [CS11]: They perform measurements until the minimum doesn't change anymore. However, we haven't implemented this approach yet.

[2]`http://linuxcommand.org/man_pages/taskset1.html`

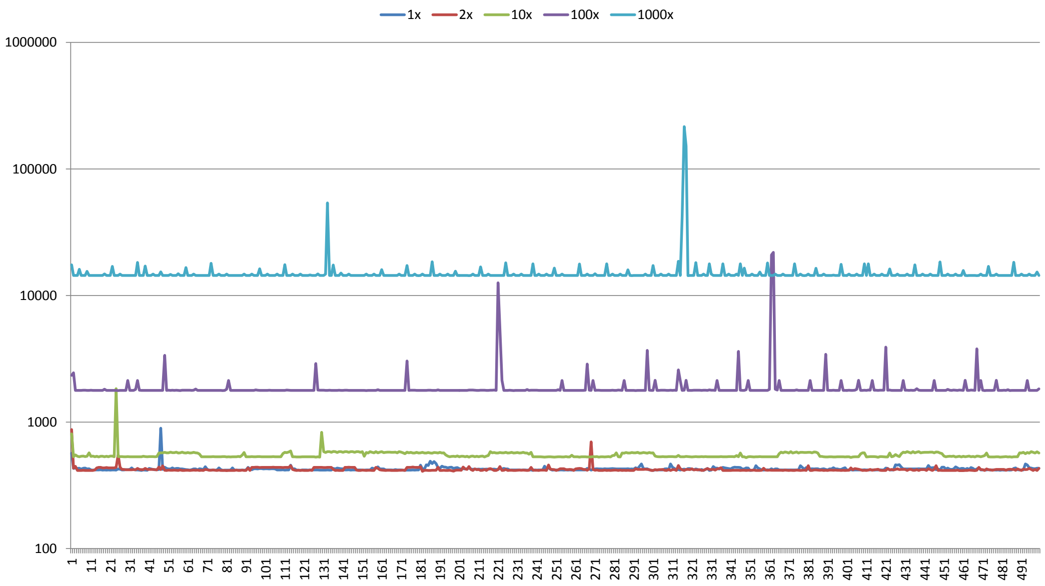


Figure 5.6: Cache misses when accessing all blocks in the cache n times, with concurrently running program.

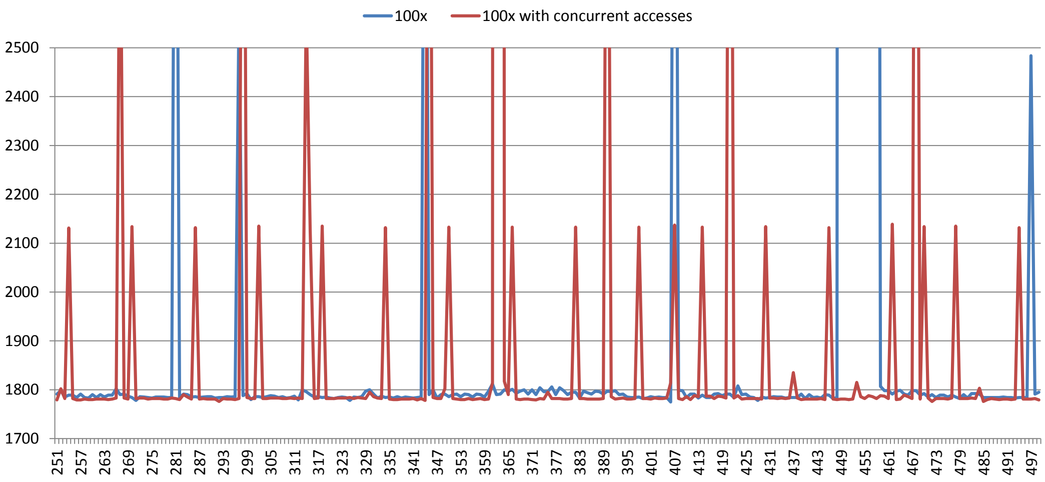


Figure 5.7: Cache misses when accessing all blocks in the cache 100 times, with and without concurrently running program.

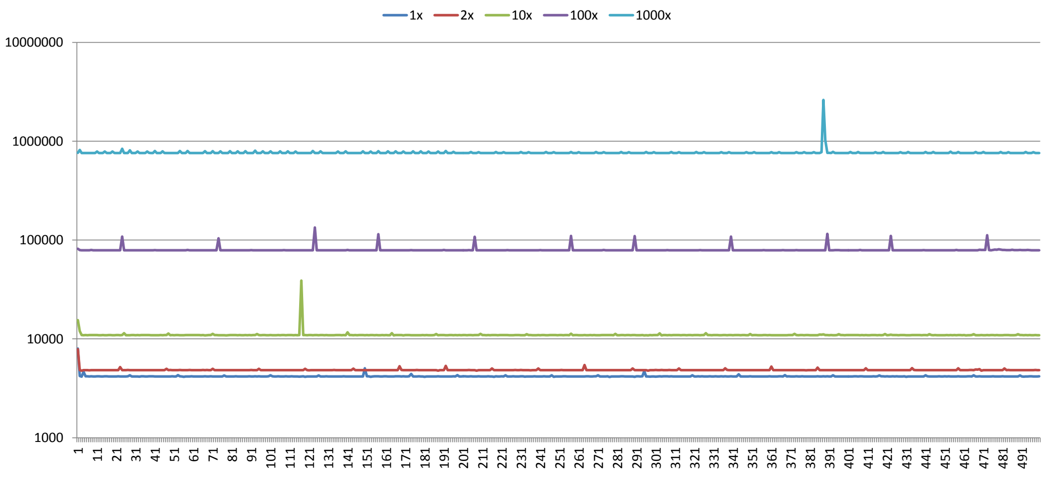


Figure 5.8: Execution time for accessing all blocks in the cache n times.

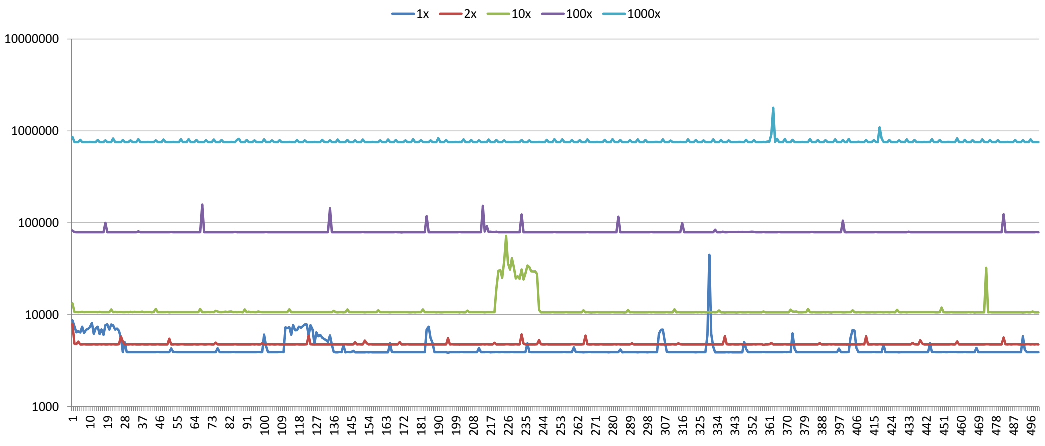


Figure 5.9: Execution time for accessing all blocks in the cache n times, with concurrently running program.

5.3 Implementing Access Sequences

As already pointed out in Section 4, we serialize memory accesses by using a form of "pointer chasing" where each memory location contains the address of the next access. This helps us to minimize the effects of non-blocking caches and out-of-order execution. The following subsections explain how we implemented this technique both for data and instruction caches.

5.3.1 Implementing Access Sequences for Data Caches

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

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

Furthermore, to minimize the effect of cache prefetching, we have implemented a function to shuffle the sequences, using an "inside-out" version[3] of the Fisher-Yates shuffle, as implemented by Durstenfeld [Dur64].

5.3.2 Implementing Access Sequences for Instruction Caches

To analyze instruction caches of x86 processors, we first allocate a large enough memory area `code` and declare it executable by calling the mprotect[4] function. Next, we create a list with the memory locations we want to access in this memory area.

We have implemented a function `createCodeArray` that populates `code` with a sequence of jump instructions to the memory locations in the list. If the list contains the same memory location more than once, an offset is added such that the access still maps to the same cache block. Note that this limits the possible number of accesses to the same location. Furthermore,

[3]http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle#The_.22inside-out.22_algorithm

[4]http://pubs.opengroup.org/onlinepubs/7908799/xsh/mprotect.html

Listing 5.1: createCodeArray()

```
code[0]=0x55; //push %rbp
code[1]=0x48; code[2]=0x89; code[3]=0xe5; //mov %rsp,%rbp
code[4]=0x48; code[5]=0x83; code[6]=0xec;
        code[7]=0x10; //sub $0x10,%rsp
code[8]=0xc7; code[9]=0x45; code[10]=0xfc;
        code[11]=0x00; code[12]=0x00; code[13]=0x00;
        code[14]=0x00; //movl $0x0, 0x4(%rbp)
code[15]=0xeb; code[16]=0x0e; //jmp 2d
code[17]=0xb8; code[18]=0x00; code[19]=0x00;
        code[20]=0x00; code[21]=0x00; //mov $0x0,%eax
code[22]=0xe8; code[23]=0x0f; code[24]=0x00;
        code[25]=0x00; code[26]=0x00; //callq code[42]
code[27]=0x83; code[28]=0x45; code[29]=0xfc;
        code[30]=0x01; //addl $0x1, 0x4(%rbp)
code[31]=0x81; code[32]=0x7d; code[33]=0xfc; //cmpl
        *(int32_t*)(&code[34])=repetitions 1;
code[38]=0x7e; code[39]=0xe9; //jle 1f
code[40]=0xc9; //leaveq
code[41]=0xc3; //retq
```

Listing 5.2: Code used to obtain the machine instructions.

```
int main() {
        int i;
        for (i=0; i<1000 ;i++) {
                some_function();
        }
}
```

`createCodeArray` takes a parameter `repetitions` that allows us to specify how often the whole access sequence should be executed.

More precisely, `createCodeArray` fills `code` with the machine instructions for a function call, a for-loop and the corresponding jump instructions. This code can then be called with the statement `((void(*)(void))code)();`

Listing 5.1 shows an excerpt from this function. The opcodes were obtained by compiling the C-Code from Listing 5.2 with GCC, and replacing the value of the loop boundary (here: 1000) with the value of the parameter `repetitions`, and the call to `some_function` with a call to `code[42]`. `code[42]` is assumed to contain a jump to the first memory location we want to access, and the last memory location is assumed to contain a return instruction such that the control flow returns to `code[27]` for the next iteration of the loop. The code that creates the sequence of jumps is not shown here, as it is rather lengthy because it requires a lot of bookkeeping to avoid overwriting previous jump instructions when accessing the same location more than once, and to compute the correct relative jump addresses.

The advantage of our approach is that this code needs to be generated only once for every instruction set. And thus, unlike in the approach described by [Y^+06], access to the compiler is not necessary at runtime. Moreover, it seems possible to automate the process of obtaining the opcodes for a given instruction set.

5.4 Implementation of the Android App

Android apps are usually written in Java. However, Java does not allow for direct memory access, which is required by our algorithms. We therefore use the *Android Native Development Kit (NDK)*[5], which makes it possible to implement parts of an app using native-code languages like C and C++. The native code can be called from Java using the *Java Native Interface (JNI)*[6]. This technique allowed us to reuse parts of our existing implementation with only minor changes.

All Android-based devices we could test so far come with ARM processors that either use PSEUDO RANDOM or PSEUDO ROUND ROBIN replacement policies. Both of these policies can not be detected by our approach, and they also complicate the inference of second-level cache parameters as it might not be possible to ensure that all accesses lead to misses in the first-level cache. Therefore, our app currently only infers the size, the associativity, and the block size of first-level data caches.

[5]`http://developer.android.com/tools/sdk/ndk/index.html`
[6]`http://docs.oracle.com/javase/6/docs/technotes/guides/jni/`

6 Experimental Evaluation

In this chapter, we evaluate our implementation experimentally. First, we run our tools and available existing tools on a number of different x86 CPUs. After that, we describe the most interesting results.

6.1 Evaluation of chi-PC & chi-T on Different CPUs

The goal of this section is to run the two versions of our tool on different platforms, and to compare the results with existing tools. While a number of tools have been described in the literature (see Chapter 3), we were only able to find a few of them online, namely LMBENCH [MS96], CALIBRATOR [Man04], RCT [CS11], and SERVET [GD+10]. In addition, Keshav Pingali provided us with a version of X-RAY [Y+04, Y+05b, Y+05c, Y+05a, Y+06]. All of these tools use timing information to estimate the number of cache misses.

Table 6.1 shows the results of running *chi-PC*, *chi-T* and the tools mentioned above on a number of x86 processors introduced in the last twelve years. Empty cells in the table mean that the corresponding tool does not try to infer this parameter. The results for instruction caches are shown in a separate table, Table 6.2, as none of the other tools analyzes instruction caches.

Both versions of our tool were able to successfully infer the cache parameters on many machines; on the Intel Atom D525 processor, we discovered a previously undocumented approximation of LRU that will be described in more detail in section 6.1.2.

However, on some machines we were not able to infer all of the parameters. In

Table 6.1: Results of running available tools on different platforms.

		L1 Data				L2 Unified			
Architecture	Tool	Size (in kB)	Associativity	Block Size (in Bytes)	Policy	Size (in kB)	Associativity	Block Size (in Bytes)	Policy
Intel Pentium 3 900	Actual	16	4	32	PLRU	256	8	32	PLRU
	chi-PC	16	4	32	PLRU	256	8	32	PLRU
	chi-T	16	4	32	PLRU	256	8	32	PLRU[2]
	X-Ray	16	4	32		256	4	8192	
	RCT	16	4	32		256		32	
	lmbench	16		32		256		32	
	Calibrator	16		32		256		32	
	Servet	16				256			
Intel Atom D525	Actual	24	6	64	ATOM[1]	512	8	64	PLRU
	chi-PC	24	6	64	ATOM[1]	512	8	64	PLRU
	chi-T	24	6	64	ATOM[1]	512	8	64	PLRU
	X-Ray	24	6	64		256	16	4096	
	RCT	24	6	64		256		128	
	lmbench	24		128		512		128	
	Calibrator	24		64		384		512	
	Servet	16				512			
Intel Core 2 Quad Q9550	Actual	32	8	64	PLRU	6144	24	64	PLRU
	chi-PC	32	8	64	PLRU	6144	24	64	[3,4]
	chi-T	32	8	64	PLRU	6144	24	64	[3,4]
	X-Ray	32	8	64		1024	4	[6]	
	RCT	32	8	64		4096		64	
	lmbench	32		64		6144		128	
	Calibrator	32		64		6144		256	
	Servet	16				6144			
Intel Core 2 Duo E8400	Actual	32	8	64	PLRU	6144	24	64	PLRU
	chi-PC	32	8	64	PLRU	6144	24	64	[3,4]
	chi-T	32	8	64	PLRU	6144	24	64	[3,4]
	X-Ray	64	4	4096		1024	4	[6]	
	RCT	32	8	64		4096		128	
	lmbench	32		64		6144		128	
	Calibrator	32		64		6144		128	
	Servet	32				6144			
Intel Core 2 Duo E6750	Actual	32	8	64	PLRU	4096	16	64	PLRU
	chi-PC	32	8	64	PLRU	4096	16	64	[3,4]
	chi-T	32	8	64	PLRU	4096	16	64	[3,4]
	X-Ray	64	4	4096		1024	4	[6]	
	RCT	32	8	64		2560		128	
	lmbench	32		64		4096		128	
	Calibrator	32		64		4096		128	
	Servet	32				5120			
Intel Core 2 Duo E6300	Actual	32	8	64	PLRU	2048	8	64	PLRU
	chi-PC	32	8	64	PLRU	2048	8	64	PLRU
	chi-T	32	8	64	PLRU	2048	8	8	PLRU
	X-Ray	64	4	4096		1024	4	[6]	
	RCT	32	8	64		1024		64	
	lmbench	32		64		2048		128	
	Calibrator	32		64		2048		128	
	Servet	32				2048			

		L1 Data				L2 Unified			
Architecture	Tool	Size (in kB)	Associativity	Block Size (in Bytes)	Policy	Size (in kB)	Associativity	Block Size (in Bytes)	Policy
Intel Xeon W3550	Actual	32	8	64	256	8		64	PLRU
	chi-PC	32	8	64	PLRU	256	8	64	PLRU
	chi-T	32	8	64	PLRU	256	8	64	PLRU
	X-Ray	36	9	64		256	4	16384	
	RCT	32	8	64		224		64	
	lmbench	32		64		256		128	
	Calibrator	32		64		160		256	
	Servet	16				256			
Intel Core i5 460M	Actual	32	8	64	PLRU	256	8	64	PLRU
	chi-PC	32	8	64	PLRU	256	8	64	PLRU
	chi-T	32	8	64	[3]	256	8	64	PLRU[2]
	X-Ray	36	9	64		256	4	16384	
	RCT	32	8	64		224		64	
	lmbench	32		64		256		0	
	Calibrator	32		64		256		256	
	Servet	16				256			
AMD Athlon 64 X2 4850e	Actual	64	2	64	LRU	512	16	64	PLRU
	chi-PC	64	2	64	LRU	[5]	[5]	[5]	[5]
	chi-T	64	2	64	LRU	[5]	[5]	[5]	[5]
	X-Ray	64	2	64		728	91	64	
	RCT	64	2	64		384		64	
	lmbench	768		128		[7]		[7]	
	Calibrator	80		32		512		128	
	Servet	64				512			
AMD Opteron 8360SE	Actual	64	2	64	LRU	512	16	64	PLRU
	chi-PC	64	2	64	LRU	[5]	[5]	[5]	[5]
	chi-T	64	2	64	LRU	[5]	[5]	[5]	[5]
	X-Ray	64	2	64		6144	48	[6]	
	RCT	64	2	64		512		64	
	lmbench	64		64		512		64	
	Calibrator	64		128		512		128	
	Servet	64				2048			

[1] see section 6.1.2
[2] the results were not stable, i.e., only some runs reported the correct value
[3] the replacement policy could not be inferred, i.e., the result did not form a permutation
[4] see section 6.1.1
[5] these CPUs have exclusive L2 caches; this is currently not supported by our tools
[6] `ERROR: hashtable_string_integer__is_value_null(j) ./source/nbm/cache_prepare.h(300)`
[7] lmbench detected no L2 cache on this CPU

Table 6.2: Instruction cache results.

Architecture	Tool	L1 Instruction			
		Size (in kB)	Associativity	Block Size (in Bytes)	Policy
Intel Pentium 3 900	Actual	16	4	32	PLRU
	chi-PC	16	4	32	PLRU
	chi-T	16[1]	4	32	[2]
Intel Atom D525	Actual	32	8	64	PLRU
	chi-PC	32	8	64	PLRU
	chi-T	32	8	64	[2]
Intel Core 2 Quad Q9550	Actual	32	8	64	PLRU
	chi-PC	32	8	64	PLRU
	chi-T	32	8	64	[2]
Intel Core 2 Duo E8400	Actual	32	8	64	PLRU
	chi-PC	32	8	64	PLRU
	chi-T	32	8	64	[2]
Intel Core 2 Duo E6750	Actual	32	8	64	PLRU
	chi-PC	32	8	64	PLRU
	chi-T	32	8	64	[2]
Intel Core 2 Duo E6300	Actual	32	8	64	PLRU
	chi-PC	32	8	64	PLRU
	chi-T	32	8	64	[2]
Intel Xeon W3550	Actual	32	4	64	PLRU
	chi-PC	32	4	64	[2]
	chi-T	32	4	64	[2]
Intel Core i5 460M	Actual	32	4	64	PLRU
	chi-PC	32	4	64	[2]
	chi-T	32	4	64	[2]
AMD Athlon 64 X2 4850e	Actual	64	2	64	LRU
	chi-PC	64	2	64	LRU
	chi-T	64	2	64	LRU
AMD Opteron 8360SE	Actual	64	2	64	LRU
	chi-PC	64	2	64	LRU
	chi-T	64	2	64	LRU

[1] the results were not stable, i.e., only some runs reported the correct value

[2] the replacement policy could not be inferred, i.e., the result did not form a permutation

the following paragraphs, we will describe these cases more in detail, except for the problems regarding the L2 replacement policies of several Core 2 Duo and Core 2 Quad CPUs, which will be analyzed more thoroughly in section 6.1.1.

AMD Athlon 64 X2 4850e and Opteron 8360SE Both AMD CPUs have exclusive second-level caches with the same way size as their first-level caches. Our algorithms are currently not able to infer their L2 cache parameters.

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[1].

Time-based inference of instruction cache replacement policies Using our time-based inference algorithms, we were only able to infer the instruction cache replacement policies of the AMD processors we examined. On the Intel CPUs, the reported results did not form a permutation, though they resembled the PLRU policy in most cases. We have not been able to find an explanation for this behavior yet. One reason might be that the individual measurements take only about $4 - 10\mu s$ (on an Intel Core 2 Duo E6750), and the expected differences between the measurements are less than $1\mu s$. This might be too low to get reproducible results. Another reason might be an additional optimization technique used by Intel CPUs that we are not aware of.

6.1.1 Core 2 Duo & Core 2 Quad Replacement Policies

The L2 replacement policy could not be inferred by our algorithm on an Intel Core 2 Duo E6750 (4 MB, 16-way set-associative), an Intel Core 2 Duo E8400 (6 MB, 24-way set-associative), and an Intel Core 2 Quad Q9550 (6 MB, 24-way set-associative), i.e., the new positions determined by the function `newPosOfBlockInPerm` did not form a permutation. However, on an Intel Core 2 Duo E6300 (2 MB, 8-way set-associative), the PLRU replacement

[1]`http://www.bit-tech.net/hardware/cpus/2008/11/03/intel-core-i7-nehalem-architecture-dive/5`.

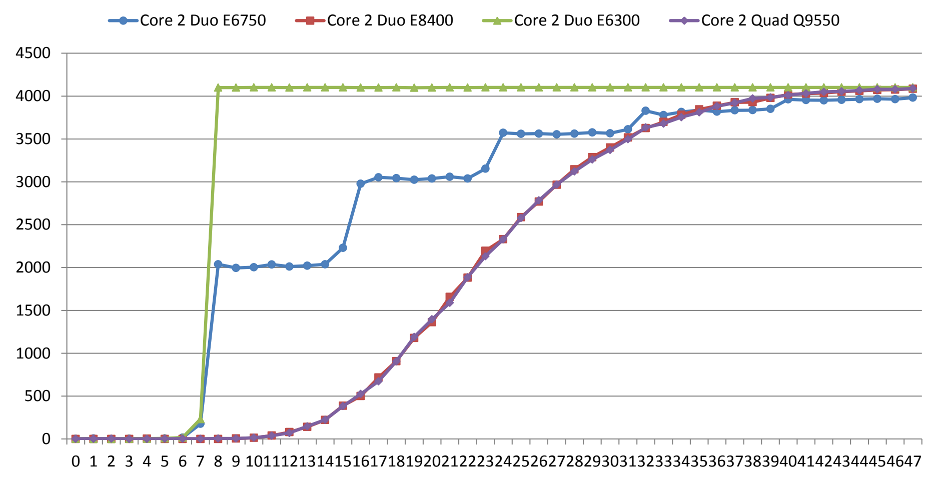


Figure 6.1: Experimental analysis of L2 cache behavior of the Intel Core 2 Duo E6750, E6300 and E8400, and Core 2 Quad Q9550.

policy was inferred. According to Intel, all of these CPUs "use some variation of a pseudo LRU replacement algorithm" [Sin12]. 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 number of cache sets many misses otherwise. Figure 6.1 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 and the Core 2 Quad machine, 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.

$$\begin{aligned}
\Pi_0^{\mathrm{ATOM}} &= (0,1,2,3,4,5)\\
\Pi_1^{\mathrm{ATOM}} &= (1,0,2,4,3,5)\\
\Pi_2^{\mathrm{ATOM}} &= (2,0,1,5,3,4)\\
\Pi_3^{\mathrm{ATOM}} &= (3,1,2,0,4,5)\\
\Pi_4^{\mathrm{ATOM}} &= (4,0,2,1,3,5)\\
\Pi_5^{\mathrm{ATOM}} &= (5,0,1,2,3,4)
\end{aligned}$$

Figure 6.2: Permutation vectors for Intel Atom D525.

6.1.2 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 6.2 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 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 seems to approximate LRU. Previously described implementations of pseudo-LRU [AZ+04] were based on perfect binary trees and thus required the associativity to be a power of two.

While we are not sure how this replacement policy is actually implemented, we have figured out one possible way to implement such a policy: One can divide the cache into three groups, with each such group holding two cache lines. Then both the groups and the elements of the groups are managed by LRU policies. So, upon a cache hit, the group containing the accessed element becomes the most-recently-used group and the accessed element becomes the most-recently-used element of this group. Upon a cache miss, the least-recently-used element in the least-recently-used group is evicted. Figure 6.3 illustrates this approach with two example access sequences. Here, the least-recently-used groups and elements are marked red, the most-recently-used groups and elements green, and the remaining group gray.

Using Relacs[2] [RG08], we have determined that this replacement policy is $(1,0)$-competitive relative to LRU at associativity 4. This means that all cache analyses previously developed for LRU can be immediately applied in WCET analyses of the Intel Atom D525.

[2]`http://rw4.cs.uni-saarland.de/~reineke/relacs`.

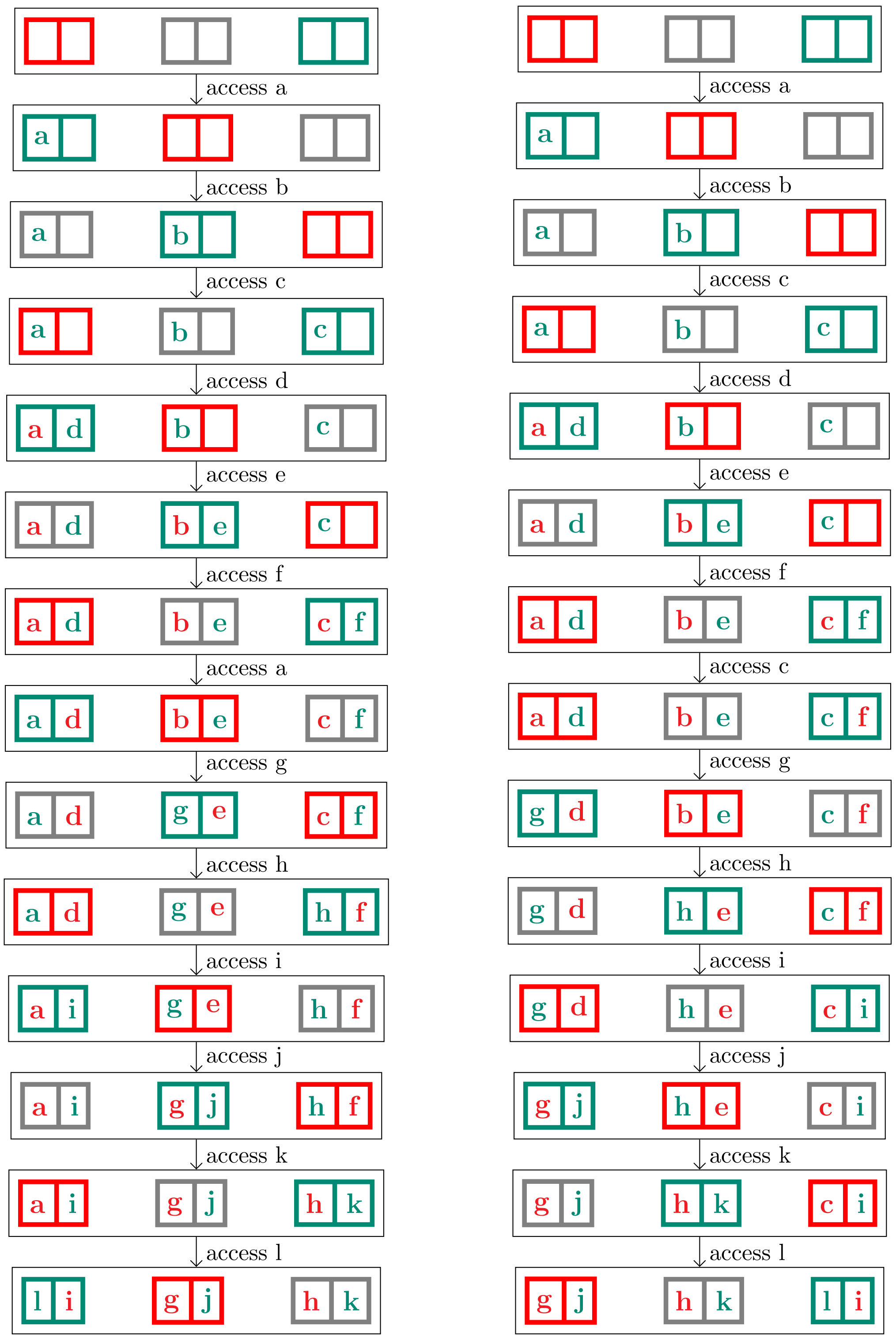


Figure 6.3: Example sequences for a possible implementation of the replacement policy used by the Intel Atom D525. The least-recently-used elements and groups are marked red, the most-recently-used ones green.

6.1.3 Discussion

It is important to mention that we had access to most of the machines we used for the evaluation while implementing our algorithms. Though we were able to infer the replacement policies of most of these systems, this does not necessarily indicate how well our implementation would perform on other systems.

6.1.4 Experimental Setup

The experiments were performed using different versions of Linux, depending on what was already installed on the machines we examined. Aside from enabling support for huge pages (which is required for our current implementation of the L2 replacement policy inference algorithm), 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 counters, we used PAPI in version 4.4.0. The code was compiled with GCC at optimization level 0 to avoid compiler optimizations that could influence the measurement results. The execution time of our algorithm was usually less than one minute.

We used the other tools, to which we compared our implementation, in the following versions: lmbench 3.0-a9, Calibrator 0.9e, Servet 2.0, RCT as of June 13, 2012 and X-Ray as of October 19, 2006.

Device	L1 Size	L1 Associativity	L1 Block Size
LG GT540	32kB	4	32 bytes
HTC Nexus One	32kB	16	32 bytes
Samsung Galaxy Nexus	32kB	4	32 bytes
Samsung GT-I9100	32kB	4	32 bytes
Samsung GT-I9300	32kB	4	32 bytes
Samsung GT-N8000	32kB	4	32 bytes
HP Touchpad	32kB	16	32 bytes

Table 6.3: Result of evaluating the Android app.

6.2 Evaluation of the Android App

Table 6.3 shows the result of evaluating our Android app on a number of smartphones and tablets. Unfortunately, reliable and sufficiently precise documentations of the processors used in these devices are often unavailable or hard to find. We therefore only present the values that our tool reported.

7 Conclusions

We developed a novel algorithm to automatically infer the replacement policy of a cache using a series of measurements. To this end, we introduced permutation policies, a class of replacement policies that admits efficient inference and includes widely used 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.

Our algorithm requires knowledge of a number of other cache parameters, namely the cache size, the associativity and the block size. We designed and improved algorithms to automatically infer these parameters as well. We developed novel approaches for handling both physically indexed caches and instruction caches.

Based on these algorithms, we implemented two tools, *chi-PC* and *chi-T*, that can automatically detect the cache sizes, the associativities, the block sizes, and the replacement policies of first- and second-level data and instruction caches. To estimate the number of cache misses, *chi-PC* uses hardware performance counters, while *chi-T* measures the execution time. Both tools can be run on standard Linux systems without any modifications. Furthermore, in order to extend the scope of our work to embedded processors, we implemented an Android App that can infer the size, the associativity, and the block size of first-level data caches.

We evaluated our approach on a number of different systems. We successfully determined the replacement policies on most of the evaluated systems. On the Intel Atom D525, we discovered a—to our knowledge—previously undocumented approximation of least-recently-used replacement.

7.1 Future Work

The work presented in this thesis can be extended in a couple of directions. Short-term goals include

- investigating further why the replacement policies of the Core 2 Duo E6750 and E8400, and the Core 2 Quad Q9550 could not be detected.
- evaluating our tools on more platforms, in particular on embedded systems.
- implementing algorithms to infer other cache parameters like the latency or the write policy.

In the medium term, one could

- find a more general class of replacement policies that still admits efficient inference.
- analyze trace and victim caches.
- implement algorithms to deal with strictly inclusive or exclusive caches.
- develop approaches to infer properties of other architectural features such as TLBs, branch predictors or prefetchers.
- analyze cache coherency protocols used by shared caches in multi-core processors.

In the long term, one could thus arrive at a tool that could automatically detect all micro-architectural parameters necessary to build WCET analyzers or cycle-accurate simulators.

Bibliography

[Ang87] D. Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[Ang88] D. Angluin. Queries and concept learning. *Machine learning*, 2(4):319–342, 1988.

[AR12] 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.

[AR13] A. Abel and J. Reineke. Measurement-based modeling of the cache replacement policy. In *RTAS*, April 2013. To appear.

[ARM10] ARM. Cortex-A9 Technical Reference Manual, 2010.

[AZ$^+$04] H. Al-Zoubi et al. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In *ACM-SE 42*, pages 267–272, New York, NY, USA, 2004.

[BACD97] 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.

[BC00] J. M. Blanquer and R. C. Chalmers. MOB: A memory organization benchmark. `http://www.gnu-darwin.org/www001/ports-1.5a-CURRENT/devel/mob/work/mob-0.1.0/doc/mob.ps`, 2000.

[BT09] V. Babka and P. Tůma. Investigating cache parameters of x86 family processors. In *Proceedings of the 2009 SPEC Benchmark Workshop*, pages 77–96, 2009.

[C$^+$10] K. D. Cooper et al. The platform-aware compilation environment, preliminary design document. `http://pace.rice.edu/uploadedFiles/Publications/PACEDesignDocument.pdf`, September 2010.

[C$^+$11] S. Cassel et al. A succinct canonical register automaton model. In *ATVA*, pages 366–380, 2011.

[CD01] C. Coleman and J. Davidson. Automatic memory hierarchy characterization. In *ISPASS*, pages 103–110, 2001.

[Cha11] V. Chandran. Robust method to determine cache and TLB characteristics. Master's thesis, The Ohio State University, 2011.

[CS11] K. Cooper and J. Sandoval. Portable techniques to find effective memory hierarchy parameters. Technical report, Rice University, 2011.

[CWPD01] R. Clint Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–35, 2001.

[D$^+$04] J. Dongarra et al. Accurate cache and TLB characterization using hardware counters. In *ICCS*, pages 432–439, 2004.

[DSGP08] A. Duchateau, A. Sidelnik, M. Garzarán, and D. Padua. P-ray: A suite of micro-benchmarks for multi-core architectures. In *Proc. 21st Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC'08)*, volume 5335, pages 187–201, 2008.

[Dur64] R. Durstenfeld. Algorithm 235: Random permutation. *Commun. ACM*, 7(7):420–, July 1964.

[FH04] C. Ferdinand and R. Heckmann. aiT: Worst-case execution time prediction by static program analysis. *Building the Information Society*, pages 377–383, 2004.

[FJ94] K. Farkas and N. Jouppi. *Complexity/performance tradeoffs with non-blocking loads*, volume 22. IEEE Computer Society Press, 1994.

[FJ05] M. Frigo and S. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[GD$^+$10] J. González-Domínguez et al. Servet: A benchmark suite for autotuning on multicore clusters. In *IPDPS*, pages 1–9. IEEE, 2010.

[GLYY12] N. Guan, M. Lv, W. Yi, and G. Yu. WCET analysis with MRU caches: Challenging LRU for predictability. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 55–64. IEEE, 2012.

[Gru12] D. Grund. *Static Cache Analysis for Real-Time Systems – LRU, FIFO, PLRU.* PhD thesis, Saarland University, 2012.

[H$^+$12] F. Howar et al. Inferring canonical register automata. In *VMCAI*, pages 251–266, 2012.

[HP11a] D. Hardy and I. Puaut. WCET analysis of instruction cache hierarchies. *Journal of Systems Architecture*, 57(7):677–694, 2011.

[HP11b] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[HSU$^+$01] G. Hinton, D. Sager, M. Upton, D. Boggs, et al. The microarchitecture of the Pentium® 4 processor. *Intel Technology Journal*, 1:2001, 2001.

[Int12a] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* Number 248966-026. April 2012.

[Int12b] Intel Corporation. *Intel® Processor Identification and the CPUID Instruction.* Application Note 485. May 2012.

[JB07] T. John and R. Baumgartl. Exact cache characterization by experimental parameter extraction. In *RTNS*, pages 65–74, Nancy, France, 2007.

[LGBT05] C. Liu, I. Ganusov, M. Burtscher, and S. Tiwari. Bridging the processor-memory performance gap with 3D IC technology. *Design & Test of Computers, IEEE*, 22(6):556–564, 2005.

[LT98] E. Li and C. Thomborson. Data cache parameter measurements. In *Computer Design, International Conference on*, page 376, Los Alamitos, CA, USA, 1998. IEEE.

[M$^+$99] P. J. Mucci et al. PAPI: A portable interface to hardware performance counters. In *Proceedings of the DoD HPCMP Users Group Conference*, pages 7–10, 1999.

[Man04] S. Manegold. The calibrator (v0.9e), a cache-memory and TLB calibration tool. `http://homepages.cwi.nl/~manegold/Calibrator/`, June 2004.

[MS96] L. McVoy and C. Staelin. lmbench: portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 23–23, Berkeley, CA, USA, 1996.

[NS03] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic notes in theoretical computer science*, 89(2):44–66, 2003.

[NS07] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 42(6):89–100, 2007.

[Rei08] J. Reineke. *Caches in WCET Analysis.* PhD thesis, Universität des Saarlandes, November 2008.

[RG08] J. Reineke and D. Grund. Relative competitive analysis of cache replacement policies. In *LCTES*, pages 51–60, New York, NY, USA, 2008. ACM.

[S^+04] T. S. B. Sudarshan et al. Highly efficient LRU implementations for high associativity cache memory. In *ADCOM*, pages 87–95, Ahmedabad, India, 2004.

[SCW^+02] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 21–21. IEEE, 2002.

[Sin12] R. Singhal. Personal communication, August 2012.

[Spr02] B. Sprunt. The basics of performance-monitoring hardware. *Micro, IEEE*, 22(4):64 – 71, jul/aug 2002.

[SS95] A. J. Smith and R. H. Saavedra. Measuring cache and TLB performance and their effect on benchmark runtimes. *IEEE Trans. Comput.*, 44(10):1223–1235, October 1995.

[TY00] 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*, pages 383–390, July 2000.

[U^+08] L. Uhsadel et al. Exploiting hardware performance counters. In *FDTC*, pages 59–67, 2008.

[UM09] V. Uzelac and A. Milenkovic. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *ISPASS*, pages 207–217. IEEE, 2009.

[W^+10] H. Wong et al. Demystifying GPU microarchitecture through microbenchmarking. In *ISPASS*, pages 235–246, 2010.

[WD10] V. Weaver and J. Dongarra. Can hardware performance counters produce expected, deterministic results. In *3rd Workshop on Functionality of Hardware Performance Monitoring, Atlanta, GA (December 4, 2010)*, 2010.

[Y^+04] K. Yotov et al. X-ray: Automatic measurement of hardware parameters. 2004.

[Y^+05a] K. Yotov et al. Automatic measurement of hardware parameters for embedded processors. 2005.

[Y^+05b] K. Yotov et al. Automatic measurement of memory hierarchy parameters. In *SIGMETRICS*, pages 181–192, New York, NY, USA, 2005. ACM.

[Y^+05c] K. Yotov et al. X-ray: A tool for automatic measurement of hardware parameters. In *QEST*, pages 168–177, Washington, DC, USA, 2005. IEEE.

[Y^+06] K. Yotov et al. Automatic measurement of instruction cache capacity. In *LCPC*, pages 230–243. Springer-Verlag, 2006.

[You07] M. T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *ISPASS*, pages 23–34, 2007.

[ZV04] C. Zhang and F. Vahid. Using a victim buffer in an application-specific memory hierarchy. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 1, pages 220–225. IEEE, 2004.

List of Figures

List of Tables

List of Algorithms