Automatic Cache Modeling by Measurements

Andreas Abel and Jan Reineke
Saarland University
abeand@studcs.uni-saarland.de, reineke@cs.uni-saarland.de

Abstract

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

In this paper, we propose a method to infer a number of these properties automatically. In particular, we describe algorithms to detect the cache size, associativity and block size.

1 Introduction

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

As a consequence, engineers are forced to either create "conservative" models, which yield imprecise WCET estimates without being provably sound, or to obtain a better understanding of the microarchitecture by other means. An engineer might contact the processor manufacturer to inquire more details, which the manufacturer is often not willing or able to provide to protect his intellectual property. As a last resort, the engineer is often found tinkering with evaluation boards, performing measurements on microbenchmarks, iteratively refining his models of architectural features, until he is sufficiently confident in his conjectured model. Clearly, this process is both costly and error-prone.

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

In this paper, we focus on the memory hierarchy. The memory hierarchy is an attractive target, because it has a strong influence on execution times. Further, modern microarchitectures feature hardware performance counters, which can be used to count the number of cache accesses and misses that a program generates.

We have developed algorithms to determine the capacities, associativities, block sizes of first-level data caches. At a very high level of abstraction, our algorithms all follow the following scheme:

1. Generate multiple sequences of memory accesses.
2. Measure the number of cache misses (alternatively: the execution time) on each of the sequences.
3. Deduce the property of interest from the measurement results and some structural assumptions.

In the following section, we review related work. In Section 3, we provide the necessary background regarding caches, performance counters, and time measurements. Section 4 then describes our algorithms and Section 5 their implementation. In Section 6, we conclude with a summary of potential future work.

2 Related Work

Several publications have presented approaches for determining parameters like the cache size, the associativity and the block size through measurements. Some of these approaches try to detect the actual parameters automatically while others require the user to interpret the measurement results manually. However, most of the proposed tools are not available online, except LMBENCH [5] and CALIBRATOR [4]. In addition, we were able to obtain X-RAY [7] by contacting one of the authors.

Experiments with the existing tools on modern hardware have shown that the results are often unreliable. We assume

that this is due to the complex optimizations used in the memory hierarchies of recent CPUs, like prefetching and non-blocking caches.

Moreover, some of the proposed approaches require special real-time operating system environments and need to be adapted to every different processor architecture [3]. Others make assumptions on certain cache parameters, e.g. that the cache size is a power of two, which is not the case on several modern processors.

There are two different ways to perform the measurements: one can either measure the time a particular part of the program needs or one can use hardware performance counters to record the number of cache misses.

Using timing information has the advantage of better portability since hardware performance counters are not available on all platforms. A number of papers use this approach (e.g. [7, 1]). Using performance counters, on the other hand, yields more accurate measurement results. More recent papers tend to prefer this approach (e.g. [2, 3]).

3 Cache Organization

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

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

4 Cache Parameter Inference Algorithms

We follow an iterative approach. First, we infer the capacity C of the cache. We then determine the associativity A of the cache, under the assumption of the capacity derived before. Given both capacity and associativity, we derive the block size.

4.1 Capacity

We use the following algorithm to infer the size of the cache.

Algorithm 1: Cache Size

```
char a[maxSize]
m_old ← 0
curCap ← 1024
while curCap < maxSize do
    reset performance counters
    access 1000x a[0], a[1], ..., a[curCap]
    m_old ←number of cache misses
    if m − m_old > threshold then
        return curCap − 1
    curCap ← curCap + 1024
    m_old ← m
```

The algorithm accesses a contiguous memory area of $curCap$ bytes repeatedly. As long as $curCap$ is less than or equal to the actual capacity, all accessed bytes fit into the cache and thus, no misses should occur. If $curCap$ exceeds the capacity, we expect to see a sharp increase in the number of misses.
Figure 1a shows the result of running this algorithm on an Intel Atom D525 CPU which features a 24kB L1 Cache with associativity 6 and block size 64.

4.2 Associativity

Given the cache size, the following algorithm determines the associativity.

The algorithm uses the fact that the cache size is 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.
Figure 1b shows the result of running this algorithm on the architecture mentioned above.

Algorithm 2: Associativity

Input: cache size cs
$char\ a[maxAssoc * s]$
$m_{old} \leftarrow 0$
$curAssoc \leftarrow 1$
while $curAssoc < maxAssoc$ **do**
  reset performance counters
  access 1000x $a[cs], a[2 * cs], ..., a[curAssoc * cs]$
  $m_{old} \leftarrow$number of cache misses
  if $m - m_{old} > threshold$ **then**
    return $curAssoc - 1$
  $curAssoc \leftarrow curAssoc + 1$
  $m_{old} \leftarrow m$

4.3 Block Size

Given the cache size and the associativity, the following algorithm infers the block size.

Algorithm 3: Block size

Input: cache size cs, associativity assoc
$char\ a[associativity * cachesize]$
$m_{old} \leftarrow 0$
$curSize \leftarrow 1$
while $true$ **do**
  reset performance counters
  access 1000x
  $a[0], a[cs], ..., a[\lceil assoc/2 \rceil - 1) * s], a[\lceil assoc/2 \rceil * cs + curSize], ..., a[assoc * cs + curSize]$
  $m_{old} \leftarrow$number of cache misses
  if $m - m_{old} < threshold$ **then**
    return $curAssoc - 1$
  $curSize \leftarrow curSize * 2$
  $m_{old} \leftarrow m$

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. Figure 1c shows again the result of running this algorithm.

5 Implementation of Inference Algorithms

In this section we describe how we implemented our algorithms. First, we show the techniques we used to measure cache misses. Then, we examine a number of challenges arising on modern hardware and how we deal with them.

5.1 Measuring Cache Misses

To measure the number of cache misses, existing approaches have either used hardware performance counters or timing information.

Performance Counters Hardware Performance Counters are special-purpose registers that can count the number of various hardware-related events, including the number of hits and misses in different stages of the memory hierarchy. They are available on many modern processors. The "Performance Application Programming Interface" (PAPI) [6] provides a common interface to access these counters on a number of different platforms, including all recent Intel and AMD x86 processors.

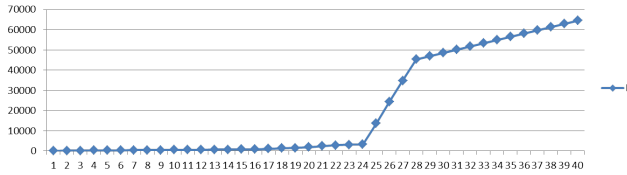
Measuring Execution Time Another approach is to measure the execution time of short code fragments. Since a cache miss takes more time than a cache hit, this can be used to estimate the number of hits/misses. This approach has the advantage of being available on virtually all platforms. However, there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

- A processor might not offer a timer that is precise enough to capture the time differences when executing very short code fragments.
- Modern CPUs feature non-blocking caches which can serve other requests while fetching the data for a miss.

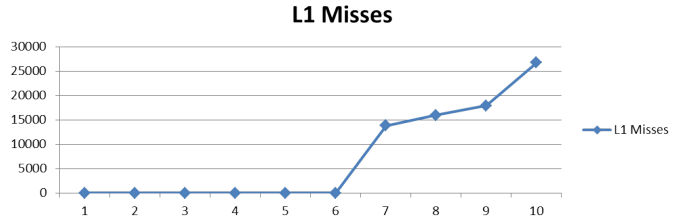
In our implementation, we leverage hardware performance counters using PAPI whenever they are available because of the higher accuracy they offer. Furthermore, we have implemented a second version of our algorithms that uses timing based information. This allows us to obtain a high accuracy on many modern processors while at the same time also supporting as many different platforms as possible.

5.2 Dealing with Interference

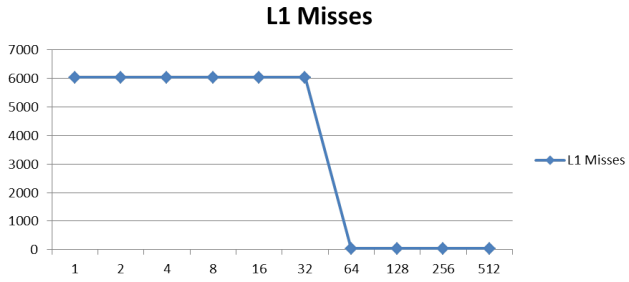
Although the performance counters count events on a per process basis, other processes running in parallel can alter the cache contents and thus increase the number of cache misses. To minimize this problem, we can perform the same measurements repeatedly and keep the minimum of the measurement results.



(a) Cache size.



(b) Associativity.



(c) Block size.

Figure 1: Result of running the algorithms to infer the cache size, associativity, and block size on an Intel Atom D525 CPU.

5.3 Prefetching

The memory hierarchies of modern processors often make use of prefetcher that can detect constant strides in memory accesses and memory locations are brought into the cache before they are actually accessed. To deal with this problem, we "shuffle" our access sequences so that the accesses are performed in a random order. Moreover, we implemented a form of "pointer chasing" where each memory location contains the address of the next access.

6 Conclusions and Future Work

We have presented algorithms to infer the size, associativity, and block size of an L1 data cache through measurements. Initial experimental evaluation of these algorithms on the Intel Atom D525 has been promising.

In the future, we plan to extend our algorithms to L2 and L3 caches, as well as to instruction and unified caches. Generating a particular sequence of instruction accesses, determined at runtime, is more challenging than generating a sequence of data accesses. We would also like to infer a model of the replacement policy through measurements.

We also plan to develop algorithms to infer other aspects of the microarchitecture that affect a program's execution time. Our vision is to eventually automate most of the work involved in the construction of timing models for worst-case execution time analysis.

References

[1] J. M. Blanquer and R. C. Chalmers. MOB: A memory organization benchmark. Technical report, 2000.

[2] C. Coleman and J. Davidson. Automatic memory hierarchy characterization. In *Performance Analysis of Systems and Software, 2001. ISPASS. 2001 IEEE International Symposium on*, pages 103 –110, 2001.

[3] T. John and R. Baumgartl. Exact cache characterization by experimental parameter extraction. In *Proceedings of the 15th International Conference on Real-Time and Network Systems (RTNS)*, pages 65–74, Nancy : INPL, 2007.

[4] S. Manegold and P. Boncz. Cache-memory and tlb calibration tool. online: http://homepages. cwi. nl/˜ manegold, 2009.

[5] L. McVoy, C. Staelin, et al. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294. San Diego, CA, USA, 1996.

[6] P. J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[7] K. Yotov, K. Pingali, and P. Stodghill. Automatic measurement of memory hierarchy parameters. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '05, pages 181–192, New York, NY, USA, 2005. ACM.