INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Masterarbeit

CacheHound: Automated Reverse-Engineering of CPU Cache Policies in Modern Multiprocessors

Simon Hilchenbach

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Masterarbeit

CacheHound: Automated Reverse-Engineering of CPU Cache Policies in Modern Multiprocessors

Simon Hilchenbach

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Sergej Breiter Dr. Karl Fürlinger
Abgabetermin:	24. September 2024

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 23. September 2024

. . . . . .
(Unterschrift des Kandidaten)

Abstract

In modern multiprocessors, hardware manufacturers employ a hierarchy of CPU caches to mitigate the considerable latency associated with accessing main memory. These CPU caches leverage the temporal and spatial locality of an application's data access patterns to serve a portion of the main memory at significantly reduced latencies. The operation of CPU caches is governed by cache policies.

While this solution is effective in the majority of scenarios, an application may encounter difficulties in performing optimally under a given cache policy, potentially leading to issues such as thrashing. Awareness of the policy would facilitate the restructuring of the application to align with it. Such knowledge can be further applied to the domain of cache-based side-channels, from both a hardening and an attacker perspective.

However, manufacturers typically refrain from disclosing the details of their cache policies, particularly those pertaining to the placement and replacement of data within the cache. Prior research has focused on the reverse-engineering of replacement policies, yet we are not aware of any investigation into placement policies. Moreover, to the best of our knowledge, there is currently no generic framework for the reverse-engineering of CPU caches.

In this work, we devise such a framework and also develop a methodology for the reverse-engineering of placement policies. We provide a corresponding open-source implementation, called CacheHound, and benchmark it on several x86- and ARM-based systems. Finally, we employ the gained knowledge to explore use cases in the fields of security and high-performance computing (HPC).

Contents

1. Introduction

The ongoing advancements in CPU design, multi-core computing and semiconductor technology have resulted in a situation where the speed of processors is outpacing the rate at which data can be transferred with the main memory. Recent benchmarks show a memory access latency of 70ns on the Intel Skylake and an even higher latency of 110ns on the newer Intel Cannonlake architecture [Lem19]. In light of the aforementioned access latencies, a typical processor with a clock frequency of 3.4GHz must await the availability of requested data for 238 and 374 cycles, respectively. It is important to note that each instruction fetch entails a memory read. Consequently, even CPU-bound applications are concerned about memory speed. The phenomenon whereby a specific threshold is reached and memory speed subsequently becomes the limiting factor for performance is referred to as the memory-wall problem [WM95].

CPU manufacturers employ various approaches to heighten this threshold or, figuratively speaking, to move the memory-wall further into the future: Techniques such as pipelining, superscalar execution, instruction-level parallelism, branch prediction, speculative execution, out-of-order execution and prefetching are employed with the objective of hiding memory access latency [Mac02].

This work focuses on CPU caches which are another technique to hide the latency. CPU caches are SRAM units situated between the CPU and main memory. These CPU caches exploit the temporal and spatial locality of an application's data access patterns to serve a fraction of the main memory at significantly lower latencies. A CPU cache must operate according to a specific logic that determines which data to retain in the relatively smaller SRAM and which to discard. This operation is governed by a cache policy.

It is common practice among processor manufacturers to refrain from disclosing the details of their cache policies, particularly the placement and replacement policies, which dictate where data is stored within the cache and which data is evicted when the cache reaches its capacity. The objective of this work is to develop a solution for reverse-engineering these CPU cache policies.

1.1. Motivation

Applications that are structured in a manner that optimizes the use of the CPU cache are referred to as cache-friendly. In the domain of high-performance computing (HPC), cache-friendliness is a highly sought-after property, as it has a considerable influence on the overall performance of the application. Consequently, one of the initial improvements implemented by software developers is the optimization of an algorithm's memory accesses.

Typically, these optimizations consider only the cache size and neglect to take the cache policy into account, e.g., the widely applied technique *Cache blocking* processes data in cache-sized chunks [Sha19]. Incorporating the cache policy into the optimization process can facilitate additional enhancements.

For instance, one potential issue that can arise with CPU caches is thrashing. Thrashing describes a situation in which the same cache line is repeatedly swapped in and out of memory [Han98a]. This phenomenon occurs when two or more data accesses in a hot code path are handled by the same cache set. Knowledge about the placement policy would enable compilers and linkers to layout the code in such a way to prevent this situation. This knowledge could also be employed to use a CPU cache as scratchpad memory. They differ in that scratchpad memory offers explicit control whereas a CPU cache is transparent to the user.

Lastly, the placement policy is also interesting from a security perspective since many studied cache-based attacks implicitly assume a textbook index function and break otherwise. Knowledge of the non-standard placement policy would allow these attacks to be carried out.

1.2. Contribution

Prior research has concentrated on the reverse-engineering of cache replacement policies (cf. section 3.2), employing a variety of methodologies. Nevertheless, to the best of our knowledge, no investigation into placement policies has been conducted thus far. This apparent absence of research can be attributed to the fact that the placement policy is typically expected to be based on an index function that performs a simple modulo on the memory address, as can be found in any textbook on CPU caches [Han98b; HP18; Fox24a]. However, the Fujitsu A64FX ARMv8 microprocessor deviates from this pattern by employing a non-trivial index function [Fuj22]. We believe that this is the only microprocessor which is publicly documented not to follow the textbook.

We note that both architectures considered in our work, x86 and ARMv8, allow for such deviation from the textbook function. In the case of x86, the "Deterministic Cache Parameters Leaf" on the `CPUID` instruction includes a "Complex Cache Indexing" bit which denotes that a "complex function is used to index the cache, potentially using all address bits" [Int24]. This bit is usually set for the sliced L3 cache on Intel. For ARMv8, the architecture reference manual states that "the set number is an `IMPLEMENTATION DEFINED` function of an address" [Arm24a].

Moreover, prior reverse-engineering methodologies have been developed within the context of the analyzed platform, which has typically been Intel machines. We are not aware of any attempts to develop a generic framework within which various cache reverse-engineering techniques can be implemented.

Our contribution encompasses these two elements. In particular, we present a mathematical model of cached memory that serves to abstract the reverse-engineering of CPU caches. In this model, we present our approach for inferring the placement policy. Additionally, we demonstrate how an existing approach for the reverse-engineering of replacement policies can be implemented within this model.

Furthermore, we contribute an open-source implementation of our framework and methodology which we call CacheHound. CacheHound features an architecture that minimizes noise when reverse-engineering the CPU cache by offloading measurements to a dedicated CPU core. The accompanying CacheHound kernel module is compatible with 64-bit Intel, AMD and ARMv8 systems running Linux.

1.3. Overview

In order to achieve the objective of this work, we first explore the design of CPU caches and cache policies in chapter 2. This is followed by a literature review on existing studies on this or related to this topic in chapter 3. In chapter 4 we devise strategies and approaches of reverse-engineering, and implement these in chapter 5. We then use the implementation to reverse-engineer cache policies on various systems in chapter 6. Table 1.2 serves as an overview of the systems we examine. Lastly, in chapter 7 we explore use cases in HPC as well as security, and discuss this work.

This work is only concerned with CPU data and unified caches. There are other caches present within modern CPUs, such as the TLB, BTB and the Micro-OP-Cache, which are not covered. Although the TLB and instruction caches fall outside the scope of our implementation, they are compatible with our approach. We discuss how they can be incorporated into CacheHound in the final chapters.

System / Mainboard	**CPU**	**Instruction set**	**Memory**
Raspberry Pi 5	BCM2712 (Cortex-A76)	ARMv8	4GiB
Supermicro X9DRH-7TF	Intel Xeon E5-2680 v2	x86_64 (Ivy Bridge EP)	64GiB
HPE ProLiant DL385 Gen10 Plus	AMD EPYC 7302	x86_64 (Zen 2)	128GiB
Cray CS500	Fujitsu A64FX	ARMv8	64GiB

Table 1.2.: Overview of systems selected for examination.

2. Cache design

CPU caches are static random-access memory (SRAM) units located between the CPU and main memory to serve a fraction of the memory at significantly lower latencies. This chapter provides an overview of CPU caches, including their types, parameters and implementation principles. The second half of the chapter discusses the various choices for cache policies, specifically cache line replacement options.

2.1. Principle of locality

Main memory can be accessed randomly, but practice has shown that program data accesses typically follow a certain pattern, known as the principle of locality (or locality of reference). This means that data which has been accessed recently is likely to be accessed again shortly (temporal locality) and other data which is in close proximity to it is more likely to be accessed than data further away (spatial locality) [Han98c; Fox24a].

Listing 2.1 demonstrates the principle of locality: The function `int sum(int*, size_t)` calculates the sum over all entries in the contiguous C-array starting at `int* ptr`. For the implementation, the variable `int acc` acts as the currently accumulated value while a `for`-loop iterates over the entries from low to high addresses.

```
int sum(int* ptr, size_t len) {
    int acc = 0;
    for(size_t i = 0; i < len; i++) {
        acc += ptr[i];
    }
    return acc;
}
```

Listing 2.1: Exemplary code which follows the principle of locality.

The iteration over the array exhibits spatial locality as the accessed entry is in close proximity to the previously read one. The variable `int acc` demonstrates temporal locality as it is updated regularly, specifically in every iteration. If one were to trace the memory accesses during code execution, the resulting plot would resemble[1] fig. 2.1.

2.2. Memory hierarchy

The memory hierarchy leverages the principle of locality by storing frequently accessed data in small but fast storage, while still providing the advantages of larger but slower storage. The fastest storage available in a computer system is that provided by CPU registers.

[1] In practice, a compiler is likely to store the variable `int acc` in a processor register.

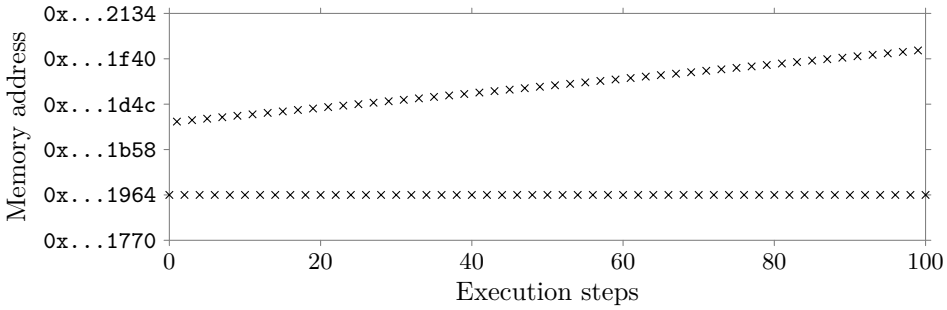

Figure 2.1.: Exemplary memory accesses plotted over time.

However, only a limited number of data words are available per core. The slowest available storage is dependent upon the system in question and may range from flash storage, spinning disks, or even magnetic tape, which the operating system is able to utilize for memory swapping [Fox24b].

CPU caches constitute a specific type of storage situated between the CPU registers and main memory within the aforementioned hierarchical memory structure. Typically, there are multiple levels of CPU caches, denoted as L1, L2, and so on, with lower levels being closer to the CPU. The specific composition of the caches varies depending on the CPU manufacturer and model. Some caches are dedicated to each CPU core, whereas others are shared between multiple cores [Fox24a].

The modern computer typically adheres to the *von Neumann architecture*, whereby the main memory stores both data and code. This is distinct from the first level of CPU caches, which commonly follows the *Harvard architecture* with a split instruction cache (L1i) and data cache (L1d).

Table 2.1 serves as a reference for the typical latencies and sizes of CPU registers, different levels of caches and main memory.

Storage class	**Latency**	**Size**
CPU register	0.5 cycles	64 bits
L1d cache	5 cycles	48 KiB
L2 cache	13 cycles	512 KiB
L3 cache	42 cycles	8 MiB
Main memory	42 cycles + 64 ns	16 GiB

Table 2.1.: Overview of typical sizes and latencies of storage classes [Pav19].

One of the systems examined in this work is equipped with the AMD EPYC 7302 processor, which comprises 16 physical CPU cores (32 SMT threads). Figure 2.2 shows the cache hierarchy on this chip. Each core, and, thus, each pair of SMT threads, is equipped with a 32KiB L1 instruction cache and a 32KiB L1 data cache. The instruction cache is exclusively utilized for the retrieval of instructions, whereas the data cache is employed for the reading and writing of data to memory. Each pair of L1 instruction and data caches is backed by a single L2 cache which is significantly larger at 512KiB. A single L3 cache, with a capacity of

128MiB, is shared among the entire socket.[2] This cache is also referred to as the Last-Level Cache (LLC), as it represents the final cache before the main memory. Although the cache organization used by AMD is also found in other chips, such as those from Intel, there is no standardization across different manufacturers.

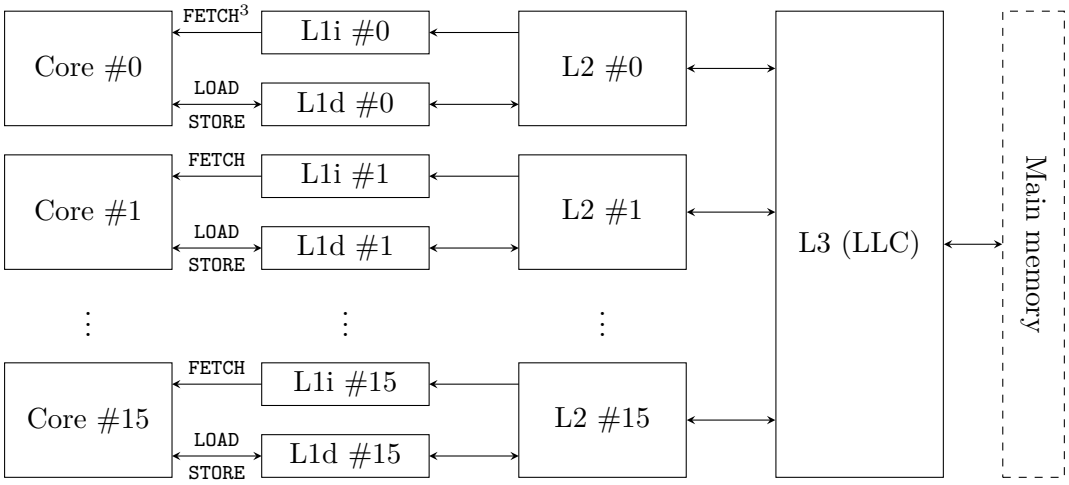

Figure 2.2.: Cache hierarchy of the AMD EPYC 7302.

2.3. High-level overview of caches

As illustrated in fig. 2.2, all loads and stores pass the cache hierarchy in a layer-by-layer fashion. The CPU is not directly connected to the main memory; rather, it is linked to the first level of caches. In the event that the first level cache is unable to satisfy a request (*cache miss*), the request is then forwarded to the second level cache, and so on. This process is repeated until either a cache is able to fulfill the request (*cache hit*) or the main memory is reached [Fox24a].

In this manner, the caches are managed transparently by the CPU. From the perspective of software, including the operating system, caches are hidden and operate implicitly, without requiring explicit control. It is noteworthy that the figures illustrate the logical configuration. Hardware designers have various implementation options and some architectures permit the deliberate bypassing of caches. Additionally, the L3 cache is not typically a single, large SRAM unit, but rather divided among the number of cores, with each core holding a *slice* of the L3 cache. This allows for the parallelization of L3 cache accesses [HWH13], and is likely also done to modularize the chip and aid in product binning.

2.3.1. Cache lines

A cache is comprised of multiple entries, known as cache lines. In order to operate, it is necessary for the system to have a method for determining whether a requested memory

[2]The information on the cache sizes was obtained by exploring the files under the sysfs directory `/sys/devices/system/cpu/cpu0/cache/` on the AMD EPYC 7302 system running Linux.

[3]A `FETCH` is performed during the Fetch-Decode-Execute cycle to obtain the next instruction.

location is *cached*, that is, whether a cache line exists that contains a copy of that location. To achieve this, each line is assigned a *tag* based on the address in the main memory [Han98b]. Cache lines are not single bytes or words, but rather larger, to take advantage of spatial locality and the more efficient transfer of contiguous chunks of memory in DRAM [Fox24a]. The size of cache lines in contemporary processors varies at the discretion of the processor manufacturer. Intel and AMD processors have 64-byte cache lines, while Apple's M1 processor lineup and other ARM processors like the A64FX use 128-byte and 256-byte cache lines, respectively.

Each time the CPU accesses a location in main memory, the entire cache line is loaded into the cache. The memory address that points to the beginning of a cache line is obtained by zeroing the lower bits according to the line size. These lower bits are used as the offset into the cache line [Fox24a]. Given a cache line size of $2^{b_{\text{offset}}}$ bytes, b_{offset} lower offset bits are used. For instance, a 64-byte cache line uses a 6-bit offset.

The remaining upper bits uniquely identify the cache line in the cache. Depending on the cache parameters, the upper bits are split between tag and index bits. The following sections will address this topic in greater detail.

2.3.2. Associativity

An additional crucial element of a cache, in conjunction with its capacity, is its associativity. A cache is classified as a *fully associative cache* if it possesses the capability to store a copy of a given memory location within any of its cache lines. In this type of cache, the tag of the memory address is compared to each cache line to determine if the address is cached [HP18].

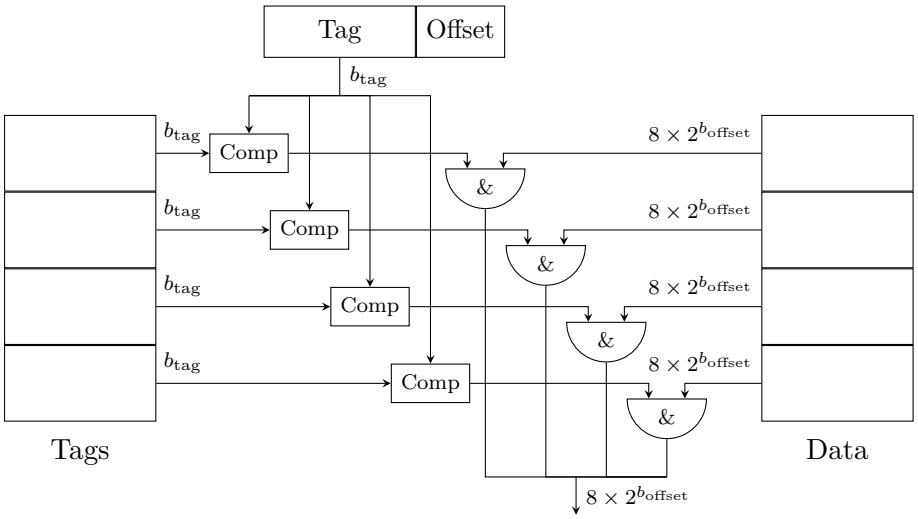

Figure 2.3.: Schematic of reading from a fully associative cache [Dre07].

Figure 2.3 depicts the schematic of this setup, with the cache lines divided into the tag section on the left and the data section on the right. The width of the wire is indicated by the accompanying letter or term, or by 1 in the absence of a letter. Each cache line is equipped with a comparator and an AND-gate which are used to make a comparison between the stored tag bits and the b_{tag} tag bits of the requested address. In the event of a match, the cached copy of $2^{b_{\text{offset}}}$ bytes ($8 \times 2^{b_{\text{offset}}}$ bits) is returned.

The schematic depicts a cache with only four lines, yet a 2 MiB cache with 64-byte lines would have nearly 33,000 entries in a fully associative configuration. This would necessitate 33000 comparators, each requiring b_{tag} XNOR-gates for bit-level matching and $log_2(b_{\text{tag}})$ AND-gates to aggregate the results. These gates are composed of several transistors each, resulting in a large chip space requirement. An iterative comparison could reduce the number of transistors needed for implementation at the expense of increased latency which is incompatible with the nature of caches [Dre07]. In practice, most caches are not fully associative.

At the opposite end of the associativity spectrum is the *direct-mapped cache*, in which each memory location can only be stored in a particular cache line. As a result, the direct-mapped cache only needs to check a single cache line, unlike the fully associative cache, which must compare the tag to every line [HP18]. Figure 2.4 illustrates this concept.

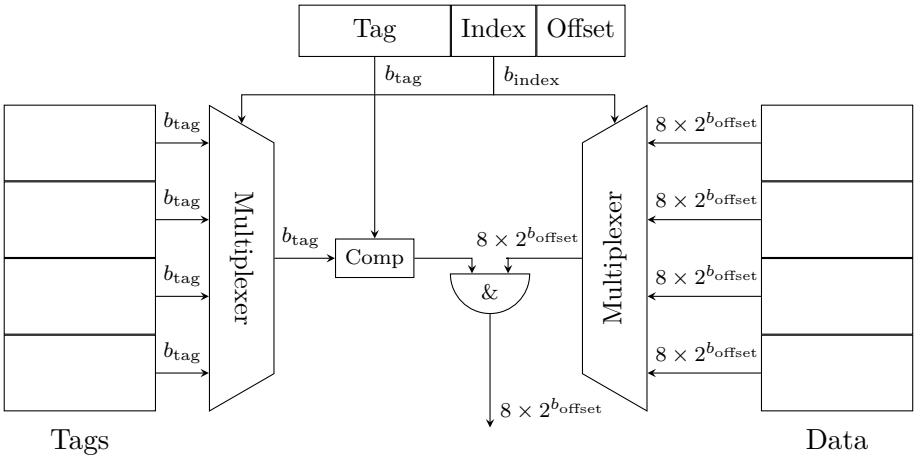

Figure 2.4.: Schematic of reading from a direct-mapped cache [Dre07].

The implementation of this cache necessitates only a single comparator, as only one cache line must be compared. However, a multiplexer is necessary to select the line to compare. The number of transistors required to implement a multiplexer grows in the order of $\mathcal{O}(k \log N)$ where N is the number of cache lines and k is the width of the wire [Dre07]. In the depicted case there are $N = 2^{b_{\text{index}}}$ cache lines and $k = b_{\text{tag}} + 8 \times 2^{b_{\text{offset}}}$ bits required per line. Therefore, the chip space necessary for a direct-mapped cache grows only logarithmically and it is already smaller to begin with.

Note that a portion of the requested address is used to select the cache line. There are b_{index} index bits to select from one of the $2^{b_{\text{index}}}$ cache lines. Since every $2^{b_{\text{offset}}}$ addresses the next cache line is mapped to, the b_{index} lower bits are redundant and, thus, stripped from the tag [Dre07].

Although a direct-mapped cache is easier to implement, it has the drawback of not handling unevenly distributed memory accesses well. In situations where a program frequently accesses two or more addresses that map to the same cache line, the cache lines get replaced repeatedly, reducing the effectiveness of the cache [Dre07].

To address this issue, both techniques can be combined to create an *n-way set-associative cache*. In this setup, cache lines are grouped into sets of size n, where the associativity n is a relatively small number like four as shown in fig. 2.5. Most modern CPU caches typically have

an associativity n of 2, 4, 8 or 16. An associativity of 1 corresponds to the direct-mapped cache. In the set-associative cache, each memory address can be stored in a single set of n cache lines. The n lanes of cache lines are also referred to as *ways* [Dre07].

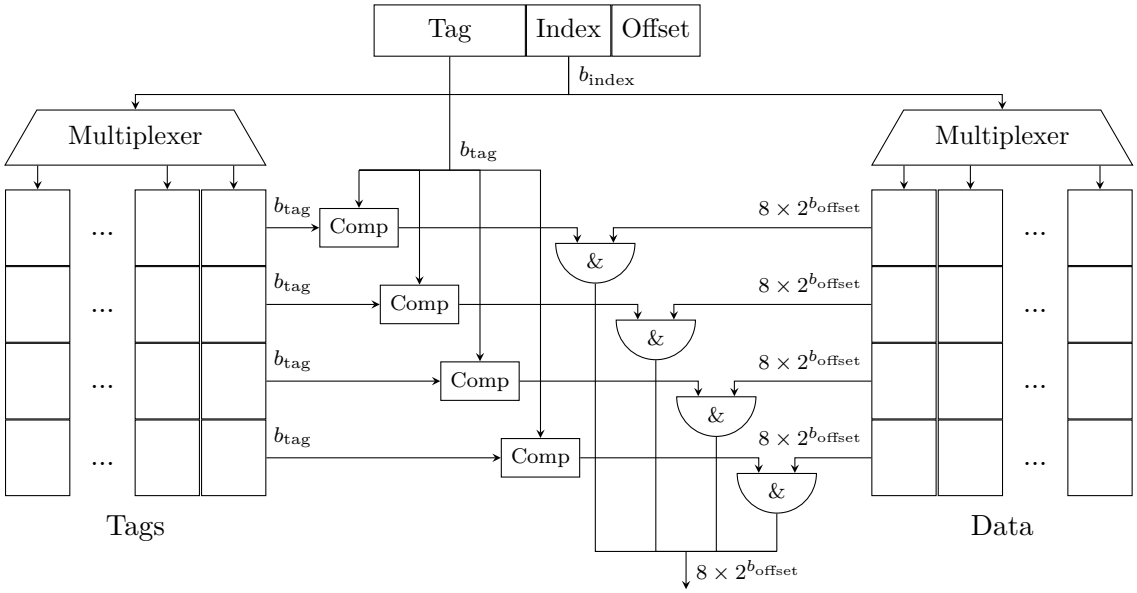

Figure 2.5.: Schematic of reading from a 4-way set-associative cache [Dre07].

This setup operates in two steps. Initially, a multiplexer is used, analogous to that employed in a direct-mapped cache, to select the set for the requested address. Then, in the second step, n comparators are utilized to determine the cache line in which the data is stored.

2.3.3. Replacement

The preceding section addresses the process of reading from caches in the context of data that has already been cached. When data that has not been previously cached is loaded into the cache and the cache's storage capacity is reached, it must remove a cache line to create space for the new data. This process of removal is also referred to as *eviction* [Han98d].

The cache lines that are permitted to be evicted are contingent upon the associativity of the cache. Naturally, in the case of a fully associative cache, any cache line may be evicted. In contrast, for a set-associative cache, it is all cache lines in the same set that may be evicted. The direct-mapped cache is the simplest case, as there is only one possible cache line to evict.

When multiple cache lines are permitted to be evicted, the cache adheres to the specified *replacement policy*. The replacement policy is designed in such a way that the number of cache misses is minimized for the targeted workload. There are numerous strategies for replacements – a random cache line could be selected, for example. The majority of strategies necessitate the incorporation of supplementary metadata bits within each cache line or set for the purpose of bookkeeping [Han98d]. A comprehensive examination of replacement policies can be found in section 2.4.

2.3.4. Writing

In addition to the removal or overwriting of a cache line, the eviction of data from a cache may also entail the writing of any changes back to the next layer of memory, with the exception of the L1 instruction cache. The precise process is determined by the write policy [Han98d; Fox24a].

In a *write-through* cache, any write to the cache is immediately written to memory. This approach simplifies the system design, as it ensures that the memory always has an up-to-date copy of the cache line. This is particularly the case in the context of shared-memory multiprocessor systems, where it is necessary for writes initiated by one processor to be observable by another processor eventually. However, in such systems a write-through policy would also generate high memory bus traffic as every single write of every processor would be announced [Han98d; Fox24a].

The solution is a *write-back* policy in which the update to a cache line is only copied back during eviction. The implementation is more complex, requiring particular attention to be paid to the issues of memory coherence which is concerned about keeping multiple caches and main memory coherent. The typical cache employs a write-back policy [Han98d; Fox24a].

2.3.5. Placement

Direct-mapped and set-associative caches require a methodology for mapping a cache line to a corresponding cache set deterministically. Figures 2.4 and 2.5 assume the non-offset bits of the requested address to be divided up into the upper bits for tagging and the lower bits for indexing. This is the textbook example which presupposes an index function based on modular arithmetic as the *placement policy*. An index function maps an input address to the index used for set selection, i.e., $[0, 2^{64}) \mapsto [0, 2^{b_{\text{index}}})$ on a 64-bit machine. The index function as used in figs. 2.3 to 2.5 is of the form $I(a) = \left\lfloor \frac{a}{2^{b_{\text{offset}}}} \right\rfloor \bmod 2^{b_{\text{index}}}$ or, expressed in code, `(a >> offset_bits) & ((1 << index_bits) - 1)`.

This work is concerned with reverse-engineering more complex index functions such as the one used in the L2 cache on the Fujitsu A64FX processor which is publicly documented [Fuj22]. Figure 2.6 shows the index function.

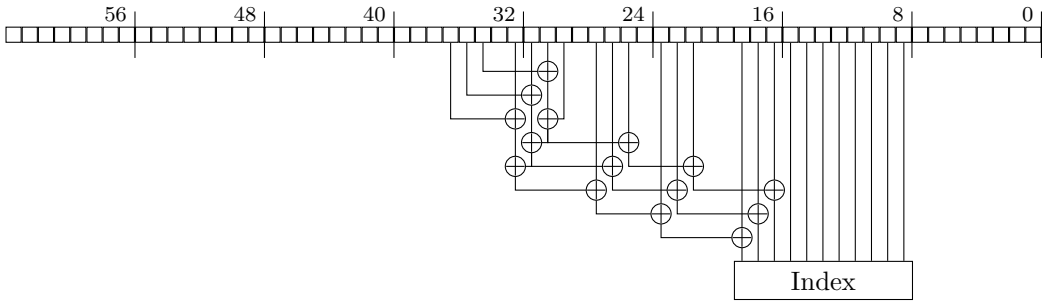

Figure 2.6.: Schematic of the Fujitsu A64FX L2 cache index function.

The index function uses the lower eleven non-offset bits of the physical address as the index. However, unlike the textbook example, the upper three bits of the index are XORed in triplets with bits 36 to 34, 32 to 30, 31 to 29, 27 to 25, as well as bits 23 to 21 of the address.

2.3.6. Virtual and physical indexing

Caches can operate on either virtual or physical memory and are designated as *virtually indexed* (VI) or *physically indexed* (PI), respectively. The advantage of virtual indexing is that no translation of the address is required, which is costly in the event of a TLB miss. Conversely, the use of virtual addresses for cache coherence introduces an additional overhead, as multiple virtual addresses may refer to the same physical address. This phenomenon is also referred to as *aliasing*. Moreover, when context switching between processes, a virtually indexed cache must be flushed, as the virtual address space of the two processes may overlap and map to different memory locations. Finally, it is difficult to share a virtually indexed cache between multiple cores, as it is unlikely that they will all execute the same process with the same virtual memory simultaneously [Bot04].

Some of the issues associated with virtual indexing can be mitigated by tagging the cache line using the physical address in lieu of the virtual address. Such *physically tagged* (PT) cache is also referred to as a VIPT cache. In a VIPT cache, the lookup of the cache line and the address translation, i.e. TLB lookup, can commence simultaneously. Once both results become available, the tag can be verified [Bot04].

Private L1 caches are typically implemented using VIPT, whereas shared higher-level caches use PIPT [Lip+20]. In a processor employing SMT, the L1 caches are shared between multiple SMT threads. In such instances, a thread identifier is typically incorporated into the tag, leading to threads competing on the cache.

2.3.7. Inclusiveness

In cache hierarchies, the *inclusion policy* controls whether data available in a lower-level cache is retained in the higher-level cache. A cache can be *inclusive*, *exclusive* [Dre07; HP11] or *non-inclusive* [HP11] of its lower-level cache.

An inclusive cache contains any data which is also present in the lower-level cache. Conversely, an exclusive cache is never populated with data from the lower-level cache [Dre07; HP11]. In contrast to those two strict policies is the non-inclusive policy. A non-inclusive cache may or may not contain data from the lower-level cache [HP11].

It is evident that an inclusive cache results in the inefficient use of chip area due to the presence of redundant copies of the data. Although Intel has previously employed inclusive caches, many contemporary processors utilize non-inclusive caches. AMD has always used non-inclusive caches [Dre07].

2.4. Replacement policies

There are various approaches for replacement policies that describe which cache line of a set to evict to make new space. This cache line is called the *victim*. The choice of the policy is a tradeoff of different factors, including the anticipated workload and the associated management overhead. In this section we present a variety of options for policies, beginning with fundamental ones and progressing to more sophisticated ones. While there are an infinite number of potential replacement policies, we focus only on the most well-known and documented ones.

2.4.1. Random and Round-robin

The *Random replacement policy*, as the name implies, selects a random cache line for eviction. That is, an integer in the range of the number of ways is drawn from a pseudo-random number generator (PRNG). Since the *Random replacement policy* does not store any historic data in a set, it requires no extra bits in the cache. The exception is the shared PRNG which usually requires some bits to store the state.

The *Round-robin replacement policy*, or *Cyclic replacement policy*, is more predictable. It keeps a *victim counter* per set which points to the next cache line to evict and is incremented afterwards. It cycles back to 0 when it reaches the maximum number of ways. This policy requires $\lceil log_2(n) \rceil$ number of extra bits per set to store the victim counter where n is the number of ways, or associativity.

The round-robin policy can also be viewed in a *First-in-First-out (FIFO)* fashion. In that view, each cache set is treated as a queue. The victim cache line is picked from the tail of the queue and re-added to the head when filled with new data. Note that only misses change the queue. On a hit, the queue remains unchanged. Given the queue of cache lines $[a,\ b,\ c,\ d]$, line d is selected as victim in the case of a miss. The new queue becomes $[d,\ a,\ b,\ c]$ [Rei+07].

Both policies make no use of prior cache accesses and, thus, do not take full advantage of temporal locality. Use cases include situations where the implementation needs to be simple or easy to simulate. Both policies can be found in the cache of a ARM Cortex-R Series processor which are designed for real-time applications [Arm14].

An extension to the random replacement is the *Not Last Used (NLU)* replacement policy which excludes the last accessed cache line from the selection process. The idea behind NLU is that the hit rate can be increased by avoiding the inherent possibility of evicting the last used line. The NLU implementation requires $\lceil \log_2(n) \rceil$ bits per set to store the last accessed way [Han98b]. It is also alternatively referred to as *Not Most Recently Used (NMRU)* [ZMN08].

2.4.2. Least Frequently Used (LFU)

In the *Least Frequently Used (LFU)* replacement policy, a frequency counter on each cache line keeps track of the number of accesses. The cache line with the lowest counter value (i.e. fewest accesses) is chosen as the victim to evict. Special care needs to be taken to prevent stale cache lines from polluting the cache. Hence, an aging policy is typically employed as well [ZMN08].

Besides the significant space requirements to store the counters for each cache line, the implementation also requires a way to compare the counters. This can either happen iteratively or through a tree of comparators, both solutions adding to the latency of the cache.

2.4.3. Least Recently Used (LRU)

The *Least Recently Used (LRU)* replacement policy uses a queue that is updated on every access of a cache line. This is in contrast to FIFO (cf. section 2.4.1) where only a cache miss updates the queue. On any cache hit, the cache controller looks up the cache line in the queue and moves it to the head. The head is also referred to as the MRU position. On a cache miss, the line at the tail position, or LRU position, is chosen as the victim to evict [Han98b].

LRU is expensive to implement in hardware since every possible state of the queue needs to be encoded resulting in a space requirement of $\lceil \log_2(n!) \rceil$ bits per cache set making for super-linear space complexity $\mathcal{O}(n \log n)$.

Additional to the space requirements, there is significant overhead involved in updating the state. Firstly, the position of the hit cache line needs to be decoded from the queue state. In a software-implemented cache this would typically be solved either by iterating over the queue or using a hash map of pointers to queue node for constant lookup. The compact state encoding in the hardware cache requires a complex decoding unit. Secondly, the new compact state needs to be encoded and written back. Crucially, the write back has to happen in the same processor cycle in order to serve the next request [Han98b].

2.4.4. Pseudo-LRU (PLRU)

Intel developed a replacement policy called *Pseudo-LRU (PLRU)* which approximates LRU at the advantage of a simplified implementation. This policy maintains a binary tree per set where each node keeps a bit that points to the half that was accessed more recently. A cache hit updates all bits along the path to point to the MRU position. On a cache miss, the cache line in the LRU position is evicted. The LRU position is obtained by going the opposite ways of the pointers. An exemplary PLRU binary tree is depicted in fig. 2.7. Solid lines mark the half that was accessed more recently.

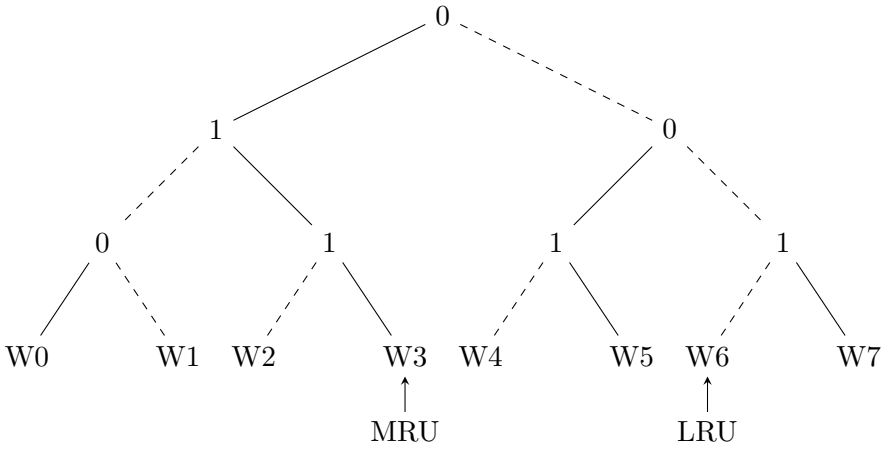

Figure 2.7.: Exemplary PLRU tree for an 8-way set-associative cache.

2.4.5. Not Recently Used (NRU) and Most Recently Used (MRU)

The *Not Recently Used (NRU)* replacement policy is another variant of approximating LRU behavior. Alternative names are *bit-PLRU* [PJ15] or *PLRUm* [AMM04], with the traditional *PLRU* being called *tree-PLRU*.

This policy is implemented using a NRU-bit per cache line. The NRU-bit denotes whether the cache line was recently accessed (0) or not (1). When a cache line is filled, the corresponding NRU-bit is set to 0. The victim cache line to evict is selected as the first line (e.g. left-to-right) in the set with the NRU-bit set to 1. In the case that all NRU-bits are 0, all bits except for the first one are set back to 1 and the first line is evicted [Jal+10].

A variation of this policy, the *Most Recently Used (MRU)* replacement policy, differs only in the implementation of the last step. Instead of resetting the NRU-bits back to 1 at victim selection, they are already updated after the last NRU-bit is zeroed [AR20; Rei+07]. Note that there is no established consensus on the policy naming. For instance, in [GD11] MRU replacement actually means choosing the most recently accessed cache line as a victim. This denotation is congruent with the naming of the other policy names.

2.4.6. Hybrid policies

While LFU is preferred for workloads with frequent accesses, it fails for workloads where recency (e.g., LRU) is the appropriate heuristic. Conversely, for some workloads a replacement policy based on recency is not appropriate. For example, a memory-intensive application that regularly scans over a working set larger than the cache size will result in thrashing under the LRU replacement policy [Han98a]. This is because while a new cache line is initially inserted at the MRU position, it traverses to the LRU position without ever receiving a cache hit. The continuous traversal cycle results in inefficient use of cache space [Qur+07].

There are replacement algorithms that take both, recency and frequency, into account such as *LRFU* [Don+01], *LRU-K* [OOW93], or *FBR* [RD90]. However, they are not designed for CPU caches as several parameters need to be tuned on a per-workload basis to make use of them. As noted by Jaleel et al., several adaptive replacement algorithms that self-tune these parameters exist. Nevertheless, algorithms like *ARC* [MM03] or *CAR* [BM04] significantly increase the hardware overhead and complexity [Jal+10].

One solution is a hybrid approach where the cache controller can choose between multiple replacement policies at runtime, depending on the detected workload. Intel introduced a mechanism called *Set Dueling* for selecting between two competing policies. In Set Dueling, a few sets of the cache are dedicated to each policy. The policy that performs better on the *dedicated sets* is inherited by all other sets, termed the *follower sets*. It has been shown that 32 to 64 dedicated sets are sufficient for Set Dueling to choose the better policy [Qur+07].

The concrete implementation discussed by Intel separates the LRU replacement policy into an *insertion policy* and an *eviction policy*. In the traditional LRU policy (cf. section 2.4.3), the eviction policy selects the cache line at the LRU position as the victim. The traditional insertion policy inserts new cache lines at the MRU position. Intel introduced further insertion policies such as the *LRU Insertion Policy (LIP)* which puts newly filled cache lines into the LRU position. The enhanced *Bimodal Insertion Policy (BIP)* is randomized and puts most cache lines into the LRU position. Set Dueling is employed to build a *Dynamic Insertion Policy (DIP)* that switches between BIP and the traditional insertion policy [Qur+07].

When Set Dueling detects that, for instance, the traditional LRU policy performs poorly on the current working set, it switches to LIP (or BIP). Given the same aforementioned situation where the working set is larger than the cache size, since LIP puts all new cache lines into the LRU position, this traversal cycle is broken and cache lines are kept in the cache until they are eventually accessed again and result in a hit. Note that the insertion policy is adopted by the whole follower set. Hence, for applications which mix incompatible working sets, with DIP it is not possible to dedicate different policies to different ranges of cache sets [Jal+10].

2.4.7. Re-Reference Interval Prediction (RRIP)

Another class of replacement policies introduced by Intel are based on *Re-Reference Interval Prediction (RRIP)*. The idea of RRIP policies is that, instead of ordering cache lines by their recency of access as in the case of LRU, they are ordered by their predicted re-reference. The head of the queue points to the cache line that is predicted to have *near-immediate* re-reference, i.e., it is expected to be access again soon. On the other side, the line at the tail of the queue is predicated for *distant* re-reference. The victim is always selected as the cache line at the tail. In the RRIP point of view, LRU is just another policy that uses recency as the predictor for re-reference [Jal+10].

Instead of establishing a strict total-order on all cache lines in a set, the *M-bit Static RRIP (SRRIP)* uses 2^M buckets of predictions. An M-bit register per cache line is used to store the index of the bucket, termed the *Re-reference Prediction Value (RRPV)* where 0 denotes a *near-immediate* re-reference prediction and $2^M - 1$ a *distant* re-reference prediction. Initially, a cache line receives a RRPV that represents a *long* re-reference interval $2^M - 2$ [Jal+10].

How the RRPV is lowered is governed by the *RRIP hit promotion policy*. The *Hit Priority (HP)* policy approximates LRU behavior by predicting that a cache line which receives another hit will be re-referenced in *near-immediate* future. Thus, any further hit updates the RRPV from $2^M - 2$ to 0. Alternatively, the *Frequency Priority (FP)* policy takes frequency of re-references into account. It decrements the RRPV by one on every hit but not less than 0 [Jal+10].

In victim selection, the first cache line (e.g., left-to-right) with an RRPV of $2^M - 1$ is chosen. If no such line exists, all RRPV registers are incremented by one and the process is repeated. Note that 1-bit SRRIP is equivalent to the NRU replacement policy (cf. section 2.4.5) [Jal+10].

Analogous to the BIP (cf. section 2.4.6), there is also the *Bimodal RRIP (BRRIP)* which uses an RRPV of $2^M - 1$ at high probability and only occasionally assigns the RRPV of $2^M - 2$. Similarly, the *Dynamic RRIP (DRRIP)* uses Set Dueling to switch between SRRIP and BRRIP [Jal+10].

Intel refers to the family of 2-bit RRIP policies as *Quad-Age LRU (QLRU)* [Jah+12; AR20] since the 2-bit RRPV per cache line is supposed to represent four levels of age. Modern Intel processors starting at Skylake employ variations of QLRU in the higher-level CPU caches while low-level caches use PLRU [Abe20]. Presumably, this design was chosen because low-level caches have a filtering effect on data accesses handled by the higher-level cache, i.e., L2 accesses are L1 misses and L1 hits do not touch L2. Hence, PLRU is not considered a proper policy for handling the L2 access pattern [KHM01; Jal+10].

3. Literature review

In this chapter we present prior literature on the topic of reverse-engineering of CPU cache policies. We are particularly interested in the placement and replacement policies.

As noted in chapter 1, we are not aware of prior research on the reverse-engineering of placement policies. However, there has been research on other hash functions found in modern multiprocessors, termed microarchitectural hash functions, which we present in section 3.1. The presented research includes LLC slice functions, which are related to placement policies.

In section 3.2 we cover prior work on reverse-engineering of replacement policies.

3.1. Microarchitectural hash functions

Many components of modern CPUs, such as sliced L3 caches, TLBs and DRAMs, use hash functions. The advent of sliced L3 CPU caches in the Intel Sandy Bridge architecture sparked interest in reverse-engineering the undocumented hash function which maps physical addresses to cache slices.

Hund, Willems, and Holz [HWH13] were one of the earliest to reverse-engineer this hash function to allow for side-channel attacks against KASLR using the x86 `rdtsc` instruction. Their approach was to allocate a large chunk of physically contiguous memory, and then finding pairs of set-aligned addresses differing in one bit to observe whether they collide on the same slice through timed probing. Through the obtained information they were able to manually recover the hash function. Their results were later verified and confirmed by Seaborn [Sea15].

Maurice et al. [Mau+15] extended on this work in that they used performance counters instead of the `rdtsc` instruction. They also automated the recovery process in that they used the pairs of addresses to find how flipping a bit changes the slice. The large chunk of physically contiguous memory was allocated using huge pages. A similar approach was taken by McCalpin [McC21]. This approach was further adopted for the reverse-engineering of other microarchitectural hash functions, including work on DRAM addressing [Pes+16] and TLB attacks [Gra+18; Kos+20].

Irazoqui, Eisenbarth, and Sunar [IES15] proposed to model the hash function as a concatenation of linear boolean functions. They recovered the hash function by solving each of the boolean functions using a system of linear equations. Lipp et al. [Lip+20] adopted this approach.

Previously cited work is constrained to hash functions with 2^n possible outputs. Yarom et al. [Liu+15; Yar+15] studied the hash functions of L3 caches featuring 6 or 10 slices. They used the captured mapping of addresses to cache slices in order to reverse-engineer the hash function manually.

In their study of microarchitectural hash functions, Gerlach et al. [Ger+24] developed an automated method for reverse-engineering a wide range of hash functions, particular those exhibiting non-linear characteristics as identified in [Yar+15]. However, their approach is

constrained by the inherent limitations of the proposed framework, which "is rather slow, requiring days in order to recover the targeted function" [Ger+24].

To the best of our knowledge, we are the first to propose a solution for reverse-engineering the placement policy. Our approach relies on modeling the index function as an affine transformation that can be recovered by collecting mappings of addresses to set indices.

3.2. Replacement policy reverse-engineering

Few authors have approached the problem of learning CPU cache replacement policies with most prior work being authored by Abel and Reineke.

John and Baumgartl [JB07] studied the LRU replacement policy and its derivatives. They devised an approach of identifying the underlying replacement policy by observing which way is evicted given a manually prepared sequence of loads. The evicted way is determined by counting cache misses when accessing a prepared set of addresses a second time after they have been accessed initially on an invalidated cache.

Abel [Abe12] found a flaw in their approach and introduced the class of *permutation policies*. Permutation policies are replacement policies which maintain an order of the ways, i.e., from the MRU- to the LRU-position. On a hit on a particular way, a permutation vector for that way describes how the order is adjusted. Likewise, there is a permutation vector corresponding to a miss. Abel reverse-engineered the class of permutation policies by first establishing a known initial state, then triggering a particular permutation on the replacement policy and lastly reading out the transitioned-to state. This approach has been reused in [AR12; AR13; AR14; Abe20; AR20].

Rueda Cebollero [Rue13] noted that some replacement policies cannot be modelled as permutation policies such as MRU. This restriction also includes SRRIP. He suggested to use automata learning as implemented in the LearnLib library[1] in order to obtain the replacement policy as a mealy machine. However, while the solution worked in theory, it took more than 72 hours of runtime to learn the replacement policy of a cache with more than six ways. Also, the author failed to run the implementation against a real system.

Vila et al. [Vil+20] adopted Rueda Cebollero's approach to build Polca and CacheQuery which establish a link between the LearnLib library and the actual system. They further implemented a function to synthesize a replacement policy automaton into readable code. While their approach reverse-engineers policies such as FIFO, LRU and LIP in a few seconds, it takes 4 hours to learn the MRU and 34 hours to learn the PLRU policy on a software-simulated cache with an associativity of 12 and 16, respectively. The authors mention an overhead of 1500x when applied on a real system.

Abel [Abe20] later introduced a second approach to [Abe12] based on a pre-defined set of replacement policies. In this approach, sequences of loads are generated and each sequence is executed on the system under test as well as all replacement policy candidates. A hit counter keeps track of the number of hits that occurred during the execution. Candidates with a hit counter different from the one obtained from the system are discarded until one (or no) candidate remains.

Given the significant runtime of the solutions based on automata learning is impracticable for our purposes, we base our approach of reverse-engineering replacement policies on the work of Abel [Abe20].

[1] `https://learnlib.de/`

4. Methods

The objective of this research is to develop an automated system for reverse-engineering CPU cache policies. To achieve this, we first construct a reverse-engineering framework tailored to CPU caches, which we describe in this chapter. Based on this framework, we then develop algorithms for obtaining cache policies, specifically the placement and replacement policy. The subsequent chapter is devoted to the software implementation of this framework.

4.1. Mathematical model of cached memory

We present a mathematical model and notation for cached memory to provide a formal description of our approach. A reference of all mathematical symbols used in this document is provided in appendix B.

4.1.1. Abstract memory

The fundamental model underlying our approach is termed abstract memory. This term refers to regions, or chunks, of allocated memory that are backed by a set-associative cache (hierarchy) on which memory accesses can be performed. It is assumed that accesses are serialized and that a state cannot decay into another, i.e., there are no side-effects.

We define abstract memory M as the quintuple $\langle A,\ L,\ S,\ s_0,\ \delta \rangle$, where:

- A is the set of accessible addresses (a subset of the available address space) where no two distinct addresses join a cache line,
- L is a strictly totally ordered set of symbols representing each cache level, including one for the main memory, where the minimum element $min(L)$ represents the level closest to the CPU (i.e. L1),
- S is the set of possible states of M, including all caches,
- s_0 is the initial state, and
- $\delta : S \times A \to S \times L$ is the accessor function that, given an address, transitions M into the next state and yields the symbol representing the cache layer that served the access.

The model of abstract memory is intentionally devoid of further constraints that have no impact on the functioning of the subsequent algorithms. To give a practical example, contiguous abstract memory can be described by $A = \left\{a \in \mathbb{N} \mid a_b \leq a < (a_b + n) \wedge a \mod 2^{b_{\text{offset}}} = 0\right\}$, where a_b represents the base address and n denotes the number of bytes. Assuming three levels of CPU caches, the set of cache level symbols can be represented by $L = \{1, 2, 3, 4\}$, where 1 represents the L1 cache and 4 the main memory.

Assuming an initial state s_0 in which all CPU caches are empty (or invalid), when one accesses the same address twice consecutively, the L1 cache is expected to serve the second request. In other words, $\forall a \in A : \delta(\delta(s_0, a)_1, a)_2 = min(L)$.[1]

The above definition is not concerned with virtual and physical address spaces. In practical terms, any memory allocation routine in user space and kernel space operates on virtual memory. Consequently, A is typically a subset of the virtual address space.

Given a mapping of physical to virtual addresses $m : A_p \to A_v$, new abstract memory M_p can be constructed by encapsulating the virtual memory's accessor function δ_v in the new function $\delta_p(s, a_p) = \delta_v(s, m(a_p))$. This approach allows any algorithm to operate with both, the virtual and physical address spaces, by substituting the accessor function. This concept is essential for reverse engineering physically indexed caches. Further details are discussed in section 5.3.2.

4.1.2. Replacement Policy

In addition, our model incorporates the concept of replacement policies. A replacement policy Π is managed per cache set, where a cache set consists of α ways.

We define replacement policy Π as the quintuple $\langle W, R, r_0, \delta_H, \delta_M \rangle$, where:

- W is a set where each symbol represents a way in the corresponding cache set, e.g. $\{0, 1, \ldots, 7\}$ for an associativity α of 8,
- R is the set of possible states of the replacement policy,
- r_0 is the initial state of the replacement policy,
- $\delta_H : R \times W \to R$ is the hit function which is invoked whenever the cache line at $w \in W$ receives a hit, and
- $\delta_M : R \to R \times W$ is the miss function which is invoked whenever space for a new cache line must be made in the cache set. The victim cache line that should be replaced is given by $w \in W$.

We further introduce the property *cyclically evicting*. A replacement policy Π is called *cyclically evicting for a state* $r \in R$ if and only if repeated application of the miss function δ_M forms a cyclic group with respect to the yielded way symbol. In other words, let $\delta_M^1(r) = \delta_M(r)$ and $\delta_M^i(r) = \delta_M(\delta_M^{i-1}(r)_1)$ in order to denote the i^{th} application of δ_M on state r. Then the first α misses should touch every way once, i.e., $\{\delta_M^1(r), \delta_M^2(r), \ldots, \delta_M^\alpha(r)\} = W$, and the pattern repeats cyclically, i.e., $\forall i, t \in \mathbb{N}\backslash\{0\} : \delta_M^i(r) = \delta_M^{i+\alpha t}(r)$.

We denote a replacement policy that is cyclically evicting for r_0 as *initially cyclically evicting* and one that is cyclically evicting $\forall r \in R$ as just *cyclically evicting*. Table 4.1 gives an overview on this property on common replacement policies.

Policies based on set dueling (cf. section 2.4.6) can only be modeled per cache set using this approach. Consequently, the interoperation between dedicated sets and follower sets would require modeling at a higher level.

[1] Note that indices 1 and 2 refer to the first and second element of the tuple.

Policy	**Initially cyclically evicting** (r_0)	**Cyclically evicting** ($\forall r \in R$)
FIFO	Yes	Yes
LRU	Yes	Yes
PLRU	Yes	Yes
2-bit SRRIP-HP	Yes	No

Table 4.1.: Overview of cyclically evicting property on common replacement policies.

4.1.3. Caching behavior

While the devised algorithms operate directly on our model of abstract memory, the concept of replacement policies is hidden by the cache implementation. The caching behavior bridges these two concepts.

Without loss of generality, we consider only the case of a single cache. We consider multiple levels of caches in section 4.5. We also assume that the cache is set-associative, which is consistent with all major processor architectures. Note that fully associative and direct-mapped caches are special cases of set-associative caches. Therefore, they are also covered by this assumption.

We denote the associativity by α ($= |W|$) and the number of sets by β. A fully associative cache is implied by $\beta = 1$ and a direct-mapped cache by $\alpha = 1$. An index function $I : A \mapsto [0, \beta)$ maps an address to the corresponding set in the cache. The set-associative cache maintains a replacement policy per set, denoted as $\Pi_0, \Pi_1, \ldots, \Pi_{\beta-1}$ and initialized as $r_{(\Pi_0)} \leftarrow r_{0(\Pi_0)}, r_{(\Pi_1)} \leftarrow r_{0(\Pi_1)}, \ldots, r_{(\Pi_{\beta-1})} \leftarrow r_{0(\Pi_{\beta-1})}$ where the set of way symbols is given by $W = \{0, 1, \ldots, \alpha - 1\}$. Typically, the cache is expected to use the same replacement policy for all sets, except in the case of set dueling, where the replacement policy used in a follower set can change at runtime.

The cache contents are given by a $\beta \times \alpha$ 0-indexed matrix Γ of cache lines. For the purposes of this work, we have no interest in the data that is actually in the cache. For this reason, only the cache line tag is considered. A value of $\square$ denotes an empty, or invalid, cache line, in contrast to a tag value given by the tag function $\tau : A \mapsto T$. The cache is initially empty:

$$\Gamma = \overbrace{\begin{pmatrix} \square & \square & \cdots & \square \\ \square & \square & \cdots & \square \\ \vdots & \vdots & \ddots & \vdots \\ \square & \square & \cdots & \square \end{pmatrix}}^{\alpha \text{ ways}} \Bigg\} \beta \text{ sets}$$

When abstract memory is accessed with the accessor function δ, the cache first determines the set corresponding to the address as $i = I(a)$. Then it checks whether there is a cache line for this address in set i, i.e., $\exists w \in W : \Gamma_{i,w} = \tau(a)$. The positive case constitutes a cache hit, hence $r_{(\Pi_i)} \leftarrow \delta_{H(\Pi_i)}(r_{(\Pi_i)}, w)$. In the case of a miss, the victim way is given by the replacement policy as $r_{(\Pi_i)}, w \leftarrow \delta_{M(\Pi_i)}(r_{(\Pi_i)})$. $\Gamma_{i,w}$ is evicted and replaced by $\tau(a)$.

4.2. Eviction Strategy

We introduce the concept of an *eviction strategy* which is used to evict all cache lines from a given cache set. By loading an address into the cache, evicting a cache set using this strategy and then retesting whether the address remains in the cache, it is possible to map addresses to cache sets. This mapping allows the index function $I(a)$ to be reversed-engineered.

We devise two eviction strategies:

- The *Eviction Set Strategy* is a platform-independent strategy using eviction sets, and
- The *CISW Eviction Strategy* is based on the `DC CISW` ARM instruction [Arm24b].

Figure 4.1 illustrates how both eviction strategies evict the second cache set and implicitly evict target address $t \in A$. A reaccess of t would reveal that t is no longer cached and, thus, must have indeed been mapped into the second cache set.

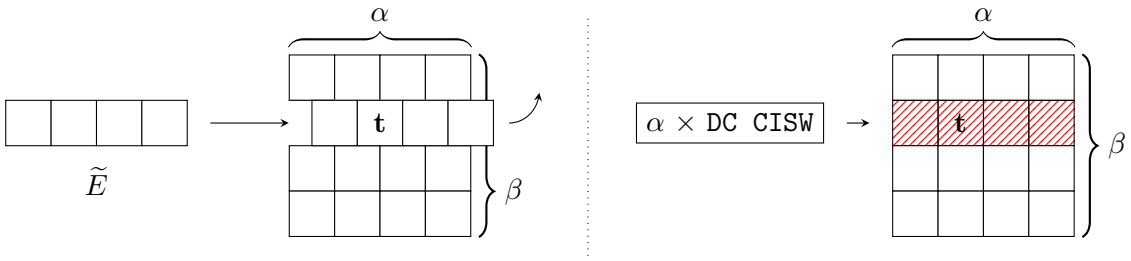

Figure 4.1.: Visual comparison of the eviction strategies based on eviction sets and the `DC CISW` ARM instruction.

4.2.1. Eviction Set Strategy

The eviction set strategy relies on the primitive of eviction sets. We call a set E of addresses an eviction set if and only if it contains a subset $\widetilde{E}$ of at least α addresses which map to the same cache set. We define the space of eviction sets as

$$\mathcal{E} = \left\{ E \in \wp(A) \mid \exists \widetilde{E} \subseteq E : \left|\widetilde{E}\right| \geq \alpha \wedge \left(\forall a, b \in \widetilde{E} : I(a) = I(b) \right) \right\}.$$

We call $\widetilde{E}$ a *congruent eviction set.*

We are interested in eviction sets specific to a particular cache set which we denote as E_i with i being the cache set index, or specific to a particular target address a which we denote as $E_{I(a)}$ with $a \in A$ and $a \notin E_{I(a)}$. We use the same notation for $\widetilde{E}$ and $\mathcal{E}$.

The term eviction set originates from research on side-channel attacks on CPU caches such as Flush+Reload and Prime+Probe [OST05]. Section 7.2.1 is devoted to these attacks. The intuition is that when all addresses in the eviction set are accessed, the contents ($\notin E$) in the cache set get evicted. Once one congruent eviction set is established per cache set, one can map any address to its corresponding cache set, allowing for inferring (or approximation) the index function $I(a)$.

We devise a methodology for testing, finding, reducing, minimizing and locating eviction sets.

Testing

Testing whether an arbitrary set of addresses forms an eviction set for some target address t involves checking if accessing the addresses causes t to get evicted. To check whether t gets evicted, t is first loaded into the cache by accessing it. After accessing the addresses from the eviction set, a second access of t is performed which should not be answered by the (first level) cache. We adopt this routine from Vila, Köpf, and Morales which is formalized in algorithm 1 and has a linear time complexity of $\mathcal{O}(\left|E_{I(t)}\right|)$ [VKM18].

Environment: Memory $M = \langle A, L, S, s_0, \delta \rangle$ and current state $s \in S$
Data: Eviction set candidate $E_{I(t)}$ and target address t
begin
  $s, \_ \longleftarrow \delta(s, t)$
  for $a \in E_{I(t)}$ **do**
    $s, \_ \longleftarrow \delta(s, a)$
  end
  $s, l \longleftarrow \delta(s, t)$
  return $l \neq min(L)$
end

Algorithm 1: IsEvictionSet (taken from [VKM18])

The following conditions must be met for the algorithm to work correctly generically:

1. The underlying replacement policy for the set i to which t is mapped must be cyclically evicting for state s,

2. The first access to t must constitute a cache miss, and

3. The first α accesses to addresses from $E_{I(t)}$ which map to set i must constitute a cache miss.

The rationale is as follows: Due to the cyclic eviction property, the initially accessed t will be emplaced in way w, and, after α misses, way w, i.e., address t, will eventually get evicted again. If condition 2 did not hold, there would be no guarantee that the replacement policy would be cyclically evicting for the successor state after the cache hit on t. Similarly, if condition 3 did not hold and any of the first α accesses to set i produced a cache hit, there would be no guarantee either.

Exceptions exist when applied to certain replacement policies. Notably, in the case of cyclically evicting ($\forall s \in S$) policies such as FIFO, LRU and PLRU condition 2 can be dropped and condition 3 can be relaxed to require α contiguous cache misses (not interrupted by cache hits) on set i.

In the case of LRU in particular, the conditions can be relaxed even further. In LRU, the first access to t, no matter whether hit or miss, moves t to the MRU position. The next $\alpha - 1$ hits or misses then move t into LRU position, and the α^{th} access will evict t.

It is also important to consider that the algorithm is not deterministic. Firstly, the addresses of $E_{I(t)}$ can be accessed in any order, which causes an access to a to hit the cache on some occasions and miss it on others due to accesses to predecessor addresses.

However, there is another effect that, when taken into account, allows the use of sets in place of sequences. It is conceivable that a call to the routine with a positive result may yield a negative result in a subsequent call due to alterations to the abstract memory state induced by the initial call. In such a scenario, the conditions that were applicable prior to the initial call may no longer be applicable prior to the subsequent call to `IsEvictionSet`, potentially leading to the erroneous classification of a set as not being an eviction set. This phenomenon occurs when the first call loads the addresses from the eviction set into the cache and in the second call condition 3, thus, no longer holds.

It is possible to accept false negatives in certain cases (cf. section 4.2.1), but in other situations (cf. section 4.2.1) a probabilistic result is not desirable. It is therefore essential to devise a solution that ensures the three aforementioned conditions are always met prior to the execution of the algorithm in question.

One potential avenue for ensuring that all accesses constitute a cache miss (conditions 2 and 3) is to introduce a form of noise access, or cache pollution, using a set of disjoint addresses, i.e., $A \backslash (E_{I(t)} \cup \{t\})$, in the anticipation that these evict any previously cached lines. An extension to the algorithm redoes the pollution if during any access a cache hit is detected.

An alternative solution relies on an extension to the model of abstract memory. This extension introduces a new invalidation function, δ_{inv}, which removes a cache line for a given address from the cache. In essence, the invalidation function serves as the inverse of the accessor function, in that $\forall s \in S, a \in A : \delta(\delta_{\text{inv}}(s, a), a)_1 \neq min(L)$. Initially, the algorithm invalidates all addresses that are to be accessed. Availability of the invalidation function depends on architectural support. For example, the `CLFLUSH` instruction in the x86 instruction set serves this purpose in that it flushes the specified address from all caches.

In the case of policies that are not cyclically evicting but initially cyclically evicting, it is necessary to bring the replacement policy into state s_0 in order to ensure that condition 1 holds. On x86, it is possible to bring the entire memory and implicitly the replacement policies into state s_0 using the `INVD/WBINVD` instruction. Both instructions invalidate all levels of caches, the difference being that latter writes-back any modified cached data before invalidating. This is essentially equivalent to $s \leftarrow s_0$.

Finding

The methodology for finding eviction sets is based on the generation of sets of addresses and subsequent testing to determine if they constitute an eviction set for the target address t.

One straightforward approach is to generate random sets of addresses or even consider the entire address space as an eviction set. However, our methodology relies on the comparatively expensive routine of minimizing eviction sets (cf. section 4.2.1). Therefore, when finding eviction sets, it is crucial to ensure that they are as small as possible. At the same time, the process of identifying eviction sets should be expeditious. It is possible to generate random sets of size α, but the probability of such a set being an eviction set would be $\beta^{-\alpha}$. However, the probably of a hit is significantly low even on the smallest CPU caches.

A practical approach is to start generating random sets at a certain size n and, following a series of unsuccessful attempts x, multiply the size by a fixed factor q in an exponential

backoff fashion. Algorithm 2 implements this approach.

```
Environment: Memory M = ⟨A, L, S, s_0, δ⟩ and current state s ∈ S
Data: target address t, starting size n, attempts x and factor q
begin
    while true do
        for i ⟵ 0; i < x; i ⟵ i + 1 do
            C ⟵ selectRandomSubset(A, n)
            if IsEvictionSet(C, t) then
                return C
            end
        end
        n ⟵ ⌈n × q⌉
    end
end
```

Algorithm 2: FindEvictionSet

The objective is to minimize both, the average eviction set size and the time complexity of the algorithm by identifying optimal parameters for n, x, and q. To that end, we begin by describing the probability that a set of addresses is an eviction set for some address t. We assume that addresses are distributed uniformly across cache sets, i.e., the probability of an address to map to a particular set is $p = 1^{-\beta}$. The random variable of X addresses mapping to the same set as t can be modelled as a binomial distribution $X \sim B(n, p)$ [VKM18].

The probability of finding k addresses mapping to the same set as t can be approximated as

$$P(X = k) \approx \frac{\lambda^k e^{-\lambda}}{k!} \text{ with } \lambda = np.$$

A valid eviction set for address t is formed if at least α addresses map to the same cache set:

$$\begin{aligned} P(X \geq \alpha) &= 1 - P(X < \alpha) \\ &\approx 1 - e^{-\lambda} \sum_{k=0}^{\alpha-1} \frac{\lambda^k}{k!} \end{aligned}$$

We define a cost function $C_{\alpha,\beta}(n, q)$ to find the probability-weighted cost of running the algorithm with the given parameters. To simplify the calculations, the number of attempts x is fixed as 1. We note that a factor adjusted for the number of attempts can be approximated as $q' = q^x$. We define the cost of testing an eviction set as n and the cost of minimizing an eviction set as n^2 which is the worst case scenario (cf. section 4.2.1).

$$\begin{aligned} C_{\alpha,\beta}(n, q) &= n + (P(\text{is eviction set}) \cdot n^2 + P(\text{is not eviction set}) \cdot C_{\alpha,\beta}(\lceil n \cdot q \rceil, q)) \\ &= n + ((1 - p) \cdot n^2 + p \cdot C_{\alpha,\beta}(\lceil n \cdot q \rceil, q)) \text{ with } p = e^{-\lambda} \sum_{k=0}^{\alpha-1} \frac{\lambda^k}{k!}, \lambda = \frac{n}{\beta} \end{aligned}$$

We can find the optimal parameters for a given cache configuration by solving for

$$(n^*, q^*) = \arg\min_{(n,q)} C_{\alpha,\beta}(n, q).$$

To account for multiple possible cache configurations $1, 2, \ldots, m$, the cost can be minimized for the average case by solving for

$$(n^*, q^*) = \arg\min_{(n,q)}\left(\frac{C_{\alpha_1,\beta_1}(n,q)}{\min\ C_{\alpha_1,\beta_1}(n,q)} + \frac{C_{\alpha_2,\beta_2}(n,q)}{\min\ C_{\alpha_2,\beta_2}(n,q)} + \ldots + \frac{C_{\alpha_m,\beta_m}(n,q)}{\min\ C_{\alpha_m,\beta_m}(n,q)}\right).$$

Reducing

The process of reducing an eviction set entails the removal of addresses from the set without rendering it invalid. There are two reasons for why an address can be removed:

1. The address maps to a different cache set than t. Therefore, removal has no impact.
2. Without the address there are still at least α addresses that map to the same cache set as t.

Two approaches are considered for reducing eviction sets. The first approach, as described by Liu et al., involves removing an address from the set and verifying whether it still constitutes an eviction set. In the negative case, the address is added back and marked as critical to the set. This process is repeated until the target size n is reached. An empty set is returned if the reduction failed. The time complexity is bounded by $\mathcal{O}(\left|E_{I(t)}\right|^2)$ [Liu+15].

The second approach, introduced by Vila, Köpf, and Morales, uses group testing to find a solution in linear time. In each iteration, the eviction set is split into $n+1$ equal-sized subsets, designated as $P_1, \ldots, P_{n+1}$. Since n is given as the target size, there must be at least one subset which can be subtracted from the eviction set without invalidating it: $\exists P : (E_{I(t)} \backslash P) \in \mathcal{E}_{I(t)}$. In the event that such a subset cannot be identified, the reduction process is deemed unsuccessful. Algorithm 3 implements this approach with a time complexity of $\mathcal{O}(\left|E_{I(t)}\right|)$ [VKM18].

Environment: Memory $M = \langle A, L, S, s_0, \delta \rangle$ and current state $s \in S$
Data: Eviction set $E_{I(t)}$, target address t and target size n
begin
  if $\left|E_{I(t)}\right| = n$ **then**
    return $E_{I(t)}$
  end
  $P_1, \ldots, P_{n+1} \longleftarrow split(E_{I(t)}, n+1)$
  forall $P_1, \ldots, P_{n+1}$ **do**
    if *IsEvictionSet*$(E_{I(t)} \backslash P_x, t)$ **then**
      return *ReduceEvictionSetFast*$(E_{I(t)} \backslash P_x, t, n)$
    end
  end
  return $\emptyset$
end

Algorithm 3: ReduceEvictionSetFast

Minimizing

We introduce the notion of eviction set minimization which is the reduction to the smallest possible size, i.e., the associativity α, without knowledge of that size. Hence, through finding

and minimizing an eviction set, the cache associativity can be reverse-engineered.

The quadratic reduction can be adjusted slightly to perform minimization. The time complexity remains to be $\mathcal{O}(\left|E_{I(t)}\right|^2)$.

The linear-time reduction algorithm 3 can be transformed into the minimization algorithm algorithm 4 by performing a binary search on the target size n. The binary search is constrained by a lower bound of 1 and an upper bound of $\alpha_{\max}$ which is an educated guess. Consequently, the overall time complexity is $\mathcal{O}(\left|E_{I(t)}\right| \log \alpha_{\max})$. The resulting minimal eviction set is, by definition, a congruent eviction set.

Environment: Memory $M = \langle A, L, S, s_0, \delta \rangle$ and current state $s \in S$
Data: Eviction set $E_{I(t)}$, target address t and upper bound $\alpha_{\max}$
begin
  $E_{I(t)} \longleftarrow$ `ReduceEvictionSetFast`$(E_{I(t)}, t, \alpha_{\max})$
  $l \longleftarrow 1$
  $r \longleftarrow \alpha_{\max}$
  while $l < r$ **do**
    $m \longleftarrow l + \lfloor (r - l)/2 \rfloor$
    $C \longleftarrow$ `ReduceEvictionSetFast`$(E_{I(t)}, t, m)$
    if $C = \emptyset$ **then**
      $l \longleftarrow m + 1$
    end
    else
      $r \longleftarrow m$
      $E_{I(t)} \longleftarrow C$
    end
  end
  return $E_{I(t)}$
end

Algorithm 4: MinimizeEvictionSetFast

Locating

Finally, we introduce another algorithm used for locating the eviction set for some address t from a collection of congruent eviction sets $\widetilde{E}_0, \widetilde{E}_1, \ldots, \widetilde{E}_{n-1}$ with $n \leq \beta$. This routine is employed for the purpose of mapping an address to its corresponding cache set, thereby enabling the index function to be learned.

The naïve implementation for locating the eviction set is by iterating through all sets and returning the one that matches. Assuming an eviction set size of α, the time complexity is bounded by $\mathcal{O}(\alpha\beta)$. We also introduce algorithm 5, based on group-testing, which has the same time complexity but is more robust since the resulting eviction set is tested more than once on average.

Strategy Initialization

The introduced algorithms are employed in the initial bootstrapping. Based on this bootstrapping, inferring the placement and replacement policies is possible, as described in subsequent

Environment: Memory $M = \langle A, L, S, s_0, \delta \rangle$ and current state $s \in S$
Data: Collection of congruent eviction sets $\widetilde{E}_0, \widetilde{E}_{[1]}, \ldots, \widetilde{E}_{n-1}$ and target address t
begin
  $l \longleftarrow 0$
  $r \longleftarrow n - 1$
  while $l \leq r$ **do**
    $m \longleftarrow l + \lfloor (r - l)/2 \rfloor$
    if *IsEvictionSet* $\left(\bigcup_{i=l}^{m} \widetilde{E}_i \right)$ **then**
      $r \longleftarrow m$
    end
    else
      $l \longleftarrow m + 1$
    end
  end
  return l
end

Algorithm 5: LocateEvictionSetGT

sections.

The initialization begins with the generation of a random address $t \in A$ which is used to find the first eviction set $E_{I(t)}$. This set is then minimized using one of the minimization algorithms. The minimal eviction set, stored as $\widetilde{E}_{[0]}$, has a size of α which implies the associativity of the cache. Associativity α is typically known, allowing one the use of a reduction algorithm instead, which is faster. The minimal eviction set $\widetilde{E}_0$ is then enriched by adding t.

Next, further random addresses are generated repeatedly and the corresponding eviction set is located. If an eviction set can be located, it is enriched by the random address. If such an eviction set cannot be located, a new eviction set is found for this address. Instead of minimizing, the eviction set is reduced using the determined α as the desired size, which has a better time complexity. The reduced eviction set is stored as $\widetilde{E}_{[i]}$.

The iteration stops when the process no longer fails to locate eviction sets which happens once an eviction set has been derived per physical cache set, i.e., when β congruent eviction sets are present. The termination can be based on a pre-supplied value for β, or probabilistic. A simple probabilistic method is to use an educated guess β_{guess} to calculate the number of acceptable successful eviction set localization x with 95% confidence by solving $(1 - \beta_{\text{guess}}^{-1})^x \leq 0.05$. It is also possible to dynamically estimate β_{guess} during the initialization on the basis of the preceding hit/miss rate.

After bootstrapping, the congruent eviction sets $E_0, E_1, \ldots, E_{\beta-1}$ are in an arbitrary order, i.e., E_0 may map to a physical cache set other than 0. In theory this would not matter since the reverse-engineered index function would differ from the actual index function as implemented in hardware only by some permutation vector. However, knowledge of the physical cache set would simplify reverse-engineering and result in a more compact index function.

We have not found a satisfactory solution to reordering the eviction sets in such a way that the ordering approximates the corresponding physical cache sets. For now, we order the

eviction sets by what we would expect the physical cache set to be given a textbook index function.

4.2.2. CISW Eviction Strategy

The ARM instruction set includes instructions for the maintenance of both, instruction and data caches. Of particular note are three instructions that enable the data cache maintenance on a specific cache set and way:

- `DC ISW`: Data cache invalidate by set/way
- `DC CSW`: Data cache clean by set/way
- `DC CISW`: Data cache clean and invalidate by set/way

All three instructions take a 32-bit operand which encodes cache set, way and level to be cleaned and/or invalidated [Arm24b]. A strategy based on the `DC CISW` ARM instruction is devised to evict a particular cache set by iterating over all cache ways for this set and calling the instruction.

4.3. Reverse-engineering the placement policy

By employing an initialized eviction strategy, it is possible to map addresses to cache sets and thereby ascertain the index function.

There are numerous ways to construct an index function from the mapping. We anticipate that most complex index functions are based solely on XOR- and NOT-gates. This includes the A64FX L2 index function (cf. fig. 2.6). Our expectation can be explained as follows:

Firstly, for a uniformly distributed input these two gates produce uniformly distributed outputs whereas AND- and OR-gates produce one output over the other in three of four cases. This could lead to unbalanced mappings to cache sets.

Secondly, the placement policy must ensure that no two distinct cache lines mapped to the same cache set have the same tag value. A proper tag value derivation is challenging for an index function employing AND- and OR-gates. The straightforward approach would be to use all non-offset bits as a tag value which occupies costly chip space.

Lastly, prior work on LLC slice functions (cf. section 3.1) has shown that Intel uses XOR-based hash functions. Since these are closely related to index functions we expect similar design choices.

In the event that automated recovery of the index function is unsuccessful, it would be possible to export the mappings as a document and require a human to identify a pattern.

4.3.1. Affine transformation modeling

An index function based exclusively on XOR- and NOT-gates can be described as an affine transformation $\vec{y} = A\vec{x} + \vec{t}$ with coefficients in galois field GF(2). GF(2) encompasses the ring of integers modulo 2, i.e., $\mathbb{B} = \mathbb{Z}/2\mathbb{Z} = \{0, 1\}$, with addition being the XORing and multiplication the ANDing of the two operands. Henceforth, we denote this field as $\mathbb{B}$.

The input address $\vec{x} \in \mathbb{B}^d$ (e.g. with $d = 64$, assuming 64-bit address space) and the cache set index $\vec{y} \in \mathbb{B}^{\log_2 \beta}$ are bit-encoded. Each row in the $(\log_2 \beta) \times d$ matrix A describes which

bits of the input address are XORed into the corresponding bit of the set index. The vertex $\vec{t} \in \mathbb{B}^{\log_2 \beta}$ contains 1s for the set index bits to negate.

Figure 4.2 shows the transformation matrix A corresponding to the index function of the L2 cache on the A64FX. The translation vector $\vec{t}$ is the null vector. Characteristically, the transformation matrix typically contains diagonals of ones. These correspond to contiguous bits in the input address which are XORed into the index bits. A textbook index function would contain just the right diagonal, as marked in the matrix, which corresponds to a simple modulo on the input address without the offset bits (cf. section 2.3.5). Note also how the lower eight bits of the address, i.e., the offset bits, do not influence the result. The A64FX modifies the textbook index function slightly in that it XORes a few higher triplets of bits of the physical address onto the three highest bits of the set index.

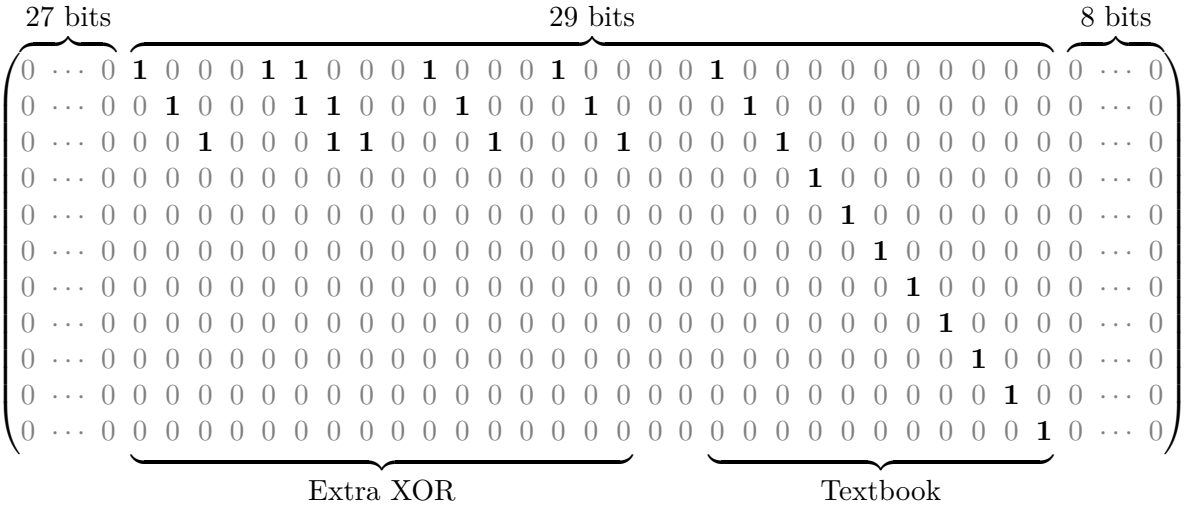

Figure 4.2.: Transformation matrix A that corresponds to the index function used in the L2 cache of the Fujitsu A64FX for mapping physical addresses to cache sets.

4.3.2. Affine transformation recovery

Given a mapping of input addresses $\vec{x}^{(1)}, \vec{x}^{(2)}, \ldots$ to cache set indices $\vec{y}^{(1)}, \vec{y}^{(2)}, \ldots$, the objective is to recover the transformation matrix A and the translation vector $\vec{t}$. We adopt the solution by Tymchyshyn and Khlevniuk [TK19] for finding an affine transformation that maps vertices of simplex $\vec{x}^{(1)}, \vec{x}^{(2)}, \ldots, \vec{x}^{(d+1)}$ into vertices $\vec{y}^{(1)}, \vec{y}^{(2)}, \ldots, \vec{y}^{(d+1)}$. The authors show that, given the aforementioned vertices and an input vertex $\vec{x}$, the mapped vertex $\vec{y}$ is

calculated as

$$\vec{y} = (-1)\frac{\det\begin{pmatrix} 0 & \vec{y}^{(1)} & \vec{y}^{(2)} & \cdots & \vec{y}^{(d+1)} \\ x_1 & x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(d+1)} \\ x_2 & x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(d+1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_d & x_d^{(1)} & x_d^{(2)} & \cdots & x_d^{(d+1)} \\ 1 & 1 & 1 & \cdots & 1 \end{pmatrix}}{\det\begin{pmatrix} x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(d+1)} \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(d+1)} \\ \vdots & \vdots & \ddots & \vdots \\ x_d^{(1)} & x_d^{(2)} & \cdots & x_d^{(d+1)} \\ 1 & 1 & \cdots & 1 \end{pmatrix}}.$$

Since division is undefined in $\mathbb{B}$, we rewrite the formula using the multiplicative inverse such that

$$\vec{y} = (-1)\det\begin{pmatrix} 0 & \vec{y}^{(1)} & \vec{y}^{(2)} & \cdots & \vec{y}^{(d+1)} \\ x_1 & x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(d+1)} \\ x_2 & x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(d+1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_d & x_d^{(1)} & x_d^{(2)} & \cdots & x_d^{(d+1)} \\ 1 & 1 & 1 & \cdots & 1 \end{pmatrix} \det^{-1}\begin{pmatrix} x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(d+1)} \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(d+1)} \\ \vdots & \vdots & \ddots & \vdots \\ x_d^{(1)} & x_d^{(2)} & \cdots & x_d^{(d+1)} \\ 1 & 1 & \cdots & 1 \end{pmatrix}.$$

We note that the multiplicative inverse is not defined in the case that the determinant is equal to 0. This would correspond to division by zero in the authors' formula. In the other case of a determinant of 1, the inverse is the element itself. Consequently, the last factor can be dropped if we can show that the determinant of the latter matrix is non-zero. Tymchyshyn and Khlevniuk show that this conditions holds given that the vertices $\vec{x}^{(1)}, \vec{x}^{(2)}, \ldots, \vec{x}^{(d+1)}$ indeed form a simplex, i.e., are affinely independent, as stated previously.

We further note that negation has no effect in $\mathbb{B}$ ($(-1) = 1$). Hence, we can also drop the factor at the front giving us the formula

$$\vec{y} = \det\underbrace{\begin{pmatrix} 0 & \vec{y}^{(1)} & \vec{y}^{(2)} & \cdots & \vec{y}^{(d+1)} \\ x_1 & x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(d+1)} \\ x_2 & x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(d+1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_d & x_d^{(1)} & x_d^{(2)} & \cdots & x_d^{(d+1)} \\ 1 & 1 & 1 & \cdots & 1 \end{pmatrix}}_{= D}.$$

We can rewrite the formula into form $\vec{y} = M\vec{x}$ by first performing Laplace expansion along the first column of the matrix. We use $\mathcal{S}_{i,j}(D)$ to denote the submatrix of D with row i and column j deleted. Note that we can neglect the Laplace cofactor (again, since negation has

no effect). Thus,

$$
\begin{aligned}
\vec{y} &= \left(\sum_{i=1}^{d} x_i \det \mathcal{S}_{i+1,1}(D)\right) + \det \mathcal{S}_{d+2,1}(D) \\
&= x_1 \det \mathcal{S}_{2,1}(D) + x_2 \det \mathcal{S}_{3,1}(D) + \cdots + x_d \det \mathcal{S}_{d+1,1}(D) + \det \mathcal{S}_{d+2,1}(D) \\
&= x_1 \det \begin{pmatrix} \vec{y}^{(1)} & \vec{y}^{(2)} & \cdots & \vec{y}^{(d+1)} \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(d+1)} \\ \vdots & \vdots & \ddots & \vdots \\ x_d^{(1)} & x_d^{(2)} & \cdots & x_d^{(d+1)} \\ 1 & 1 & \cdots & 1 \end{pmatrix} + x_2 \det \begin{pmatrix} \vec{y}^{(1)} & \vec{y}^{(2)} & \cdots & \vec{y}^{(d+1)} \\ x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(d+1)} \\ x_3^{(1)} & x_3^{(2)} & \cdots & x_3^{(d+1)} \\ \vdots & \vdots & \ddots & \vdots \\ x_d^{(1)} & x_d^{(2)} & \cdots & x_d^{(d+1)} \\ 1 & 1 & \cdots & 1 \end{pmatrix} \\
&\quad + \cdots + x_d \det \begin{pmatrix} \vec{y}^{(1)} & \vec{y}^{(2)} & \cdots & \vec{y}^{(d+1)} \\ x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(d+1)} \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(d+1)} \\ \vdots & \vdots & \ddots & \vdots \\ x_{d-1}^{(1)} & x_{d-1}^{(2)} & \cdots & x_{d-1}^{(d+1)} \\ 1 & 1 & \cdots & 1 \end{pmatrix} + \det \begin{pmatrix} \vec{y}^{(1)} & \vec{y}^{(2)} & \cdots & \vec{y}^{(d+1)} \\ x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(d+1)} \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(d+1)} \\ \vdots & \vdots & \ddots & \vdots \\ x_d^{(1)} & x_d^{(2)} & \cdots & x_d^{(d+1)} \end{pmatrix},
\end{aligned}
$$

or, written as a matrix-vector multiplication,

$$
\begin{aligned}
\vec{y} &= \underbrace{\left(\det\begin{pmatrix}\ddots\end{pmatrix}, \det\begin{pmatrix}\ddots\end{pmatrix}, \cdots, \det\begin{pmatrix}\ddots\end{pmatrix}, \det\begin{pmatrix}\ddots\end{pmatrix}\right)}_{= M} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \\ 1 \end{pmatrix} \\
&= \underbrace{(\det \mathcal{S}_{2,1}(D), \det \mathcal{S}_{3,1}(D), \cdots, \det \mathcal{S}_{d+1,1}(D), \det \mathcal{S}_{d+2,1}(D),)}_{= M} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \\ 1 \end{pmatrix}
\end{aligned}
$$

Similarly, we perform Laplace expansion along the first row for each of the submatrices.

$$
\begin{aligned}
\det \mathcal{S}_{i+1,1}(D) &= \sum_{j=1}^{d+1} \vec{y}^{(j)} \det \mathcal{S}_{1,j}(S_{i+1,1}(D)) \\
&= \sum_{j=1}^{d+1} \vec{y}^{(j)} \det \mathcal{S}_{i,j}(D') \qquad\qquad \text{with } D' = \mathcal{S}_{1,1}(D)
\end{aligned}
$$

We can, thus, calculate component $m_{i,j}$ of matrix M as

$$\begin{aligned} m_{i,j} &= (\det \mathcal{S}_{j+1,1}(D))_{(i)} \\ &= (\sum_{k=1}^{d+1} \vec{y}^{(k)} \det \mathcal{S}_{j,k}(D'))_{(i)} \qquad \text{with } D' = \mathcal{S}_{1,1}(D) \\ &= \sum_{k=1}^{d+1} y_i^{(k)} \det \mathcal{S}_{j,k}(D') \end{aligned}$$

This gives us the affine transformation in the form $\vec{y} = M(x_1, x_2, \ldots, x_d, 1)^T$. We can treat M as an augmented matrix $M = \left(A \,|\, \vec{t}\right)$ to obtain the affine transformation in canonical form $\vec{y} = A\vec{x} + \vec{t}$. An example implementation of the affine transformation recovery using Python and NumPy[2] is provided in appendix A.1.

4.3.3. Affine recovery optimizations

We have shown how A and t can be recovered using addresses $\vec{x}^{(1)}, \vec{x}^{(2)}, \ldots, \vec{x}^{(d+1)}$ and their corresponding set indices $\vec{y}^{(1)}, \vec{y}^{(2)}, \ldots, \vec{y}^{(d+1)}$. We emphasize that our approach relies on the expensive task of frequently calculating determinants of the D' matrix which scales quadratically with d. Hence, the objective is to keep d as low as possible.

This can be achieved by stripping the offset bits from all calculations since they are expected not to influence the set index. Hence, for the purpose of the recovery process, $\vec{x}$ only contains the non-offset bits of the address. After the transformation matrix is recovered, it is expanded horizontally to the right by zeroes.

4.3.4. Iterative affine recovery

For correct recovery, the (stripped) addresses $\vec{x}^{(1)}, \vec{x}^{(2)}, \ldots, \vec{x}^{(d+1)}$ must form a d-simplex, i.e., they must be affinely independent. A set of vectors $\left\{\vec{v}^{(1)}, \vec{v}^{(2)}, \ldots, \vec{v}^{(n)}\right\}$ is affinely independent if the set $\left\{\vec{v}^{(2)} - \vec{v}^{(1)}, \vec{v}^{(3)} - \vec{v}^{(1)} \ldots, \vec{v}^{(n)} - \vec{v}^{(1)}\right\}$ is linearly independent. In the following we call a set of vectors *affinely independent with respect to* if the set is affinely independent given a selection matrix is applied to each element.

In the optimal approach, affinely independent addresses would be generated systematically, e.g., by flipping every bit once. However, this approach would have to take the complexity of the set of accessible addresses A into account which is not guaranteed to be fully contiguous. We use a simpler approach based on sourcing random addresses.

We also note that, unlike 32-bit systems, 64-bit systems do not suffer from a small address space. On the contrary, it is challenging to have the accessible address space cover each address bit in both states at least once (which is required for affine independence). This is doubly true in the case of physical address space where the accessible address space is limited to how much memory the system is equipped with. To cover a 64-bit physical address space, an impracticable amount of 18 exabytes of memory would be required. We approach this problem by iteratively increasing the number of bits we consider for the recovery process, starting from the least-significant (non-offset) address bit.

[2] `https://numpy.org/`

To get started, we find two addresses which differ in their least-significant (non-offset) bit. This ensures that they are affinely independent with respect to this bit. These addresses form set X where $|X| = 2$. Iteratively, the next generated address $\vec{x}^*$ for which $\{\vec{x}^*\} \cup X$ is affinely independent with respect to the lower (non-offset) $|X|$ bits. This step is repeated until either all (non-offset) bits are covered ($|X| = d + 1$) or a threshold is reached where no new address can be found which extends the set. The vertices from X containing just the least-significant (non-offset) $|X| - 1$ bits are then passed to the recovery process. Similarly to handling the offset bits, the transformation matrix is expanded with zeroes on the left to account for the non-recoverable most-significant bits.

Checking that X is affinely independent with respect to the low bits after each iteration ensures that the recovery can be performed at any time when the loop stops. This is not true if X was checked generally for affine independence because when iteration stops and the uncovered high bits are dropped, the stripped vectors might no longer be. Vice-versa, if X is checked for affine independence with respect to the lower n bits then there can still be some address $\vec{x}^*$ such that $\{\vec{x}^*\} \cup X$ is affinely independent with respect to the lower $n + 1$ bits.

This intuition can be explained geometrically. Dropping bits from $\vec{x}$ can be visualized as projection into a lower dimension. If some addresses are affinely independent, they form a simplex. Through projection, a simplex parallel to the dropped axis is no longer a simplex in the lower dimension.

A confidence score for the recovered transformation is calculated as the quotient of recorded mappings matching the index function.

4.4. Reverse-engineering the replacement policy

The address-to-cache-set mapping also allows learning the replacement policy used for a particular cache set. To this end, a set of addresses which map to the same cache set (i.e., a congruent eviction set $\widetilde{E}$) is generated. When accessing an address from the set, one can observe how it affects which cache layer responds when accessing another address.

We adopt the elimination-based approach by Abel and Reineke [AR20] which starts off with a pre-defined set of replacement policies and performs a test for observational equivalence between each policy and the system. The test is implemented by performing random accesses and comparing the behavior. The primitive for comparing two replacement policies is that of a sequence.

We define a sequence ω as an l-tuple of integers $\langle i_0, i_1, \ldots, i_{l-1} \rangle$ with $0 \leq i_0, i_1, \ldots, i_{l-1} < b \leq |\widetilde{E}|, i \in \mathbb{N}_0$. The integers of ω represent indices into congruent eviction set $\widetilde{E}$ for the cache set for which the replacement policy should be reversed.

The hit count $c \in \mathbb{N}_0$ for sequence ω is measured by accessing the addresses in-order and registering all cache hits with exception to the first access to an address. The exception is necessary to obtain the same count no matter whether the address was already cached or not. An implementation is available as algorithm 6.

Similarly, algorithm 7 simulates the behavior of a pre-defined policy Π on the given sequence. The algorithm keeps an associative array C mapping ways to indices i, and another associative array D mapping indices i to ways. The associative array C is used to simulate a cache set in that it keeps track of which way holds which index. Array D allows for performing a constant-time invertible lookup. A hit is registered if the next i from the sequence can be found in the cache set, i.e., $i = C[D[i]]$. In the case of a miss, δ_H is called to obtain the way

Environment: Memory $M = \langle A, L, S, s_0, \delta \rangle$ and current state $s \in S$
Data: Sequence ω and congruent eviction set $\widetilde{E}$
begin
  $I \longleftarrow \emptyset$
  $c \longleftarrow 0$
  forall $i \in \omega$ **do**
    $a \longleftarrow \widetilde{E}[i]$
    $s, l \longleftarrow \delta(s)$
    if $i \notin I \wedge l = min(L)$ **then**
      $c \longleftarrow c + 1$
    end
    $I \longleftarrow I \cup \{i\}$
  end
  return c
end

Algorithm 6: MeasureSequence

Environment: Replacement policy $\Pi = \langle W, R, r_0, \delta_H, \delta_M \rangle$ and current state $r \in R$
Data: Sequence ω
begin
  $I \longleftarrow \emptyset$
  $C \longleftarrow \{[w] \Rightarrow \square,\ \forall w \in W\}$
  $D \longleftarrow \{i \Rightarrow \square,\ \forall i\}$
  $c \longleftarrow 0$
  forall $i \in \omega$ **do**
    $w \longleftarrow D[i]$
    if $C[w] = i$ **then**
      $r \longleftarrow \delta_H(r, w)$
      if $i \notin I$ **then**
        $c \longleftarrow c + 1$
      end
    else
      $r, w \longleftarrow \delta_M(r)$
      $D[i] \longleftarrow w$
      $C[w] \longleftarrow i$
    end
    $I \longleftarrow I \cup \{i\}$
  end
  return c
end

Algorithm 7: SimulateSequence

to evict and store i in.

The elimination-based approach begins with a set of pre-defined replacement policies and then generates sequences repeatedly. All replacement policies with a different hit count than the one measured on the system for that sequence are eliminated. This procedure is repeated until either one replacement policy remains or none. We note that there can be no absolute certainty regarding the replacement policy actually employed by the system unless all possible states and transitions are tested. An exemplary execution trace is shown in table 4.3.

Sequence	**Measured**	**LRU**	**PLRU**	**FIFO**	**SRRIP**
14, 19, 14, 11, 4, 4, 19, 10, 19, 3, 9, 14, 1, 3, 14, 12, 8, 12, 12, 11, 6, 0, 15, 13, 18, 7, 7, 1, 13, 1	11	11	11	11	10
5, 7, 19, 18, 1, 9, 17, 19, 17, 12, 7, 7, 4, 18, 12, 15, 6, 11, 15, 6, 15, 5, 2, 8, 12, 15, 4, 18, 18, 5	7	7	7	7	-
4, 6, 15, 18, 4, 0, 19, 10, 1, 14, 1, 7, 12, 12, 8, 6, 13, 17, 13, 17, 11, 2, 14, 8, 12, 13, 5, 9, 12, 17	6	6	6	6	-
14, 16, 9, 11, 1, 4, 9, 19, 13, 3, 0, 7, 3, 0, 11, 11, 11, 6, 2, 11, 5, 9, 6, 14, 3, 13, 17, 0, 8, 13	7	7	8	8	-

Table 4.3.: Exemplary execution trace of using the elimination-based approach to infer the replacement policy. The LRU policy remains as the only candidate.

4.5. Cache hierarchy considerations

Thus far, we covered algorithms operating on the L1 cache in that algorithmic decisions are based on comparing the responding cache level l with $min(L)$. Hence, working with a reduced set of two cache level symbols $L' = \{min(L),\ \text{Rest}\}$ would not change the execution. We can employ the same algorithms to reverse-engineer higher cache levels by adapting the underlying memory M to bypass the L1 cache. We call such adapter a *bypass adapter*.

The adapted accessor function δ_{L2} which wraps δ is the key aspect of a bypass adapter. Using δ_{L2}, an access is performed as if the L1 cache did not exist and the L2 cache was the one closest to the CPU. Naturally, the adapter provides a modified set of cache level symbols that excludes L1, i.e., $L_{L2} = L \setminus \{min(L)\}$.

Bypass adapters are designed to be stackable. Given a memory that was adapted for the L2 cache, it should be possible to adapt it again to bypass the L1 cache as well as the L2 cache. In that case, δ_{L3} wraps δ_{L2} and the set of cache level symbols is given as $L_{L3} = L_{L2} \setminus \{min(L_{L2})\}$. Without loss of generality, we only describe the first application of a bypass adapter.

The actual implementation depends on the features supported by the underlying platform. On ARMv8 systems we use a loop of `DC ISW` instructions before and after every access to invalidate all ways of the L1 cache set that the access would fall in. This forces a miss on the

L1 cache.

On x86 systems we prepare an eviction set per available L1 cache set. For any access we check which cache level responded. In the case of a response from the L1 cache, we access all addresses from the particular eviction set and perform another access to produce a hit in L2. Otherwise, we remain with just the first access as a bypass is implied by the L1 cache not responding.

Note that in both cases the index function for the L1 cache needs to be known to identify the proper cache set. Hence, higher-level caches can only be reverse-engineered after the index functions for all lower-level caches are found.

5. Implementation

In this chapter we describe how the models and algorithms are implemented in CacheHound.

5.1. Low-level considerations

We target the Linux operating system as it is the defacto standard in the area of high-performance computing. An alternative would be to have CacheHound run bare-metal without any operating system, similar to MemTest86[1]. However, this would be not as easy to use, and would require extensive source code to support the various hardware components.

5.1.1. Accessor function

A central component of the model of abstract memory is the accessor function δ. At its core, this function performs a memory access, specifically a load in our implementation. However, modern multiprocessors employ various techniques such as out-of-order execution which can distort the results. To address this issue, we utilize platform-specific serializing instructions which ensure that memory accesses occur in program order. These are `CPUID`/`LFENCE` under x86 and `ISB SY` under ARM.

Additionally, the accessor function yields the cache level that responded to the loading memory access. One potential solution to obtain this cache level is to define latency ranges for each level, measure the latency of that memory access and return the corresponding cache level. However, there are various side effects which could distort such measurement. One common occurrence is an interrupt from the CPU timer, used for process preemption, or other hardware, such as a network interface card. Additionally, the latency is susceptible to fluctuations depending on the workload on the bus and the TLB state in the case of virtually-indexed caches.

An alternative approach is the utilization of hardware performance counters. These are specialized registers integrated within the CPU that collect and store various metrics related to system performance. Each performance counter register can be configured to track a particular event, such as those for CPU cache hits or misses. By reading the counter before and after a memory access, it is possible to distinguish a hit from a miss.

The typical approach to using performance counters is through an interface such as perf[2], PAPI[3], or Likwid[4] under Linux. These interfaces incur an overhead caused by a routine when reading a performance counter which is acceptable when many events are measured over a longer time period. However, in our case, an overhead which modifies the memory state is not acceptable as it distorts the measurements.

[1] `https://www.memtest86.com/`
[2] `https://perf.wiki.kernel.org/index.php/Main_Page`
[3] `https://linux.die.net/man/3/papi`
[4] `https://hpc.fau.de/research/tools/likwid/`

Consequently, in the implementation, we directly use the instruction to interact with the performance counters. These are `RDPMC` under x86 and `MRS` with the `PMEVCNTR#_EL0` registers under ARMv8.

5.1.2. Noise-free environment

The presented methodology assumes the abstract memory M to be deterministic and predictable and one state cannot decay into another (cf. section 4.1.1). This implies a noise-free environment where any change on the cache state is controlled by CacheHound. In a multi-threaded, preemptive operating system, it is unavoidable to have noise since at any time the process can be preempted and execution switches into kernel space and/or another process.

Linux can be booted with the `isolcpus` flag which allows the isolation of one or multiple CPU cores from the scheduler [Kro07]. This enables a process which is explicitly bound to an isolated CPU core to run as the sole process on that core. However, this is not sufficient since it does not disable periodic timer interrupts, or ticks, as called in Linux. It is necessary to develop a method for disabling interrupts on a CPU core and granting 100% CPU time to the CacheHound process.

The Linux kernel can be compiled in "full tickless mode" using the `CONFIG_NO_HZ_FULL` option. However, in addition of affecting the ease of use, as outlined in [Jon13], this mode is a misnomer as it merely reduces the frequency of ticks to one per second, without fully eliminating them. A new task isolation feature in Linux, as described by [Mar20], appears promising. However, at the time of writing, this feature has not yet been merged into the kernel source code.

In order to achieve the desired level of isolation, the design of CacheHound encompasses two processes: The director process runs the algorithms and manages data structures such as for storing eviction sets, and the agent process performs the actual memory accesses on a dedicated core. This agent process is implemented as a kernel thread and, through kernel functions, can disable any interrupts. Both processes communicate through a core-to-core channel. That is to say, the director process communicates the address to be accessed to the agent process and the agent process responds with the cache level that answered. The setup is sketched in fig. 5.1.

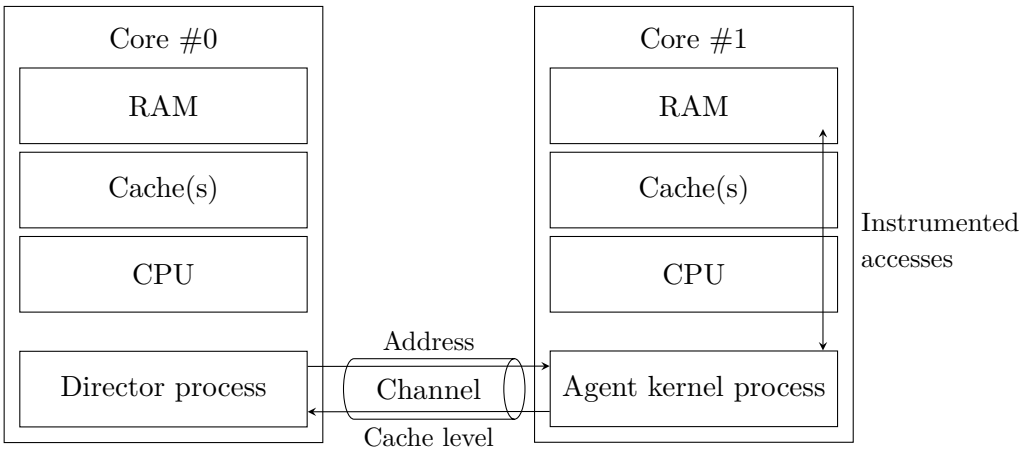

Figure 5.1.: Split architecture, comprising a director and an agent process running on different cores, used to minimize interference.

The advantage of this split approach is that the director process can be arbitrarily complex, as its implicit memory accesses on the executing core do not distort the cache state on the core running the agent process. This is in contrast to implementations such as the one described in [VKM18], where data must be managed carefully in linked lists and 10–50 measurements are needed to reduce the interference of context switches.

5.2. Project overview

We use C++ as programming language for the implementation of the director process and GNU C with inline assembly for the implementation of the agent process. The source code is available at[5]:

`https://github.com/shilch/CacheHound`

The project is separated into four parts:

- `/lib` contains the CacheHound library (`cachehound` namespace) featuring the implementation of our model and all algorithms described in the previous chapter,
- `/sim` contains a CPU cache simulator (`cachehound::sim` namespace) which can be used to test new developments against,
- `/kernel` contains the kernel module which implements the agent process, and
- `/cli` contains a command-line tool (`cachehound::cli` namespace) to work with the three other parts from the shell.

We describe the implementation of the four parts in the order they are listed.

5.3. Library

The CacheHound C++20 template library implements the our model and algorithms and is meant to be useable standalone in third-party software. Only a fraction of the code is non-templated and can be used header-only by setting the `CACHEHOUND_HEADER_ONLY` macro (which is set by default).

5.3.1. Concepts and interfaces

The model of abstract memory introduced in section 4.1.1 is expressed in terms of an interface such that the underlying implementation can be exchanged without requiring adaption of algorithms. This static polymorphism can be achieved by using concepts in C++. The concepts are described in the `concepts` directory.

The `memory` concept, defined in listing 5.1, is implemented in two stages: The `memory` concept defines address space A and provides an accessor function which does not return the cache level. The `instrumented_memory` concept which builds on top of the `memory` concept provides a member function `instrumented_access(uintptr_t) -> unsigned` that returns the cache level, i.e., this is the accessor function δ as specified in the model of abstract

[5]Archived version available at `https://zenodo.org/doi/10.5281/zenodo.13828840`

memory. Hence, the `instrumented_memory` is the primary concept that algorithms operate on.

The rational for this distinction is future extensibility. For instance, the design also includes a `timed_memory` with a `timed_access(uintptr_t) -> uint64_t` member function which returns the latency instead of the cache level. Using an appropriate `memory` adapter, or wrapper, it would be possible to construct `instrumented_memory` from `timed_memory`, as described in section 5.1.1.

Note that `access(uintptr_t)` and `instrumented_access(uintptr_t)` are expected to behave the same with the only difference of the latter returning a value. If the return value is not required, one should resort to former as the `memory` implementation might be able to perform certain optimizations. Details are discussed in section 5.5.2.

```
template<typename M>
concept memory = requires(M memory, uintptr_t address) {
    { memory.access(address) };
    { memory.regions() } -> memory_region_range;
    { memory.offset_bits() } -> same_as<uint8_t>;
};

template<typename M>
concept instrumented_memory = memory<M>
    and requires(M memory, uintptr_t address, unsigned level) {
    { memory.instrumented_access(address) } -> same_as<unsigned>;
    { memory.levels() } -> same_as<unsigned>;
    { memory.index_bits(level) } -> same_as<uint8_t>;
    { memory.ways(level) } -> same_as<size_t>;
};
```

Listing 5.1: Model of abstract memory defined as C++ concepts (simplified).

The member function `regions()` returns a list of `memory_region`. A `memory_region` is a chunk of contiguously allocated memory that can be accessed through the object implementing the `memory` concept. As shown in listing 5.2, it features functions to obtain the (virtual) base address and a size. There further exists an `extended_memory_region` which additionally supports reading the physical base address.

Concepts for placement and replacement policies are also available as shown in listing 5.3. The placement policy is a functor modeling the index function, taking the address and returning the corresponding set index. The replacement policy notably features the member functions `hit(size_t way)` and `miss() -> size_t` which model the hit (δ_H) and the miss (δ_M) functions.

Lastly, the library encompasses the concept of an eviction strategy in listing 5.4 (cf. section 4.2). The `eviction_strategy` implements the logic of evicting cache sets and features a `maps_to` member function that returns `true` if the specified address maps to any of the cache sets given by a range.

These are the core concepts used by CacheHound. Our library defines further concepts, mainly to distinguish various features on the `memory` as supported by different architectures (e.g., `armv8_memory`).

```
template<typename R>
concept memory_region = requires(R region) {
    { region.base() } -> same_as<uintptr_t>;
    { region.size() } -> same_as<size_t>;
};

template<typename R>
concept extended_memory_region = memory_region<R> and requires(R region) {
    { region.physical_base() } -> same_as<uintptr_t>;
};
```

Listing 5.2: Memory region defined as C++ concepts (simplified).

```
template<typename P>
concept placement_policy = requires(P policy, uintptr_t address) {
    { policy.index_bits() } -> same_as<uint8_t>;
    { policy(address) } -> same_as<size_t>;
};

template<typename P>
concept replacement_policy = requires(P policy, size_t way) {
    { policy.ways() } -> same_as<size_t>;
    { policy.hit(way) } -> same_as<void>;
    { policy.miss() } -> same_as<size_t>;
};
```

Listing 5.3: Placement and Replacement policies defined as C++ concepts (simplified).

5.3.2. Backends and Adapters

Backends (`backends` directory) and adapters (`adapters` directory) both implement the `memory` concept. The difference is that backends implement the core `memory` functions while adapters add layers on top by transforming, or wrapping, existing `memory` objects.

The only backend provided by the library is that of `kernel_memory` which is the userland interface to the CacheHound kernel module. Details are discussed in section 5.5.

There are three adapters provided by the library:

- `physical_adapter` which wraps `memory` to translate physical to virtual addresses,
- `bypass_adapter` which wraps `memory` to create a bypass to the second cache layer (i.e., L2 on the first application, L3 on the second), and
- `armv8_bypass_adapter` which wraps `armv8_memory` to achieve the same but using instructions specific to the ARMv8 instruction set.

The `physical_adapter` requires regions which implement the `extended_memory_region` concept to build a red-black tree mapping physical to virtual base addresses. For any physical address to be translated, it performs a binary search on the tree to find the corresponding

```
template<typename S>
concept eviction_strategy =
    requires(S strategy, uintptr_t address, /* range type */ set_range) {
    { strategy.memory() } -> memory;
    { strategy.maps_to(address, set_range) } -> same_as<bool>;
};
```

Listing 5.4: Eviction strategy defined as a C++ concept (simplified).

mapping to virtual address space. Hence, the access complexity of the underlying `memory` is multiplied by $\log(n)$ where n is the number of memory regions. This is more optimal than the naïve solution of iterating over all pages to find the containing page in order to translate a virtual address.

Figure 5.2 shows an exemplary setup of a `physical_adapter` wrapping a `kernel_memory` by translating addresses from physical to virtual, where necessary, and remapping the `base()` function on regions to their physical counterpart.

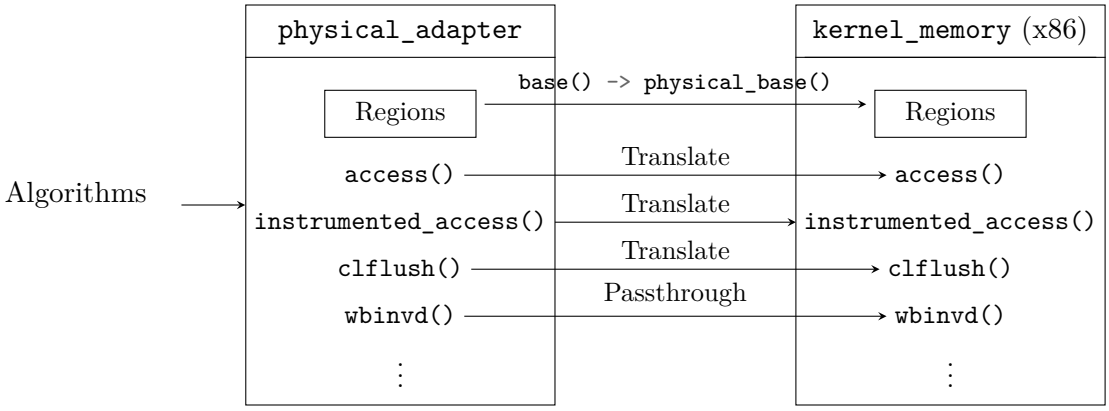

Figure 5.2.: Example case of a `physical_adapter` wrapping an instance of `kernel_memory` by translating between the physical and virtual address spaces.

The `bypass_adapter` and the specialized `armv8_bypass_adapter` form another class of adapters used to create a bypass to the second cache layer. Typically, the algorithms are designed to operate on the first cache layer. A bypass adapter can be used to reuse these algorithms for other layers of the cache hierarchy. The approach used in the implementation is described in section 4.5.

5.3.3. Eviction Strategies

The library provides implementations for both eviction strategies (cf. section 4.2) in the `strategies` directory:

- The `eviction_set_strategy` uses eviction sets in order to evict a cache set, while
- The `cisw_eviction_strategy` uses a `CISW`-instruction loop to achieve the same under the ARMv8 instruction set.

Note that the `eviction_set_strategy` only performs the eviction using pre-established eviction sets. It is the callers responsibility to initialize those and pass them in the strategy's constructor. It also requires an `is_eviction_set` functor which is passed to the eviction set algorithms. The `memory` provided through the `eviction_set_strategy` is stripped from the addresses used in the eviction sets.

5.3.4. Eviction Set Algorithms

The algorithms for finding, testing, reducing, minimizing and locating eviction sets (cf. section 4.2.1) are implemented in the `algo` directory. All algorithms take an `is_eviction_set` functor which implements the eviction set testing logic. The purpose of this design is to allow full customization by the caller to account for the various methods for testing including cache pollution, memory resetting and multiple measurements (cf. section 4.2.1).

Two building blocks are provided: The unsafe `unchecked_is_eviction_set() -> bool` and the safe `checked_is_eviction_set() -> optional<bool>` implement the core testing logic of accessing the target address first, then accessing all addresses from the eviction set, and lastly assessing whether the target address was evicted. The difference is that the checked function performs instrumented accesses to ensure that all accesses produce a cache miss. This is useful to monitor whether cache pollution needs to be repeated.

5.3.5. Pre-defined placement and replacement policies

The CacheHound library ships with a set of pre-defined cache policies in the `policies` directory. The following placement policies are available:

- `modular_placement_policy` which implements the textbook index function,
- `affine_placement_policy` which implements an index function based on an affine transformation on binary vector space (cf. section 4.3), and
- `generic_placement_policy` which wraps an arbitrary C++ lambda as the index function.

The `affine_placement_policy` further features a string conversion function to return a C-like form of the index function. This is used to print a reversed affine transformation in a human-friendly output format.

The following replacement policies are provided by the library:

- `fifo_replacement_policy`,
- `lru_replacement_policy`,
- `plru_replacement_policy`,
- `lru_plru_replacement_policy` which is a combination of LRU and PLRU which Intel uses for a cache associativity of 12,
- `mru_replacement_policy`,
- `srrip_replacement_policy` which can be templated with the bit width and the hit promotion policy, and
- `qlru_replacement_policy`.

5.4. Cache Simulation Library

Additionally to the main library, we provide an extension featuring a `simulated_memory` backend that implements a simulated cache hierarchy. As such, it is possible to test various kinds of cache policies, cache parameters and cache hierarchies against our methodology on a single development machine. The source code is part of the `cachehound::sim` namespace.

Appendix A.2 shows an example use of this simulation library.

5.5. Kernel module

For actual measurements, a physical backend is required. As explained in section 5.1.2, the presented approach necessitates a noise-free environment which is implemented using a Linux kernel module. In this section, we given an overview of the userland interface of this module (which is used by the `kernel_memory` backend), the isolation, the communication channel and the instrumentation.

5.5.1. Userland interface

Upon initialization of the kernel module, the character device `/dev/cachehound` is made available. An interface to the kernel module can then be requested by a userland application via the `open(2)` syscall. The resulting file descriptor implements the `ioctl(2)` and `mmap(2)` syscalls.

A call to `ioctl(2)` with the `CH_IOC_ALLOC_MEMORY` code is used to allocate a new memory region in kernel space. This allocation request expects an argument pointer to a `struct ch_ioc_alloc_config` (cf. listing 5.5) which allows specifying the amount of memory to be allocated. The size is expressed in terms of $2^{\texttt{order}}$ pages. Using the same structure, the virtual and physical address of the newly allocated region are returned. The `order` is limited by what the zoned buddy allocator in Linux accepts in a call to `__alloc_pages()` [Gor04], which we found is usually in the single-digit Megabyte range.

```
struct ch_ioc_alloc_config {
    /* in  */ unsigned order;
    /* out */ uintptr_t virtual_base;
    /* out */ uintptr_t physical_base;
};
```

Listing 5.5: Configuration structure used for the allocation of memory regions in kernel space.

After the desired number of memory regions have been allocated, `ioctl(2)` with the `CH_IOC_START_AGENT` code can be used to launch an agent as a kernel thread. This launch request expects an argument pointer to a `struct ch_ioc_start_config` (cf. listing 5.6) which allows specifying the target CPU core, and the isolation level. The following isolation levels are possible:

- `CH_ISOLATION_OFF`: No isolation guarantee. The kernel thread might be interrupted at any time.

- `CH_ISOLATION_NO_PREEMPT`: Disables preemption (through `preempt_disable()`), i.e., no other thread will run on the selected CPU core for the duration of the agent.
- `CH_ISOLATION_DISABLE_IRQ`: Disables preemption and interrupt handling (through `local_irq_save(flags)`).

```
struct ch_ioc_start_config {
    /* in */  unsigned cpu;
    /* in */  unsigned isolation_level;
    /* out */ uintptr_t channels_base; /* Address of the channels
                                          inside kernel address space */
    /* out */ size_t active_channel;
    /* out */ size_t channel_count; /* # of channels allocated in
                                       the kernel */
};
```

Listing 5.6: Configuration structure used for launching the agent in kernel space.

Notably, via the same structure the index of the currently active core-to-core channel and the total number of allocated channels is returned. A channel enables the low-overhead communication between userspace and kernel thread, primarily for the transmission of addresses to be accessed. We discuss details in section 5.5.2. Channels can be mounted into userspace by a call to `mmap(2)` on the file descriptor.

5.5.2. Communication channel

Communication channels between userspace and kernelspace are implemented as 64 bytes of aligned shared memory. The size is chosen such that it fits into at most one cache line that is kept hot in the cache. Hence, noise created by the communication from and to userspace is minimized.

The channel is laid out in eight 64-bit words, with the first word being the control word and the remaining seven words the data words. One data word can carry one address, allowing for up to seven addresses to be communicated per pass.

The control word can be in two states. If the least significant bit is set, the control word is non-zero and it is the agent's turn to read the addresses from the data words and perform the accesses. The next three lower bits of the control word denote the number of data words in use. If all three bits are zero, the agent terminates gracefully. If the control word is zero, it is the director's turn to read the data words which contain the performance counters for the last address before and after access.

The `kernel_memory` in the library is implemented in such a way that a call to `access()` buffers the addresses until the capacity of seven is reached and the buffer is flushed. Any performance counter values in the channel are ignored. However, a call to `instrumented_access()` always flushes the buffer afterwards to ensure that the specified address appears as the last address in the channel and is, therefore, instrumented.

Agent and director wait for their turn by spinning on the control word. Successfully gaining control of the channel follows acquire semantics. After all data words are written, the control word is written at last using release semantics.

The layout is extended to support architecture-specific instructions: On Intel x86 the next available bit in the control word is used to execute the `WBINVD` instruction. Furthermore, on each data word, the most significant bit of the address can be toggled to produce a `CLFLUSH` of this address instead of a load.

On ARMv8, the `DC CSW`, `DC ISW` and `DC CISW` instructions can be executed by toggling either or both of the two upper bits on a data word. The lower 32 bits of the data word are passed as the first operand to the corresponding instruction.

Since the channel occupies one cache line, it competes with the algorithms on the corresponding cache set. Hence, multiple channels are allocated and the active channel is rotated periodically during runtime.

5.6. Command-line interface

The CacheHound command-line interface (`cli`) can be used to interact with CacheHound from the shell. The tool is built hierarchically to fit our model with the main entrypoint for reverse-engineering being the `reverse` subcommand. The `reverse` subcommand is responsible for the provision of the `memory` backend including the application of any adapters such as for for physical to virtual address translation and bypass.

Nested below the `reverse` subcommand are the `placement` and `replacement` subcommands which operate on the `memory` as parameterized and provided by `reverse`. Both subcommands take further options for specifying the algorithm behavior such as the eviction strategy to use. All subcommands aim to make sensible default choices.

Note that all subcommands provide a `-h` flag to show the possible options and arguments. The root command `cli` can be supplied with the `-V` or `-VV` to enable debug or trace output, respectively. A usage example is shown in listing 5.7.

```
sudo insmod cachehound.ko  # Load the CacheHound kernel module

sudo ./cli -VV                          \  # Enable trace output
    reverse                             \
        -L 2                            \  # Provide bypass to the L2 cache
        -m 1073741824                   \  # Allocate 1GiB of memory
        --kernel-cpu 3                  \  # Launch the kernel agent thread
                                        \  # on CPU core 3
        --kernel-isolation disable-irq  \  # Disable IRQs in agent thread
        --pmu rpi5                      \  # Performance counter configuration
    placement                           \  # Reverse the placement policy
        -S cisw                            # CISW Eviction Strategy
```

Listing 5.7: Example of using the CacheHound command-line interface to reverse the L2 cache placement policy on the Raspberry Pi 5 using the CISW Eviction Strategy.

5.6.1. Performance counter events

As discussed in section 5.1.1, the performance counter values are read before and after a memory access. This implies that three counters need to be programmed to track L1, L2 and L3

hits/misses. On Intel x86 they are are configured by writing to the `IA32_PERFEVTSEL#` MSR, and AMD requires writing to the `CH_PERF_LEGACY_CTL#`[6] MSR instead. On ARMv8 this configuration is performed by first selecting the register using `PMSELR_EL0` and then configuring it by writing to `PMXEVTYPER_EL0`. The event codes used per system are listed in table 5.2.

CPU	--pmu	Monitored events
BCM2712 (Cortex A76)	`rpi5`	`L1D_CACHE_REFILL_RD`
		`L2D_CACHE_REFILL_RD`
		`L3D_CACHE_REFILL`
Intel Xeon E5-2680 v2	`intel`	`MEM_LOAD_UOPS_RETIRED.L1_HIT`
		`MEM_LOAD_UOPS_RETIRED.L2_HIT`
		`MEM_LOAD_UOPS_RETIRED.LLC_HIT`
AMD EPYC 7302	`amd-zen2`	`l2_cache_accesses_from_dc_misses`
		`l2_cache_req_stat.ls_rd_blk_l_hit_x`
		`l3_comb_clstr_state.request_miss`
Fujitsu A64FX	`a64fx`	`L1D_CACHE_REFILL_DM`
		`L2D_CACHE_REFILL_DM`
		`L2D_SWAP_DM`[7]

Table 5.2.: Performance counter events recorded before and after each instrumented memory access to count hits or misses per cache level.

5.6.2. Educated guess

In section 4.2.1 we describe the initialization process of the eviction set strategy which generates random addresses in the aspiration of uncovering a new cache set. Our implementation initially assumes a textbook index function to perform educated guesses on which address will uncover a new cache set and only uses random addresses as a fallback.

Similarly, when finding an eviction set we start with addresses which would be aligned under a textbook index function and only use random addresses with larger eviction set candidates. The address generation logic is implemented in the family of `*_address_distribution` classes.

5.6.3. Solving the affine equation

For the recovery of the affine transformation, described in section 4.3, a library for linear algebra is required. Our implementation uses the Eigen C++ library[8].

Eigen allows custom data types with custom operators to be defined for any type of computation. Our `bit` class implements $\mathbb{B}$ with composition over `bool` and custom `operator+` and `operator-`.

[6] Using the legacy interface allows for code reuse.

[7] Fujitsu published an errata document which states that event `L2D_CACHE_REFILL_DM` should be corrected by subtracting event `L2D_SWAP_DM` [Fuj22].

[8] `http://eigen.tuxfamily.org/`

The library also provides an optimized function for computing the determinant of a matrix. However, as of version 3.4.90 this is only possible on matrices containing floating-point types. To move around this limitation, we use the `mpq_rational` type from the Boost multiprecision library which is a wrapper around `mpq_t` from the GMP library. This type can store an arbitrary-length rational number. Due to the licensing of the GMP library, the affine transformation solver is part of the command-line interface and not the main library.

The determinant of a `bit` matrix is computed by copying it into a `mpq_rational` matrix on which the routine provided by Eigen is then called. A final value for the determinant of type `bit` is calculated by taking the `mpq_rational` value modulo 2. This workaround produces the correct result since the determinant can be defined as a series of additions and multiplications, and $\forall a, b \in \mathbb{Z} : a_{[2]} + b_{[2]} = (a + b)_{[2]}$ and $\forall a, b \in \mathbb{Z} : a_{[2]} * b_{[2]} = (a * b)_{[2]}$.

6. Evaluation

In order to evaluate CacheHound, we run it against the targeted systems to reverse-engineer both, the placement and the replacement policies. The benchmarking of the routines for reverse-engineering the placement policy is conducted in two steps. Initially, the routines are run using some initial parameterization, and we discuss the validity of the obtained results. Subsequently, the impact of modifying the parameters on the runtime, number of memory accesses and error rate is measured. Afterwards, the routines for reverse-engineering the replacement policy are benchmarked. Finally, we interpret the results.

6.1. Setup

We examine the systems outlined in table 1.2. Additionally, we provide table 6.2 which lists the operating systems and compilers used on the examined systems. The commit of the source code used to perform the measurements is tagged as `v0.1.0` in the repository.

CPU	**Operating System**	`uname -r`	**Compiler**
BCM2712	Debian 12	`6.6.28+rpt-rpi-2712`	GCC 12.2.0
Intel Xeon E5-2680 v2	Ubuntu 22.04.4 LTS	`5.15.0-107-generic`	GCC 12.3.0
AMD EPYC 7302	Ubuntu 22.04.4 LTS	`5.15.0-119-generic`	GCC 11.4.0
Fujitsu A64FX	CentOS Linux 8 (Core)	`4.18.0-193.19.1.el8_2.aarch64`	GCC 13.2.0

Table 6.2.: Overview of the operating system and compiler configuration on the examined systems.

While CacheHound is implemented for x86_64- and ARMv8-based processors, some system-specific configurations are necessary.

6.1.1. Prefetcher disablement

Modern processors observe the memory access pattern and try to predictively load data from main memory into the cache(s) before it is explicitly referenced. This mechanism, called prefetching, can lead to unexpected behavior and measurements. Nevertheless, many systems allow disabling the prefetcher.

On the BCM2712 (Raspberry Pi 5), the prefetcher can be disabled by setting the `PF_DIS` bit of the `CPUECTLR_EL1` register. This register is write-accessible in EL1 and EL2 only if bit `ECTLRLEN` of the `ACTLR_EL3` register is set. Since this is not the system default, we patch the Pi 5 bootloader to disable the prefetcher in EL3 on all cores before dropping into EL2 when booting into the operating system. We attach the patch in appendix A.3.

The Intel system allows disablement of the “MLC Streamer”, “MLC Spatial”, “DCU Streamer” and “DCU IP” prefetchers in the BIOS settings.

This is similar in the case of the AMD system which allows disablement of the "L1 Stream HW" and "L2 Stream HW" prefetchers.

Fujitsu provides a HPC extension for the A64FX which can be accessed in EL0, if enabled. The prefetcher operates in the *Stream detect mode* by default. In this mode, prefetching can be disabled by setting the bits `L1PF_DIS` and `L2PF_DIS` of the `IMP_PF_STREAM_DETECT_CTRL_EL0` register.

6.1.2. Additional system configuration

On the systems supporting *Simultaneous Multithreading* (SMT), i.e., Intel and AMD in our setup, we disable this feature to avoid noise from another thread competing for the L1 cache resources. On Intel this feature is called *Hyperthreading*. Both systems allow disablement in the BIOS settings.

On both x86 systems we further boot Linux without the NMI watchdog. The NMI watchdog issues *non-maskable interrupts* (NMI) on the CPU core occupied by the agent kernel process, thereby disrupting the execution of CacheHound and creating noise. Furthermore, when it (spuriously) detects a lockup, it produces a kernel panic requiring a reboot of the system.

We disable the NMI watchdog by setting the sysctl property `kernel.nmi_watchdog=0` in the file `/etc/sysctl.conf` and booting Linux with the `nmi_watchdog=0` argument.

6.2. Placement Policy

6.2.1. Initial parameterization

We run CacheHound against each of the systems using an initial set of parameters. For each system we start with the cache closest to the CPU.

We choose the following initial parameters:

Allocated Memory	1GiB
Isolation Level	no preempt
Eviction Strategy	CISW on ARMv8, Eviction Sets otherwise
Cache Pollution Accesses	10000 (used for Eviction Set Strategy)
Repetitions for `is_eviction_set`	1 (used for Eviction Set Strategy)

For each run we keep track of the elapsed real time, as well as the number of (plain) accesses, instrumented accesses and channel flushes. We also record the confidence value as determined by CacheHound (cf. section 4.3.4).

BCM2712 (Raspberry Pi 5)

On the Raspberry Pi 5, we reverse-engineer the placement policies of the L1D and L2 caches successfully. The results are shown in table 6.5. In both cases, a textbook index function is identified which is plausible since this is a type of function one would expect to find. The confidence is also reported at 100% for both.

We fail to reverse-engineer the L3 cache because of unexpected caching behavior. More precisely, when using the `DC ISW` or `DC CISW` instruction to evict some data from the L2 cache, we notice how the next request on the same data is handled by main memory, as

reported by the performance counters, and not by the L3 cache as one would expect. This breaks the assumptions of the bypass adapter under ARM which is based on this instruction.

We suspect that the observed behavior is a limitation of the *ARM DynamIQ Shared Unit* (DSU) which combines multiple CPUs into a multicore cluster and also implements the L3 cache [Arm23a]. Since the DSU is a separate unit it might not be fully integrated with the cache maintenance instructions offered by the CPU. Another possibility is that the data does indeed reside in the L3 cache but the performance counters are incorrect. We note that the DSU also offers its own set of performance counters. The Raspberry Pi 5 uses the DSU r4p1.

Cache	α	β	Line size	Time	Index function	Confidence
L1D	4	2^8	64 bytes	56s	`I(a) = (a[14:6])`	100%
L2	8	2^{10}	64 bytes	58s	`I(a) = (a[16:6])`	100%
L3	16	2^{11}	64 bytes	——	——	——

Table 6.5.: Cache configuration and index functions as identified on the BCM2712.

Intel Xeon E5-2680 v2

On the Intel system, we also find a textbook index function for the L1D and the L2 caches as shown in table 6.7. This is also expected as prior work on Intel caches has implicitly presupposed such index function (e.g., [Ber04; OST05; Per05; Yan+19; Abe20]) and the `CPUID` instruction reports these caches as not using "Complex Cache Indexing" [Int24].

We cannot run CacheHound against the LLC since the number of cache sets is not a power of two which is incompatible with our reverse-engineering approach based on the recovery of the affine transformation.

Cache	α	β	Line size	Time	Index function	Confidence
L1D	8	2^6	64 bytes	97s	`I(a) = (a[12:6])`	99.7%
L2	8	2^9	64 bytes	260s	`I(a) = (a[15:6])`	99.8%
LLC	20	10×2^{11}	64 bytes	——	——	——

Table 6.7.: Cache configuration and index functions as identified on the Intel Xeon E5-2680 v2.

AMD EPYC 7302

On the AMD system, we successfully reverse-engineer the placement policy of the L1 cache. The results are shown in table 6.9. We also identify a textbook index function here. Similarly, this is expected as prior work on AMD caches has implicitly assumed such index function (e.g., [Ber04; Abe20]).

Reverse-engineering of the L2 cache takes a significant amount of time compared to measurements on the Intel system or the Raspberry Pi 5. The majority of the time is spent on initializing the eviction strategy, i.e., finding an eviction set per cache set. After around

25 minutes of runtime, CacheHound finds 1000 of the 1024 eviction sets. It takes another 90 minutes to find the remaining 24 eviction sets.

After finding 1024 eviction sets, CacheHound takes another 90 minutes to record 1000 mappings of addresses to cache sets, resulting in around 4.3 billion instrumented memory accesses. The obtained placement policy is reported with a confidence of 2.8%, the lowest value we measure for any of the systems (cf. section 6.2.2).

We assume that the issue is rooted in the initialization of the eviction strategy since we find multiple eviction sets which map to the same cache set if one were to assume a textbook index function while some cache sets lack an associated eviction set.

Cache	α	β	Line size	Time	Index function	Confidence
L1D	8	2^6	64 bytes	175s	`I(a) = (a[12:6])`	98.8%
L2	8	2^{10}	64 bytes	———	———	———
LLC	16	2^{14}	64 bytes	———	———	———

Table 6.9.: Cache configuration and index functions as identified on the AMD EPYC 7302.

Fujitsu A64FX (Cray CS500)

On the Fujitsu A64FX we fail to run CacheHound because we can not get the performance counters to count cache misses in EL2, i.e., the ARM exception level that the agent process is running at in kernel space. However, we observe that other events like `LD_SPEC` do count in EL2. We investigate further and notice that cache misses are counted in EL0, i.e., user space, when utilizing the same instructions. As the `DC CISW` instruction is not available in EL0, running the agent process in EL0 is not a solution.

Cache	α	β	Line size	Time	Index function	Confidence
L1D	4	2^6	256 bytes	———	———	———
L2	16	2^{10}	256 bytes	———	———	———

Table 6.11.: Cache configuration as reported on the Fujitsu A64FX.

6.2.2. Custom parameterization

In this section, we run CacheHound with different parameterization to observe how it changes the results. On the ARM system, the Raspberry Pi 5, we also test the Eviction Set Strategy. We test all combinations of the following parameters ten times and record the minimum, maximum and average values for the elapsed real time, the number of (plain) accesses, instrumented accesses and channel flushes, as well as the confidence. Additionally, we note down the number of times that the index function does not match the one we determined in the previous section.

It should be noted that a system reboot is not performed after each execution of CacheHound. Hence, previous runs may result in memory fragmentation, which could lead to a slight

increase in overhead for subsequent runs.

Allocated Memory	1GiB
Isolation Level	off, no preempt, disable irq
Eviction Strategy	CISW (only on ARMv8), Eviction Sets
Cache Pollution Accesses	10000, 25000, 50000
Repetitions for `is_eviction_set`	1, 2, 5

BCM2712 (Raspberry Pi 5)

Table 6.13 shows the results of running CacheHound using the CISW Eviction Strategy. It takes less than one minute on average for the tool to find the placement policy of either the L1D or the L2 cache.

Cache	Isolation	Time			Accesses			Instr. accesses			Flushes			Confidence			Mismatches
		Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	
L1D	off	55s	55.4s	57s	9000	9001	9009	9000	9001	9009	298k	298k	298.3k	100%	100%	100%	0/10
	no preempt	54s	57.8s	80s	9000	9000	9000	9000	9000	9000	298k	298k	298k	100%	100%	100%	0/10
	disable irq	53s	55s	56s	9000	9000	9000	9000	9000	9000	298k	298k	298k	100%	100%	100%	0/10
L2	off	58s	58.8s	60s	0	0	0	22k	22k	22k	2.4m	2.4m	2.4m	99.9%	99.99%	100%	0/10
	no preempt	57s	59.7s	61s	0	0	0	22k	22k	22k	2.4m	2.4m	2.4m	99.9%	99.99%	100%	0/10
	disable irq	58s	59.1s	61s	0	0	0	22k	22k	22k	2.4m	2.4m	2.4m	100%	100%	100%	0/10

Table 6.13.: Results of using different isolation levels when reverse-engineering the placement policy on the BCM2712 and utilizing the CISW Eviction Strategy with `isolcpus`.

Since we do not observe a significant difference between the isolation levels, we boot the system without the `isolcpus` command-line argument and run CacheHound again without any isolation. Using some degree of isolation (such as `no-preempt`) would risk locking a process scheduled on the same core as the CacheHound agent. We run CacheHound using `taskset -c` to ensure that the director process is bound to a different core than the agent kernel process. The results are shown in table 6.14

Cache	Isolation	Time			Accesses			Instr. accesses			Flushes			Confidence			Mismatches
		Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	
L1D	off	47s	48.9s	50s	9000	9000	9000	9000	9000	9000	298k	298k	298k	99.9%	99.99%	100%	0/10
L2	off	52s	54.3s	66s	0	0	0	22k	22k	22k	2.4m	2.4m	2.4m	49.3%	94.92%	100%	1/10

Table 6.14.: Results of using different isolation levels when reverse-engineering the placement policy on the BCM2712 and utilizing the CISW Eviction Strategy without `isolcpus`.

We note that the run producing the index function mismatch for the L2 cache also reported the only low confidence value of 49.3%. CacheHound also prints a warning to the terminal hinting the user to re-run the tool:

```
[2024-08-10 14:03:26.930] [info] Placement policy confidence: 493/1000
```

```
[2024-08-10 14:03:26.930] [warning] The confidence for this placement policy is low
[2024-08-10 14:03:26.930] [warning] Consider increasing the memory size and/or adjusting the indexing method
                                    (virtual/physical)
```

We also run CacheHound using the Eviction Set Strategy. Since there are considerably more combinations of parameters to test and each run takes longer than using the CISW Eviction Strategy, we start with the L1D cache to determine suitable parameters to be used for the other CPU caches. The results are shown in table 6.15.

Cache	Isolation	Pollution	Repetitions	Time			Accesses			Instr. accesses			Flushes			Confidence			Mismatches
				Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	
L1D	off	10000	1	71s	75.1s	79s	109.4m	127.9m	153.1m	3.5m	4.5m	5.7m	19.1m	22.8m	27.6m	13.2%	60.36%	99.7%	6/10
			2	77s	80.4s	84s	150m	166.1m	184.7m	4.7m	5.7m	6.9m	26.2m	29.4m	33.3m	99.7%	99.87%	100%	0/10
			5	99s	103.1s	117s	305.7m	324.6m	429.6m	10.5m	10.8m	11.6m	54.3m	57.2m	73m	99.7%	99.91%	100%	0/10
		25000	1	90s	96.6s	103s	240.7m	295.3m	343.4m	2.8m	4m	5.1m	37.2m	46.2m	54.2m	24.3%	82.22%	99.8%	3/10
			2	106s	110.9s	120s	372.6m	401.6m	453.8m	4.7m	5.4m	6.7m	58m	62.8m	71.5m	99.7%	99.9%	100%	0/10
			5	159s	162.7s	166s	765.3m	770.6m	774m	10.5m	10.6m	10.7m	119.8m	120.7m	121.2m	99.7%	99.92%	100%	0/10
		50000	1	125s	140.9s	156s	480.9m	600.2m	701.9m	2.8m	4.1m	5.3m	71.5m	89.8m	105.6m	49.1%	89.15%	99.4%	2/10
			2	160s	166.9s	182s	742.3m	785.1m	888m	4.7m	5.2m	6.5m	110.7m	117.4m	133.4m	53.5%	95.28%	100%	1/10
			5	277s	282.9s	288s	1.5b	1.5b	1.6b	10.5m	10.6m	10.7m	229.1m	231.2m	232.9m	99.8%	99.91%	100%	0/10
	no preempt	10000	1	72s	74.6s	79s	117.6m	124m	138.8m	3.9m	4.3m	5.2m	20.7m	22m	25m	15.1%	49.82%	98.9%	8/10
			2	77s	80.6s	84s	149.3m	169.2m	185.8m	4.7m	5.9m	6.9m	26.1m	30.1m	33.4m	99.6%	99.87%	100%	0/10
			5	97s	100.7s	102s	306.1m	310.3m	313.5m	10.5m	10.6m	10.7m	54.2m	54.9m	55.5m	99.7%	99.85%	100%	0/10
		25000	1	93s	103.1s	111s	241.2m	305.1m	356.6m	2.8m	4.2m	5.4m	37.2m	47.8m	56.3m	14.3%	66.37%	99.5%	6/10
			2	111s	117.9s	127s	372.4m	406m	466.3m	4.7m	5.4m	6.9m	57.9m	63.4m	72.9m	99.7%	99.87%	100%	0/10
			5	172s	178s	215s	767m	773m	778.4m	10.5m	10.6m	10.7m	120.1m	121.1m	121.9m	99.9%	99.96%	100%	0/10
		50000	1	140s	146.2s	179s	561.9m	606.1m	816m	3.7m	4.2m	6.8m	84m	90.8m	123.4m	50.9%	94.17%	99.5%	1/10
			2	163s	171s	192s	742.5m	781.6m	927.4m	4.7m	5.2m	7m	110.8m	116.8m	139.4m	99.7%	99.9%	100%	0/10
			5	284s	297.8s	303s	1.5b	1.5b	1.6b	10.6m	10.6m	10.7m	228.6m	231.6m	233.8m	50.1%	94.89%	100%	1/10
	disable irq	10000	1	73s	78.1s	82s	110.4m	131m	142.7m	3.6m	4.7m	5.4m	19.3m	23.4m	25.8m	26.2%	81.85%	99.3%	3/10
			2	79s	82.4s	87s	149.8m	159.5m	184m	4.7m	5.3m	6.8m	26.1m	28.1m	33.1m	51.9%	90.33%	100%	2/10
			5	104s	107.4s	118s	307.4m	317.1m	384m	10.6m	11.1m	15.5m	54.4m	56.4m	70.3m	99.8%	99.87%	100%	0/10
		25000	1	96s	99.9s	105s	285.2m	297.5m	324.1m	3.8m	4m	4.7m	44.5m	46.5m	51m	47.9%	84.57%	99.4%	3/10
			2	109s	112.9s	122s	371.5m	387.8m	448.5m	4.7m	5.1m	6.6m	57.8m	60.5m	70.6m	48.8%	94.85%	100%	1/10
			5	166s	170.4s	173s	767.7m	773m	777.4m	10.5m	10.6m	10.7m	120.2m	121m	121.7m	48.7%	94.8%	100%	1/10
		50000	1	141s	149.2s	159s	575.8m	630.8m	696m	3.8m	4.5m	5.3m	86.1m	94.6m	104.7m	27%	67.61%	99%	6/10
			2	166s	174.8s	196s	746.5m	796m	920.1m	4.7m	5.3m	6.9m	111.4m	119m	138.3m	99.8%	99.94%	100%	0/10
			5	285s	295.4s	303s	1.5b	1.5b	1.6b	10.6m	10.6m	10.7m	230.5m	231.5m	232.2m	99.9%	99.91%	100%	0/10

Table 6.15.: Results of using different isolation levels, numbers of pollution accesses and `is_eviction_set` repetitions when reverse-engineering the placement policy on the L1D cache of the BCM2712 and utilizing the Eviction Set Strategy.

We note again that CacheHound correctly assigns a low confidence value to wrong results. The highest confidence reported for a

mismatch is 53.5% while the lowest confidence reported for a match is 98%.

We observe that the number of `is_eviction_set` repetitions has the most significant influence on the correctness of the result. Two repetitions are already sufficient to reduce the mismatch rate to an acceptable level. The number of accesses for cache pollution increases the runtime with little effect on the mismatch rate. Therefore, we decide to run further measurements using 10000 cache pollution accesses and two repetitions. Further more, we use an isolation level of no preempt as this does not seem to influence the results.

We find that the Eviction Set Strategy fails on the L2 cache of the BCM2712. Our test run using the aforementioned parameters takes only around 5 minutes to find 944 of the 1024 eviction sets. However, after one hour, the program is still stuck on 944 eviction sets.

Intel Xeon E5-2680 v2

In contrast to the BCM2712, we can only apply the Eviction Set Strategy to the Intel Xeon E5-2680 v2. For increasing intensity of cache pollution, we observe significantly more required runtime. Due to the total time required to perform these tests, we shorten the measurements for a pollution of 25000 and 50000 and only select the highest isolation level. The results are shown in table 6.16.

Cache	Isolation	Pollution	Repetitions	Time			Accesses			Instr. accesses			Flushes			Confidence			Mismatches
				Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	
L1D	disable irq	10000	1	96s	97.1s	98s	72.5m	73.1m	73.8m	1.2m	1.2m	1.2m	11.5m	11.6m	11.7m	52.2%	90.03%	99.9%	2/10
			2	130s	133.8s	135s	111.2m	112.9m	114.7m	2m	2m	2.1m	17.9m	18.2m	18.4m	49.6%	74.18%	99.6%	5/10
			5	244s	248s	257s	233.4m	235.3m	238.2m	4.6m	4.7m	4.7m	37.9m	38.2m	38.7m	49.1%	83.95%	99.8%	3/10
		25000	1	182s	182.5s	183s	178.7m	179.6m	180.5m	1.2m	1.2m	1.2m	26.7m	26.8m	27m	97.8%	98.62%	99.3%	0/4
			2	264s	654.6s	2996s	280.2m	514.6m	2.1b	2m	3.1m	9.6m	42.1m	76.6m	303.8m	7.5%	75.1%	100%	3/8
			5	636s	1143s	2892s	608.4m	888.3m	2b	4.7m	6.3m	12.2m	91.6m	133.2m	295.7m	11.9%	71.45%	99.6%	2/6
		50000	1	404s	483s	580s	375.5m	392m	425.7m	1.2m	1.2m	1.3m	54.9m	57.2m	62.1m	98.7%	99.4%	99.8%	0/4
			2	644s	857.8s	1480s	598.3m	772.2m	1.3b	2.2m	2.7m	4.5m	87.6m	113m	194.1m	96.7%	98.72%	99.5%	0/5
			5	1206s	2091.4s	4027s	1.2b	1.8b	3.8b	4.6m	6.2m	10.4m	178.2m	269.9m	556.6m	5.2%	61%	99.5%	3/5

Table 6.16.: Results of using different numbers of pollution accesses and `is_eviction_set` repetitions when reverse-engineering the placement policy on the L1D cache of the Intel Xeon E5-2680 v2 and utilizing the Eviction Set Strategy.

We also test the parameters when reverse-engineering the placement policy of the L2 cache and obtain the measurements shown in table 6.17.

Cache	Isolation	Pollution	Repetitions	Time			Accesses			Instr. accesses			Flushes			Confidence			Mismatches
				Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	
L2	disable irq	10000	1	251s	473.6s	1542s	207.5k	967.8k	3.4m	146.2m	294.9m	1b	146.4m	295m	1b	99.5%	99.84%	100%	0/10
			2	378s	419.3s	741s	320.9k	441k	664.1k	231.7m	257.4m	472.9m	231.8m	257.5m	473m	49.5%	94.76%	100%	1/10
			5	771s	873.6s	1234s	587.5k	813.5k	1.1m	486.3m	555.3m	799.1m	486.4m	555.4m	799.3m	99.1%	99.54%	99.8%	0/5
		25000	1	525s	733.9s	2283s	433.8k	780k	2.7m	333.2m	470m	1.5b	333.3m	470.1m	1.5b	99.6%	99.82%	100%	0/9
			2	825s	846.6s	863s	658.1k	1m	1.7m	531.8m	538.9m	547.1m	532m	539m	547.3m	49.7%	92.57%	99.9%	1/7
			5	1553s	1636.7s	1679s	1.4m	1.9m	2.9m	1b	1.1b	1.2b	1b	1.1b	1.2b	98.8%	99.13%	99.4%	0/3
		50000	1	958s	1070.3s	1283s	781k	1.1m	1.3m	655m	731.7m	879.5m	655.1m	731.9m	879.7m	99.5%	99.77%	100%	0/3
			2	1472s	1507s	1552s	1.2m	1.8m	2.4m	991.8m	1b	1.1b	992m	1b	1.1b	53.6%	84.17%	99.7%	1/3
			5	3052s	3113.7s	3180s	2.7m	3.3m	4.2m	2.1b	2.1b	2.2b	2.1b	2.1b	2.2b	99.3%	99.43%	99.7%	0/3

Table 6.17.: Results of using different numbers of pollution accesses and `is_eviction_set` repetitions when reverse-engineering the placement policy on the L2 cache of the Intel Xeon E5-2680 v2 and utilizing the Eviction Set Strategy.

AMD EPYC 7302

Similarly as in the case of the Intel processor, we can only use the Eviction Set Strategy on the AMD EPYC 7302. The results of reverse-engineering the placement policy of the L1D cache are shown in table 6.18. As mentioned before, reverse-engineering the L2 cache takes a significant amount of time and produces bad results. Hence, we do not perform any measurements on the L2 cache.

Cache	Isolation	Pollution	Repetitions	Time			Accesses			Instr. accesses			Flushes			Confidence			Mismatches
				Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	
L1D	no preempt	10000	1	170s	193.8s	216s	73.5m	74.1m	74.7m	1.2m	1.2m	1.2m	11.7m	11.8m	11.9m	98.6%	99.39%	99.8%	0/10
			2	226s	241.2s	263s	115.4m	117.1m	123.4m	2.1m	2.1m	2.3m	18.5m	18.8m	20m	50.1%	94.73%	99.9%	1/10
			5	388s	407.5s	425s	236.3m	241.5m	243.8m	4.7m	4.7m	4.8m	38.4m	39.2m	39.6m	48.6%	79.14%	99.7%	4/10
		25000	1	269s	292.6s	331s	144.6m	152m	188.6m	1.2m	1.3m	2.3m	21.8m	23.1m	29.2m	10.3%	90.69%	99.9%	1/10
			2	388s	408.8s	432s	226.7m	233.6m	252.8m	2.1m	2.2m	2.7m	34.4m	35.5m	38.8m	53.1%	95.06%	99.9%	1/10
			5	691s	727.4s	822s	473.2m	495.4m	569.3m	4.7m	5.3m	7.3m	72.3m	76m	88.6m	7.6%	85.2%	99.7%	2/10
		50000	1	2118s	4984.9s	7748s	1.6b	4b	6.2b	1.3m	1.4m	1.5m	236.1m	572.4m	889.5m	47.2%	86.88%	99.7%	2/8
			2	6978s	7324.5s	7671s	5.6b	5.9b	6.1b	2.3m	2.3m	2.3m	803.2m	838.6m	873.9m	99.2%	99.55%	99.9%	0/2

Table 6.18.: Results of using different numbers of pollution accesses and `is_eviction_set` repetitions when reverse-engineering the placement policy on the L1D cache of the AMD EPYC 7302 and utilizing the Eviction Set Strategy.

6.3. Replacement Policy

We also run CacheHound on each system to reverse-engineer the replacement policies using the elimination-based approach (cf. section 4.4). To attain confidence in the measurements, we perform 100 executions per cache and system. The runtime is negligible, rarely exceeding more than seven seconds on any system. Table 6.19 summarizes the results.

CPU	Cache	Replacement Policy	Confidence
BCM2712	L1D	PLRU	100/100
	L2	Unknown	100/100
	L3	———	———
Intel Xeon E5-2680 v2	L1D	PLRU	100/100
	L2	Unknown	100/100
	LLC	———	———
AMD EPYC 7302	L1D	PLRU	72/100
	L2	———	———
	LLC	———	———
Fujitsu A64FX	L1D	———	———
	L2	———	———

Table 6.19.: Replacement policies as identified per each of the examined systems.

We emphasize that an "unknown" replacement policy means that the elimination left no policies in the candidate set. Hence, the L2 caches on the BCM2712 and the Intel Xeon E5-2680 v2 utilize a replacement policy which is not yet implemented in CacheHound.

6.4. Interpretation

In this section, we interpret the results of reverse-engineering the placement and replacement policies.

We observe that CacheHound is capable of identifying the placement and replacement policies of the L1D cache on all examined systems, with the exception of the Fujitsu A64FX, for which the performance counters are unable to count cache misses in EL2. In all instances, a textbook index function was deduced, consistent with the anticipated outcome (cf. section 6.2.1). In the case of the L2 cache of the Fujitsu A64FX, the index function is expected to be complex, as documented in the manual (cf. [Fuj22]).

The Raspberry Pi 5 allows for a comparison of the two eviction strategies. The CISW Eviction Strategy was found to be faster, with an average runtime of less than 60 seconds, resulting in minimal mismatches. In order to achieve a similarly low mismatch rate, the Eviction Set Strategy requires a repetition count for `is_eviction_set` of two. However, this results in a runtime that is approximately 50% longer on the L1D cache.

The discrepancy widens as one progresses to higher-level caches. While the difference in runtime required for reverse-engineering the placement policy on the L1D and L2 cache, respectively, of the Raspberry Pi 5 is negligible, the average runtime for reverse-engineering the L2 cache of the Intel system is approximately four to five times that of the L1 cache.

This is to be expected, given that the bypass adapter based on eviction sets performs an additional $\alpha + 1$ accesses when a hit is produced in the L1 cache. As the intensity of pollution increases, the ration between the two runtimes decreases, as the majority of pollution results in a cache miss, which does not produce any overhead in the bypass adapter.

CacheHound identifies a PLRU replacement policy on the L1D cache in all working systems. In the case of the BCM2712, these findings are consistent with the technical reference manual for the ARM Cortex-A76 [Arm23b]. An analysis by Abel and Reineke concluded that the Intel Core i5-3470, which is another processor based on the Ivy Bridge microarchitecture, utilizes the same replacement policy in the L1D cache [AR24]. Therefore, we argue that our findings are accurate. We are not aware of any literature documenting the replacement policies used by AMD.

The same ARM manual mentions that the L2 cache of the Cortex-A76 uses a "dynamic biased replacement policy" which is not implemented in CacheHound and for which no further explanations are provided in the document [Arm23c]. Abel and Reineke state that the L2 cache of the Intel Core i5-3470 utilizes a PLRU replacement policy as well [AR24]. However, this cannot be confirmed for the Intel Xeon E5-2680 v2.

7. Discussion and Conclusion

In this chapter we discuss our methodology, including its limitations. We also outline possible applications of the gained knowledge in the fields of HPC as well as security. Lastly, we discuss future work and conclude this research.

7.1. Methodology

A framework for reverse-engineering CPU caches was constructed based on a mathematical model of cached memory. The mathematical model is comprised of three distinct components: a model of abstract memory, a model of replacement policies, and a model of caching behavior.

Furthermore, we presented a methodology for the reverse-engineering of placement policies that employs the aforementioned mathematical model. The methodology is based on a so-called eviction strategy, which is used to evict all cache lines from a given cache set. Two such strategies were introduced: one based on the theory of eviction sets and the other based on the `DC CISW` ARM instruction. We have shown that both eviction strategies produce the same results with the one based on `DC CISW` requiring less runtime. Additionally, we integrated an existing methodology for the inference of replacement policies into our model.

Our implementation, called CacheHound, employs a split architecture, comprising a director and an agent process running on different cores, which minimizes interference. In this setup, the director process can manage data arbitrarily without affecting the cache state on the CPU running the agent process.

7.1.1. Limitations

Our approach was found to have two major shortcomings. In the case of non-ARM processors, an eviction set is prepared for each L1 cache set, with the objective of performing accesses on the L2 cache as if there was no L1 cache. This bypass strategy has the consequence of markedly extending the runtime when reverse-engineering caches of a higher level than L1. In the case of the AMD system, the overhead is as high as 70x of the time required for the L1 cache. While measurements on the Intel L2 cache indicate that the increase in runtime is tolerable, with the runtime for the L2 cache roughly being a product of the runtime for the L1 cache and the L1 cache's associativity, i.e., $t_{L2} = (\alpha_{L1} + 1) \times t_{L1}$, we expect the runtime for the L3 cache to be $\alpha_{L2} + 1 = 9$ times that of the L2 cache, which amounts to more than 4000 seconds in the most favorable case.

Another limitation associated with eviction sets pertains to the mapping of eviction sets to cache set hardware indices during the initialization of the Eviction Set Strategy. This information is essential for the purpose of modeling the index function as an affine transformation. A solution is required that identifies the actual hardware index to which an eviction set maps. Alternatively, an approximation of the hardware indices would be adequate, provided that the estimated indices differ from the actual indices by a fixed bit vector. The current solution is predicated on the assumption of a textbook index function,

which is conflicting with the objective of identifying the index function. It should be noted that this limitation does not apply to the CISW Eviction Strategy, which allows for the accurate identification of hardware indices.

7.1.2. Correctness

An inherent issue associated with a measurement-based reverse-engineering methodology is the correctness of the obtained results. In the absence of an exhaustive exploration of the entire address space, in the context of placement policies, and the entire state space, in the context of replacement policies, any outcome is inevitably an approximation of the actual cache policy. This is due to the possibility of a unique case that was not covered in any of the measurements. For instance, there might be some special address for which the index function returns a set index that is different from what it would return when adhering to the deduction used for all other addresses. It is still necessary to account for measurement errors even when exploring the entirety of the address or state space.

By default, our implementation records a greater number of measurements than are necessary for inference to achieve a high level of confidence in the results. In order to recover the affine transformation, it is sufficient to record only $d + 1$ mappings of addresses to set indices (e.g., 65 on 64-bit machines). However, our implementation records 1000 mappings for the purpose of verifying the obtained function. Similarly, when eliminating replacement policies from the set of candidate policies, further iterations are performed even if only one policy remains, with the objective of confirming the results. It is important to note that, in our evaluation, the tool was run several times, which contributed to the overall confidence in the results.

7.2. Uses in HPC and Security

In this section we discuss various ways of how the knowledge of the cache policies could be utilized in the fields of high-performance computing and security.

7.2.1. Cache side-channels

Even though modern operating systems segregate process memory, thereby preventing one process from reading data from another (unless explicitly allowed to), the cache remains a shared physical resource that is susceptible to side-channels [Per05; OST05].

Covert channel

Percival presented a technique for using the cache state to construct a covert channel between two isolated processes running on the same physical core but on distinct SMT threads. The sender process writes into the covert channel in chunks of β-bit words by reading a cache line from those cache sets whose corresponding bit in the word is set. The receiver constantly accesses a block of memory which fits exactly into the cache. When it encounters a miss for a particular cache set (e.g. through latency measurements), it can infer that the corresponding bit in the word sent by the sender is set [Per05].

The accompanying implementation assumes an index functions based on modular arithmetic. The sender allocates $\beta \cdot 2^{b_{\text{offset}}}$ bytes of contiguous memory whereas the receiver allocates

$\alpha \cdot \beta \cdot 2^{b_{\text{offset}}}$ contiguous bytes. Both are assumed to be aligned. This setup fails when a non-standard index function is used. Knowledge of the index function would allow to fix such setup.

It should be noted that explicitly allocating both chunks of memory at the same virtual address is not a solution to not knowing the index function. While the first $\beta \cdot 2^{b_{\text{offset}}}$ bytes may map exactly the same way into the cache, the remaining $(\alpha - 1)$ slices of the receiver process may not.

Further, we propose an improvement to this technique. Through the knowledge of the (deterministic) replacement policy, it is possible to determine in advance which way of a particular cache set is chosen next as the victim. That way, instead of scanning all α ways, the receiver process only needs to scan the victim way, resulting in a theoretical speedup of α. Depending on the replacement policy, the policy state per cache set might be manageable in CPU registers. Otherwise, some sets of the cache might need to be reserved.

This technique can also be employed at higher-level caches, enabling processes across cores to communicate through their shared CPU cache [Per05; Liu+15].

Attacks

The principle of the covert channel can be adjusted to construct a side-channel attack. In this attack, the victim process implicitly acts as the sender through memory accesses. The attacker process (i.e. the receiver) continuously accesses its memory block and monitors for cache hits/misses to determine which cache sets the victim process accessed. This attack is known as Prime+Probe [OST05].

The primary target for this attack involves processes which use secret data as an array index, such as AES relying on S-Boxes to achieve a high-speed implementation. As shown by Bernstein, an AES key can be recovered from known-plaintext timings [Ber04]. Osvik, Shamir, and Tromer have introduced the Prime+Probe attack and have demonstrated AES key extraction using it.

The implications of knowing the placement and replacement policies are the same as in the case of the covert channel. That is, an index function not based on modular arithmetic breaks the attack but knowledge of the index function fixes it.

7.2.2. Rowhammer attack

The rowhammer attack exploits a side-effect of DRAM where interacting with a memory cell changes the contents of a neighboring memory row due to leaking electrical charges [Kim+14]. This attack relies on a method to circumvent the cache in order to access the DRAM, traditionally using `CLFLUSH` on x86. However, since cache flushing instructions are not available on all platforms and/or environments, eviction sets have been considered as an alternative. Knowledge of the index function allows efficient generation of such eviction sets.

7.2.3. Page coloring

Page coloring, or cache coloring, is a technique for cache partitioning. In this technique, the cache sets are divided into several colored ranges. Memory pages are also assigned colors depending on the range of cache sets it maps to as specified by the index function. The primary motivation for the employment of this technique is to ensure that virtually

contiguous pages map to a distinct range of cache sets, i.e., they do not collide, to maximize cache utilization.

This technique is implemented in several operating systems, such as Windows [Bug+96], but notably not in Linux due to code complexity and the advent of associative caches which decrease the negative performance impact of colliding pages [Tor03].

We also suggest a secondary use case of page coloring that improves process isolation by rendering cache-based side-channels impossible. Processes could also be (multi-)colored and the operating system would ensure that the address space of such process consists only of memory pages of the conforming coloring. This could be useful for hypervisors or systems running processes of various classification levels on the same hardware. In Linux, this could be implemented via the namespaces feature or cgroups. Note that also in this use case, the operating system needs to know the index function of the cache(s).

7.2.4. Scratchpad memory

Scratchpad Memory is high-speed memory which is provided in some processors and can be used to for temporary storage. In that way, it is comparable to CPU caches but with explicit control. With knowledge of the cache replacement and placement policies it would be possible for a program to calculate how some memory access changes the cache state. Consequently, it would be possible to develop a library that allows for explicit caching of data. In order to not interfere with transparent caching, the cache could be separated (colored) into a transparently managed and a library-managed partition.

7.2.5. Cache-Policy-guided optimization

The results could further be employed for reducing the number of cache misses in performance-critical executables. Linkers, which combine multiple object files into a single executable binary, could be modified to ensure that frequently accessed variables do not map to the same cache set. To this end, the placement policy must be known by the linker. Attributes offered by the compiler could be utilized to mark all such variables. A similar procedure could be used for the text segment, i.e., instructions.

When it comes to manually tuning program performance, modern profilers allow for monitoring cache hits and misses in any section of the program. Examples include the previously mentioned perf[1] and Likwid[2] under Linux. However, we are not aware of any profiler that shows why some particular access causes a high rate of cache misses to total accesses. A profiler that takes the placement and replacement policies into account could simulate the caching behavior and provide statements such as “In 20% of the cases, variable A is not in the cache due to a prior access to variable B.”

7.3. Future work

In this work, we have only covered a subset of cache policies and cache hardware. However, the model of abstract memory facilitates further reverse-engineering which we touch briefly on.

[1] `https://perf.wiki.kernel.org/index.php/Main_Page`
[2] `https://hpc.fau.de/research/tools/likwid/`

7.3.1. Instruction caches

At the time of writing, the implementation focuses only on data and unified caches but does not provide a backend for L1 instruction caches. In contrast to data caches, instruction caches are not involved during a memory load but in an instruction fetch. Consequently, a corresponding backend would have to perform an instruction fetch on the address passed to the accessor function. This could be implemented by filling the allocated range of memory with machine code that jumps to a previously set-up handler. The accessor function would then just have to perform a jump to the desired address. Performance counters could be monitored before the jump and in the handler.

7.3.2. Translation Lookaside Buffer

The Translation Lookaside Buffer (TLB) is another kind of cache inside the CPU which is responsible for caching virtual to physical address mappings. TLBs follow the same hardware design as CPU caches with many architectures distinguishing instruction and data TLBs as well as L1 and L2 TLBs. Hence, TLBs provide for interesting hardware, adjacent to CacheHound's primary use case.

Reverse-engineering of TLBs necessitates a backend that performs TLB lookups instead of memory loads. In this setup, A is not a subset of the (virtual) address space but of the available page numbers. During initialization, the backend has to allocate some pages, and, on access, it performs a load on some address in the specified page monitoring the TLB-specific performance counters. Processors kindly provide instructions for TLB flushing which are required when page structures are modified (e.g., during a context switch). For example, x86 provides `INVLPG` which is the TLB-equivalent to `CLFLUSH`.

7.3.3. Fallback methodologies

Prior research has studied the utilization of automata learning for reverse-engineering of replacement policies [Rue13; Vil+20]. Furthermore, Gerlach et al. [Ger+24] presented an automated approach for deriving a non-linear hash function from a given mapping. Both methodologies have been shown to require a significant amount of runtime but could be implemented on top of CacheHound as a fallback when the methodology presented in this work fails.

7.3.4. Last-Level Cache slice mapping

In section 3.1 we explored prior work on the reverse-engineering of the Intel LLC slice function which is related to our approach for learning the placement policy as described in section 4.3. CacheHound could automatically reverse Intel LLCs if made aware of the LLC slice in which a cache line is placed. To that end, a new `eviction_strategy` can be implemented which operates on slices instead of cache sets. The mapped-to slice can be determined using performance counters. In order to reach the last-level cache, bypass adapters must be used.

An exception are processors with a number of slices that are not a power of two (cf. [Yar+15]). This prevents us from running CacheHound on the LLC of the Intel system (cf. section 6.2.1). A fallback approach described in section 7.3.3 could work.

7.3.5. Systematic memory page mapping

The kernel module currently relies on the `memremap()` interface for mapping memory pages into virtual address space. This interface uses the next available virtual address. Recovery of the index function is optimal when the accessible address space is spread out as far as possible to cover most/all of the address bits. In the current setup, we allocate a significant amount of pages in an attempt to cover most of the bits. In a more systematic approach, only a few pages would be needed to achieve this.

7.3.6. Further cache properties

Future work can extend our methodology to reverse-engineer further cache properties. For example, an extension to the model of abstract memory that allows for writing (instead of only reading) could be used to find details on the write policy, monitoring TLB performance counters would enable distinguishing VIVT and VIPT caches, and algorithms using two backends on two distinct cores could learn more about the coherency protocol.

7.4. Conclusion

We provide CacheHound, an open-source implementation of our methodology, comprised of a command-line interface and a kernel module. It uses a split architecture encompassing a director and an agent kernel process to minimize noise during any measurements and allow the director process to be arbitrarily complex. The results demonstrate that the tool functions reliably on the L1D caches and some of the L2 caches of several processor architectures based on x86 and ARMv8. The runtime is satisfactory, as the tool is only required to be executed once per processor model and requires less than a minute in the best case and less than an hour in the worst case.

Furthermore, we have demonstrated how the knowledge of cache policies identified by the tool can be utilized in the domains of high-performance computing and security. Use cases include cache side-channels attacks and mitigations, as well as code optimizations. Future work could extend the capabilities of CacheHound by incorporating support for additional types of caches, such as instruction caches or TLBs. Moreover, the model lends itself to the support of strategies for reverse-engineering LLC slice mappings and other cache properties.

The tool was unable to run on the Fujitsu A64FX due to the performance counters not counting cache misses in EL2. This is unfortunate, as the documented non-textbook index function of the A64FX motivated this research, and this CPU model would have been an interesting candidate to prove that CacheHound can identify such complex placement policy. We will investigate this case further and attempt to provide a solution.

A. Additional source code

A.1. Recovering the affine transformation matrix

```
import numpy as np
import sys

np.set_printoptions(threshold=sys.maxsize, linewidth=np.inf)

inputs = [0xe94abdfcb21cb7, 0x0bc41b0a5d0b15, 0x176ba78278b3ee, 0x080d3417888231,
          0x6311575eb5b269, 0xfcae88a30d4bbd, 0x2d464ebe2224bf, 0xecdcebf73d4f84,
          0xde0f7df3b43b04, 0x94e300f2079ab8, 0xe53ee4bdbc1e6f, 0x74fda4cf4ed3c1,
          0x870d7b1bcd7d7b, 0xbaa85348f858bb, 0x5b40e6b0e916ea, 0x037a11a978f713,
          0xc0b5313c519e8c, 0x81120023ae6dbb, 0x96e08a4d2eb02e, 0x27c508f62afbf7,
          0xd83351e639a29a, 0x3351ee1d6b68eb, 0x5addf718356f03, 0xe1d58796bca31d,
          0xc1e54430676c09, 0x2c41ca2c9bcc02, 0xa3ed040eaf70c1, 0x9144de3d15200d,
          0x3c0c2e597a728b, 0x4cdb1db9a304df, 0xcaaf4a8f14b60e, 0x013482e8cee5a9,
          0x116a803825667b, 0x71bfea20bbd77b, 0x32860b2bc71312, 0x3d530b7cf39953,
          0x48d17e44aba129, 0xd8b0f071b9398a, 0x04fa6085c6b006, 0x590c4f75e4527d,
          0x3d50efeea5c21c]
input_bits  = len(inputs) - 1

outputs = [0x5b7, 0x415, 0xee, 0x531, 0x569, 0x3bd, 0x2bf, 0x384, 0x704, 0x1b8,
           0x46f, 0x6c1, 0x77b, 0xbb, 0x2ea, 0x13, 0x68c, 0x2bb, 0x42e, 0x5f7,
           0x39a, 0x7eb, 0x103, 0x21d, 0x209, 0x202, 0xc1, 0xd, 0x48b, 0xdf,
           0x60e, 0x2a9, 0x7b, 0x37b, 0x312, 0x753, 0x729, 0x38a, 0x606, 0x77d,
           0x21c]
output_bits = 11

assert len(inputs) == len(outputs)

def to_bitvector(num: int, bits: int):
    num = num & ((1 << bits) - 1)
    return np.transpose(
        np.matrix(
            [int(bit) for bit in bin(num)[2:].zfill(bits)],
            dtype='int'
        )
    )

# D' matrix
d_prime = np.vstack([
```

```
    np.hstack([
        to_bitvector(inp, input_bits)
        for inp in inputs
    ], dtype='int'),
    np.ones(shape=(1, input_bits + 1), dtype='int')
])

def submatrix_det(i: int, j: int) -> int:
    a = np.delete(d_prime, (i), axis=0)
    b = np.delete(a, (j), axis=1)
    return int(round(np.linalg.det(b))) % 2

M = np.zeros(shape=(output_bits,len(inputs)), dtype='int')

for i in range(output_bits):
    for j in range(len(inputs)):
        M[i, j] = sum(
            to_bitvector(outputs[k], output_bits)[i, 0] and submatrix_det(j, k)
            for k in range(len(outputs))
        ) % 2

print(M)

# Outputs the A64FX matrix (augmented):
#[[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
# [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
# [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
# [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
# [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
# [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
# [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
# [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
# [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
# [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
# [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
# [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]]
```

A.2. Using the cache simulation library

```
using namespace cachehound;

static constexpr uintptr_t base_address = 0x0;
static constexpr size_t size = 1 << 20
                     , ways = 4, index_bits = 6, offset_bits = 6;

sim::simulated_memory memory {
    base_address, size, offset_bits,
```

```
    /* L1 */ sim::set_associative_cache {
        sim::basic_set_associative_policy {
            index_bits,
            modular_placement_policy{offset_bits, 1 << index_bits},
            lru_replacement_policy{ways}
        }
    },
    /* L2 */ sim::set_associative_cache {
        sim::basic_set_associative_policy {
            1 + index_bits,
            modular_placement_policy{offset_bits, 1 << (1 + index_bits)},
            lru_replacement_policy{2 * ways}
        }
    }
};
```

A.3. Disabling the Raspberry Pi 5 prefetcher

Disabling the Raspberry Pi 5 prefetcher requires writing into the `CPUECTLR_EL1` (also known as `s3_0_c15_c1_4`) which is only possible in the bootloader. Hence, we fork the "Trusted Firmware-A"[1] bootloader project and adjust bootloader stage 3-1 (BL31) as follows.

We patch the `bl31/bl31_main.c` to disable the prefetcher on the primary core:

```
    @@ -101,6 +101,9 @@ void bl31_setup(u_register_t arg0, u_register_t
    ↪  arg1, u_register_t arg2,
    /* Perform early platform-specific setup */
    bl31_early_platform_setup2(arg0, arg1, arg2, arg3);

+   /* Disable prefetcher */
+   asm volatile("MSR s3_0_c15_c1_4, %[val]" :: [val] "r"(0x96156B020) :
↪   "memory");
+
    /* Perform late platform-specific setup */
    bl31_plat_arch_setup();
```

We also patch the platform-specific secondary setup at `plat/rpi/common/aarch64/plat_helpers.S` to disable the prefetcher on all other cores as well:

```
    @@ -120,6 +120,10 @@ endfunc plat_wait_for_warm_boot
    * ----------------------------------------------------
    */
func plat_secondary_cold_boot_setup
+   mov     x1, #0xb020
+   movk    x1, #0x6156, lsl #16
+   movk    x1, #0x9, lsl #32
+   msr     s3_0_c15_c1_4, x1
```

[1] `https://www.trustedfirmware.org/projects/tf-a/`

```
    b       plat_wait_for_warm_boot
endfunc plat_secondary_cold_boot_setup
```

We compile the BL31 and copy the binary file as `armstub8-2712.bin` to the boot partition.

B. Glossary of mathematical symbols

Symbol	Description
b_{offset}	Number of offset bits in the address where $2^{b_{\text{offset}}}$ is the cache line size in bytes
b_{index}	Number of index bits in the address when assuming a textbook index function where $2^{b_{\text{index}}}$ is the number of cache sets β
b_{tag}	Number of tag bits in the address when assuming a textbook index function
α	Number of ways / associativity of a cache
β	Number of sets of a cache
M	Abstract memory which is defined as the quintuple $\langle A,\ L,\ S,\ s_0,\ \delta \rangle$
A	Set of addresses accessible on M (a subset of the address space) where no two distinct addresses join a cache line
a, t	Address from A where a is some arbitrary address and t a target address
L, l	Strictly totally ordered set L of symbols l representing each cache level in M, including one for main memory
S, s	Set S of possible states s of M, including all caches
$\delta : S \times A \to S \times L$	Accessor function that, given an address, transitions memory M into the next state and yields the symbol representing the cache layer that served the access
δ_{inv}	Optional invalidation function which removes a cache line for a given address from the cache
Π	Replacement policy which is managed per cache set and defined as the quintuple $\langle W, R, r_0, \delta_H, \delta_M \rangle$
W, w	Set W of symbols where every symbol w represents a way in Π
R, r	Set R of possible states r of the replacement policy Π

Symbol	Description
$\delta_H : R \times W \to R$	Hit function which is invoked whenever the cache line at $w \in W$ receives a hit
$\delta_M : R \to R \times W$	Miss function which is invoked whenever space for a new cache line must be made in the cache set, with the victim cache line that should be replaced given by $w \in W$
δ_M^n	n^{th} application of the miss function
$\widetilde{E}$	Congruent eviction set which is a set of at least α addresses which map to the same cache set
E	Eviction set which is a superset of $\widetilde{E}$ and might contain addresses which map to a different cache set
$\mathcal{E}$	Space of eviction sets
$I : A \mapsto [0, \beta)$	Index function which maps an address to a cache set
T	Set of cache line tags
$\tau : A \mapsto T$	Tag function which maps addresses to cache line tags
Γ	$\beta \times \alpha$ 0-indexed matrix representing the cache contents where elements are either cache line tags or $\square$ (empty)
GF(2), $\mathbb{B}$, $\mathbb{Z}/2\mathbb{Z}$	Ring of integers modulo 2, i.e., 0 and 1, with addition being the XORing and multiplication the ANDing of the two operands
ω	Sequence which is a l-tuple of integers $\langle i_0, i_1, \ldots, i_{l-1} \rangle$ with $0 \leq i_0, i_1, \ldots, i_{l-1} < b \leq \lvert\widetilde{E}\rvert, i \in \mathbb{N}_0$

List of Figures

List of Tables

List of Listings

Bibliography

[Lem19] Daniel Lemire. *Memory-level parallelism: Intel Skylake versus Intel Cannonlake*. Daniel Lemire's blog. Jan. 1, 2019. URL: https://lemire.me/blog/2019/01/01/memory-level-parallelism-intel-skylake-versus-intel-cannonlake/ (visited on 11/26/2023).

[WM95] Wm. A. Wulf and Sally A. McKee. "Hitting the Memory Wall: Implications of the Obvious". In: *ACM SIGARCH Computer Architecture News* 23.1 (Mar. 1995), pp. 20–24. ISSN: 0163-5964. DOI: 10.1145/216585.216588. URL: https://dl.acm.org/doi/10.1145/216585.216588 (visited on 01/24/2024).

[Mac02] Philip Machanick. *Approaches to Addressing the Memory Wall*. 2002. URL: https://homes.cs.ru.ac.za/philip/Publications/Techreports/2002/Reports/memory-wall-survey.pdf (visited on 01/23/2024).

[Sha19] Amanda K. Sharp. *Cache Blocking Techniques*. Intel. Mar. 26, 2019. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/cache-blocking-techniques.html (visited on 09/22/2024).

[Han98a] Jim Handy. "Thrashing: Getting the most out of the cache". In: *The Cache Memory book, 2nd edition*. Acad. Press, 1998, pp. 22–27. ISBN: 0-12-322980-4.

[Han98b] Jim Handy. "Cache data and cache-tag memories". In: *The Cache Memory book, 2nd edition*. Acad. Press, 1998, p. 18. ISBN: 0-12-322980-4.

[HP18] John Hennessy and David Patterson. "Basics of Memory Hierarchies: A Quick Review". In: *Computer Architecture: A Quantitative Approach, 6th Edition*. Elsevier, 2018, p. 81. ISBN: 978-0-12-811905-1.

[Fox24a] Charles Fox. "Caches". In: *Computer Architecture: From the Stone Age to the Quantum Age*. 1st ed. No Starch Press, 2024, pp. 226–233. ISBN: 978-1-71850-286-4.

[Fuj22] Fujitsu Limited. *A64FX Microarchitecture Manual*. Version 1.8.1. Nov. 2022. URL: https://github.com/fujitsu/A64FX/blob/master/doc/.

[Int24] Intel Corporation. "CPUID – CPU Identification". In: *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Vol. 2A. 2024, p. 805. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[Arm24a] Arm Limited. "D7.5.8.1.1 Terminology for cache maintenance instructions operating by set/way". In: *Arm® Architecture Reference Manual for A-profile architecture*. Mar. 20, 2024, p. 6531. URL: https://developer.arm.com/documentation/ddi0487/latest/.

[Han98c] Jim Handy. "The concept of locality". In: *The Cache Memory book, 2nd edition*. Acad. Press, 1998, pp. 6–8. ISBN: 0-12-322980-4.

[Fox24b] Charles Fox. "The Memory Hierarchy". In: *Computer Architecture: From the Stone Age to the Quantum Age*. 1st ed. No Starch Press, 2024, pp. 215–217. ISBN: 978-1-71850-286-4.

[Pav19] Pavlov. *Intel Skylake - 7-Zip LZMA Benchmark*. Sept. 5, 2019. URL: https://www.7-cpu.com/cpu/Ice_Lake.html (visited on 07/21/2024).

[HWH13] Ralf Hund, Carsten Willems, and Thorsten Holz. "Practical Timing Side Channel Attacks against Kernel Space ASLR". In: *2013 IEEE Symposium on Security and Privacy*. 2013 IEEE Symposium on Security and Privacy. ISSN: 1081-6011. May 2013, pp. 191–205. DOI: 10.1109/SP.2013.23. URL: https://ieeexplore.ieee.org/document/6547110/?arnumber=6547110 (visited on 07/27/2024).

[Dre07] Ulrich Drepper. "What Every Programmer Should Know About Memory". In: *Red Hat, Inc* (2007), pp. 15–20.

[Han98d] Jim Handy. "Choosing cache policies". In: *The Cache Memory book, 2nd edition*. Acad. Press, 1998, pp. 49–77. ISBN: 0-12-322980-4.

[Bot04] James Bottomley. *Understanding Caching*. Linux Journal. 2004. URL: https://www.linuxjournal.com/article/7105 (visited on 09/02/2024).

[Lip+20] Moritz Lipp et al. "Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors". In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security. Taipei Taiwan: ACM, Oct. 5, 2020, pp. 813–825. ISBN: 978-1-4503-6750-9. DOI: 10.1145/3320269.3384746. URL: https://dl.acm.org/doi/10.1145/3320269.3384746 (visited on 07/27/2024).

[HP11] Damien Hardy and Isabelle Puaut. "WCET analysis of instruction cache hierarchies". In: *Journal of Systems Architecture* 57.7 (Aug. 2011), pp. 677–694. ISSN: 13837621. DOI: 10.1016/j.sysarc.2010.08.007. URL: https://linkinghub.elsevier.com/retrieve/pii/S1383762110001074 (visited on 09/02/2024).

[Rei+07] Jan Reineke et al. "Timing predictability of cache replacement policies". In: *Real-Time Systems* 37.2 (Sept. 14, 2007), pp. 99–122. ISSN: 0922-6443, 1573-1383. DOI: 10.1007/s11241-007-9032-3. URL: http://link.springer.com/10.1007/s11241-007-9032-3 (visited on 01/23/2024).

[Arm14] Arm Limited. *ARM Cortex-R Series Programmer's Guide*. 2014. URL: https://documentation-service.arm.com/static/60ffb7c39ebe3a7dbd3a78b7 (visited on 09/23/2024).

[ZMN08] Jason Zebchuk, Srihari Makineni, and Don Newell. "Re-examining cache replacement policies". In: *2008 IEEE International Conference on Computer Design*. 2008 IEEE International Conference on Computer Design (ICCD). Lake Tahoe, CA, USA: IEEE, Oct. 2008, pp. 671–678. ISBN: 978-1-4244-2657-7. DOI: 10.1109/ICCD.2008.4751933. URL: http://ieeexplore.ieee.org/document/4751933/ (visited on 01/23/2024).

[PJ15] Xiaoyue Pan and Bengt Jonsson. "A modeling framework for reuse distance-based estimation of cache performance". In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Philadelphia, PA, USA: IEEE, Mar. 2015, pp. 62–71. ISBN: 978-1-4799-1957-4. DOI: 10.1109/ISPASS.2015.7095785. URL: http://ieeexplore.ieee.org/document/7095785/ (visited on 01/24/2024).

[AMM04] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. "Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite". In: *Proceedings of the 42nd annual Southeast regional conference*. ACM SE04: ACM Southeast Regional Conference 2004. Huntsville Alabama: ACM, Apr. 2, 2004, pp. 267–272. ISBN: 978-1-58113-870-2. DOI: 10.1145/986537.986601. URL: https://dl.acm.org/doi/10.1145/986537.986601 (visited on 01/25/2024).

[Jal+10] Aamer Jaleel et al. "High performance cache replacement using re-reference interval prediction (RRIP)". In: *Proceedings of the 37th annual international symposium on Computer architecture*. ISCA '10: The 37th Annual International Symposium on Computer Architecture. Saint-Malo France: ACM, June 19, 2010, pp. 60–71. ISBN: 978-1-4503-0053-7. DOI: 10.1145/1815961.1815971. URL: https://dl.acm.org/doi/10.1145/1815961.1815971 (visited on 01/23/2024).

[AR20] Andreas Abel and Jan Reineke. "nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems". In: *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Aug. 2020, pp. 34–46. DOI: 10.1109/ISPASS48437.2020.00014. arXiv: 1911.03282[cs]. URL: http://arxiv.org/abs/1911.03282 (visited on 01/23/2024).

[GD11] Xiaoming Gu and Chen Ding. "On the Theory and Potential of LRU-MRU Collaborative Cache Management". In: *ACM SIGPLAN Notices* (2011).

[Qur+07] Moinuddin K. Qureshi et al. "Adaptive insertion policies for high performance caching". In: *Proceedings of the 34th annual international symposium on Computer architecture*. SPAA07: 19th ACM Symposium on Parallelism in Algorithms and Architectures. San Diego California USA: ACM, June 9, 2007, pp. 381–391. ISBN: 978-1-59593-706-3. DOI: 10.1145/1250662.1250709. URL: https://dl.acm.org/doi/10.1145/1250662.1250709 (visited on 01/23/2024).

[Don+01] Donghee Lee et al. "LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies". In: *IEEE Transactions on Computers* 50.12 (Dec. 2001), pp. 1352–1361. ISSN: 0018-9340. DOI: 10.1109/TC.2001.970573. URL: http://ieeexplore.ieee.org/document/970573/ (visited on 01/25/2024).

[OOW93] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. "The LRU-K page replacement algorithm for database disk buffering". In: *ACM SIGMOD Record* 22.2 (June 1993), pp. 297–306. ISSN: 0163-5808. DOI: 10.1145/170036.170081. URL: https://dl.acm.org/doi/10.1145/170036.170081 (visited on 01/25/2024).

[RD90] John T. Robinson and Murthy V. Devarakonda. "Data cache management using frequency-based replacement". In: *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems - SIGMETRICS '90*. the 1990 ACM SIGMETRICS conference. Univ. of Colorado, Boulder, Colorado, United States: ACM Press, 1990, pp. 134–142. ISBN: 978-0-89791-359-1. DOI: 10.1145/98457.98523. URL: http://portal.acm.org/citation.cfm?doid=98457.98523 (visited on 01/25/2024).

[MM03] Nimrod Megiddo and Dharmendra S. Modha. "ARC: A Self-Tuning, Low Overhead Replacement Cache". In: *2nd USENIX Conference on File and Storage Technologies (FAST 03)*. San Francisco, CA: USENIX Association, Mar. 2003. URL: https://www.usenix.org/conference/fast-03/arc-self-tuning-low-overhead-replacement-cache.

[BM04] Sorav Bansal and Dharmendra S. Modha. "{CAR}: Clock with Adaptive Replacement". In: 3rd USENIX Conference on File and Storage Technologies (FAST 04). 2004. URL: https://www.usenix.org/conference/fast-04/car-clock-adaptive-replacement (visited on 08/30/2024).

[Jah+12] Sanjeev Jahagirdar et al. "Power management of the third generation intel core micro architecture formerly codenamed ivy bridge". In: *2012 IEEE Hot Chips 24 Symposium (HCS)*. 2012 IEEE Hot Chips 24 Symposium (HCS). Cupertino, CA, USA: IEEE, Aug. 2012, pp. 1–49. ISBN: 978-1-4673-8879-5. DOI: 10.1109/HOTCHIPS.2012.7476478. URL: http://ieeexplore.ieee.org/document/7476478/ (visited on 01/26/2024).

[Abe20] Andreas Abel. "Automatic Generation of Models of Microarchitectures". PhD thesis. 2020. URL: https://publikationen.sulb.uni-saarland.de/bitstream/20.500.11880/29336/1/thesis.pdf (visited on 09/23/2024).

[KHM01] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. "Cache decay: Exploiting generational behavior to reduce cache leakage power". In: *Proceedings of the 28th annual international symposium on Computer architecture*. 2001, pp. 240–251.

[Sea15] Mark Seaborn. *Lacking Rhoticity: L3 cache mapping on Sandy Bridge CPUs*. Lacking Rhoticity. Apr. 27, 2015. URL: https://lackingrhoticity.blogspot.com/2015/04/l3-cache-mapping-on-sandy-bridge-cpus.html (visited on 07/27/2024).

[Mau+15] Clémentine Maurice et al. "Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters". In: *Research in Attacks, Intrusions, and Defenses*. Ed. by Herbert Bos, Fabian Monrose, and Gregory Blanc. Vol. 9404. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 48–65. ISBN: 978-3-319-26361-8 978-3-319-26362-5. DOI: 10.1007/978-3-319-26362-5_3. URL: http://link.springer.com/10.1007/978-3-319-26362-5_3 (visited on 07/07/2024).

[McC21] John D. McCalpin. *Mapping Addresses to L3/CHA Slices in Intel Processors*. TR-2021-03. ACELab, Sept. 10, 2021. URL: https://hdl.handle.net/2152/87595 (visited on 07/27/2024).

[Pes+16] Peter Pessl et al. "{DRAMA}: Exploiting {DRAM} Addressing for {Cross-CPU} Attacks". In: 25th USENIX Security Symposium (USENIX Security 16). 2016, pp. 565–581. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl (visited on 07/27/2024).

[Gra+18] Ben Gras et al. "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with {TLB} Attacks". In: 27th USENIX Security Symposium (USENIX Security 18). 2018, pp. 955–972. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/gras (visited on 07/27/2024).

[Kos+20] Jakob Koschel et al. "TagBleed: Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs". In: *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2020 IEEE European Symposium on Security and Privacy (EuroS&P). Sept. 2020, pp. 309–321. DOI: 10.1109/EuroSP48549.2020.00027. URL: https://ieeexplore.ieee.org/document/9230388/?arnumber=9230388 (visited on 07/27/2024).

[IES15] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. *Systematic Reverse Engineering of Cache Slice Selection in Intel Processors*. Publication info: Published elsewhere. 18th EUROMICRO Conference on Digital System Design 2015. 2015. URL: https://eprint.iacr.org/2015/690 (visited on 07/27/2024).

[Yar+15] Yuval Yarom et al. *Mapping the Intel Last-Level Cache*. Publication info: Preprint. 2015. URL: https://eprint.iacr.org/2015/905 (visited on 01/23/2024).

[Liu+15] Fangfei Liu et al. "Last-Level Cache Side-Channel Attacks are Practical". In: *2015 IEEE Symposium on Security and Privacy*. 2015 IEEE Symposium on Security and Privacy (SP). San Jose, CA: IEEE, May 2015, pp. 605–622. ISBN: 978-1-4673-6949-7. DOI: 10.1109/SP.2015.43. URL: https://ieeexplore.ieee.org/document/7163050/ (visited on 01/23/2024).

[Ger+24] Lukas Gerlach et al. *Efficient and Generic Microarchitectural Hash-Function Recovery*. 2024. URL: https://publications.cispa.saarland/3983/1/hash_recovery_sp24.pdf (visited on 09/23/2024).

[AR14] Andreas Abel and Jan Reineke. "Reverse Engineering of Cache Replacement Policies in Intel Microprocessors and Their Evaluation". In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). CA, USA: IEEE, Mar. 2014, pp. 141–142. ISBN: 978-1-4799-3606-9 978-1-4799-3604-5. DOI: 10.1109/ISPASS.2014.6844475. URL: http://embedded.cs.uni-saarland.de/publications/ISPASS14.pdf (visited on 01/23/2024).

[JB07] Tobias John and Robert Baumgartl. "Exact cache characterization by experimental parameter extraction". In: *Proceedings of the 15th International Conference on Real-Time and Network Systems RTNS07*. 2007, pp. 65–74.

[Abe12] Andreas Abel. "Measurement-based Inference of the Cache Hierarchy". Master's Thesis. Universität des Saarlandes, Germany, 2012. URL: http://embedded.cs.uni-saarland.de/literature/AndreasAbelMastersThesis.pdf.

[AR12] Andreas Abel and Jan Reineke. "Automatic Cache Modeling by Measurements". In: *6th Junior Researcher Workshop on Real-Time Computing (in conjunction with RTNS)*. Nov. 2012. URL: http://embedded.cs.uni-saarland.de/publications/CacheModelingJRWRTC.pdf.

[AR13] A. Abel and J. Reineke. "Measurement-based modeling of the cache replacement policy". In: *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS). Philadelphia, PA: IEEE, Apr. 2013, pp. 65–74. ISBN: 978-1-4799-0187-6 978-1-4799-0186-9 978-1-4799-0185-2. DOI: 10.1109/RTAS.2013.6531080. URL: http://ieeexplore.ieee.org/document/6531080/ (visited on 04/21/2024).

[Rue13] Guillem Rueda Cebollero. "Learning Cache Replacement Policies using Register Automata". Master's Thesis. 2013. URL: https://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-212677 (visited on 07/28/2024).

[Vil+20] Pepe Vila et al. *CacheQuery: Learning Replacement Policies from Hardware Caches*. May 26, 2020. arXiv: 1912.09770[cs]. URL: http://arxiv.org/abs/1912.09770 (visited on 01/25/2024).

[Arm24b] Arm Limited. "C5.3 A64 System instructions for cache maintenance". In: *Arm® Architecture Reference Manual for A-profile architecture*. Mar. 20, 2024, pp. 994–1062. URL: https://developer.arm.com/documentation/ddi0487/latest/.

[OST05] Dag Arne Osvik, Adi Shamir, and Eran Tromer. *Cache attacks and Countermeasures: the Case of AES*. Publication info: Published elsewhere. Unknown where it was published. 2005. URL: https://eprint.iacr.org/2005/271 (visited on 07/08/2024).

[VKM18] Pepe Vila, Boris Köpf, and José Francisco Morales. *Theory and Practice of Finding Eviction Sets*. Dec. 10, 2018. arXiv: 1810.01497[cs]. URL: http://arxiv.org/abs/1810.01497 (visited on 01/23/2024).

[TK19] Vitalii Tymchyshyn and Andrii Khlevniuk. *Beginner's guide to mapping simplexes affinely*. Apr. 14, 2019. DOI: 10.13140/RG.2.2.13787.41762.

[Kro07] Greg Kroah-Hartman. "Linux Kernel in a Nutshell: A Desktop Quick Reference". In: *Linux Kernel in a Nutshell: A Desktop Quick Reference*. Illustrated Edition. Beijing, Köln: O'Reilly Media, Jan. 23, 2007, p. 97. ISBN: 978-0-596-10079-7.

[Jon13] Jonathan Corbet. *(Nearly) full tickless operation in 3.10*. May 8, 2013. URL: https://lwn.net/Articles/549580/ (visited on 04/29/2024).

[Mar20] Marta Rybczyńska. *A full task-isolation mode for the kernel*. Apr. 6, 2020. URL: https://lwn.net/Articles/816298/ (visited on 04/29/2024).

[Gor04] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Bruce Perens' Open Source series. Upper Saddle River, NJ: Prentice Hall, 2004. 727 pp. ISBN: 978-0-13-145348-7.

[Arm23a] Arm Limited. *Arm DynamIQ Shared Unit Technical Reference Manual*. June 30, 2023. URL: https://developer.arm.com/documentation/100453/0401 (visited on 09/22/2024).

[Ber04] Daniel J Bernstein. *Cache-timing attacks on AES*. Nov. 11, 2004. URL: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf (visited on 09/23/2024).

[Per05] Colin Percival. *Cache missing for fun and profit*. 2005.

[Yan+19] Mengjia Yan et al. "Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019 IEEE Symposium on Security and Privacy (SP). San Francisco, CA, USA: IEEE, May 2019, pp. 888–904. ISBN: 978-1-5386-6660-9. DOI: 10.1109/SP.2019.00004. URL: https://ieeexplore.ieee.org/document/8835325/ (visited on 01/23/2024).

[Arm23b] Arm Limited. *L1 data side memory system*. Arm Cortex-A76 Core Technical Reference Manual. June 30, 2023. URL: https://developer.arm.com/documentation/100798/0401/L1-memory-system/About-the-L1-memory-system/L1-data-side-memory-system (visited on 09/22/2024).

[AR24] Andreas Abel and Jan Reineke. *uops.info*. 2024. URL: https://uops.info/cache.html (visited on 01/23/2024).

[Arm23c] Arm Limited. *About the L2 memory system*. Arm Cortex-A76 Core Technical Reference Manual. June 30, 2023. URL: https://developer.arm.com/documentation/100798/0401/L2-memory-system/About-the-L2-memory-system (visited on 09/22/2024).

[Kim+14] Yoongu Kim et al. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors". In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA). Minneapolis, MN, USA: IEEE, June 2014, pp. 361–372. ISBN: 978-1-4799-4394-4 978-1-4799-4396-8. DOI: 10.1109/ISCA.2014.6853210. URL: http://ieeexplore.ieee.org/document/6853210/ (visited on 07/08/2024).

[Bug+96] Edouard Bugnion et al. "Compiler-directed page coloring for multiprocessors". In: *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*. ASPLOS96: 7th Conference on Architectural Support of Programming Languages & Operating Systems. Cambridge Massachusetts USA: ACM, Sept. 1996, pp. 244–255. ISBN: 978-0-89791-767-4. DOI: 10.1145/237090.237195. URL: https://dl.acm.org/doi/10.1145/237090.237195 (visited on 08/31/2024).

[Tor03] Linus Torvalds. *Re: Page Colouring (was: 2.6.0 Huge pages not working as expected)*. E-mail. 2003. URL: https://yarchive.net/comp/linux/cache_coloring.html (visited on 08/31/2024).