SCIENCE ▪ PASSION ▪ TECHNOLOGY

Aaron Giner

MultiCache: Evaluating a Secure and Fast L1 Cache Design

BACHELOR'S THESIS

Bachelor's degree programme: Computer Science

Supervisors

Martin Schwarzl

Institute of Applied Information Processing and Communications
Graz University of Technology

Graz, October 2022

Abstract

Cache side-channels allow for a malicious agent to reconstruct and retrieve secrets like cryptographic keys from other programs, even when the victim program does not have any software bugs. This works by inferring the behavior of the victim process through observing memory access latency.

In this thesis, we evaluate the feasibility and performance limitations of `MultiCache`, a new L1 cache design. `MultiCache` aims to eliminate attacks on the L1 cache by introducing multiple process-exclusive caches, meaning that each of these L1 caches can only be occupied and used by one process at a time. This way, an attacker can no longer infer information from cache hits and cache misses on the L1 cache, as processes no longer share cache lines.

We test this design with typical data center and private server applications. Our findings indicate that `MultiCache` is feasible in certain contexts but heavily dependent on the condition that the number of heavy-load processes running on a core is limited. We find improvements in L1D cache miss ratio of up to 35% for benchmarks, where servers run in parallel on the hardware threads of the same core.

Contents

1. Introduction **5**

2. Background **7**

2.1. CPU Caches . . . 7

2.1.1. Designs . . . 7

2.1.2. Replacement Policies . . . 8

2.1.3. Cache Hierarchy . . . 9

2.2. Simultaneous Multithreading . . . 11

2.3. Cache Side-Channel Attacks . . . 12

3. Idea **14**

4. High-Level Overview **17**

4.1. Benchmarking . . . 17

4.2. Experiment Design . . . 18

5. Implementation **21**

5.1. System Setup . . . 21

5.1.1. Core Isolation . . . 21

5.2. Kernel Modifications . . . 22

5.2.1. Tracepoints . . . 22

5.2.2. Context Switch Information . . . 22

5.2.3. Cache Invalidation . . . 24

5.3. Data Analysis Pipeline . . . 24

6. Evaluation **26**

6.1. Baselines . . . 26

6.2. Individual Evaluations . . . 27

6.2.1. Problem . . . 32

6.3. Combined Evaluations . . . 33

7. Conclusion **36**

Bibliography **37**

A. Appendix **41**

1. Introduction

Caches constitute an essential building block of virtually all modern computers. They are compact, high-speed memory storage that sits in between CPU cores and RAM and is used to decrease memory access time. In recent years, an increasing amount of research has focused on the security of CPU caches, as they have proven to be susceptible to a wide variety of attacks, most of them regarding the abuse of side-channels to derive secret information from metadata.

These attacks include, e.g., the recovery of cryptographic keys [11] and recording inter-keystroke timings [45]. Most prominently, side-channels have been exploited in `Meltdown` [25] and `Spectre` [22], where cache covert channels are used to communicate secret information to the attacker. Attacks such as `ZombieLoad` [35] or `Foreshadow` [42, 46, 3] also have the L1 cache as a target. Techniques like `Flush+Reload` [51, 10] or `Prime+Probe` [26] even allow for cross-core attacks.

Preventing these attacks has been an open research topic for many years. Some proposals deal with side-channel leakage on a hardware level. One example of this is `ScatterCache` [47]. This design essentially eliminates eviction-based attacks for the L3 cache, but it is infeasible for the L1 cache due to its high complexity and performance overhead.

Contributions. In this thesis, we present the `MultiCache` design idea and evaluate whether the design is feasible in practice and can even improve upon the performance of current state-of-the-art cache designs.

We provide statistical analyses of the design's performance in a cloud computing and data center context. These include the application of the design on dedicated, single-service servers and servers running different combined workloads, like privately hosted websites. For this, we record cache activity and process information during the system's context switches and simulate the behavior of our design in these different situations.

The `MultiCache` design builds on the idea of process-exclusive memory. The first level of the cache hierarchy is expanded, using multiple caches per core, which can only be occupied by one process at a time. When another process requires a cache, the entire cache is invalidated. This eliminates the possibility of timing-based side-channel attacks on the L1 cache because processes can no longer draw conclusions about the victim process from measuring the cache memory access latency as processes do not share cache lines in L1 caches.

Outline. The paper is organized as follows. In Chapter 2, we lay out the necessary background information on CPU caches, simultaneous multithreading and cache side-channel attacks. In Chapter 3, we describe the `MultiCache` design idea in detail. In Chapter 4, we give a high-level overview of our experiment design. In Chapter 5, we describe our implementation and in Chapter 6, we showcase the results of our performance evaluations. We conclude with Chapter 7.

2. Background

In this chapter, we provide background on CPU caches, cache side-channel attacks, and simultaneous multithreading.

2.1. CPU Caches

CPU caches are hardware caches used to decrease memory access latency. First presented by Maurice V. Wilkes in the 1965 paper `Slave Memories and Dynamic Storage Allocation` [49] they have since become an integral part of every modern computer architecture. Over time, many architecture optimizations have been developed, including, e.g., a hierarchy of caches and splitting the L1 cache into data- and instruction caches.

Fetching data from main memory can take upwards of five times longer than fetching data from the cache. However, caches have a significantly lower memory capacity, and the memory used for caches (*SRAM*) is more expensive than what is used for the main memory (*DRAM*).

When a physical address is accessed and the corresponding data is not yet cached, the data is fetched from main memory and placed at some location in the cache. Subsequent accesses to the same address are now extremely fast because the data can be retrieved from the cache instead. When an address is accessed, and this address is currently cached, we speak of a *cache hit*. Otherwise, we speak of a *cache miss*.

2.1.1. Designs

In essence, caches are simple arrays with 2^i entries, where each entry consists of an *address tag*, a *cache line* - which holds 2^b bytes of data (typically 64 bytes) - and some additional meta information. For each entry, there exists a set of physical addresses that map to it. The address is divided into sections, as shown in Figure 2.1. The middle i bits of an address are used to determine which entry in the cache the address maps to, b bits are used to retrieve data from the cache line at byte granularity, and the remaining bits are used as the tag. As multiple addresses can map to the same entry, the tag is used to identify the currently cached address.

Directly Mapped Cache. In a *Directly Mapped Cache*, multiple addresses compete for one table entry. Since caches have a low memory capacity, there is a high number of *congruent addresses* that map to the same entry in the table. A program with alternating accesses to two congruent addresses will have a cache miss ratio of 100% at this location in the cache, meaning the data has to be fetched from main memory and the currently cached data has to be evicted.

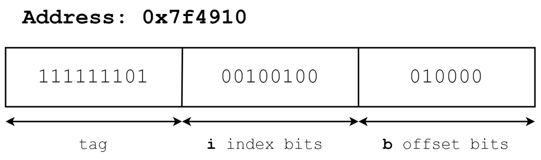


Figure 2.1.: Address sections

Fully Associative Cache. In contrast to the Directly Mapped Cache, the *Fully Associative Cache* only has one entry. But this entry has *n ways*, where *n* is the total number of cache lines in the cache. Every address can be placed in any one of these ways, allowing for the use of a *replacement policy* to decide which cache lines should be evicted, leading to a higher hit ratio. But for every access, potentially, the entire cache has to be searched and the tags compared, generating significant overhead in both power consumption and runtime.

Set Associative Cache. This is why modern caches are *set associative.* This design decision represents a trade-off between the two previously described designs. There are 2^i entries, but for each entry, there are *n* ways, and for each entry, there exists a set of congruent addresses that map to it. They do not directly map to one cache line but instead a set of cache lines that form a so-called *cache set.* And like for the Fully Associative Cache, this design allows for the use of replacement algorithms. Current Intel processors typically have between four and sixteen ways.

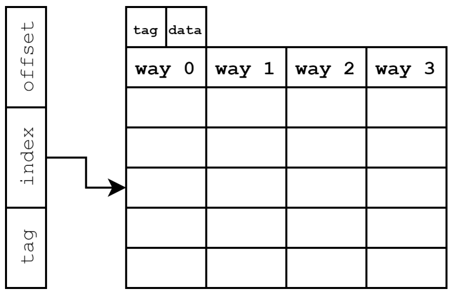


Figure 2.2.: 4-way set associative cache

2.1.2. Replacement Policies

When each cache line in a cache set is occupied and an address that is not currently cached is accessed, one of the cache lines must be evicted. Selecting the optimal cache line for eviction is not a trivial task. Optimally, a cache line that will least frequently be used in the future should be evicted. How this eviction candidate is selected depends on the replacement algorithm, but they all follow the *principle of temporal locality*, which assumes that a recently accessed address is more likely to be reaccessed in the near future.

Optimizing user programs in order to exploit the cache replacement policy can therefore lead to a significant performance increase.

Least Recently Used. One of the most basic and widely used replacement policy concepts is *LRU* [6]. In this policy, the cache line that was least recently accessed is evicted. For this, expensive bookkeeping structures are necessary.

For highly associative caches, the hardware cost of keeping track of the necessary information is extremely high, which is why modern caches implement modified versions of LRU, like *Pseudo-LRU* [21], which only requires $n_{\text{ways}} - 1$ additional bits of meta information per cache set. Unlike LRU, PLRU uses approximative measures to select the best eviction candidate. It does so by implementing a binary tree of information bits. Each bit divides a section of the cache set into two halves, and the bit's value indicates which half was previously used. E.g., the top-level bit splits the cache set into two halves, and its value indicates in which half the most recent access happened. The bits in the lower levels of the binary tree work the same way. When a cache line must be evicted, all the bits along the path are inverted to find the eviction candidate.

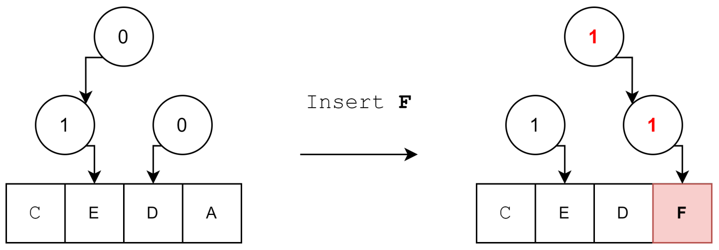


Figure 2.3.: PLRU replacement

2.1.3. Cache Hierarchy

To improve performance even further, modern processors have a hierarchy of caches. In recent Intel processors, this hierarchy consists of three levels.

The Level 1 (*L1*) cache is closest to the processing units. It is the smallest one, typically divided into data (*L1D*) and instruction (*L1I*) caches, each in the range of 32 to 64 KiB, respectively. It also has the lowest access time at four cycles. The Level 2 (*L2*) cache typically has a size of 256 KiB and an access time of about 12 cycles. The Last Level cache (*LLC*) has a capacity of multiple MiB and an access time of about 30 cycles [15].

On a multi-core system, each core has its own private L1 and L2 cache, while the LLC is shared between cores and usually logically divided into *slices*.

Inclusion Policy. The inclusion policy defines some essential properties about the content of each of the different caches in the cache hierarchy. There are three main types of inclusivity: *inclusive*, *exclusive* and *non-inclusive* [4].

An inclusive cache always contains the entire data of the upper-level caches (L1 and L2). This design simplifies *cache coherence* across the caches of the CPU but comes at

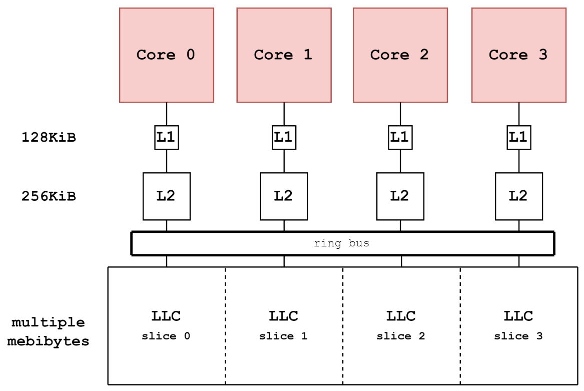


Figure 2.4.: Cache Hirarchy

the cost of performance and also needs more space for the duplicate data. The LLC's inclusiveness is essential for cross-core side-channel attacks like `Prime+Probe` to work.

Exclusive caches never contain any data from upper-level caches. The advantage of this design is that there is no duplicate data within a core's private caches. Similar to the usage of higher associativity in caches, this also leads to a higher cache hit ratio and therefore improves performance. It does come at the cost of higher power consumption due to more complex cache coherence protocols and because more data has to be moved in general.

Non-inclusive caches are a trade-off between the two aforementioned policies. Data can be stored in multiple caches of the hierarchy, but it can also just exist in one of them. When an address is initially accessed, and the data is fetched from RAM, it is placed in all lower levels. In contrast to inclusive caches, the data must not always be present in the LLC. An eviction on the LLC does not lead to an eviction on the higher levels. This increases the best-case effective size to that of exclusive caches, i.e., the summed size of all caches in the hierarchy. Again, the main disadvantage of this policy is its higher complexity when trying to maintain cache coherence.

The LLC in modern Intel processors is typically inclusive, while the L1 and L2 caches are non-inclusive [15].

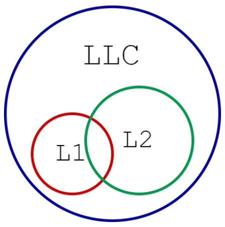


Figure 2.5.: Intel's inclusion policy

2.2. Simultaneous Multithreading

Simultaneous Multithreading, first developed by Susan Eggers and Hank Levy in 1995 [40], is a way of utilizing the unused resource of a processor core by dividing the physical core into a set of *logical cores* that can perform work in parallel. This idea stands in contrast with Multithreading in software, which does not represent real parallelism, as there is only ever one software thread running on a CPU at a time.

More specifically, this works by taking advantage of leftover resources in the pipeline architecture of modern processors.

Pipelining. The task of executing a single instruction consists of multiple phases, and, depending on the instruction, executing a single instruction can take a long time. For example, when an instruction needs data from main memory, this can take upward of 10 nanoseconds. If every instruction has to wait until the previous instruction finishes, these expensive instructions cause heavy delays.

This is why modern processors implement `instruction pipelines`. Instead of executing an instruction in a single cycle, the execution is split into different stages. The following are the phases of a classic RISC Pipeline: Instruction Fetch (*IF*), Instruction Decode (*ID*), Execute (*E*), Memory Load/Store (*M*) and Register Write Back (*WB*).

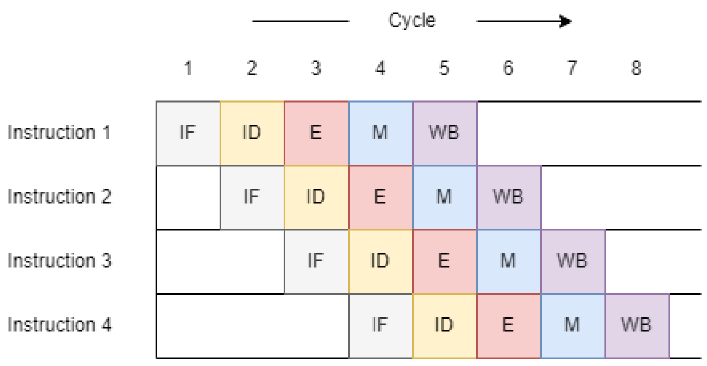


Figure 2.6.: Ideal Pipeline

Figure 2.6 shows an ideal pipeline - a pipeline where all pipeline stages are full at any time. However, in practice, due to stalling instructions, this is very often not the case. E.g., this happens because not all instructions need the same stages or because an instruction is dependent on another that has not yet provided the required result. There are several methods to more effectively use the instruction pipeline more effectively, like `Out-Of-Order Execution` or `Simultaneous Multithreading`.

Parallel Execution. Dividing physical cores into logical cores enable true parallelism, i.e., running multiple software threads in parallel. Each logical core maintains its own architectural state by having a separate set of hardware registers, but they share other resources, like the instruction pipeline of the physical core and, most importantly for this thesis, the private L1 and L2 caches of the physical core. This way, multiple hardware threads (depending on the number of threads per core) access the caches simultaneously, and each thread can subsequently affect the cache performance of the others by evicting cache lines that these threads may need again.

Simultaneous Multithreading does not come without its drawbacks and is not equivalent to adding more physical processor cores, but Intel claims that their implementation of this concept - Intel's `HyperThreading` technology - can lead to a performance increase of up to 30% in server applications [18].

2.3. Cache Side-Channel Attacks

Generally, a side-channel attack describes the process of indirectly obtaining secret information by observing some meta information. A famous real-world example of using meta-information to infer information is Clever Hans [34], a horse that was supposedly able to - among other things - perform numerical calculations. It would answer by tapping its hoof the correct number of times. In reality, the horse was trained to recognize the subtle reactions of the audience and would stop when tapping its hoof when it noticed these reactions. In this analogy, the audience's behavioral changes are the meta-information that Clever Hans uses to make an educated guess.

In the case of cache side-channel attacks, this meta-information is the difference in cache access time between a cache hit and a cache miss. An attacker can observe changes in access latency and derive secret information from it. Cache side-channel attacks have been exploited to break cryptographic algorithms and recover crypto-keys [11, 19, 20], key-logging [45], bypass kernel ASLR (Address Space Layout Randomization) [7, 9] and more. With one of the many potent techniques, this type of attack is even viable in "realistic cross-core scenarios." [47] Side-channel techniques also played a vital part in exploiting security vulnerabilities such as Meltdown [25] and Spectre [22].

Flush+Reload. `Flush+Reload` [51, 10] is a cache side-channel attack technique that relies on sharing pages between the attacker and victim process, e.g., through file-based deduplication (shared libraries, binaries) or some other sort of page deduplication. Using the `clflush` instruction, the attacking process can evict certain memory locations from the cache hierarchy. This way, if the victim process uses these locations, it will reload the flushed memory into the cache. The attacking process can then also reaccess this location and measure the access latency. Using a certain threshold (e.g., $\mathbf{t < 180\ cycles}$), the attacking process can deduce whether the address was accessed by the victim process between the `flush` and the `reload` of said address by measuring the reload access time. This information can then be used to deduce secret information by observing specific memory locations that indicate certain behavior. For example, a specific memory location might be accessed every time the `stdio` library prints the letter 'a'. The `Flush+Reload` attack is a variant of `Prime+Probe`.

Prime+Probe. The `Prime+Probe` [26] technique is a more general and powerful technique than `Flush+Reload` because it does not rely on sharing pages with the victim process. This attack consists of two phases: In the **Prime** phase, the attacker evicts cache sets using an eviction set, e.g., a set of memory locations that evict an entire cache set when accessed. After this, the attacking process waits for some time for the victim to perform some task. In the **Probe** phase, the attacking process again accesses all eviction sets and measures the time this takes for each set. If a cache line in one of the evicted cache sets has been accessed by the victim process, the total access latency of this cache set will be measurably higher. The attacker can infer secret information by repeating this process many times and recognizing memory access patterns. For example, an encryption algorithm may have a different cache access pattern depending on the secret key that is used to encrypt the data. If the encryption algorithm is known to the attacker, it is possible to recover the key.

Mitigation. These techniques work on all levels of the cache hierarchy. Due to the core-shared nature of the LLC cache, even cross-core attacks are possible. One of the proposed designs to mitigate cache side-channel attacks on the LLC cache is `ScatterCache` [47], in which the fixed link between addresses and cache sets is removed. Instead, addresses are linked to randomly generated cache sets, which makes finding eviction sets ”highly unlikely.” However, this design is highly complex and also introduces a performance overhead, which makes it infeasible for application on the L1 cache.

3. Idea

The `MultiCache` design we evaluate in this thesis shall prevent timing-based cache side-channel attacks on the L1 data cache. These attacks work because all processes running on the same core share the L1 and L2 caches in traditional processor designs. In this chapter, we give an overview of the design idea.

Motivation. The most straightforward way to prevent these attacks would be to invalidate the cache on each context switch, which would limit the power of the L1 cache to accesses in the current scheduling period. According to a 2019 paper by Bourgeat et al. [2], flushing the micro-architectural state on every context switch (TLB, L1 Caches, branch predictors) leads to an "average overhead in execution time is 5.4%". If the caches in the `MultiCache` design stay assigned to a process over an extended period, this should constitute a practical improvement.

Conceptual View. Instead of a single L1 data cache, the `MultiCache` design uses multiple L1 data caches per core and provides one for each process to use exclusively.

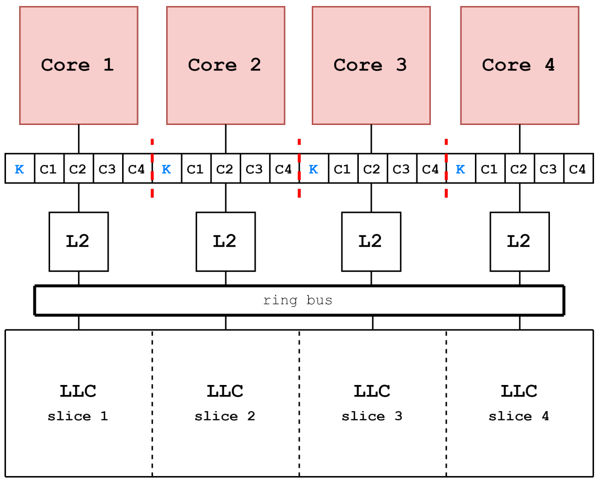
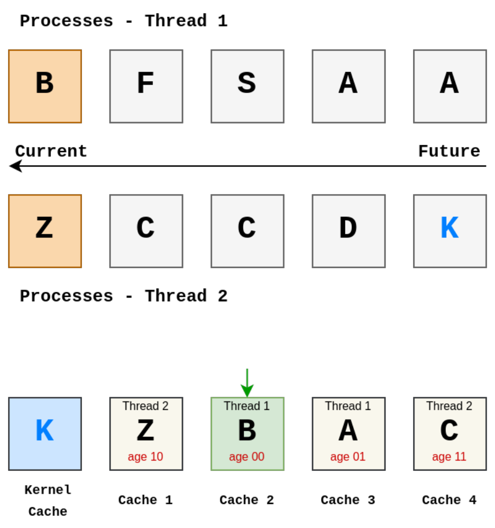


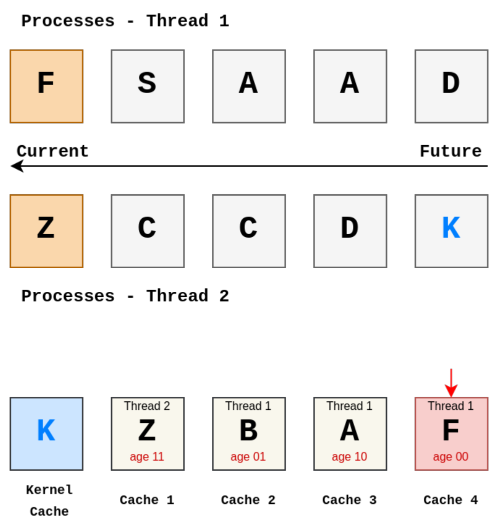
Figure 3.1.: MultiCache Design

Figure 3.1 shows a conceptual view of the resulting memory hierarchy. It has $CPC = 5$ L1 data caches per core and one L1 instruction cache (not shown in the graphic). One of these 5 caches is reserved for kernel threads.

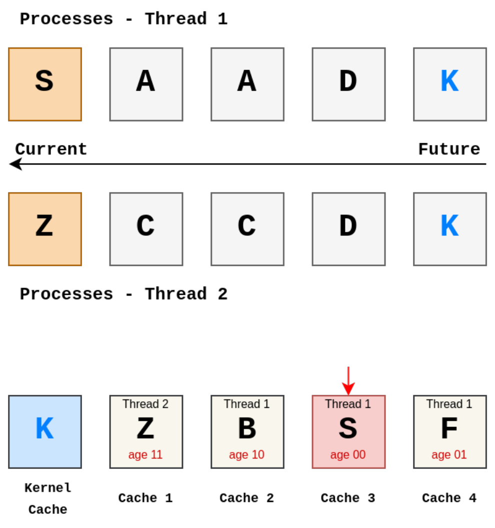
Integration. Upon a context switch - unless the new process already has a dedicated cache - one of these caches is assigned to the next process in hardware, and the contents of the chosen cache are invalidated. The cache is determined based on a modified version of the Least Recently Used replacement policy. Specifically one that takes Simultaneous Multithreading into account.



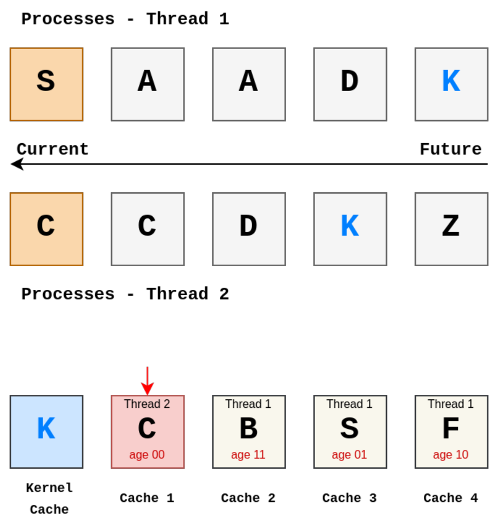
(a) Thread 1: B - Hit



(b) Thread 1: F - Miss



(c) Thread 1: S - Miss



(d) Thread 2: C - Miss

Figure 3.2.: MultiCache Replacement Policy

Figure 3.2 shows four representative scheduling steps on a setup with two-way SMT, i.e., two hardware threads per physical core. The letters represent threads of a specific process that are currently running or will run in the future.

In step 3.2a, CPU 1 schedules process B, but process B already has a cache, so no other cache has to be evicted. In step 3.2b, CPU 1 schedules process F. This time, one cache has to be evicted, and the cache that was least recently used is chosen, invalidated, and assigned to process F. In step 3.2c, Thread 1 schedules a thread of process B and needs a new cache once again. This time, Cache 1 was least recently replaced, but it is currently used by process Z, which is currently running on CPU 2 and should not be evicted because it is likely currently in use. Instead, the second oldest cache is invalidated. After that, CPU 2 schedules process C in step 3.2d, and now Cache 1 is used.

4. High-Level Overview

In this chapter, we give a high-level overview of the experiments we perform to analyze the feasibility of our design.

4.1. Benchmarking

This thesis aims to evaluate the performance of `MultiCache` based on typical data center workloads in a cloud environment. We use the same benchmarks as Virt-LM [13] and PalmsCloud [50], which are specifically designed for cloud environments. For all our workloads, we use 10, 16 or 32 simultaneous client connections per server (*CPS*) to make requests and to reach full CPU utilization. As we will further discuss later, the number of clients connected to the server can significantly impact the performance of our proposed design.

Application Server. Apache Tomcat [39] is a widely used, Java-based, open-source web and application server software. It implements specifications for Java Servlets, Java Server Pages, and other Java-based software components to allow for the execution of Java web applications. We use Tomcat version 9.
For our client-side benchmarking software, we use ApacheBench [37]. It is a widely used, generic benchmarking tool for HTTP servers that support HTTP/1.0 and HTTP/1.1.

Database Server. As our server software, we use MySQL [31]. MySQL is an open-source Relational Database Management System (DBMS) that is – according to a recent study by statista.com [36] – currently the third most used database system globally.
On the client side, we use sysbench [23], which is mainly used for database benchmarks but also implements various other features.

File Server. Samba [33] is a software package initially developed for Windows and later ported to Linux. It implements the Server Message Block (SMB) protocol, which enables transparent interoperability of Linux and Windows when sharing "files, directories, printers, serial ports, and other resources on the Network." [14]
We use DBENCH [1] to benchmark the Samba server using a predefined set of read/write/create requests. DBENCH is a popular NFS server benchmarking tool.

Mail Server. For our mail server, we use Postfix [32]. Initially developed at IBM Research, Postfix is a lightweight, easy-to-use, secure alternative still widely used today. On the client side, we use smtp-source [43], a simple command line tool also developed at IBM research that can generate Simple Mail Transfer Protocol (SMTP) mail requests with a specified payload length and repeatedly send them to our server.

Stream Server. FFserver is a broadcasting software that is part of the FFmpeg [5] software package. It supports multiple input- as well as output sources. In our setup, the input files are stored locally and not received through an input stream from another location.
As no benchmarking software is available for ffserver that we are aware of, we use openRTSP [27] to start multiple threads that open streaming connections to the broadcasting server. openRTSP is a command-line program that can be used to open single receive connections to streaming servers requesting a specific file. These threads request one of three files we prepared for this purpose: two video files and one audio file.

Web Server. For the web server, we use the Apache HTTP Server [38]. It is open-source software and one of the most used HTTP servers in the world.
On the client side, we again use ApacheBench.

4.2. Experiment Design

For our evaluation, we are primarily interested in L1D cache activity. Specifically, the percentage of L1 data cache accesses that miss in relation to those that hit (L1D miss ratio). In designing our experiments, we make two fundamental assumptions:

Assumption 1. From our measurements (discussed in Section 6.1), we know that invalidating the L1D cache on each context switch results in a higher miss ratio than if it is not invalidated when certain cache lines stay in the cache are reused later on. A cache invalidation on each context switch equals a 100% eviction ratio (*ER*). We assume that the increase in miss ratios from an ER of 0% to an ER of 100% is linear.

Assumption 2. We further assume that the miss ratio of a process with a dedicated cache is roughly equivalent to the miss ratio of this process when it is running in isolation on a traditional system with a process-shared L1D cache because we expect only minimal interference from kernel threads on an isolated core.

Baselines. For each of the servers we use in our benchmarks, we measure the miss ratios for eviction ratios of $ER = [0\%, 100\%]$. These pairs of values make up the miss ratio baselines for our evaluation.

Workloads. In this thesis, we test several selected workload combinations that we think are a representative sample for our evaluation (Table 4.1).
We also measure the miss ratios for each of these workloads and simulate the `MultiCache` design to see how often the caches assigned to a server's processes are evicted. For the

Designation		CPS
Basic Workloads		
A	App Server	32
D	Database Server	32
F	File Server	32
M	Mail Server	32
S	Stream Server	32
W	Web Server	32
Single-Core Workloads		
A_D	App + Database Server	16
A_F	App + File Server	16
D_M	Database + Mail Server	16
D_W	Database + Web Server	16
A_D	App + Database Server	16
F_W	File + Web Server	16
S_W	Stream + Web Server	16
A_D_W	App + Database + Web Server	10
D_F_W	Database + File + Web Server	10
D_S_W	Database + Stream + Web Server	10
SMT Workloads		
A_D_SMT	App + Database Server	32
A_F_SMT	App + File Server	32
A_M_SMT	App + Mail Server	32
A_S_SMT	App + Stream Server	32
A_W_SMT	App + Web Server	32
D_F_SMT	Database + File Server	32
D_M_SMT	Database + Mail Server	32
D_S_SMT	Database + Stream Server	32
D_W_SMT	Database + Web Server	32
F_M_SMT	File + Mail Server	32
F_S_SMT	File + Stream Server	32
F_W_SMT	File + Web Server	32
M_S_SMT	Mail + Stream Server	32
M_W_SMT	Mail + Web Server	32
S_W_SMT	Stream + Web Server	32

Table 4.1.: Workloads

single-core workloads, the servers run on the same core with HyperThreading disabled, while for the SMT workloads, the two servers in each combination run on two hardware threads of the same core.

In the `MultiCache` design, each process would get its own cache, and according to Assumption 1, the miss ratio in an implementation of `MultiCache` would be a value on the line segment between the 0% and the 100% baselines for this process. Figure 4.1 gives a conceptual overview of our experiment design. It shows the two baseline values at 0% and 100% eviction. The Y-axis represents the measured miss ratio, and the X-axis the simulated eviction ratio. If the measured miss ratio (X-value) is above the line segment connecting the baselines, we expect a performance gain using `MultiCache` because the expected miss ratio in a dedicated cache would be on this line segment. On the other hand, if the miss ratio is below, we expect a decrease in performance.

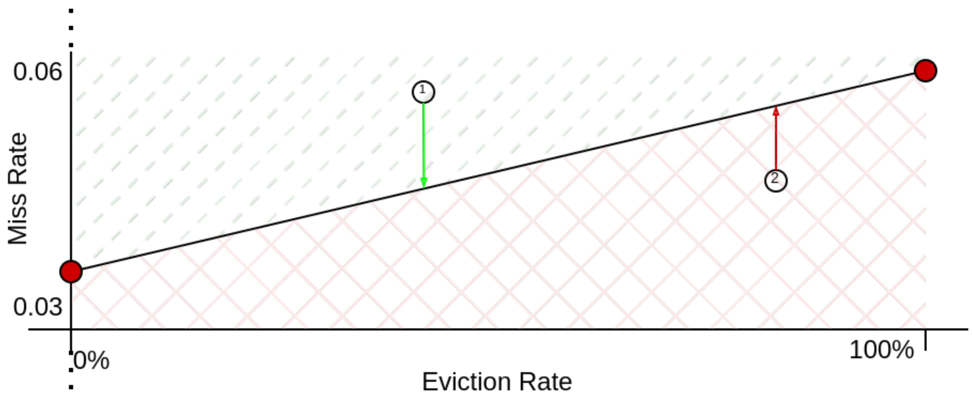


Figure 4.1.: Conceptual Experiment Overview

5. Implementation

In this chapter, we describe our implementation. In Section 5.1, we talk about the system we use and the modifications we make to it to run our benchmarks. In Section 5.2, we describe how we modify the Linux kernel to trace scheduling information. Finally, we introduce our data analysis pipeline in Section 5.3.

5.1. System Setup

We use Ubuntu Server 20.04 (Linux kernel v5.13) for all our experiments, running on an Intel Core i7-7700k processor. Due to resource constraints, the same machine acts as both the client and the server. For this to work and also to record the miss ratio baselines, we use core isolation.

5.1.1. Core Isolation

To evaluate `MultiCache`, we are interested in the closest approximation of the true miss ratio a process would have at 0% as well as 100% eviction in our design, so we use core isolation as a means of minimizing the interference of other processes running on the system, including the tools we use for the server benchmarks, which are strictly separated from the server software this way.

isolcpus. Linux allows for the isolation of cores using the `isolcpus` kernel boot command option and choosing one or more CPUs to isolate. The Linux scheduler will subsequently not run non-essential processes on the isolated cores. We also use the `managed_irq` parameter to isolate the specified CPUs from managed interrupts.

CPU Affinity. Core isolation is useless if we cannot let the servers run on the isolated cores. For this, Linux provides the `sched_setaffinity` system call to change a process affinity mask. This way, it is still possible to let specific software run on isolated cores. We do not use this system call directly but instead provide the desired affinity to the Linux service configuration files for each of the servers we use. In the case of the Mail Server, we use the `taskset` command-line tool to manually set the affinity mask of some of the Postfix processes, as this is not done automatically.

5.2. Kernel Modifications

This section describes how we modify the Linux Kernel to obtain the most accurate data possible for our evaluations.

5.2.1. Tracepoints

Tracepoints [41] are a feature of the Linux kernel that allows us to quickly trace arbitrary information about what is currently happening in the kernel at some previously specified point. Each tracepoint has a function connected to it. They act as hooks that we place at points of interest in the kernel code. The trace data is then written to a buffer in the main memory.
Tracepoints allow control of several important configuration options at runtime:

Each of the CPUs in our system has a dedicated buffer trace buffer. We can control the size of these individual buffers by writing a numerical value to /sys/kernel/debug/tracing/per_cpu/cpu**X**/buffer_size_kb. This way, we can either increase or decrease the buffer size of core **X**. This provides the necessary flexibility to keep RAM usage as low as possible.

The Linux kernel already contains many tracepoints, but we do not want them to be active most of the time. We can enable tracepoints by writing their name to /sys/kernel/debug/tracing/set_event and disable them by prepending the name with an exclamation mark ('!') character. This way, we can precisely control when the trace starts and ends.

For our evaluation, we are only interested in tracing information from the cores that our servers are running on. Trace files can get very large, so it is advantageous only to save the traces of the cores we are interested in. We can do this by writing a hexadecimal bitmask to /sys/kernel/debug/tracing/tracing_cpumask. Each bit represents one logical core. The least significant bit represents core 0, the next core 1, and so on.

The contents of the individual trace buffers are aggregated in /sys/kernel/debug/tracing/trace, which we copy to disk after tracing is finished.

5.2.2. Context Switch Information

To perform our analysis, we are interested in specific information whenever the kernel performs a switch from one process to the next. Specifically, we need data for the tasks described below.

MultiCache Simulation. To simulate how our `MultiCache` design would assign caches to processes, we need more detailed information about the processes running on the cores of interest. For this, we use a custom tracepoint - Sched-Switch-Verbose (SSV) - that logs the logical core number, a binary field representing either a switch from kernel-thread $\leftrightarrow$ user-thread, the Task Group IDs (*TGID*, commonly referred to as Process ID) of the current/next process, the Process IDs (*ID*, commonly called Thread ID) of the current/next threads, and lastly the names of the current/next threads.

Performance Counter Tracing. To obtain accurate data for our experiments, we use Intel's built-in hardware performance counters to log current values from performance counter events at each context switch. To achieve this, we add another tracepoint - L1-Events (*L1E*), which records the TGID, PID, and name of the next thread, as well as the current value of the performance counter registers. For this thesis, we use the `MEM_LOAD_RETIRED.L1_HIT`, `MEM_LOAD_RETIRED.L1_MISS` and `INST_RETIRED.ANY_P` performance monitoring events. The `MEM_LOAD_RETIRED.L1_HIT` event counts the number of L1D hits from retired load instructions (instructions for which the result was made architecturally visible). The `MEM_LOAD_RETIRED.L1_MISS` event counts L1D cache misses from retired load instructions. The `INST_RETIRED.ANY_P` event counts all retired instructions.

We use the two former instructions to calculate the `load miss ratio` for the L1D cache:

$$\texttt{LMR = L1\_MISS / (L1\_HIT + L1\_MISS)} \tag{5.1}$$

Intel's performance counters can be used by writing to the `IA32_PERFEVTSELx` [16] MSRs using the dedicated `wrmsr` and `rdmsr` instructions. In the Linux kernel we use the already implemented wrapper functions `wrmsrl` and `rdmsrl`, and we use the `wrmsr/rdmsr` command line tools to write to these registers in user-mode. To read the values of the performance counters, Intel provides the `rdpmc` (Read Performance-Monitoring Counters) instruction, for which we use the `rdpmcl` wrapper in the kernel. Our system supports four of these counters per hardware thread and eight counters per core when `HyperThreading` is disabled. Figure 5.1 shows the layout of these registers and the bits we use in this thesis. The `Unit Mask` and `Event Select` fields are used to select the events to be counted. The `USER` and `OS` bits determine whether the selected event shall be counted in user- and/or kernel-mode, respectively. We count both `OS` and `USER` events for our evaluation. Setting the `ENABLE` bit enables counting of the specified event. There are other bits and bit fields that we do not use and are not represented.

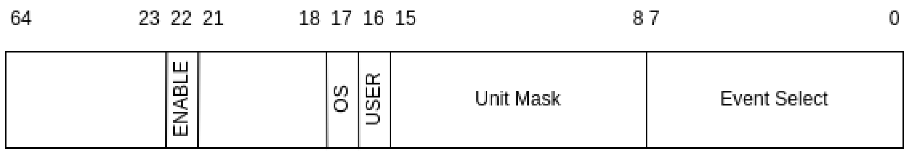


Figure 5.1.: `IA32_PERFEVTSELx` MSR layout

LMR Limitation. The metric we use does not include L1D line replacements coming from load instructions that did not retire, e.g., lines replaced due to speculative execution, but the speculative path was then discarded.

Store Miss Ratio. In this thesis, we only consider load misses in our evaluation, as Intel does not provide the L1D equivalent of the `L1_MISS` and `L1_HIT` events for store misses.

5.2.3. Cache Invalidation

To obtain the theoretical miss ratio upper bound for a process, we customize the kernel so that the L1D cache is invalidated upon each context switch. Forced cache evictions using memory accesses are slow and come at the risk of evicting cache lines from higher levels, introducing additional performance overhead. The `IA32_FLUSH_CMD` [17] model-specific register (*MSR*) provides a more fine-grained option. Writing to its `L1D_FLUSH` (0^{th}) bit triggers a writeback and invalidation of the L1 data cache.

This mechanism was introduced as a Microcode update in response to the `L1 Terminal Fault` (*L1TF*) vulnerabilities [42, 46, 3] that affected several modern Intel processors. L1TF attacks work by exploiting speculative execution paths and out-of-order execution. In these vulnerabilities, speculative execution ignores the present bit in the page table entries of virtual addresses that are accessed when the data is already in the L1 cache. Eventually, these operations raise page faults, and the results of this execution path are discarded, but in the meantime, the data may be provided to other speculative instructions. This behavior can be exploited in side-channel attacks. Because speculative execution "bypasses the extended page table (*EPT*) protection mechanism" [24], this vulnerability also enables attacks from inside virtual machines. The `L1_FLUSH` mechanism is used by the hypervisor to invalidate the L1 cache upon `VMENTER`.

Figure 5.2 shows the L1D flush time when filling different numbers of cache sets in the L1D cache. We observe that the flush time increases approximately linearly with the number of cache sets used.

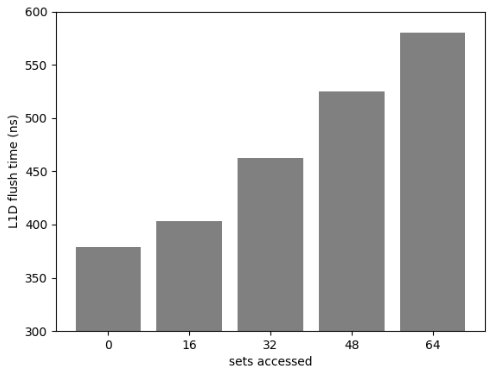


Figure 5.2.: L1D Flush Time

5.3. Data Analysis Pipeline

For our analysis, we use Python and Bash scripts to simulate the `MultiCache` design under different conditions and to parse and aggregate the data gathered from the performance counter events.

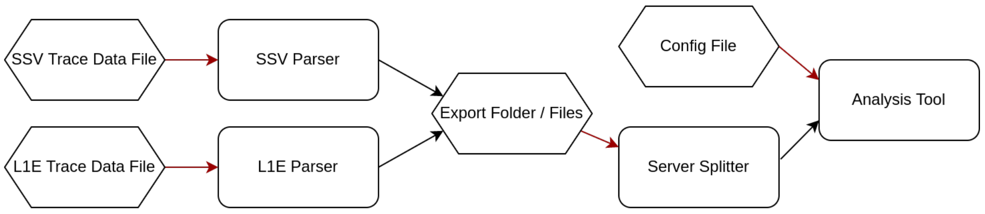


Figure 5.3.: Data-Analysis Pipeline

Figure 5.3 shows the path from collecting the tracing data to analyzing it for our evaluation. After tracing is finished, we copy the data from the aggregate buffer at /sys/kernel/debug/tracing/trace into a persistent, UTF-8 encoded file. We then either feed it to the SSV or L1E Parser, depending on the used tracepoint.

SSV Parser. The SSV Parser reads the collected SSV traces and simulates the `MultiCache` design. It implements different parsing options, most notably `Caches Per Core`, `Logical Cores` and `Physical Cores`. It then performs the simulations according to the selected options as described in Chapter 3.

L1E Parser. The L1E Parser reads the L1E traces and aggregates the collected performance counter information at core-, process-, and thread-granularity. The script exports the compiled values using the corresponding event name as the key. The events present in the trace are provided to the tool using the command line option `--events`.

Export. When running either of the Parser scripts using the `--export` command line option, the program exports the collected data in `JSON`-file format.

Splitter. The splitting tool reads these `JSON` exports and separates them into different files, one for each of the server software we use (if present). These files then only contain information about processes and threads of that software.

Analysis. Finally, we use our analysis tool to analyze the data and create plots automatically. We use `JSON` configuration files to tell the tool which data to use, which analysis to perform, and which plots to generate.

6. Evaluation

In this chapter, we evaluate the `MultiCache` design based on the data we gathered from running the different combinations of benchmarks on our system.

We denote the number of unique runs for each test with $\mathtt{N_{LMR}}$ for the load miss ratios, $\mathtt{N_{ER}}$ for eviction ratios and we denote the time with `S`. We choose the trace time based on observations we make about the variance in the results. For the single-server workloads we notice that a trace time of `S`=5s is enough, while it is not for combined workloads, which show very large variations in the results. We chose a uniform trace time of `S`=30s for all combined workloads, because we observe that the results are stable and increasing the trace time does not change the result. For all of our workloads, we perform $\mathtt{N_{LMR}}$=100, $\mathtt{N_{ER}}$=50 runs. For our analysis, we always reject outliers with a distance of 2 standard deviations from the mean.

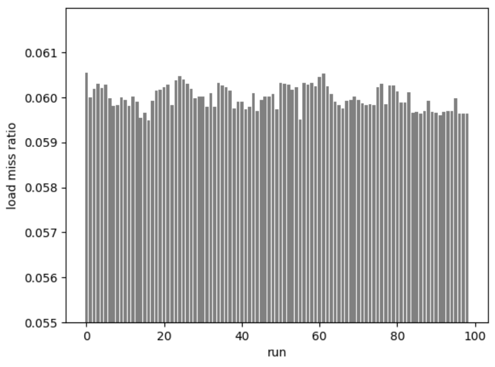
6.1. Baselines

If every process has its own exclusive cache, no other process can interfere with it. Therefore, the miss ratio of a process is dependent on how often the dedicated cache is invalidated and given to another process. This means that a process reaches its upper bound miss ratio when it receives a fresh cache 100% of the time, and when its cache is never evicted, this constitutes the lower bound miss ratio. These two cases make up the baselines in our evaluation. To get the baselines, we perform benchmarks with each of the six servers running exclusively, once without invalidating the cache and once with invalidating the cache using the `IA32_FLUSH_CMD` MSR.

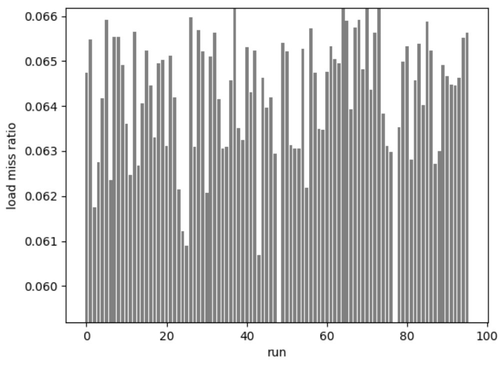
Fluctuations. After each run, we restart the server and benchmark to capture the apparent fluctuation in miss ratios accurately. Figure 6.1a shows that the fluctuations in miss ratios are insignificant if the server is not restarted in between runs, while they are if it is, as we see in Figure 6.1b.

In Table 6.1, we list the above baselines using three central tendencies: arithmetic mean, median and geometric mean. For later evaluations, we work with the arithmetic mean.

We observe that the miss ratios when the cache is invalidated upon each context switch are always higher than when it is not. To determine whether this is true with a significance level of $SL > 95\%$, we perform hypothesis tests using the alternative hypothesis that the distributions of CC datasets are likely larger than those of the normal datasets. For this, we use the `Wilcoxon-Mann-Whitney-Test` [48]. This test can be used when it is not certain that the observations are normally distributed and when we work



(a) No Restart: SEM=0.000025, IQR=0.000426



(b) Restart: SEM=0.000199, IQR=0.0022

Figure 6.1.: Miss Ratio Fluctuations for App Server Benchmarks

Server	Arithmetic Mean		Median		Geometric Mean		pvalue
	Normal	CC	Normal	CC	Normal	CC	
App	0.0641	0.0664	0.0645	0.0667	0.0641	0.0664	0.000
Database	0.0309	0.0317	0.0309	0.0317	0.0309	0.0317	0.000
File	0.0503	0.0543	0.0503	0.0543	0.0503	0.0543	0.000
Mail	0.0445	0.0468	0.0445	0.0468	0.0445	0.0468	0.000
Stream	0.0585	0.0654	0.0584	0.0655	0.0585	0.0654	0.000
Web	0.0653	0.0798	0.0653	0.0798	0.0653	0.0798	0.000

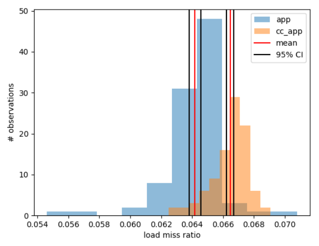
Table 6.1.: Load Miss Ratio Baselines

with unpaired (independent) samples. The `pvalue` column contains the `pvalue` of each of the tests. If the `pvalue` is smaller than $\alpha = 0.05$, the result is statistically significant. We can see that this is true in all cases.

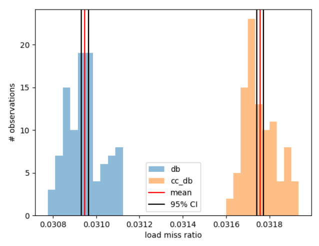
In Figure 6.2, we visualize the collected samples in a histogram, together with their means and confidence intervals. We observe that the App Server is the only server that even has an overlap between the two samples.

6.2. Individual Evaluations

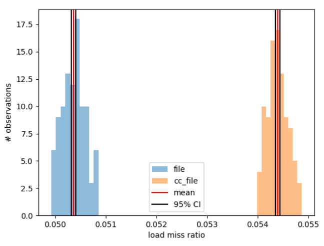
Now that we have the baselines, we collect the LMR data and simulate the ER for each workload. In this section, we first look at the expected performance differences for each of the individual servers in the workloads we selected (Table 4.1) and then aggregate the data to evaluate the performance of `MultiCache` for each of these workloads. In all of these, we simulate `MultiCache` using $CPC = 5$ caches per core.



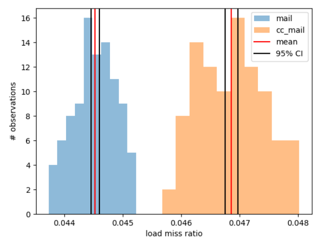
(a) App Server: Normal vs. CC



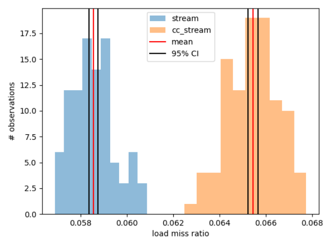
(b) Database Server: Normal vs. CC



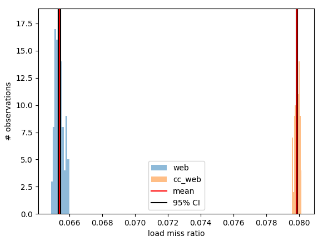
(c) File Server: Normal vs. CC



(d) Mail Server: Normal vs. CC



(e) Stream Server: Normal vs. CC



(f) Web Server: Normal vs. CC

Figure 6.2.: Load Miss Ratio Observations Histogram

In the tables below, the columns `Mean LMR` and `Mean ER` are calculated by using only the samples that correspond to the specified server. The column `Change` represents the relative change in miss ratio from the real value to the expected value in the `MultiCache` design. A negative change is desired. The `pvalue` column represents the likelihood that the data follows the Null-Hypothesis. For this, we use the alternative hypothesis that the distribution of an expected sample taken using the `MultiCache` design (= Shifted Baseline Sample (SPS)) is significantly *less* than the distribution of the measured sample for a specific workload. We define SPS in Equation (6.1).

$$SPS_S = B_{LOW} + (ER_W * (mean(B_{HIGH}) - mean(B_{LOW})) \tag{6.1}$$

B_{LOW} is the lower bound baseline sample for corresponding server S, and B_{HIGH} is the upper bound sample. ER_W represents the eviction ratio for the tested workload. We again use $\alpha = 0.05$ ($SL = 95\%$) as the threshold for the hypothesis tests and indicate whether the hypothesis was rejected in column `<`. Finally, column `>` indicates whether the opposing alternative hypothesis, i.e., the distribution of the SPS is significantly *greater* than that of the measured sample, is true.

App Server. In Table 6.2, we look at all the workloads that contain the App Server. The first thing we observe is that our results show no significant difference between the server running alone on a traditional CPU to if it would run on a CPU implementing `MultiCache`.

We see that this configuration has a simulated eviction ratio of 0%, which is not unexpected because the App Server only ever starts a single process. We also observe

Benchmark	Mean LMR	Mean ER	Change	pvalue	<	>
A	0.064 (±0.000)	0.000 (±0.000)	+0.000%	0.3875	F	F
A_D	0.071 (±0.002)	0.000 (±0.000)	-9.445%	0.1400	F	F
A_F	0.073 (±0.002)	0.007 (±0.000)	-11.900%	0.1241	F	F
A_D_W	0.070 (±0.003)	0.000 (±0.000)	-7.979%	0.2685	F	F
A_D_SMT	0.083 (±0.000)	0.000 (±0.000)	-22.938%	0.0432	T	F
A_F_SMT	0.081 (±0.000)	0.000 (±0.000)	-21.001%	0.0433	T	F
A_M_SMT	0.065 (±0.000)	0.000 (±0.000)	-0.956%	0.2558	F	F
A_S_SMT	0.063 (±0.000)	0.000 (±0.000)	+1.165%	0.7242	F	F
A_W_SMT	0.080 (±0.001)	0.000 (±0.000)	-19.806%	0.0433	T	F

Table 6.2.: Workload Evaluation - App Server

that for workloads `A_D`, `A_F`, `A_D_W`, `A_M_SMT`, and `A_S_SMT`, neither Null-Hypothesis is rejected. We interpret this as the lack of evidence that the expected distribution is either `greater` or `less`. We identify an large improvement in the miss ratio for the remaining SMT workloads. We assume that the high miss ratios of the SMT workloads are caused by the servers thrashing each other's cache lines while they are running in parallel.

Database Server. The results for the Database Server (Table 6.3) paint a similar picture as those for the App Server. We see that SMT benchmarks lead to an improvement for all combinations. For the single-core benchmarks, the improvement is minimal to insignificant, but we also see no increase in LMR. This server also only ever spawns a single process, and the eviction ratios are always close to 0%.

Benchmark	Mean LMR	Mean ER	Change	pvalue	<	>
D	0.031 (±0.000)	0.000 (±0.000)	+0.000%	0.5418	F	F
A_D	0.031 (±0.000)	0.000 (±0.000)	+0.516%	0.8213	F	F
D_M	0.031 (±0.000)	0.001 (±0.000)	-0.388%	0.0577	F	F
D_W	0.031 (±0.000)	0.000 (±0.000)	-1.209%	0.0432	T	F
A_D_W	0.031 (±0.000)	0.000 (±0.000)	+0.115%	0.5927	F	F
D_F_W	0.031 (±0.000)	0.013 (±0.000)	-0.970%	0.0433	T	F
D_S_W	0.032 (±0.000)	0.000 (±0.000)	-1.909%	0.0433	T	F
A_D_SMT	0.048 (±0.000)	0.000 (±0.000)	-35.220%	0.0432	T	F
D_F_SMT	0.044 (±0.000)	0.000 (±0.000)	-30.106%	0.0432	T	F
D_M_SMT	0.032 (±0.000)	0.000 (±0.000)	-3.175%	0.0433	T	F
D_S_SMT	0.031 (±0.000)	0.000 (±0.000)	-1.607%	0.0432	T	F
D_W_SMT	0.045 (±0.000)	0.000 (±0.000)	-31.347%	0.0433	T	F

Table 6.3.: Workload Evaluation - Database Server

File Server. Unlike the previous two servers, the Samba File Server spawns one process per client connection. As shown in Table 6.4, the eviction ratios here are all around 33%, which means 33% of the time, a process gets a fresh cache upon scheduling. Here we first observe a significant increase in miss ratio for the basic and non-SMT workloads.

Benchmark	Mean LMR	Mean ER	Change	pvalue	<	>
F	0.050 (±0.000)	0.317 (±0.001)	+2.535%	0.9568	F	T
A_F	0.050 (±0.000)	0.311 (±0.000)	+2.955%	0.9567	F	T
F_W	0.051 (±0.000)	0.311 (±0.000)	+1.238%	0.9567	F	T
D_F_W	0.050 (±0.000)	0.309 (±0.000)	+2.702%	0.9567	F	T
A_F_SMT	0.071 (±0.000)	0.328 (±0.000)	-26.958%	0.0433	T	F
D_F_SMT	0.070 (±0.000)	0.329 (±0.000)	-26.433%	0.0433	T	F
F_M_SMT	0.051 (±0.000)	0.322 (±0.000)	+1.980%	0.9568	F	T
F_S_SMT	0.050 (±0.000)	0.321 (±0.000)	+3.473%	0.9568	F	T
F_W_SMT	0.068 (±0.000)	0.356 (±0.000)	-23.586%	0.0433	T	F

Table 6.4.: Workload Evaluation - File Server

On the other hand, for the SMT workloads we see significant improvements for `A_F_SMT`, `D_F_SMT` and `F_W_SMT`, while `F_M_SMT` and `F_S_SMT` see higher expected miss ratios. We assume this is because the Mail and Stream Servers operate in different parts of the L1 cache or are scheduled significantly less often (see Figure 6.4).

Mail Server. Of the servers in our setup, the Postfix server spawns the most processes (see Figure 6.3), and we see this reflected in the ER Table 6.5. Here we again see improvements in LMR for the SMT workloads, while we see an expected loss in performance for the basic and single-core benchmarks.

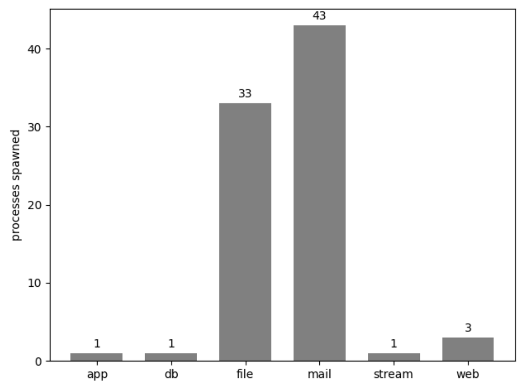


Figure 6.3.: Spawned Processes by Server

Benchmark	Mean LMR	Mean ER	Change	pvalue	<	>
M	0.045 (±0.000)	0.407 (±0.005)	+2.136%	0.9567	F	T
D_M	0.044 (±0.000)	0.458 (±0.006)	+3.177%	0.9568	F	T
A_M_SMT	0.064 (±0.000)	0.520 (±0.003)	-28.666%	0.0432	T	F
D_M_SMT	0.064 (±0.000)	0.525 (±0.003)	-28.910%	0.0433	T	F
F_M_SMT	0.062 (±0.000)	0.650 (±0.003)	-25.263%	0.0433	T	F
M_S_SMT	0.048 (±0.000)	0.511 (±0.004)	-3.860%	0.0433	T	F
M_W_SMT	0.062 (±0.000)	0.669 (±0.003)	-25.140%	0.0433	T	F

Table 6.5.: Workload Evaluation - Mail Server

Stream Server. The Stream Server constitutes an outlier in our setup in terms of behavior. Figure 6.4 shows that the FFserver is scheduled for a significantly longer period of time (∼ **10ms**), as opposed to the other servers, which all only run for < **1ms on average**.

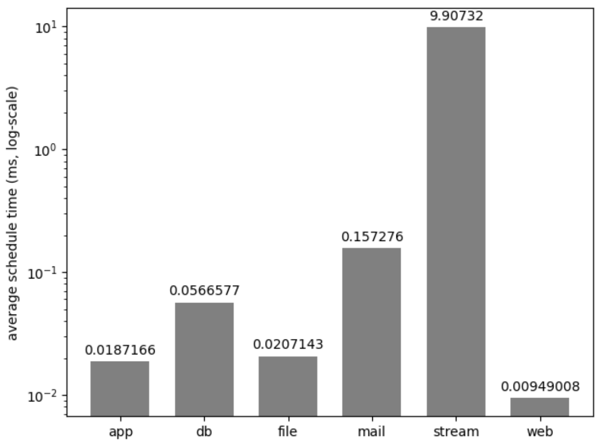


Figure 6.4.: Average Schedule Time

This explains the Mean ER of the `F_S_SMT` workload in Table 6.6. Due to its longer runtime, it is also scheduled less frequently, and other threads are scheduled in the meantime. The Mail Server takes up all the caches in the meantime, and subsequently, the ER increases to nearly 100%. Otherwise, we again observe improvements in the SMT workloads.

Web Server. For the Apache Web Server, we see comparable results as to the Database and App Servers. In the default configuration, the server has two worker processes handling up to 25 connections each [30], and one additional process handling incoming connections and load-balancing.

Benchmark	Mean LMR	Mean ER	Change	pvalue	$<$	$>$
S	0.059 (±0.000)	0.000 (±0.000)	+0.000%	0.5216	F	F
S_W	0.057 (±0.000)	0.000 (±0.000)	+2.685%	0.9568	F	T
D_S_W	0.032 (±0.000)	0.010 (±0.000)	+80.672%	0.9567	F	T
A_S_SMT	0.070 (±0.000)	0.000 (±0.000)	-16.889%	0.0432	T	F
D_S_SMT	0.075 (±0.000)	0.000 (±0.000)	-21.943%	0.0432	T	F
F_S_SMT	0.071 (±0.000)	0.979 (±0.001)	-7.553%	0.0432	T	F
M_S_SMT	0.055 (±0.001)	0.120 (±0.001)	+7.096%	0.9461	F	F
S_W_SMT	0.069 (±0.000)	0.000 (±0.000)	-14.529%	0.0432	T	F

Table 6.6.: Workload Evaluation - Stream Server

Benchmark	Mean LMR	Mean ER	Change	pvalue	$<$	$>$
W	0.065 (±0.000)	0.000 (±0.000)	+0.000%	0.5909	F	F
D_W	0.065 (±0.000)	0.000 (±0.000)	+0.882%	0.9533	F	T
F_W	0.065 (±0.000)	0.004 (±0.000)	+0.165%	0.7161	F	F
S_W	0.066 (±0.000)	0.000 (±0.000)	-0.816%	0.0433	T	F
A_D_W	0.063 (±0.000)	0.000 (±0.000)	+3.421%	0.9568	F	T
D_F_W	0.065 (±0.000)	0.004 (±0.000)	+0.988%	0.9568	F	T
D_S_W	0.065 (±0.000)	0.000 (±0.000)	+0.509%	0.8991	F	F
A_W_SMT	0.087 (±0.000)	0.000 (±0.000)	-24.636%	0.0432	T	F
D_W_SMT	0.087 (±0.000)	0.000 (±0.000)	-24.685%	0.0433	T	F
F_W_SMT	0.083 (±0.000)	0.029 (±0.000)	-20.650%	0.0432	T	F
M_W_SMT	0.063 (±0.000)	0.005 (±0.000)	+4.066%	0.9567	F	T
S_W_SMT	0.062 (±0.000)	0.000 (±0.000)	+5.850%	0.9567	F	T

Table 6.7.: Workload Evaluation - Web Server

6.2.1. Problem

In our individual evaluations, we identify one big problem: The average load miss ratios we capture from our experiments are often out of bounds of the lower and upper bound baselines for the individual servers. Our initial assumption was that they should be within these bounds (except for the SMT workloads, which can be higher than the upper bound due to parallel scheduling). We observe this effect with a $SL > 95\%$ for workloads like $\texttt{A\_F}_{\texttt{APP}}$, $\texttt{D\_S\_W}_{\texttt{STREAM}}$, $\texttt{S\_W}_{\texttt{STREAM}}$ or $\texttt{M\_W\_SMT}_{\texttt{WEB}}$ and $\texttt{S\_M\_SMT}_{\texttt{WEB}}$, among others.

We attribute this effect to possible changes in timings for different workloads configurations. E.g., we suspect that the handler thread of the App Server does more/less work depending on the number of client requests it must handle, which could be significantly different from workload to workload.

Figure 6.5 shows the difference in `Instructions Per Schedule` for the three most scheduled App Server threads between the two workloads $\texttt{A\_F}_{\texttt{APP}}$ and $\texttt{A\_F\_SMT}_{\texttt{APP}}$. We initially expected that these would be the same between the different workloads, but

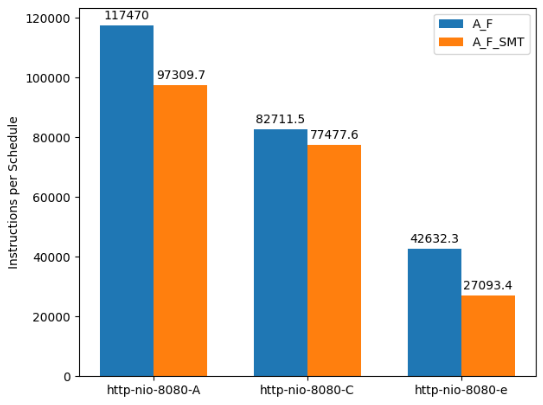


Figure 6.5.: Instructions Per Schedule Comparison - App Workloads

especially for the App Server, there are significant differences, which leads to these discrepancies.

Finding the exact cause of this problem and obtaining stable results is outside of the scope of this thesis and the subject of future work. However, we still think that the findings of this thesis provide valuable insight into the expected performance of the `MultiCache` design.

6.3. Combined Evaluations

In this section, we combine the results above into a comprehensive overview of our evaluation results to see for which of the workloads we expect an improved performance using `MultiCache`.

In Figure 6.6, we combine all the results from the previous section. Here, the Y-axis represents a factor for increase/decrease in LMR relative to the difference between the two baselines:

$$1\delta = B_{HIGH\_S} - B_{LOW\_S} \tag{6.2}$$

A Y-value of 10 would indicate that the measured miss ratio is $10 * \delta$ removed from the lower bound baseline B_{LOW_S}.

In Table 6.8, we combine the results from our individual evaluations and color the cells based on significance. Green cells indicate an at least equally good expected miss ratio using `MultiCache`, while red cells indicate a worse one. Yellow cells indicate that we do not have a high enough sample size for a significance level of $SL > 95\%$. In the cases of `A_D_SMT`, `A_F_SMT`, `A_W_SMT`, `D_F_SMT`, `D_M_SMT`, `D_S_SMT`, `D_W_SMT`, and `F_W_SMT` we observe improvements for both server in the configuration. Overall we see that our SMT workloads benefit very clearly from dedicated caches, while the single-core combined workloads produce mixed results.

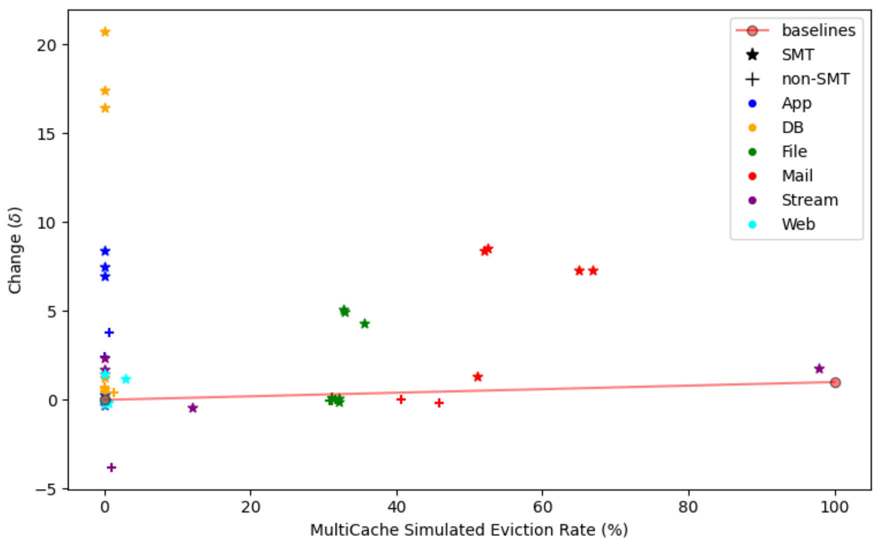


Figure 6.6.: Evaluation Results

Benchmark	App	DB	File	Mail	Stream	Web
A	+0.000%					
D		+0.000%				
F			+2.535%			
M				+2.136%		
S					+0.000%	
W						+0.000%
A_D	-9.445%	+0.516%				
A_F	-11.900%		+2.955%			
D_M		-0.388%		+3.177%		
D_W		-1.209%				+0.882%
F_W			+1.238%			+0.165%
S_W					+2.685%	-0.816%
A_D_W	-7.979%	+0.115%				+3.421%
D_F_W		-0.970%	+2.702%			+0.988%
D_S_W		-1.909%			+80.672%	+0.509%
A_D_SMT	-22.938%	-35.220%				
A_F_SMT	-21.001%		-26.958%			
A_M_SMT	-0.956%			-28.666%		
A_S_SMT	+1.165%				-16.889%	
A_W_SMT	-19.806%					-24.636%
D_F_SMT		-30.106%	-26.433%			
D_M_SMT		-3.175%		-28.910%		
D_S_SMT		-1.607%			-21.943%	
D_W_SMT		-31.347%				-24.685%
F_M_SMT			+1.980%	-25.263%		
F_S_SMT			+3.473%		-7.553%	
F_W_SMT			-23.586%			-20.650%
M_S_SMT				-3.860%	+7.096%	
M_W_SMT				-25.140%		+4.066%
S_W_SMT					-14.529%	+5.850%

Table 6.8.: Detailed Evaluation Results

7. Conclusion

This thesis aimed to evaluate the feasibility of the `MultiCache` design by measuring and simulating typical data center workloads. We collected data about a set of selected workloads. We compared the measured load miss ratios to the expected load miss ratios to see if these workloads suffer from a significant increase in load miss ratio on a processor implementing `MultiCache`.

We found that our design works best with `Simultaneous Mulithreading` enabled when the processes being executed on each hardware thread interfere heavily with each other's cache lines. Furthermore, our results indicated that our design heavily depends on the number of processes simultaneously running on the same CPU. We showed our design should perform at least equally well when this number is not higher than the number of user caches per core.

However, we also ran into problems analyzing and comparing different workloads as they do not always behave identically from benchmark to benchmark. We can, therefore, only consider the results of this thesis as a strong indicator that the `MultiCache` design has practical potential.

Bibliography

[1] Ronnie Sahlberg Andrew Tridgell. *DBENCH*. URL: `https://dbench.samba.org/` (visited on 09/21/2022).

[2] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, and Srinivas Devadas. “Mi6: Secure enclaves in a speculative out-of-order processor”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 42–56.

[3] Marco Spaziani Brunella, Sara Turco, Giuseppe Bianchi, and Nicola Blefari Melazzi. “Foreshadow-VMM: on the practical feasibility of L1 cache Terminal Fault attacks”. In: (2018).

[4] Luna Backes Drault. *Evaluation of Cache Inclusion Policies in Cache Management*. 2017. URL: `https://core.ac.uk/download/pdf/147122148.pdf`.

[5] *FFMpeg*. URL: `https://ffmpeg.org/` (visited on 09/21/2022).

[6] Damien Gille. *Study of Different Cache Line Replacement Algorithms in Embedded Systems*. 2007. URL: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.217.3594&rep=rep1&type=pdf`.

[7] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. “ASLR on the Line: Practical Cache Attacks on the MMU.” In: *NDSS*. Vol. 17. 2017, p. 26.

[8] Daniel Gruss. *Side-Channel Security - Introduction*. 2021. URL: `https://www.iaik.tugraz.at/wp-content/uploads/teaching/side-channel-security/2021/chapter1.pdf`.

[9] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. “Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR”. In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, pp. 368–379.

[10] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. *Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches*. 2015. URL: `https://online.tugraz.at/tug_online/voe_main2.getvolltext?pCurrPk=85809`.

[11] Berk Gülmezoğlu, Mehmet Sinan İnci, and Gorka Irazoqui. *A Faster and More Realistic Flush+Reload Attack on AES*. 2015. URL: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.700.8069&rep=rep1&type=pdf`.

[12] Jim Handy. *The cache memory book*. eng. 2. ed.. 1998. ISBN: 0123229804.

[13] Dawei Huang, Deshi Ye, Qinming He, Jianhai Chen, and Kejiang Ye. "Virt-LM: a benchmark for live migration of virtual machine". In: *Proceedings of the 2nd ACM/SPEC International Conference on Performance engineering.* 2011, pp. 307–316.

[14] IBM. *SMB protocol.* URL: `https://www.ibm.com/docs/en/aix/7.2?topic=management-smb-protocol` (visited on 09/20/2022).

[15] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* 2016. URL: `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`.

[16] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide.* URL: `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.` (visited on 10/15/2022).

[17] Intel Corporation. *L1 Terminal Fault.* URL: `https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/intel-analysis-l1-terminal-fault.html` (visited on 10/15/2022).

[18] Intel Corporation. *What Is Hyper-Threading?* URL: `https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html` (visited on 10/10/2022).

[19] Gorka Irazoqui, Mehmet Sinan IncI, Thomas Eisenbarth, and Berk Sunar. "Know Thy Neighbor: Crypto Library Detection in Cloud." In: *Proc. Priv. Enhancing Technol.* 2015.1 (2015), pp. 25–40.

[20] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Lucky 13 strikes back". In: *Proceedings of the 10th acm symposium on information, computer and communications security.* 2015, pp. 85–96.

[21] Kamil Kedzierski and Miquel Moreto. *Pseudo-LRU based Cache Partitioning Algorithms.* 2009. URL: `https://kamilonlinedotcom.files.wordpress.com/2013/10/kkedzier_pact_2009.pdf`.

[22] Paul Kocher, Jann Horn, and Anders Fogh. *Spectre Attacks: Exploiting Speculative Execution.* 2018. URL: `https://spectreattack.com/spectre.pdf`.

[23] Alexey Kopytov. *sysbench.* URL: `https://github.com/akopytov/sysbench` (visited on 09/20/2022).

[24] *L1TF - L1 Terminal Fault.* URL: `https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/l1tf.html` (visited on 10/14/2022).

[25] Moritz Lipp, Michael Schwarz, and Daniel Gruss. *Meltdown: Reading Kernel Memory from User Space.* 2018. URL: `https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-lipp.pdf`.

[26] Fangfei Liu, Yuval Yarom, and Qian Ge. *Last-Level Cache Side-Channel Attacks are Practical.* 2015. URL: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7163050`.

[27] Live Networks, Inc. *openRTSP - A command-line RTSP client.* URL: `http://www.live555.com/openRTSP/` (visited on 09/21/2022).

[28] Matthew Martin. *SRAM vs. DRAM: Know the Difference.* 2022. URL: `https://www.guru99.com/sram-vs-dram-difference.html`.

[29] *N/A.* URL: `https://uops.info/cache.html` (visited on 01/22/2022).

[30] Dele Omotosho. *apache2.conf - GitHub.* URL: `https://gist.github.com/deletosh/5960912#file-apache2-conf-L118` (visited on 10/09/2022).

[31] Oracle. *MySQL.* URL: `https://www.mysql.com` (visited on 09/19/2022).

[32] Postfix. *Postfix.* URL: `https://www.postfix.org/` (visited on 09/21/2022).

[33] SAMBA. *About Samba.* URL: `https://www.samba.org/` (visited on 09/20/2022).

[34] Laasya Samhita and Hans J Gross. "The "clever Hans phenomenon" revisited". In: *Communicative & integrative biology* 6.6 (2013), e27122.

[35] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. "ZombieLoad: Cross-privilege-boundary data sampling". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security.* 2019, pp. 753–768.

[36] Statista. *Ranking of the most popular database management systems worldwide, as of August 2022.* URL: `https://www.statista.com/statistics/809750/worldwide-popularity-ranking-database-management-systems/#statisticContainer` (visited on 09/19/2022).

[37] The Apache Software Foundation. *ab - Apache HTTP server benchmarking tool.* URL: `https://httpd.apache.org/docs/2.4/programs/ab.html` (visited on 09/19/2022).

[38] The Apache Software Foundation. *Apache - HTTP Server Project.* URL: `https://httpd.apache.org/` (visited on 09/21/2022).

[39] The Apache Software Foundation. *Apache Tomcat.* URL: `https://tomcat.apache.org/` (visited on 09/19/2022).

[40] Dean M Tullsen, Susan J Eggers, and Henry M Levy. "Simultaneous multithreading: Maximizing on-chip parallelism". In: *Proceedings of the 22nd annual international symposium on Computer architecture.* 1995, pp. 392–403.

[41] *Using the Linux Kernel Tracepoints.* URL: `https://docs.kernel.org/trace/tracepoints.html` (visited on 10/15/2022).

[42] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the Keys to the Intel {SGX} Kingdom with Transient {Out-of-Order} Execution". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 991–1008.

[43] Wietse Venema. *smtp-source - parallelized SMTP/LMTP test generator.* URL: `https://www.postfix.org/smtp-source.1.html` (visited on 09/21/2022).

[44] Pepe Vila, Boris Köpf, and José F. Morales. *Theory and Practice of Finding Eviction Sets.* 2018. URL: `https://arxiv.org/pdf/1810.01497.pdf`.

[45] Daimeng Wang, Ajaya Neupane, and Zhiyun Qian. *Unveiling your keystrokes: A Cache-based Side-channel Attack on Graphics Libraries.* 2019. URL: `https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_05B-3_Wang_paper.pdf`.

[46] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. "Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution". In: (2018).

[47] Mario Werner, Thomas Unterluggauer, and Lukas Giner. *ScatterCache: Thwarting Cache Attacks via Cache Set Randomization.* 2019. URL: `https://www.usenix.org/system/files/sec19-werner.pdf`.

[48] Frank Wilcoxon. "Individual comparisons by ranking methods". In: *Breakthroughs in statistics.* Springer, 1992, pp. 196–202.

[49] M. V. Wilkes. "Slave Memories and Dynamic Storage Allocation". In: *IEEE Transactions on Electronic Computers* EC-14.2 (1965), pp. 270–271. DOI: `10.1109/PGEC.1965.264263`.

[50] Hao Wu, Fangfei Liu, and Ruby B Lee. "Cloud server benchmark suite for evaluating new hardware architectures". In: *IEEE Computer Architecture Letters* 16.1 (2016), pp. 14–17.

[51] Yuval Yarom and Katrina Falkner. *Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack.* 2014. URL: `https://eprint.iacr.org/2013/448.pdf`.

A. Appendix

Table A.1 shows the evaluation results using $CPC = 4$ caches per core. We observe similar results as with $CPC = 5$ for all workloads.

Benchmark	App	DB	File	Mail	Stream	Web
A	+0.000%					
D		+0.000%				
F			+2.636%			
M				+2.898%		
S					+0.000%	
W						+0.000%
A_D	-9.445%	+0.516%				
A_F	-11.900%		+3.052%			
D_M		-0.388%		+3.789%		
D_W		-1.209%				+0.883%
F_W			+1.332%			+0.165%
S_W					+2.806%	-0.816%
A_D_W	-7.959%	+0.140%				+3.476%
D_F_W		-0.970%	+2.843%			+0.989%
D_S_W		-1.887%			+101.418%	+0.551%
A_D_SMT	-22.938%	-35.220%				
A_F_SMT	-21.000%		-26.765%			
A_M_SMT	-0.956%			-27.945%		
A_S_SMT	+1.165%				-16.888%	
A_W_SMT	-19.806%					-24.636%
D_F_SMT		-30.106%	-26.245%			
D_M_SMT		-3.175%		-28.203%		
D_S_SMT		-1.607%			-21.943%	
D_W_SMT		-31.347%				-24.684%
F_M_SMT			+2.083%	-24.657%		
F_S_SMT			+3.580%		-7.549%	
F_W_SMT			-23.454%			-18.815%
M_S_SMT				-3.225%	+7.183%	
M_W_SMT				-24.515%		+4.216%
S_W_SMT					-14.321%	+5.850%

Table A.1.: Detailed Evaluation Results (CPC=4)

The same is true for Table A.2, which contains the results for $CPC = 3$. Overall, we conclude that the results for the workloads that we used for this thesis are impacted by different configurations of the `MultiCache` design, but we see no difference in single-process configurations, while configurations where multiple processes are running would suffer decreased performances.

Benchmark	App	DB	File	Mail	Stream	Web
A	+0.000%					
D		+0.000%				
F			+2.884%			
M				+3.782%		
S					+0.000%	
W						+0.000%
A_D	-9.445%	+0.516%				
A_F	-11.900%		+3.335%			
D_M		-0.388%		+4.691%		
D_W		-1.178%				+0.947%
F_W			+1.621%			+0.166%
S_W					+14.645%	-0.795%
A_D_W	-7.956%	+0.148%				+3.516%
D_F_W		-0.968%	+3.100%			+0.993%
D_S_W		-1.876%			+101.418%	+0.603%
A_D_SMT	-22.938%	-35.220%				
A_F_SMT	-20.998%		-26.361%			
A_M_SMT	-0.956%			-27.019%		
A_S_SMT	+1.165%				-16.785%	
A_W_SMT	-19.801%					-17.734%
D_F_SMT		-30.106%	-25.837%			
D_M_SMT		-3.175%		-27.280%		
D_S_SMT		-1.607%			-21.842%	
D_W_SMT		-31.347%				-17.795%
F_M_SMT			+2.332%	-23.993%		
F_S_SMT			+3.815%		-7.545%	
F_W_SMT			-23.132%			-14.059%
M_S_SMT				-2.471%	+7.604%	
M_W_SMT				-23.925%		+4.597%
S_W_SMT					-4.600%	+5.850%

Table A.2.: Detailed Evaluation Results (CPC=3)