MASTER'S THESIS 2021

Cacheray - A Trace Based Cache Simulator

Hannes Åström, Wilhelm Lundström

ISSN 1650-2884
LU-CS-EX: 2021-43

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2021-43

Cacheray - A Trace Based Cache Simulator

Cacheray - En trace baserad cache simulator

Hannes Åström, Wilhelm Lundström

Cacheray - A Trace Based Cache Simulator

Hannes Åström
dat15has@student.lu.se

Wilhelm Lundström
fte13wlu@student.lu.se

September 22, 2021

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se
Kim Gräsman, kgrasman@sandvine.com

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

For many developers, the processor cache is perceived as a black box that increases the speed at which programs can execute. There are tools for analyzing the cache performance of programs. These are know as *cache simulators*. These tools simulate the cache interactions of the running program and provides feedback to the user where to improve. However, these tools can be too resource demanding to run on certain systems, such as on a bare-metal operating system. To get around this problem, the simulation could be done after the programs execution, using a trace/recording of the program.

This thesis develops a system which can do this, which we call Cacheray. Cacheray provides a way of recording the cache affecting operations into a 'trace file'. This trace can then be used by the Cacheray simulator to simulate cache interactions. It can even be used to simulate changes to certain memory structures, without recompiling and executing the program again.

After testing the simulator, we conclude that it shows some degree of correctness when compared to an established simulator, but does not reach the same degree of correctness. It is possible to use the simulator to change the inner layout of structs and have that change correctly shown in the simulation results. The system is not without its faults. It is slow, produces massive traces, and does not collect sufficient data to allow for more correct simulations.

Keywords: Cache, Trace, Simulator, C

Acknowledgements

We would like to thank our supervisor at Sandvine, Kim Gräsman, for providing us with much help during the thesis. We would also like to thank Anders Waldenborg, also at Sandvine, for extending our discussions and providing both insight and technical support. A large thanks to Sandvine as a whole for sponsoring this master thesis.

Finally, a big thanks to our supervisor at LTH, Jonas Skeppstedt, and our examiner Flavius Gruian.

Contents

Chapter 1
Introduction

The Central Processing Unit (CPU) is the core of a computer. It is responsible for executing instructions, handling memory and sending signals. Inside the modern CPU is a hardware component called a *CPU cache*, often shortened to just *cache*. The cache is a temporary storage for recently accessed data, designed to reduce the time needed to read and write data from the computer's main memory. It operates much faster than main memory, and is physically closer to the CPU in hardware, minimizing the time needed to access data. The trade-off is that the cache is more expensive (in terms of physical size, energy consumption and price per stored byte) and as a consequence has a relatively small amount of memory available.

Because of how data is generally stored, the performance boost provided by the cache is substantial. Most modern computers have several layers of cache, using different technologies to allow each level to have greater capacity at the cost of some slowdown, while still being faster than main memory.

Despite being a very powerful and important part of the CPU, the cache is often poorly understood by programmers. This matters, as writing code and designing software that uses the cache efficiently can give great performance improvements. Thus, increased understanding of the cache among developers and coders is highly desirable. One way of achieving this is to help developers to actually see which parts of their programs are performing badly with the cache. Cache performance analysis by way of simulation, using a *cache simulator*, can be used to accomplish that.

It is worth noting here that a cache simulator can be quite processor heavy. The company sponsoring this thesis, Sandvine, has a system with a very basic operating system, insofar that an on-line cache simulator (i.e. a simulator that performs analysis in parallel with the execution of the program being analyzed) has no way of executing, as the resource requirements become to large. A way around this limitation would be to instead produce a trace of the program execution, containing the relevant memory events, and pass the trace to a more powerful system which in turn can run the simulator.

Furthermore, it would be useful to be able to simulate structural changes of the program in the simulator, without changing and recompiling the source code. Data used by the com-

puter is often organized in different *data structures*. Though there are many types of data structures, in this thesis, we will focus on a basic data type used by the C programming language. This data structure is called a *struct* and is a composite data type. This means that it can contain several different segments of data of differing, or the same, type. These segments/variables will be referred to as *members* or *struct members*.

1.1 Research Questions

- What amount of resources are needed for the tested program to generate a trace file?
- What level of correctness can we achieve with a trace based simulator?
- Using the simulator, can a developer change the order of struct members and get correct results without needing to recompile the original program?

To clarify what this means, we need to investigate what resources are needed for trace file generation; file sizes, memory usage, execution time and trace sizes. We also need to establish that the simulator outputs correct data, i.e. the results of the simulator reflect how an actual cache behaves. Finally, we want to see if it is possible to change struct layouts in the simulator and still get correct data results, without needing to touch the original program again.

1.2 Goals

The main objective of this thesis is to construct a cache simulator, use it and see if it can correctly simulate the cache hits and misses and how data will be moved in and out of cache. This simulator should work based on **two** principles:

1. The simulator is trace based, thus not requiring the tested program to be executed alongside it.
2. The simulator is able to simulate changes in the order of struct members in the tested program, without needing the program to be recompiled or executed again.

1.3 Division of labor

The focus from Hannes has mainly been on the runtime and the handling of DWARF debug data. While it has been a joint effort with a lot of discussion in the process, Hannes has taken main lead as architect behind the layout of the system as a whole. Attributing code, Hannes has written most of the runtime, the dwarf2json script, large parts of the simulator such as the input parsing and its tests, as well as designing the input files for the simulator. Chapter 2, 5 and 6 were mainly written by Hannes, as well as sections 1.6, 4.4 and 4.5, and large parts of chapter 7 and 8.

The focus from Wilhelm has mainly been on LLVM and the LLVM Intermediary Representation (LLVM IR) and the core of the simulator. A lot of time was put on understanding the inner workings of the LLVM system (specifically the IR and the passes) and the theory

behind cache memory. Attributing code, Wilhelm has written the MallocTracker pass and much of the core of the simulator code, such as the cache system logic and the simulator backbone, as well as some of the testing. Chapter 1, 3 and 4 were mainly written by Wilhelm, as well as sections 2.1, 2.4 and 5.1, and large parts of chapter 7 and 8.

1.4 Sandvine

Sandvine Incorporated is a company specializing in network solutions with a focus on network policy and deep packet inspection. Sandvine provides network operators with products that gives real time insight into network traffic and use that to apply policies. This can improve the end-users quality of experience and the operator's expenses.

This thesis is a joint venture between the authors and Sandvine. Sandvine provided the idea for the thesis and significant help with the technical side of the thesis.

1.5 Related Works

Analyzing the cache is not a new idea, and throughout the years several tools to do this have been created. One of the most prominent among them is **Cachegrind** [5], a part of Valgrind. In this thesis it will be used as a point of reference, comparing the results from Cachegrind with the results from our simulator. It was selected among a number of other simulators (which are further discussed in the following paragraphs). The reason for this choice being that it is an already established and well documented tool, so if our results match those of Cachegrind we have reason to believe that our results are correct.

Cachegrind is a so called on-line tool, meaning it is run in parallel with the program being profiled. This has some advantages, but also means Cachegrind can not run in a bare metal environment. This is of the advantages of a trace based cache simulator: as long as a trace file can be generated, the simulation can be run off-line on a separate machine capable of running the simulator. As the simulation part is (typically) the most demanding, this allows for analysis of programs that run on systems that are unable to run existing on-line tools.

There are several cache simulators currently available, with different advantages and limitations. "A Survey of Cache Simulators" [2] provides a good overview of the current options. It also gives quick insight into which simulators are trace based, and which are not. As our work is concerned with a trace based approach, we will be focusing on simulators that provide such an approach. Based on figure 1 in [2], this means that we want to look closer at the simulators Moola, CASPER, vCSIMx86, SMPCache, pycachesim and DineroIV.

DineroIV [11] is a somewhat old simulator developed by Mark D. Hill and Jan Edler at the University of Wisconsin in the late 90's. It is trace based and capble of simulating single core systems (no multicore support). It allows for simulation of several layers of cache. It can give the user information about expected cache hit/miss rates of a program for a specified cache configuration [16]. DineroIV provides some information regarding the format of the trace files it uses, referred to as "din" files. They consist of a sequence of space separated operation - address pairs (eg "0 f1ha27ed", where 0 corresponds to a read operation). It is up to the end user to generate trace files of this format.

Moola [26] and CASPER [15] are both trace based multicore cache simulators. Casper

was developed in 2003, more than 10 years before Moola (2015). They are both capable of simulating the behaviour of single CPU caches as well as shared caches on a multicore system. Moola can provide estimates on the real execution time of the program being simulated. Moola is highly configurable, and the code is open source, making it a solid option for cache analysis. One shortcoming, however, is the lack of documentation, particularly regarding the trace file format. In order to use Moola, the user would need to figure know the trace format, and as far as we have been able to find the only way of doing this is either by investigating the code or consulting the creators of the system. Conversely, the CASPER system can create trace files, making it easier to get started with. CASPER can be used to obtain statistics regarding hit/miss rates, cache eviction rates, sharing characteristics (for shared cache), back-invalidations and snoop filter statistics. It is powerful and has a lot to offer, but is is closed source, and is not readily available for use.

Pycachesim is somewhat different to the previous simulators in that both the configuration and the trace (meaning the sequence of operations found in a trace file) are specified in a Python script. In other words, pycachesim does not read a trace of operations from a file. As such it is difficult to use pycachesim for more than educational purposes, where smaller examples can be coded by hand to illustrate the effects of the cache.

vCSIMx86 [18] is a trace based cache simulation framework developed to address the issue that most full system simultors at the time had issues dealing with a virtualized x86 host. vCSIMx86 provides a way to generate trace files using a modified version of QEMU. It then simulates the cache using a modified version of DineroIV that allows it to have multicore support. The outputs give information regarding cache hit/miss rates, with special regard taken to the effects cause by the use of Virtual Machines.

Finally, SMPCache [30] is a trace based multiprocessor simulator. It was developed mainly for educational purposes, featuring a full graphical interface for ease of use. It simulates cache systems on symmetric multiprocessors (hence the SMP in SMPCache) that use bus-based shared memory. It gives statistics about cache hit/miss rates as well as information regarding bus traffic, number of bus transactions, block transfers on the bus, and state transitions. The simulator accepts multiple trace formats, such as the Dinero (din) format and PDATS [17]. SMPCache is closed source, and is also limited to running on Windows systems.

There are of course several more simulators out there, to many to cover in this paper. For a more comprehensive overview we recommend starting with the aforementioned survey [2].

1.6 Contributions

This thesis contributes to the fields of *computer architecture* and *software design* in its creation of the runtime and simulator.

For the field of computer architecture, we have constructed a system which has the ability of measuring the cache performance on different kinds of architectures, post-execution. At the moment of writing, the system has the potential for use in research where other solutions are inappropriate. It also has the potential for further development to collect more data.

The reorder feature of the simulator contributes to the field of software design by giving developers a method of testing changes in struct orderings, without changing the source code of the program and compiling it again.

Chapter 2
Background

In this chapter, we present an overview of the different concepts that are important to the thesis and give detailed explanations of the parts that are needed to understand it. First, a reminder of what a CPU cache is and how it works. Then, a brief overview of the C programming language, with focus on its data structures. Finally, some details regarding the pre-made applications and interfaces used in this thesis. These include DWARF, LLVM, and ThreadSanitizer.

2.1 CPU cache

The cache is a part of the Central Processing Unit, *CPU*, which acts as an intermediate data storage between the CPU and the main memory. The purpose of the cache is to decrease the time needed to access data in memory, increasing the operating efficiency of the CPU. It does this by leveraging certain properties of the **memory hierarchy**.

2.1.1 Memory hierarchy

The optimal memory for general purpose computing is said to be fast, have a large storage capacity, and be cheap [22]. Unfortunately, these traits often work against each other, which results in memory having to make some sort of trade-off. For example, a hard disk drive has a large storage space and is relatively cheap, but accessing the data can take a lot of time, relative to the speed of the CPU. On the other hand, the array of registers a CPU has can operate incredibly fast, but with a very small capacity. In table 2.1, the difference between speed and cost per byte in different storage mediums can be seen to illustrate this.

This phenomenon gives rise to the concept of *Memory hierarchy*[10, p. 72]. The memory hierarchy in a computer is comprised of several levels of memory where the storage capacity of each level increases as we move further away from the processor. As the storage capacity goes up, so does the latency for fetching data to the CPU. Therefore, the lower in this hierarchy the

CPU needs to look for data, the faster it will load. Since the cache is typically implemented using SRAM technology [10] and main memory, *RAM*, is typically implemented using DRAM technology [10], we can look at table 2.1 and compare them. There, we can see that the cache will be many times faster than the main memory but it will cost a lot more per byte. It should be noted that the data in the table is 8 years old and does not accurately reflect the exact cost of these parts at present. However, the principle still stands.

These drastic differences in speed between the different levels of the memory makes the potential gain from efficient use of cache quite large.

Table 2.1: Access Time vs. Cost for different storage mediums

Type of Memory	Access Time [22]	Est. cost (2012, dollars/GiB) [22]
SRAM memory	0.5 - 2 ns	$500 - $1000
DRAM memory	50 - 70 ns	$10 - $20
Flash memory	5,000 - 50,000 ns	$0.75 - $1.5
Magnetic disk	5,000,000 - 20,000,000 ns	$0.05 - $0.10

2.1.2 What makes Cache useful

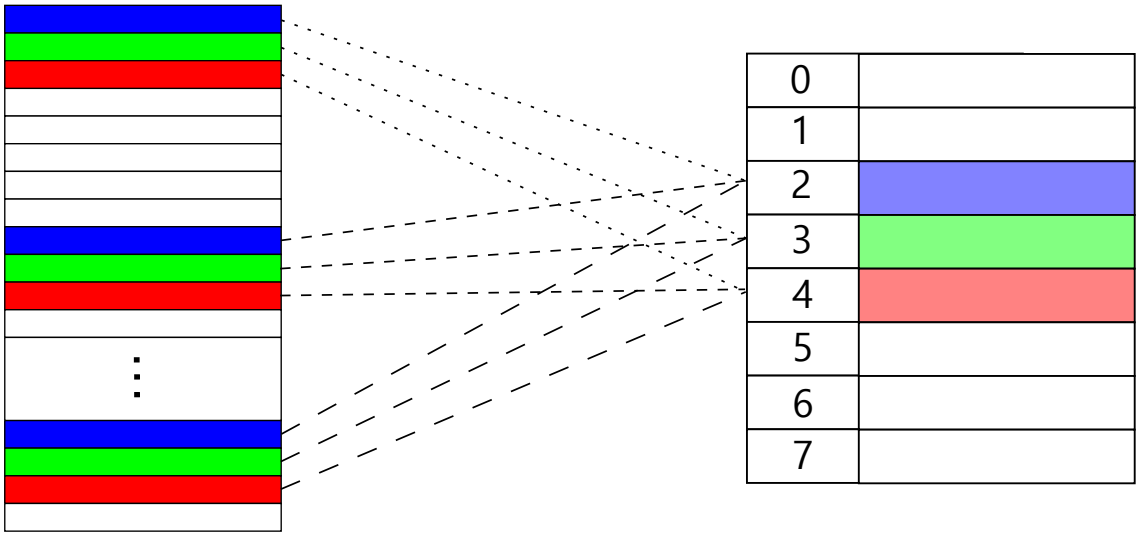
One of the main reasons why the cache is so useful and efficient is because it takes advantage of the *principle of locality*[1][27]. This principle states that the processor tends to access the same area of memory repeatedly in a small time frame. The principle can be divided into two main parts: **temporal locality** and **spatial locality**.

Temporal locality means that the time between accesses to a piece of data is relatively small. Temporal locality can then be taken advantage of by putting data that is accessed a lot in a place that is fast to access, since data thats recently been accessed will likely be accessed again shortly. As can be seen in table 2.1, accessing the cache is much faster than accessing RAM or flash memory. Following the principle of locality, by placing data that has recently been accessed in the cache, one can reduce the time taken for subsequent accesses.

Spatial locality means that as a piece of data is accessed, it is likely that adjacent data will soon be accessed as well. The cache takes advantage of this as well. The cache is made up of sets of sequential data chunks called **cache lines**. Each cache line has an address and a block of data associated with it. When a CPU needs to load some arbitrary data from memory, it will check the cache if there exists a block inside it with this data. If it finds the data in cache a so called **cache hit** occurs, and the data can be fetched from cache in a speedy fashion. Otherwise, if the data is not found in cache, a **cache miss** occurs. As a consequence the data has to be fetched from lower in the memory hierarchy, taking more time to do so. However, when fetching the requested data from RAM (or lower) the data is also put into the cache. If the cache is already full this of course means that some of the old data in cache is evicted to make room for the new. The amount of data fetched and put into cache is the same as the size of a cache line, meaning that if a particular byte of data is needed by the CPU, several surrounding bytes will be fetched as well. For a sense of how much extra data is fetched, a common size for cache lines is 64 bytes. This is a very advantageous feature when, for instance, iterating over an array (assuming it is an array of elements that are smaller than

[1] Also known as *Locality of References*

Figure 2.1: Simplified view of a direct-mapped cache, with main memory blocks to the left and cache lines of the cache to the right along with their index.



a cache line, such as char or int), as the data for later iteration loops will be put in cache in an earlier iteration loop.

It is possible for a piece of data to straddle cachelines. That is, the first part can be at the end of one cache block while the second part is at the beginning of the next block. This means that for instance a `long` that straddles cache lines will take two cachelines to store in cache, despite being much smaller than the block.

2.1.3 Placement policy

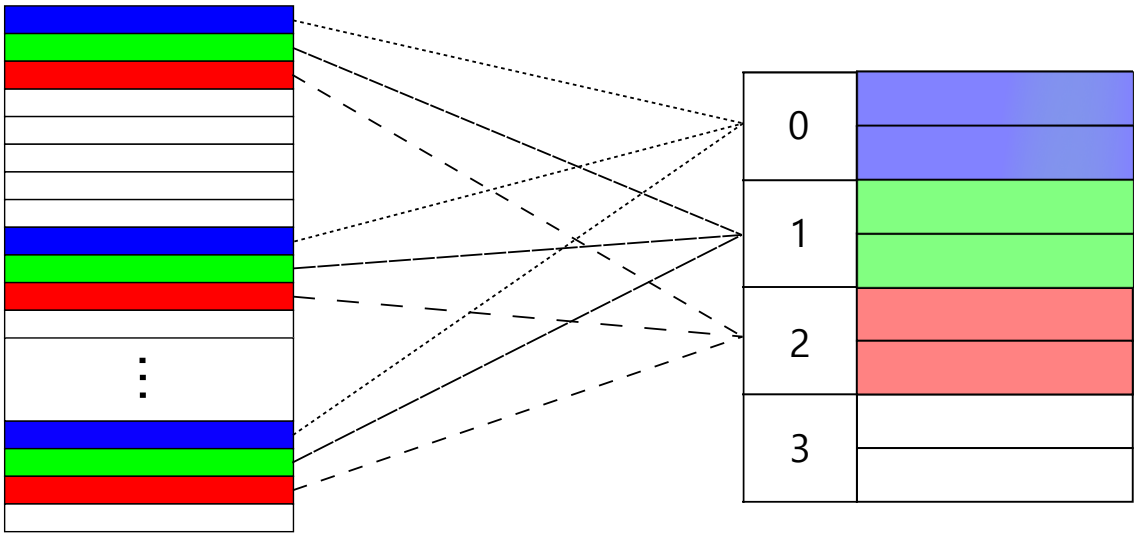
As mentioned earlier, the cache consists of sets of cache lines. How many cache lines can be held in a set is determined by the associativity of the cache. If each set only holds one cache line it is called a **direct-mapped** cache, an example of which can be seen in fig 2.1. If a single set holds all the cache lines it is called a **fully-associative** cache, as can be seen in fig 2.3. If each set holds 2, 4, 8 ... N cache lines we have a **set-associative** cache, often referred to as a N-way set-associative cache for clarity. A 2-way set-associative cache can be seen in fig 2.2. In a way, direct-mapped cache and fully-associative cache are simply special cases of a set-associative cache.

The figures for each type of cache show how blocks of data in memory can be mapped to the cache, also indicating that multiple blocks of data can map to the same location in cache.

The way the placement of cache lines in the cache is determined depends on the associativity of the cache and the effective address of the cache line. When the CPU needs a piece of data, it requests it from an address. On for instance a 64-bit system, this address consists of 64 bits. The bits of this address are used to find and place cache lines in the cache, by dividing the bits into a tag, index and offset. The least significant bits of the address make up the offset, and will not matter for the placement within the cache (as these bits simply become an offset within a cache line). In other words, if a cache line consists of 64 bytes, the

lowest 6 bits ($2^6 = 64$) of the address are the offset to access bytes within the same cache line.

Figure 2.2: Simplified view of a 2-way set-associative cache, with main memory blocks to the left and sets of cache lines to the right. The number is the set index.

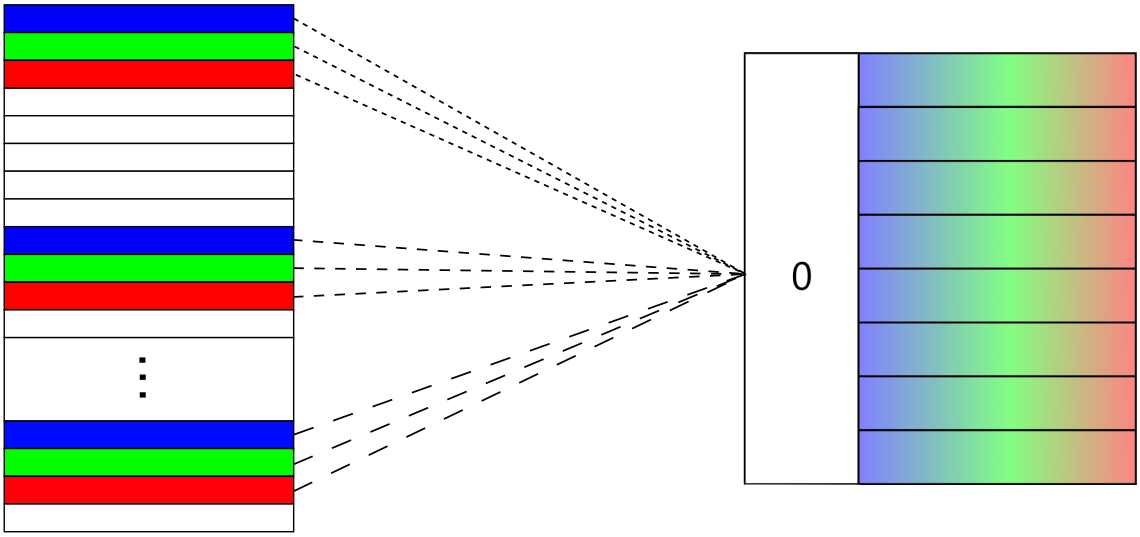


The index corresponds to the set within the cache that a cache line is placed into. This means that each set has an unchanging index that can be determined using cache size, line size and number of sets. For instance, in a 128KiB[2] 4-way set-associative cache with line size of 64B, each set can hold $4 \cdot 64 = 256$ bytes. So we have 128KiB divided by 256 equalling 512 sets, each with its own index from 0 to 511. So out of the remaining 58 bits from the earlier address, the 9 least significant bits ($2^9 = 512$) make up the index that identifies the set. Regardless of the other bits of the address, the data at that address will be placed in a cache line in a set with that particular index. Looking at the simplified cache illustrations of fig 2.1 and fig 2.2, the numbers on the left side of the cache correspond to an index, represented in decimal form. The number on the fully associative cache is a pseudo index, as zero bits are required to identify the one set that is present in a fully associative cache.

Continuing the example, the remaining 49 most significant bits of the address make up the tag. As might have become obvious from the previous section, many more cache lines will be mapped to the same set than can be held by that set. Each set might be able to hold 4 cache lines, but clearly there are more than 4 cache lines that will have those 9 bits that identify the index of that set. As such, the tag is what is checked against when looking for data in the cache. The offset does not change what cache line we search for, and the index determines the set, so the tag becomes the identifier used to check whether the requested cache line is in the cache. Figure 2.2 shows a simplified 2-way set-associative cache, where we can see that the blocks of data in memory coded green go into the same set, and 2 blocks can be in that set at once.

[2]KiB short for kibibyte, the binary counterpart to kB (kilobyte). One kibibyte is 1024 bytes, whereas one kilobyte is 1000 bytes.

Figure 2.3: Simplified view of a fully-associative cache, with main memory blocks to the left and a set of cache lines to the right. The number is a pseudo index.



To sum up, the data in cache is stored in a cache line that consists of the data, a tag, and one or two flag bits (the tag and flag bits do not take up the same space in hardware the data does, and are thus not counted towards the total size of the cache [23]). We will not go into detail about the flag bits, but they are basically used to indicate the validity of the data in cache, if it is outdated or not. When the CPU requests a piece of data e.g. an int, the address of the data is split into tag, index and offset, and the cache is checked using the tag to see if the data is present there. This process is handled by the Memory Management Unit (MMU), a piece of hardware that beyond handling the cache also takes care of bus arbitration, virtual memory management and memory protection. The details of the MMU are not important for the work done in this thesis, and we will therefore not cover the more details of how it works.

2.1.4 Replacement policy

The advantage of a direct-mapped cache is that the lookup for data can be done very quickly, as essentially only one cache line has to be checked. The trade-off is that only one cache line of a particular index can be in cache at any given time. So if a program happens to need data from two places in memory that share index, only one can be in the cache. As soon as the other is needed, it is fetched from lower in the memory hierarchy and also replaces the one in cache.

In instances where the cache is not *direct-mapped*, more than one cache line can be in cache at once, but the lookup now less straight forward, as multiple cache lines need to be checked. Once a set is full we are also presented with a different issue, what cache line should be evicted when a new one is being added? The cache will need to make a choice on which cache line to replace when a new block is fetched from memory. It does this by following a *replacement policy*. The most commonly used scheme is called *least recently used*, or *LRU* [22]. With this

policy, the block which has remained unused for the longest time gets replaced by the new block. This policy seems very reasonable when considering how the cache utilizes temporal locality; the data that has been in the cache the longest without being used is the least likely to be used soon again. One can however imagine scenarios in which this is disadvantageous. Say we have a 4-way set-associative cache and a program that so happens cycle between using data from 5 cache lines that have the same index. With the LRU policy we will get a lot of cache misses, as the cache line that is evicted will be the next one the program accesses while in this cycle.

Another policy is the *Most Recently Used* (MRU) policy, in which the most recently accessed cache line is the first one evicted. This runs somewhat counter to temporal locality, but in the example we just constructed it would result in cache hits for roughly 4 out of every 5 accesses. There are also less contrived instances where MRU performs better than LRU [12].

There are several more policies, such as *First in First Out*, *Random Replacement*, etc. We will not go through the advantages or disadvantages of these policies here. This is just to point out that there are multiple different replacement policies, and they have an effect on how well a program can utilize the cache.

2.1.5 Write Policy

When a a piece of data in memory gets written to, it is first written into the cache, so that subsequent uses of that data utilize the cache. Write policy then determines whether the changed cache line is also written back to main memory directly, or if we wait until the cache line is about to be replaced and write to main memory then instead.

The first *write policy* is called *write-through*. With write-through, blocks changed in cache immediately gets written back to main memory. This can be slow, and potentially wasteful since the computer might not need the data yet.

The other policy is called *write-back*. With this policy, cache lines that are written to are *not* updated in main memory instantly. Instead, they are only written back to memory once the cache line is about to be replaced.

2.2 The C Programming Language

C is a general-purpose programming language developed around 1978 [19]. There is a lot that can be said about C, it is a powerful language with decades of history. For this report, however, we will focus on how C handles data structures and how it handles dynamic memory and its allocation. More on the reasoning behind this in section 3.1.

2.2.1 Structs

A *struct* is a compound data type in the C programming language. In its most basic form, it enables the programmer to define types with multiple values and/or types for each unit of this struct[27, p. 340]. These values are called **struct members**. Struct members may be of any type, including pointers, unions or other structs. A basic example of a struct and its use can be seen in listings 2.1 and 2.2.

Listing 2.1: Struct example

```
struct Position {
    float longitude;
    float latitude;
};
```

Listing 2.2: Using the position struct

```
int main()
{
    struct Position pos; // declaring a struct variable

    pos.longitude = 3.141f; // writing to a struct member
}
```

While the struct members are defined to follow each other in the order they are declared, the actual position in memory of each struct member is not as well-defined by the ISO C Standard [14]. This is because there *may* exist padding between the struct members. Padding is basically empty bytes put in between struct members to achieve alignment, which increases performance. According to the 2017 ISO C Standard, padding may be applied anywhere within the struct except before the first member [14, p. 115]. This means that the compiler decides where and how much padding should be applied. Often, *natural alignment* is used. This means that members want to be placed on an address thats a multiple of its size [27, p. 341]. In fig 2.4 we can see how padding is used to align struct members. Note that due to padding the size of a struct is affected by its layout. This can also affect what cache line the struct members are on, and whether or not they are on the same cache line or not.

2.2.2 Memory Allocation

The most common way of allocating memory outside of the stack is using the functions defined in **stdlib.h** [27]. These functions are listed in listing 2.3.

Listing 2.3: Memory Allocation Functions

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
void free(void *ptr);
```

These functions, generally, take the desired number of bytes as an input argument and return a pointer, an address, where memory of that size has been allocated. Using these functions, larger arrays of data and/or structures can be allocated on-the-fly. Once the memory is no longer in use, the memory can be deallocated (or freed) with a call to *free()*.

These functions often deal with memory containing structures, and as such it will be useful for the purposes of this thesis to track when these operations occur.

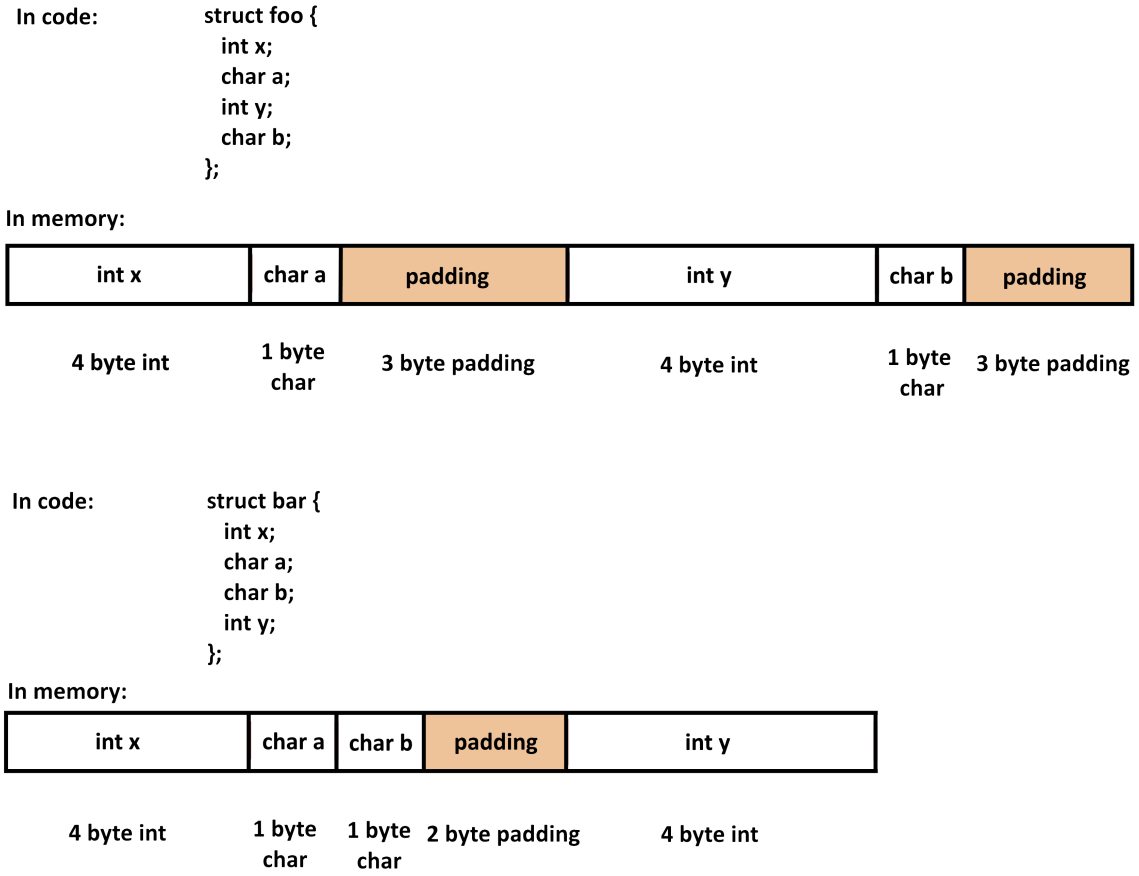


Figure 2.4: Example of how the two structs foo and bar would look in memory. Padding is used to align struct members and does not hold any data.

2.3 Linux Executables

A C program compiled under a Linux operating system will, in general, be compiled into an *ELF file*. ELF, *Executable and Linkable Format*, is a file format composed of different sections [1, p. 824]. There is a section for the machine instructions, a section for static data, a section for debug data, and many more. The most important section for this project is the debug data section.

2.3.1 DWARF Debug Data

The DWARF Debugging format is a format which is used to add debugging information to executable files [6]. Using the format, a debugger can see the names of variables, their sizes, their offsets and a lot more information to ease the debugging process. It needs this since most of the types and their names are abstracted away during the compilation process.

The DWARF data is very compact and compressed so as to not encumber the executable as much. It is also structured in a tree-like data structure. This structure is based upon nodes, called *Debugging Information Entries* or *DIEs* [6]. These entries have attributes which can help the user, or an automated tool as in our case, determine certain information about the variable, function, base type etc. that the DIE represent. In our case, we are interested in structs. In the debug data, a struct type will be represented by the DIE called *DW_TAG_structure_type*. This DIE will contain information about which line that declared it, which file that declared it and it might even have a size, in bytes, declared. Since the DIEs are organized as a tree, it makes perfect sense that the struct members are children to the structure type nodes. These DIEs, called *DIE_TAG_member*, contain information like offset, name and where its base type is defined [3].

One shortcoming of the format, or perhaps the C language as a whole, is that some structs *attributes*[3] might not be present in the corresponding DIE. For example, a *DW_TAG_structure_type* DIE *might* contain a *DW_AT_byte_size* attribute to indicate its size, but if the size is unknown, or not constant, it might be omitted from the attribute list. Such a case is the use of the C99 construct *flexible array members* [27], in which an array might be placed as the last member of a struct man may contain no set size [3].

In this project we will use this data to see the names and attributes of different variables and structures.

2.4 Code Instrumentation

With code instrumentation we are referring to the process of adding code into the program that is not present in the source code, with the aim of extracting information about the program during its execution. Code instrumentation is a common strategy to use for profiling. It can be done at various levels, with some tools such as the code coverage analysis tool Clover instrumenting the source code [28], while others such as Valgrind instrument the binary at runtime [20]. ThreadSanitizer, a tool for identifying race conditions and which will be important for this thesis, instruments the intermediary representation (IR) of the code at compile time in LLVM.

There are of course different advantages to instrumenting at different levels. Tools that instrument at the source level is generally easier to build, and (typically) retain the same level of portability as the original program. On the other hand, tools that instrument at the binary level can be reused for analysis of programs of multiple languages, as they can be instrumented by the same tool given that they are compiled to a common machine representation [9].

[3]Attributes [14] are a part of the C language that can be thought of as instructions for the compiler on how to deal with variables, functions, types, etc. For example, the struct attribute "packed" (specified as __attribute__((packed))) makes it so no padding is added. Attributes can be compiler and/or target specific, making them less portable.

Chapter 3
Approach

A system designed to fulfill the goals presented in 1.2 will consist of at least two main components (and likely more), a runtime to generate the trace and a simulator. This chapter will give a brief explanation of the purpose for each part and the method used for testing them, as well as include the scope for this thesis.

3.1 Tracing Execution

The first goal/requirement states the the simulator is to be *trace based*, i.e. we need to generate a trace of the programs execution. To do this, one either needs to modify the program in such a way that the program generates a trace while executing, or run the program in a virtual environment so that system calls and memory accesses can be intercepted and logged (thus the environment generates the trace file without needing to modify the inspected program). The trace itself should contain all the reads and all the writes that the program makes during its execution. Since the cache uses the address of the accessed memory to determine where in the cache it gets stored, the trace must note which addresses the read/write was performed on. Furthermore, it should record how many bytes were actually read/written, as if a chunk of memory straddles (or overlaps) 2 cache lines, two accesses are made in order to retrieve all the data.

The second goal states that the simulator should be able to change the order of struct members and simulate them as if they were changed in the source code. To do this, we must know 2 things: What structs are present in the code and where they are instantiated in memory. To solve the first question, the information about all of the structs needs to be extracted from the compiled program. This information can be extracted from the DWARF debug data and be formatted into a useful format. To resolve where the structs are located in memory during execution is more difficult. One approach is to annotate all operations regarding the struct, and record those in the trace files along with the memory access.

None of the pre-existing simulators discussed in the Related Works section provide a

way of generating a trace file as outlined here. As such, our approach in this thesis work was to write our own system for generating trace files. As the work involved in constructing a virtual environment, writing a virtual machine, is massive, we decided to instead use a (by us) modified version of LLVM to instrument the code along with a custom runtime environment to generate the trace files. The instrumentation occurs at the level of the intermediary representation (LLVM IR). The type data for the structs is extracted from the DWARF data in program executable using a custom python script. The implementation details of all this are provided in the next chapter.

The scope for tracing execution includes sequentially tracing reads and writes that occur in the program, including the memory addresses and size (in bytes) of those operations. However, we are not able to track indirect reads and writes that occur as a result of library calls, as these go outside the code we have instrumented. We are also specifically tracking structs, but due to time constraint we were only able to trace the memory address of structs allocated on the heap, and not those on the stack. Regarding the type data extraction, we were focused on information regarding the structs and their members. The reason why structs are prioritized in this way is because it is something that Sandvine has shown interest for and it is the most suitable for the reorder feature. We can not rearrange the layout of arrays, and unions are less common than structs in most code bases. We want to add support for tracking of these data types as well if time allows for it.

As a first step, the focus will be making the tracing work for the programs written in the C language and making sure that achieves our goals before potentially expanding to include C++ and other C-like languages. The decision to start with only C comes from both it being a smaller language and it, in theory, allowing us to extend the system to support other C-like languages more smoothly.

3.2 Simulator

The simulator should read the events in the trace and sequentially simulate how their execution affects the cache. These events can be reads, writes as well as struct allocation and deallocation.

The simulator should be able to construct a cache model on which to simulate the read and write operations. This cache model should behave as a theoretical cache. This means that it should keep track of accessed addresses and update its state according to the incoming events, but not actually track or store any data. For example, if it registers a read, it should check if the accessed address is already in the cache. If it is, it should register a cache hit and if it is not, it should register a cache miss. The cache model should be configurable so as to be able to model different cache configurations, meaning different replacement and write policies, cache sizes and cache line sizes, and even multiple levels of caches.

With the type data provided, the simulator should be able to use it to add further information about the allocated structs in the program. This data should be able to determine which struct members are accessed during the execution and other information about the structs, such as their names, their members' names and the size of their members. Furthermore, the simulator should be able to change the order of the struct members and correctly simulate how this affects the hit/miss rate.

The initial scope for the simulator does not include simulation of multi-core systems,

focusing instead on programs executing on a single core. If there is time to allow for it, simulating multi-core systems will be added to the scope. Also, the simulator will initially not track or estimate clock cycles spent performing the cache operations. This may be added if we have sufficient time, but our first priority is to as correctly as possible calculate the number of hits and misses in the cache. The initial scope also excludes the split of the L1 cache into data- and instruction cache, modelling it instead only as a data cache.

Chapter 4
Implementation: The Cacheray System

This chapter will cover all the parts that have been built and put together to use the simulator system, which we have chosen to call Cacheray. It will give insight into how the Cacheray works and how it can be used, as well as its limitations.

4.1 System Overview

For the purposes of instrumenting the code we decided to use LLVM as a base. LLVM is an open source compiler for C and C-like languages, and it has extensive documentation. Its modular design with passes that operate on the intermediary representation[1] (LLVM IR) make it relatively easy to add functionality such as instrumentation to the compilation process.

Figure 4.1 shows how the Cacheray system works and all the parts that are included in it. Beginning with the source program, we compile it using our modified version of Clang/LLVM, which includes the ThreadSanitizer (TSAN) pass and our custom LLVM pass, referred to as MallocTracker. Including TSAN and MallocTracker in the compilation process allows us to instrument the code. From this we obtain the instrumented executable of the program. By executing the instrumented program with our custom runtime we can generate a trace file. We also need to use the Dwarf2json script on the executable to extract the type data. Supplying the trace file and type data file, along with the configuration file (which specifies things such as cache size, line size, replacement policy, etc) to the simulator, a simulation can be performed. There is an optional fourth file that can be included, referred to as the *Reorder Structs file*, which allows for simulation of the same program but with struct members reordered. This reordering feature allows users to investigate how struct layouts affect cache

[1] LLVM compiles the source code into intermediary representation, then performs various optimization and analysis (based on what flags are specified with the compile command) before finally compiling the intermediary representation down to the machine specific instructions. The analysis and optimizations are performed using passes that run over the entire program or subsections of it.

utilization without having to rewrite the source program and starting the process over. The simulator finally outputs two files, one containing statistics on how many times the structs in the program were accessed, and one which contains the events the simulator simulated in a human readable format.

4.2 ThreadSanitizer

In 2.4 the ThreadSanitizer tool was briefly mentioned. In this section we will focus on what it is and how it is utilized with regards to the goals of this thesis. ThreadSaniter (TSAN) is a tool provided by Clang/LLVM, with the goal of identifying race conditions in C/C++ code. Race conditions are not very interesting with regards to cache performance, but TSAN is still interesting for the purposes of this work. TSAN basically consists of two parts: the TSAN runtime (which does the analysis) and the TSAN LLVM pass, which instruments the code so the runtime can do its analysis. The runtime we can do without, but the TSAN LLVM pass is interesting with its instrumentation. The TSAN pass is transforming the code by adding callbacks at points of interest. These callbacks are meant to be caught by the TSAN runtime for analysis. However, the points of interest for TSAN greatly overlap with the points of interest for this work, namely read and write operations.

To take advantage of the interesting parts of TSAN (i.e. the instrumentation done by the TSAN LLVM pass) we can compile a program with the -fsanitize=thread flag set, making the compiler include the pass in the compilation process. Then, instead of linking in the TSAN runtime we link in our own custom runtime which handles all the function calls added by the pass during compilation instead. The custom runtime is covered more thoroughly in 4.4.

It should be noted that this means the use of ThreadSanitizer in our thesis is not the intended use. As stated, TSAN is intended to identify race conditions, not to correctly mark all operations that interact with the cache. This is not necessarily a problem, but could potentially bring with it some unforseen quirks. The reason TSAN is used is simply because of the large overlap between the TSAN instrumentation and the instrumentation necessary for this thesis, meaning we free up some time that would otherwise need to be spent to get code instrumentation working.

4.3 MallocTracker

ThreadSanitizer does not collect all the events we are interested in. Some of the most important ones for the purposes of this thesis are the memory allocations on the heap. Information about the operations malloc, calloc, realloc and free is not collected by the TSAN pass. Since these calls are commonly used to allocate space for structs and other data structures we need to collect information about when and how these calls are made. It is for this reason that we have created MallocTracker. MallocTracker is a custom LLVM pass that allows tracking of the four aforementioned operations. If there is an instance of any of these operations (not just malloc, as the name might imply) a callback is inserted. The newly inserted callback reports what method was called, the relevant memory address, and (where relevant) the memory size. Similarly to TSAN, the custom runtime handles the added function calls.

It is important to note that MallocTracker has some limitations. The way the methods

are identified in the code is by identifying call instructions in the IR and then inspecting the name, checking if it matches as "malloc", "calloc", "free", or "realloc". Therefore it will miss any custom implementation of a method that manipulates memory on the heap. Solving this problem in general would not be possible using a pass since there is no way to directly identify memory manipulation on the heap in the LLVM IR. The core issue is that we are instrumenting the source program, not the library that implements the for instance the malloc method call. To solve this more generally would involve intercepting the sbrk and brk system calls, which would be doable with a virtual environment.

MallocTracker along with the TSAN pass make up what is referred to as the **instrumentation of the code**, in fig 4.1 these are the "LLVM pass" part. They are relevant at compile time in order to add in the function calls to the program that is being analyzed.

4.4 Cacheray Runtime

The custom runtime, referred to as the *Cacheray Runtime* (and sometimes as just *the runtime*), is important during program execution. The runtime is designed to be linked into an existing project that has been instrumented as described above. It is responsible for handling the callbacks added by the instrumentation, capturing reads, writes and struct creation events. These operations can then be written to a write buffer associated with the runtime. The size of the associated write buffer is configurable and can be arbitrarily large (it is limited by what the system the runtime is running on can handle rather than by the runtime itself). If a tracefile is desired, the library *cacheray-utils.h* has functions that can create one or more traces. The trace file is necessary in order to perform a simulation, so it is typically desired.

It is possible to pause and resume the recording at any point during the execution of the program. The runtime only writes to the write buffer while it is actively recording. While not recording, the function calls added by the instrumentation are still handled by the Cacheray runtime, which basically just receives and ignores them.

Everything the runtime records is regarded as an *event* in the trace file. The events have the following properties. All the events record which processor that the operation was performed on[2] (i.e. the processor the event occurred on) as well as **2** flags; one which says if the operation was an atomic operation or not and one that says if the read/write was unaligned or not. Each event is also categorized as being one of two different types.

The first type of event is the *Memory Event*. These are the simple events such as *read* and *write* but are also used for the *free* event. The Memory events contain whether it was a read, write, or free operation, the relevant address, and the size of the operation (in the free event, this is not used).

The second type of event is the *RTTA Event* (Run-Time Type Annotation) or *Struct Event*. Its purpose is to mark a certain part of memory which is deemed interesting. In this project, it is used to mark structs and arrays of structs. It tracks the amount of structs in the arrays and the variable name which represents the array in the code.

The reason events are separated into two types this way is to allow for the "Reorder structs" feature. A RTTA Event needs to be treated specially in the case where a reorder

[2]Though the system as a whole does not support simulating multicore systems, it was relatively simple to include this information in the trace format. It has therefore been included in case we add multicore support in the future.

file has been provided so that the new struct member layout can be accounted for in the simulation. The events are added in the order that they are created during execution. This means that they have no built in index and should be processed in the same order they are saved.

4.5 Typedata collection

The simulator needs some information about the different structs used by the program. The MallocTracker program will give the addresses and names of the structs contained within them, but no more. Some interesting information about structs that would be beneficial for the program to know are:

- Struct size
- Struct member types
- Struct member sizes

All this information about the program is stored within the executable file, in the DWARF format. It is therefore possible to extract this information from the executable, and use it to amend the simulators knowledge of the trace. This extraction process is accomplished by the script **dwarf2json.py**. It uses the library *pyelftools* [7] to read the DWARF data and translate it into a file with the data formatted according to the JSON standard [13]. This file can be referred to as a *type-data file*.

The type-data file contains 2 fields. The first field is a list of all the different struct types contained in the program. Each struct has a unique numeric id, a name, and a list of struct members. Each struct member has a numeric id, a name, a type, an offset (from the start of the struct) and a size. The second field is a list of typedef mappings. Each mapping shows the mapping from a typedef to a struct. This list exists to track the usage of the structs through potential typedefs.

This data file is very important for the simulation because it gives the simulator the ability to model the internal structure of the struct. This means that the simulator can see at what offset each struct member and what size they are.

4.6 Cacheray Simulator

The Cacheray Simulator is the core of this thesis. It is written in Java and is heavily object oriented in nature.

When simulating, the simulator first reads from the configuration file and sets up the cache accordingly. It then reads from the trace file generated by the cacheray runtime and simulates the effects that the recorded events would have on the cache. In the end it produces the Struct Statistics and the Trace even execution list.

4.6.1 Inputs

To function, the Cacheray Simulator requires three files, and optionally can take a fourth.

The first file is the trace file. It consists of a series of events of the two types described in section 4.4. The trace file does not need any specific header or anything identifying it as a trace file. Instead, the simulator performs a validation and will halt if it thinks that the trace file is malformed. There is support taking in multiple trace files, in which case they are simulated in the sequence in which they are provided.

The second file needed is the type data information file, also known as the *dwarf2json file*. This is the output from dwarf2json.py, described in section 4.5.

The third and final file that is required is the configuration file. It is written in a JSON fromat for ease of use, where the developer can specify configurations such as many caches to use, their sizes, their policies etc. An example 2-level cache configuration can been seen in listing 4.1. The sizes are specified as powers of 2, so in this case the L1 cache has a size of $2^15 = 32768$ bytes. The write policy is set with either a 0 or a 1, 0 being write-through and 1 being write-back. Set associativity is specified as the number of cache-lines per set, with 0 representing a fully-mapped cache. Replacement policy is specified with a number (0-4), each number corresponding to a different policy. These are, in numerical order from 0 to 4, LRU (Least Recently Used), FIFO (Fisrt In First Out), LIFO (Last In First Out), FILO (First In Last Out), and Random.

If the block size, cache size, and associativity are not compatible, eg if block size is greater than the cache size or the number of cache lines per set is an uneven number (other than 1), the simulator will not run the simulation and instead throw an error.

Listing 4.1: A Cacheray configuration file

```
{
  "caches" : [
    {
      "name" : "L1",
      "cache_size" : 15,
      "block_size" : 6,
      "write_policy" : 0,
      "set_assoc" : 8,
      "rep_policy" : 1
    },
    {
      "name" : "L2",
      "cache_size" : 18,
      "block_size" : 6,
      "write_policy" : 0,
      "set_assoc" : 8,
      "rep_policy" : 1
    }
  ]
}
```

There is also the possibility of adding a fourth *member re-order* file. This file changes the member order of one or more structs defined in the type data information. This is used for

the reorder feature. An example file can be seen in listing 4.2. In the reorder file, the user lists the structs intended for reorder and give pairs of variable names and offsets. The offset tells which new offset the named variable should have. For example, in the file in listing 4.2, for struct *A*, the struct member *a* will be placed at offset 0 and struct member *b* will be placed at offset 4.

Listing 4.2: Remap file

```
1 struct A=a,0;b,4;pad,8;pad2,68
```

4.6.2 How it works

Once all files are correctly formatted and the program is started, the actual process of running the simulation can begin. First, the simulated cache is set up in accordance with the configuration file. Then the events are played up in the order that they arrive, i.e. the order that they were written to the trace file. They are decoded and are simulated in the cache system, meaning that each read/write operation is passed to the simulated cache, and the effects of that operation on the cache are calculated.

The simulation does not actually fetch and store any data that the original program would have used. Instead it simulates this process by keeping track of which cache lines' indices are loaded into the simulated cache. A read then amounts to simply checking whether the desired cache line(s) are in the simulated cache or not, and if not causes an update. Every cache hit and miss is tracked by the simulator in order to produce useful output at the end.

The way the simulator counts hits is simply to check whether a desired cache line is in cache. It does *not* count individual bytes of a read as individual hits. A request for 4 bytes of data will result in one hit or miss. There is an argument to be made for counting individual byte hits, but as one of the features is to give statistics on how struct members have interacted with cache (giving hit/miss rates for individual struct members), we believe counting bytes simply becomes more confusing than helpful.

4.6.3 The output

Once the simulator has finished processing the event, an output is created. The output has two parts, constituting two separate output files.

The first part of the output is the *struct statistics*. In this output, the structs affected by the execution are listed with individual access statistics for each struct type. This information includes reads, writes, member reads/writes and hit/miss statistics. This part of the output is the most pertinent to the programmer looking to improve a programs cache utilization.

The second part of the output is a *trace event execution list*. It lists the events carried out on the simulator in chronological order, with all their information included. This can be used to debug the simulator, as well as doing certain information look ups, post simulation.

4.6.4 Limitations

There are some limitations to the simulator that are worth reiterating. Firstly, the simulator cannot simulate a multi-core setup. It assumes the program runs on a single core and bases the

simulated cache interaction on this. The Cacheray runtime is basically capable of generating a trace for a multi-core system, and a future improvement on the simulator would be to add support for this. It would be time consuming to do this, however, and as it is not a core feature that is necessary for this thesis it has not been included. It can be added here that a shared cache is completely outside the scope of this project.

The simulator does not take into consideration any split in the L1 cache into instruction cache and data cache. Instead the cache only handles data and results regard only hits and misses on data in cache. Adding consideration to instruction cache would no doubt provide interesting results regarding scheduling and such, however since this is not the focus or goal of this project it has been excluded.

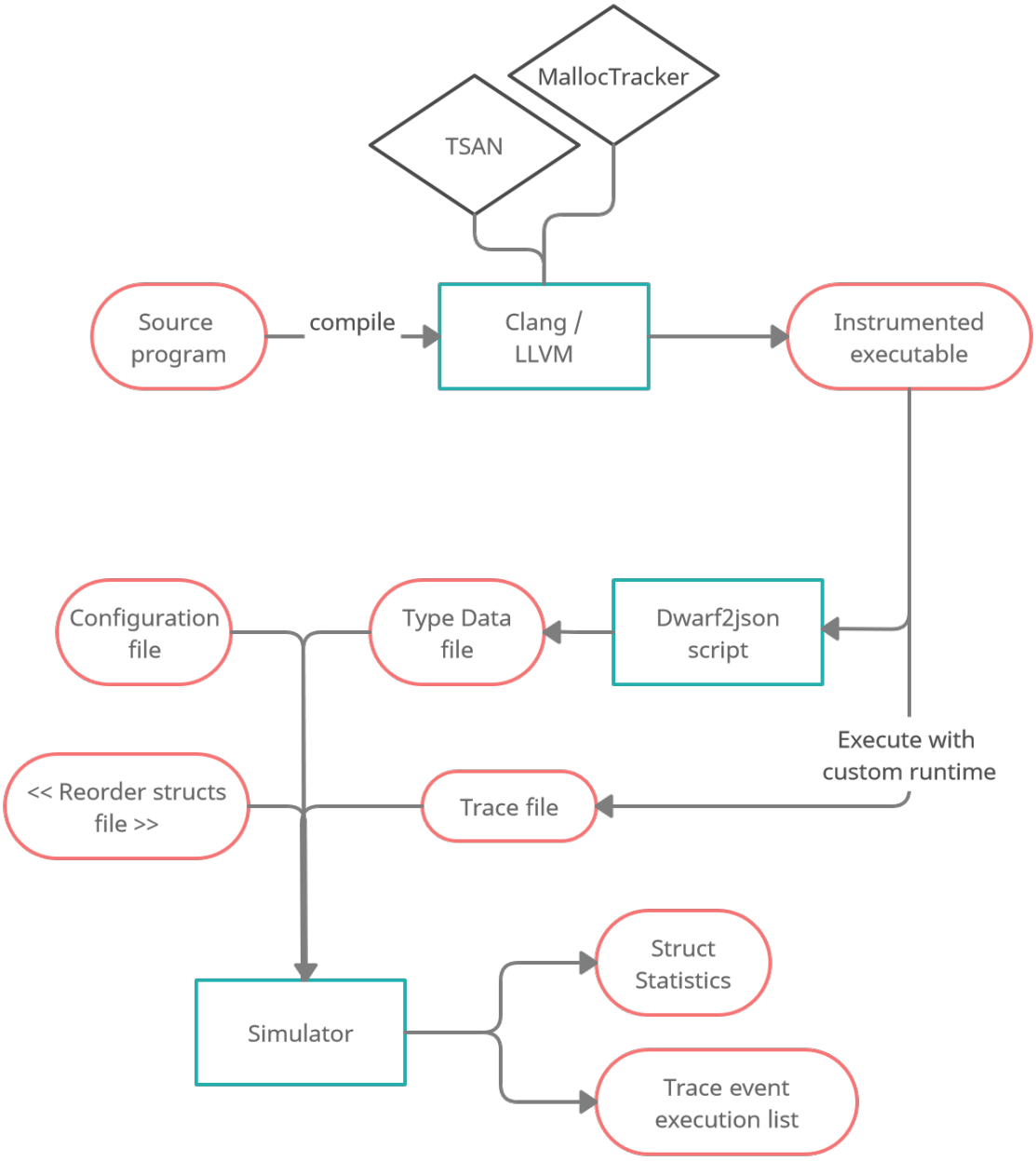


Figure 4.1: Overview of the flow of using the cacheray system. The red rounded boxes are files (also notes that the files on the left side of the graphic are provided by the user, while the other files are generated in the process of using cacheray). Observe that the "« »" brackets on the "Reorder structs file" indicate that this file is optional. The cyan boxes are programs or scripts that are used in the process. The black rhomboids are additions to LLVM.

Chapter 5
Testing the System

This chapter presents the testing procedure that was used to obtain the results presented in chapter 6. It will also explain why these procedures are a good way of testing the Cacheray system and what types of metrics might be interesting to know about the simulator. Finally, it will describe in detail what supporting programs were used and how they were used. They are described so as to allow future researchers to reproduce these results.

5.1 Testing approach

The testing should be set up in such a way that the results allow us to answer our research questions posed in section 1.1 properly. As such we are dividing our testing into three categories, related to the runtime and the simulator.

1. Resource requirements and usage (Cacheray runtime)
2. Simulator correctness (Simulator)
3. Struct reorder feature correctness (Simulator)

These three contain multiple tests and measurements, which will be further explained in the following subsections.

5.1.1 Resource Requirements and Usage

To determine the resource requirements and resource usage for the system, there are a few interesting measurements to make.

The first is **storage overhead**. This is the extra space the program compiled and linked with the Cacheray runtime will take up. To measure this, the program is compiled with and

without the instrumentation and linking. The sizes are then compared to see what overhead the system requires.

The second interesting metric is **memory usage**, specifically peak memory usage during the execution of the instrumented programs. This will be measured by running the program, with and without instrumentation, under a memory-profiler and observing the peak memory usage. The programs will run without saving and producing a trace in order to make the results easier to analyze. The results can then be compared against each other to see the memory usage requirement.

The **size of the trace files** that are generated is also interesting from a resource perspective. The sizes of these traces can be measured as soon as the instrumented program is done executing and the trace has been created.

The **execution time** of the instrumented programs as compared to their non-instrumented counterparts is also one of the more interesting metrics to collect. To measure this, both the instrumented and the non-instrumented program should be executed, with the same inputs, and the difference in execution time should be measured. The execution, of both the instrumented and non-instrumented programs, should be performed several times to yield the best, worst, and average execution time, as well as the standard deviation of times. A special program will be used to achieve this, which we will go over in section 5.2.3.

5.1.2 Simulator Correctness

Perhaps the most important feature of a simulator is that it is **correct**. This brings up an interesting question, how would one measure how correct Cacheray is when determining hits and misses?

We can not simply inspect the physical cache hardware during program execution and compare the actual behavior to the behaviour reported by the simulator. Instead, to determine whether the simulator is correct or not, it will be compared against the results of an established cache simulator. In doing this, we will assume that the chosen established cache simulator is correct, or at least sufficiently correct for our purposes.

To this end we have chosen the simulator Cachegrind, which is a part of the performance testing suite Valgrind [5]. Since Cachegrind is an on-line simulator, it will be used with the non-instrumented test program and its results will be compared to the Cacheray Simulator results from the instrumented program's trace.

5.1.3 Reorder Correctness

Since one of the features of the simulator is to change the ordering of struct members, it is pertinent to also test this.

To measure the correctness of the reorder operation, a test different to the test for simulator correctness is used. A program with a particular struct will be compiled and executed twice. The second time around, the particular struct will have a different layout of its members. Everything else will be the same for the two executions. The layout change will be made in a way that, in theory, gives different cache performance for the same number and order of read and write operations. The programs will be compiled into instrumented and a non-instrumented versions.

The instrumented versions of the programs will produce traces to be used by the Cacheray simulator. The non-instrumented version of the programs will be measured by Cachegrind. The results from the Cacheray and Cachegrind simulations will then be compared to ensure that any performance difference noticed by Cacheray is also noticed by Cachegrind. We do this to establish that the difference in struct layout has a noticeable effect on the performance of the program, one struct layout being better and one being worse. If there is a clear difference for both Cacheray and Cachegrind, this means that we can make valid conclusions regarding the reorder feature independently of whether or not the Cacheray simulator is simulating correctly.

Finally, the Cacheray simulator's reorder feature will be used. It will change the layout of the particular struct for the trace with worse cache performance into the layout of the struct in the instance with better cache performance. The simulator will perform a simulation with the trace with worse cache performance and the reorder file. Then, the Cacheray simulator will simulate the trace from the other program, the program containing the struct that the reorder feature is attempting to form. The results of both of these tests will be compared to see if the reorder feature has achieved a cache performance similar to the program with the better struct layout. If the reorder feature works as intended we should expect to see the same output for both simulations.

5.2 Test Bench

This section describes which programs were used for the different tests and all configuration information needed to replicate these tests.

5.2.1 Test programs

There are **5** different programs which will be used to test the Cacheray simulator and runtime. **3** of the programs will be existing application which will be instrumented with Cacheray, and 2 of them will be written by the authors. The **3** already existing programs are:

- gzip
- bzip2
- oggenc

These were chosen because of their ease of compilation and for their relatively small size. The programs were downloaded from `https://people.csail.mit.edu/smcc/projects/single-file-programs/`.

The **2** programs written by the authors are:

- colrow-copy
- struct-copy

These programs were written to test specific cache effects and performance changes. The following sections contain a more detailed description of the programs and how they will be used for testing.

gzip

The program **gzip** is a compression utility program [8]. It takes an input file and attempts compress it down to a smaller file size.

During the tests, the program will be run in test mode, which checks the file integrity of gzip files. The file it will test is a small file filled with arbitrary (random) data, which has already been compressed by gzip.

Listing 5.1 contains short snippet that shows the command used to perform the test:

Listing 5.1: gzip test

```
1 $ ./gzip -t bin.gz
```

bzip2

The program **bzip2** is also a compression utility program [24]. The bzip2 tests will be performed in the same way the gzip tests are performed. See the short snippet in listing 5.2.

Listing 5.2: bzip2 test

```
1 $ ./bzip2 -t bin.bz2
```

oggenc

The program **oggenc** is an audio encoding program that encodes different audio formats into the Ogg Vorbis audio format.

To test the program, it will be given a short audio file in the WAVE format. It will then encode the audio into the Ogg Vorbis format and discard the output. See the short snippet in listing 5.3.

Listing 5.3: oggenc test

```
1 $ ./oggenc -Q test.wav -o /dev/null
```

colrow-copy

The program colrow-copy is a special program, written specifically for this project. It is intended to show the effects of program structure and how it affects cache performance.

There are 2 different versions of colrow-copy. Both versions perform the same number of reads and writes but do it in different orders. The ordering should give rise to a difference in cache performance between the versions.

struct-copy

Like colrow-copy, struct-copy is a special program, written specifically for this project. It is intended to show a potential difference in cache performance where the only difference is the ordering of certain struct members.

Also like colrow-copy, there are 2 different versions of struct-copy. Both version perform the same number of reads and writes. The difference between the program lies in the layout of the structs used.

5.2.2 Compilation and Configuration

The test-programs are compiled with clang version 10.0.1 with MallocTracker compiled in and enabled.

There are a lot of flags during compilation. The base flags are **-g -O0**. These are all the flags needed by the non-instrumented files. **-g** enables debug data, which is needed by dwarf2json.py to extract the type data. **-O0** sets the optimization level to minimum. This is to prevent loop-optimizations and other optimizations which might make the result unpredictable.

The instrumented files also used a few extra flags. These are **-fsanitize=thread -Xclang -load -Xclang LLVMMallocTracker.so**. The **-fsanitize=thread** flag enables ThreadSanitizer instrumentation which is needed to capture read and writes. **-Xclang -load -Xclang LLVMMallocTracker.so** loads and enables the MallocTracker LLVM pass which allows for recording further information regarding allocations on the heap.

5.2.3 Resource Test Configuration

The first and easiest resource metric to test is the size difference. This is done using the linux utility program **ls**. This method was also used to obtain the sizes of the traces. See listing 5.4. This will display the sizes of the executables in bytes.

Listing 5.4: List file sizes

```
$ ls -l *.out # list executable sizes
$ ls -l *.trace # list trace sizes
```

The execution times are collected using the program **hyperfine** [25]. It is a benchmarking program which runs a program several times and averages the execution time as well as displaying the standard deviation of these results. The version of hyperfine used is 1.11.0. Listing 5.5 shows the commands for running the program.

Listing 5.5: Execution time testing

```
$ hyperfine --warmup 3 --runs 100 --style basic \
      './gzip.norm -t test.gz' # instrumented run
$ hyperfine --warmup 3 --runs 100 --style basic \
      './gzip.ins -t test.gz' # non-instrumented run
```

The peak memory usage is tested by a tool called **massif**, which is a part of the valgrind suite. This tool measure heap memory usage over the program life-time. The version used is 3.16.1. To extract the max memory usage, the commands in listing 5.6 were run. The first line generates the memory usage over the programs life-time. The second line finds the largest value, which should be equal to the largest amount of memory allocated.

Listing 5.6: Memory usage testing

```
$ valgrind -q --tool=massif --pages-as-heap=yes \
      --massif-out-file=m.out ./gzip.norm
$ grep mem_heap_B m.out | sed -e 's/mem_heap_B=\(.*\)/\1/' | sort -g | tail -n 1
```

5.2.4 Simulator Configurations

Several cache configurations will be used for both Cacheray and the reference simulator (Cachegrind). Each run will vary the line size and the associativity of the cache.

The first configuration item is the line size. This is the size, in bytes, of each cache line. The sizes are: 32, 64, 128 and 256 bytes.

The second configuration item is the cache associativity. This is the amount of lines each associativity group has. The amount of lines tested are: 1, 2, 4 and 8.

For the Reorder Test, the type data from the struct_copy.out executable is extracted by using the dwarf2json.py program, see listing 5.7. The resulting file, *struct_copy.dwarf.json*, is then used as an input to the simulator.

Listing 5.7: Type data extraction

```
1 $ dwarf2json.py struct_copy.out
```

5.2.5 Cachegrind Configuration

The simulation and displaying of the result can be done with the commands shown in listing 5.8. The program cg_annotate is part of the valgrind suite, and is used to display the results from the run. The program is particularly useful since it enables line-by-line inspection of the tested program, with read, write and cache interaction information for each line of the source code [4]. The version of valgrind and cg_annotate is 3.16.1. In the tests, the total number of read (marked as **Dr** in cg_annotate) and total number of cache misses in this first cache (marked as **D1mr**) will be used to compare the simulators. This feature needs to be used to accurately compare Cachegrind and the Cacheray Simulator since Cachegrind measure more of the program than Cacheray does.

Listing 5.8: Running Cachegrind

```
1 $ valgrind --cachegrind-out-file=a.out.cg \
2             --tool=cachegrind \
3             --D1=<size>,<associativity>,<line size> \
4             ./a_n.out &> /dev/null
5 $ cg_annotate a.out.cg # displays cache information
```

Chapter 6
Results

In this chapter, the results of the tests are presented. First the results relating to resource usage are presented, then the comparative simulation results. These results are then to be referenced in the discussion in chapter 7.

6.1 Resources Usage Results

Table 6.1 shows the file sizes of the instrumented and non-instrumented executables, ordered by non-instrumented size ascending. It also shows the relative size difference of the files (instrumented / non-instrumented) in both absolute and relative terms.

Table 6.1: Static File Size (in bytes)

File	Non-Instrumented Size	Instrumented Size	Size difference
colrow-copy	17560	51744	34184 (2.95)
struct-copy	18104	57072	38968 (3.15)
gzip	154928	275208	120280 (1.78)
bzip2	191920	448272	256352 (2.34)
oggenc	2466744	2926336	459592 (1.19)

Table 6.2 shows the memory usage by the instrumented and the non-instrumented executables, ordered by non-instrumented usage ascending. It also shows the difference between them, as a percentage.

Table 6.3 shows the file sizes of the trace output by Cacheray, when run on different commands.

Table 6.5 and table 6.4 show the results of the execution time experiments. Table 6.5 shows the results for the non instrumented version of the programs, while table 6.4 shows the results for the instrumented version. The results in all both tables have had their values rounded to 4 significant digits.

Table 6.2: Max Memory Usage (in bytes)

Command	Non-Instrumented	Instrumented	Difference
colrow-copy	17560	51744	34184 (2.95)
struct-copy	18104	57072	38968 (3.15)
gzip	154928	275208	120280 (1.78)
bzip2	191920	448272	256352 (2.34)
oggenc	2466744	2926336	459592 (1.19)

Table 6.3: Traces Sizes (in bytes)

Command	Size
colrow-copy (Good/Bad)	56567808
struct-copy (Good/Bad)	74907707
gzip	10047690
bzip2	229266270
oggenc	180543996

Table 6.4: Execution time (Instrumented) (in seconds)

Command	Min	Max	Mean	Std. Dev.
bzip2	1.564	1.600	1.579	0.007280
gzip	0.05847	0.09026	0.07619	0.003504
oggenc	1.219	1.444	1.238	0.02349
colrow-copy (Good)	0.3911	0.4083	0.3994	0.003420
colrow-copy (Bad)	0.3274	0.4058	0.3770	0.006465
struct-copy (Good)	0.4272	0.6776	0.4985	0.01205
struct-copy (Bad)	0.4289	0.6501	0.4989	0.01461

Table 6.5: Execution time (Not Instrumented) (in seconds)

Command	Min	Max	Mean	Std. Dev.
bzip2	0.006981	0.01324	0.01046	0.001996
gzip	0.001021	0.002787	0.001547	0.0003154
oggenc	0.01074	0.01844	0.01549	0.002107
colrow-copy (Good)	0.004783	0.01144	0.007726	0.001777
colrow-copy (Bad)	0.005091	0.01245	0.007896	0.001548
struct-copy (Good)	0.04849	0.07202	0.05238	0.001939
struct-copy (Bad)	0.04866	0.06117	0.05258	0.001885

6.2 Simulation Results

Table 6.6 shows the test programs, without instrumentation, run on the established simulator *Cachegrind*. Table 6.7 shows the results of the Cacheray tests. The table shows the miss rates of all the different test programs, using different cache setups. The miss rate output from cachegrind is rounded to one decimal place, so we have decided to round the cachegrind

results in the same way to make comparisons between the two easier.

Table 6.6: Missrate of various programs for different cache layouts according to Cachegrind

Cache size	16384															
Associativity	1				2				4				8			
Line size	32	64	128	256	32	64	128	256	32	64	128	256	32	64	128	256
bzip2	0.6	0.6	0.8	1.0	0.4	0.4	0.4	0.6	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4
gzip	0.7	0.7	2.5	4.6	0.5	0.3	0.3	0.6	0.5	0.3	0.2	0.1	0.5	0.3	0.2	0.1
oggenc	2.5	2.3	7.6	7.9	0.4	0.4	0.4	0.5	0.5	0.4	0.3	0.3	0.2	0.1	0.1	0.0
colrow-copy (Good)	1.1	0.6	0.5	0.5	1.0	0.5	0.3	0.1	1.0	0.5	0.3	0.1	1.0	0.5	0.3	0.1
colrow-copy (Bad)	4.6	4.3	4.3	4.4	4.5	4.3	4.1	4.1	4.5	4.3	4.1	4.1	4.5	4.3	4.1	4.1
struct-copy (Good)	3.5	2.1	1.4	0.8	3.5	2.1	1.4	0.7	3.5	2.1	1.4	0.7	3.5	2.1	1.4	0.7
struct-copy (Bad)	4.1	2.8	1.4	0.8	4.1	2.8	1.4	0.7	4.1	2.8	1.4	0.7	4.1	2.8	1.4	0.7

Table 6.7: Missrate of various programs for different cache layouts according to Cacheray

Cache size	16384															
Associativity	1				2				4				8			
Line size	32	64	128	256	32	64	128	256	32	64	128	256	32	64	128	256
bzip2	0.7	0.8	0.9	1.2	0.9	0.9	0.9	1.1	0.7	0.7	0.6	0.8	0.2	0.1	0.0	0.0
gzip	1.1	0.9	5.6	10.7	0.9	0.4	0.2	0.1	0.9	0.4	0.2	0.1	0.6	0.3	0.1	0.0
oggenc	0.8	0.8	0.9	1.2	1.1	0.8	0.4	0.3	0.3	0.2	0.3	0.1	0.1	0.0	0.0	0.0
colrow-copy (Good)	6.2	3.1	1.5	0.7	12.5	6.2	3.1	1.5	6.2	3.1	3.1	0.7	6.2	3.1	1.5	0.7
colrow-copy (Bad)	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0
struct-copy (Good)	0.3	0.3	0.3	0.1	0.3	0.3	0.3	0.1	0.3	0.3	0.3	0.1	3.8	3.8	3.8	1.9
struct-copy (Bad)	0.7	0.7	0.3	0.1	0.7	0.7	0.3	0.1	0.7	0.7	0.3	0.1	7.6	7.6	3.8	1.9

6.3 Reorder feature correctness

Table 6.8 shows the results of the Cacheray reorder tests. As a reminder, the **Good** and **Bad** program types are ostensibly the same program but with a different struct member order on the structs which they manipulate.

Table 6.8: Reorder Correctness Table

Program (type)	Accesses	Hits	Misses	Miss Rate
struct-copy (Good)	2080768	2072576	8192	0.3937%
struct-copy (Bad)	2080768	2064384	16384	0.7874%
struct-copy (Bad) (into Good)	2080768	2072576	8192	0.3937%

Chapter 7
Discussion

In this chapter we will discuss the results presented in chapter 6. The three core testing categories will be covered: resource usage, simulator results and struct reorder results. These results will be discussed and, for simulator results, comparisons between Cacheray and Cachegrind will be made. Finally, there is a section on possible error sources which may have affected the results of the tests.

7.1 Resource Usage

In our tests we have covered resource usage in terms of four main areas: file size, memory usage, trace size, and execution time. The results are discussed and analyzed below.

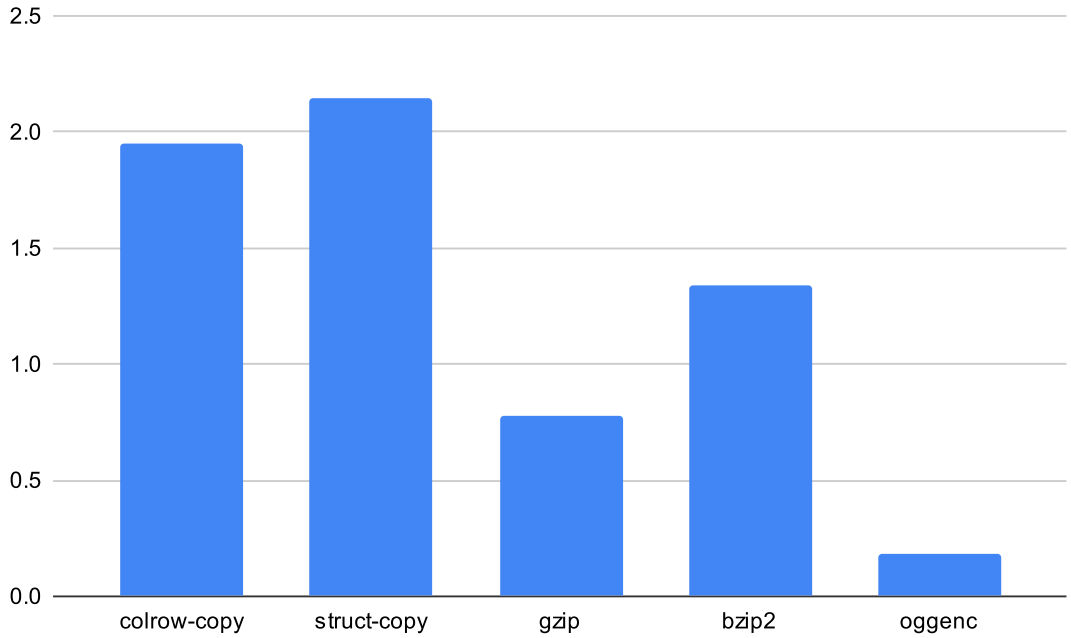
7.1.1 File Size Difference

From the data in table 6.1, we can see that the size of the compiled executable increases when instrumentation is added. This is to be expected, as the Cacheray runtime and the extra function calls, that are compiled in for instrumentation, will take up additional space in the executable. We should expect to see a consistent static addition to the executables size from the inclusion of the runtime, as well as a dynamic size addition from the added function calls. As described before, these function calls are put in wherever the program makes calls that can affect the cache, meaning that this size addition scales as the number of cache affecting, e.g. read and write, operations in the program increases. Figure 7.1 shows a plot of the size of the non-instrumented executables against the instrumented executables, based on the data in table 7.1. The relative size differences of the instrumented and non-instrumented executables decreases as the size goes up. In other words, for small programs the size difference is noticeable, but becomes less significant for larger programs.

Table 7.1: File size difference (in bytes) and relative difference. Relative difference is calculated as {instrumented file size}{non-instrumented file size}

File	Size difference	Relative Difference (%)
colrow-copy	34184	2.95
struct-copy	38968	3.15
gzip	120280	1.78
bzip2	256352	2.34
oggenc	459592	1.19

Figure 7.1: File size comparison between instrumented and non-instrumented versions of the programs



7.1.2 Memory Usage

Comparing table 6.2 and table 6.1, one can observe that the memory usage is the same as the file size. This is because the entire file is loaded into memory, and no more. Since Cacheray, in test mode, discards all generated data and does not store any trace data for later use, the memory usage will be the same as the file size. However, this will of course change if data is to be held before writing to disk. In that case, the memory usage will grow as the trace gets temporarily saved to memory, pending writing to the trace file. Since Cacheray is configurable to store an arbitrary amount of memory before writing to file, the amount of memory usage itself will depend on the configuration (for example, if we set the write buffer to 1kB, the memory overhead will increase by 1kB).

What this test shows that there is no extra memory usage incurred by Cacheray, with the

exception of the extra space taken by the instrumentation and the trace data temporarily held in memory. The increased memory overhead can be contrasted with the increase expected when running a program with ThreadSanitizer. The increase then is expected at around 5x-10x. In our case, we stripped away the ThreadSanitizer back-end and substituted with the Cacheray runtime.

7.1.3 Trace Sizes

The trace file that is generated by the Cacheray runtime is quite large. Looking at the results in table 6.3, we see that for a program such as *bzip2* the file size reaches a size of 229266270 bytes (almost 230MB). Despite the program not being all that big, the trace file is relatively large. In certain cases this may be a limiting factor, as the trace simply becomes to large to be practical when increasing program size and/or duration of analysis. This problem can to an extent be mitigated by the fact that the tracing can be turned on and off (see section 4.4). This means that if there is a particular section of the program that is interesting, you can setup Cacheray to only record to the trace when executing in that section. This will reduce the size of the trace file, but the trade off is then that the simulation becomes less correct.

The simulator has no way of knowing the state of the cache during unrecorded segments, so the start of each recorded segment will have an inaccurate layout of the cache in the simulator. We can either ignore this fact, meaning that we basically assume that the state of the cache at the end of one segment is the same as the state of the cache at the beginning of the next segment, or we can insert a cache flush command, meaning we assume the cache is empty at the start of each segment. Both of these are inaccurate with regards to the actual state of the cache at the start of each segment. In our implementation we decided to go with the former approach, since neither seems to provide an advantage over the other.

7.1.4 Execution Time

Finally, in table 6.4 and table 6.5, the different execution times for instrumented and non-instrumented versions of the programs can be seen. The interesting point here is the increase in execution time from non-instrumented to instrumented versions. The relative increase, calculated as execution time of the instrumented version over the non-instrumented, can be sen in table 7.2.

The increase in execution time for the instrumented versions can be quite large, ranging from 9.5 to 150 times their non-instrumented counterparts, which is quite noticeable. In the tests for execution time we are not actually writing to a trace file, so the increase in time is not due to added I/O operations. This also means that the execution time is further increased when actually writing the trace file.

It is difficult to pinpoint a definitive reason as to why the execution time increases so drastically here, but we believe there are two main contributing factors. The first is simply the increase in instructions to execute. The instrumentation adds in callbacks to the Cacheray runtime at every read/write, causing several more instructions to be run any time a read/write is executed. The second factor is that our system affects the cache performance of the program we are investigating. Our runtime has several read/write operations that can cause cache evictions, which of course affects how efficiently the original program uses the cache. Note that this doesn't affect our results, as we are tracing the operations and then simulating the

cache interactions, so the actual state of the cache during execution and trace generation is not really relevant.

Some slowdown is to be expected when performing this kind of analysis, but reaching execution times 150 times greater than the original may be going to far. For reference, one source [31] suggests that Cachegrind slows down execution by about 50 times, but that also includes analysis and generating the relevant results. ThreadSanitizers runtime can be expected to slow down program execution by 5-15 times [29]. This large decrease in execution speed limits the possible use cases for Cacheray, since long-running programs might become to slow to efficiently analyze.

Table 7.2: Comparison of Mean Execution Times

Command	Time Increase
bzip2	150.0
gzip	48.48
oggenc	79.92
colrow-copy (Good)	51.70
colrow-copy (Bad)	47.75
struct-copy (Good)	9.517
struct-copy (Bad)	9.488

7.2 Correctness of Simulation Results

To show the degree of correctness for Cacheray, two data sources will be used. Firstly, some observations to see if the amount of read/write events captured by Cacheray is correct, with regards to the input program. Secondly, a direct comparison between the calculated miss-rate of the programs when run on both Cacheray and Cachegrind.

7.2.1 Captured events in Cacheray

Figure 7.2 contains the critical loop which is performed by colrow-good. This section of code is what will be captured by Cacheray.

Figure 7.2: Critical loop of colrow-copy (Good)

```
1 int i, j;
2 // begin recording...
3 for (i = 1; i < rows; i++) {
4     for (j = 0; j < cols; j++) {
5         array[i * cols + j] = array[i * cols + j] + array[(i - 1) * cols + j];
6     }
7 }
8 // end recording.
```

Using this, we can calculate the hypothetical amounts of reads and writes. All of these will come from the row iteration loop and the column iteration loop, both of which have the same number of reads and writes. The amount of reads in the test program can therefore be calculated with:

$$N_{reads} = 2N_{cols}(N_{rows} - 1) \tag{7.1}$$

Conversely, the amount of writes can be estimated with a similar equation:

$$N_{writes} = N_{cols}(N_{rows} - 1) \iff N_{writes} = 2N_{reads} \tag{7.2}$$

With $N_{cols} = 1024$ and $N_{rows} = 1024$ we get $N_{reads} = 2095104$ and $N_{writes} = 1047552$. After running the Cacheray Simulator and looking at the output, there are 2095104 reads and 1047552 writes. Since the theoretical number of reads and writes match the obtained number of reads and writes, we can assume that the simulator is collecting the interactions correctly.

To make it easier to see the difference between the read and write captures between Cacheray and Cachegrind we can compare the outputs of the two tools for the code above. Table 7.3 shows the amount of reads, writes and the miss rates obtained by both Cacheray and Cachegrind. What is clear to see is that Cachegrind registers a lot more reads and writes than Cacheray does.

Table 7.3: Captured Read and Writes from *colrow-copy*

System	Reads	Writes
Cacheray	2095104	1047552
Cachegrind	17861900	8398057

The difference is to be expected, as Cacheray depends on traces generated by code instrumentation, while Cachegrind performs live analysis at runtime using Dynamic Binary Instrumentation [21]. Since Cacheray collects information based on instrumented source code this in turn means that code outside the instrumented source code, such as code in imported libraries, is not included in the simulation. Therefore, if a program imports functions from an external library (that is not itself also instrumented), the cache effects of that function will not be recorded in the trace, and are consequently missed in the Cacheray simulator. This limitation is not present in Cachegrind.

A perhaps even more succinct showing of the difference between Cacheray and Cachegrind is the result of analyzing a virtually empty program: `int main{ return 0; }`. Here Cacheray records 0 reads and writes, while Cachegrind records 23546 reads and 10,621 writes. The source code doesn't have any read or write operations, which is why Cacheray doesn't record any. However, when compiled, the program includes several commands in order to become runnable that aren't visible in the source code, which are captured by Cachegrind.

7.2.2 Cacheray and Cachegrind

Tables 6.6 and 6.7 show the miss rate of the test programs, using different cache configurations, according to Cachegrind and Cacheray respectively. To see if Cacheray has some

level of correctness, a comparison needs to be made between the results from the two tools. Looking at and comparing the tables, it is immediately clear that they don not have the same results, which was discussed in section 7.2.1 above.

However, there are still some valuable observations we can make when comparing the results of the two tables. There are certain trends which appear in both the Cachegrind and the Cacheray results. For example, in both the Cachegrind and the Cacheray tests, the performance degradation of colrow-copy when comparing the good and the bad version can be observed. The same can be seen in the comparison with struct-copy-good and struct-copy-bad. When increasing the line size, both Cachegrind and Cacheray observe the same decreasing miss rate. Perhaps the strongest point of comparison is the results for gzip. Both Cachegrind and Cacheray show significant increases to missrate as line size increase with a 1-way associative cache. The trends also overlap for the other associativities, with one exception being that Cachegrind has an increased missrate when increasing line size to 256 bytes for a 2-way associative cache, whereas Cacheray shows a decreased missrate.

The results for colrow-copy (Bad) in Cacheray does warrant a further comment. From the results, it appears that it always produces a 50% miss-rate, regardless of cache configuration. Although this might look quite strange, it is not a clear indication of error in the simulator, since the program was deliberately constructed to perform poorly with the cache.

There are also some results that show dissimilarities between the systems. While some numbers change in the same direction as the line size and/or associativity increases, the actual miss rates are not near close.

7.3 Reorder Feature

Looking at table 6.6 and table 6.7, there is a clear difference in the miss-rate between the optimized and non-optimized version of struct-copy, when run on the Cacheray simulator. The non-optimized version of struct-copy (Bad) has a higher miss-rate than the optimized version of struct-copy (Good).

When the reorder feature is used, to turn the bad struct into the good struct, the results can be seen in table 6.8. It is clear the non-optimized program achieved the same number of misses, and therefore the same miss-rate, as the optimized version. This shows that the reorder feature works as intended. Futhermore, it achieves the original goal of being able to change the underlying structs of a program without performing a change and recompilation of that program.

7.4 Possible Error Sources

As mentioned in earlier sections, ThreadSanitizer is a tool intended to catch race conditions. In this thesis it has been used to instead identify reads and writes that occur in a given program. As this is not the original intent of the tool, it is not guaranteed to correctly do this. In fact, at a very late stage in the project we came across an issue that was the result of TSAN not instrumenting the code in the way we had expected. Using the += operator makes TSAN miss a read that should be there. The following two operations are expected to be equivalent.

```
1  a = a + b
```

```
2    a += b
```

We would expect these to be causing two reads (one to a and one to b) and a write (to a). However, when inspecting the LLVM IR generated after the TSAN pass we found that one of the reads was missing in code that was using the += operator. As such we only got one read and one write recorded, giving some strange results when running the simulation. Currently the only way around this is to avoid using the += operator. We have not found other limitations of this kind yet, but that does not mean they do not exist.

As mentioned earlier, the method used for identifying writes on the heap (MallocTracker) can miss custom implementations of heap writing methods. This is a less common problem, but could still cause issue in some certain projects.

The fact that the instrumentation and runtime can not capture cache affecting operations that occur in external libraries is a source of errors. These external function calls can affect the cache in the same way that operations within the source program do. This means that Cacheray will statistically become less correct the more the analyzed program utilizes libraries. Theoretically this issue can be resolved by adding the instrumentation to the libraries, but practically it would be a big barrier to entry to ask developers to recompile the libraries they use in order to get correct data from Cacheray.

Also mentioned earlier is the fact that the simulator misses the effects of function calls to external (non-instrumented) libraries. This along with above error sources can cause problems for the simulator. The trace based simulation relies on accurate playback of a sequence of event, where each event depends on - and affects - the current state of the cache. Thus, missing events *can* give misleading conclusions about the programs interaction with the cache.

However, it should be noted that the aim is to provide a tool that is able to help developers create programs that make better use of cache memory. To this end it is more important to see the interaction with cache in aggregate, and missing a small percentage of the events will often not have a significant effect, and the output can still be useful for the developer.

Chapter 8
Conclusion

In this final chapter, a conclusion of the thesis, based upon the results and the discussion of them is presented. Also, possible improvements to the simulator and the project as a whole is discussed, along with some ideas on further work within this field.

8.1 Summary

In chapter 1, the following research questions were posed:

1. What amount of resources are needed for the program-under-test to generate a trace file?
2. What level of correctness can we achieve with a trace based simulator?
3. Using the simulator, can a developer change the order of struct members and get correct results without needing to recompile the original program?

These questions will be answered below.

8.1.1 Research Question 1

Based on the discussion in section 7.1 we can draw our conclusions with regards to the four areas we investigated. In terms of file size increase we deem Cacheray is within reasonable bounds, especially looking at the trend visible in figure 7.1. We see that the increase in file size appears significant for smaller files, but for bigger programs it can be seen that the increase is very manageable.

The memory usage is, just like the increase in file size, very manageable. The fact that the size of the write buffer can be configured means that developers have some control of the added overhead.

The increase in execution time is significant, in the worst case increasing 150 fold. While the issue can be mitigated by only recording segments of the program, it certainly limits the usability of Cacheray. Still, the system is usable of analysis of smaller programs. While slower than Cachegrind, it is not so slow as to be unusable.

The trace files produced by Cacheray is very large, even for simpler programs which short execution duration. For one of the tests, over 200MB of data was produced. This was for a program which, without instrumentation, ran for around 10 milliseconds. This is unnecessarily large, and may prevent certain use cases.

With all this in mind, we conclude that the use of Cacheray is not limited in terms of increase to file size or memory usage, since these extra resource requirements are relatively small. Cacherays usability *may* be limited by the increase in execution time in some cases. Finally, the size of the trace files *may* be a limiting factor in some cases.

8.1.2 Research Question 2

As shown in chapter 7, specifically section 7.2, Cacheray appears to achieve some level of correctness when compared to the established simulator Cachegrind. However, as the amount of data collected from the running of the program is only a subset of the total amount of actual events occurring during the execution, the simulator is lacking the data for absolutely correct simulation.

This level of correctness could possibly be enough for many use cases, but obviously lacking in others. As it stands, the level of correctness is fine but we do not believe it is high enough for accurate analysis.

8.1.3 Research Question 3

As shown in chapter 7, section 6.3, the simulator is able to successfully change the order of struct members and have the simulation correctly simulate the change. Of course, the level of correctness is limited to the assumed correctness of the simulator, which in it self is not entirely correct (see 8.1.2).

Nevertheless, struct reordering is possible with the Cacheray system, and based on our testing it works as intended.

8.2 Possible Improvements

Though it is true that code can virtually always be improved and further optimised, that is not the focus of this section. Rather, we will present some shortcomings currently in the cacheray system that could be improved upon given more time.

8.2.1 Efficiency

There are 2 major points of inefficiency currently in the system: Execution time and the size of the trace files.

Because of the need to perform a number of extra calls and computation for each read and write event in the instrumented file, the execution time becomes increased by a large amount. One way to improve this would be to pay close attention to how efficient the Cacheray code is, since it will be executed so many times. There are many ways of improving the performance of C code and if effort was made to make the system run more efficiently, the execution time could most likely be improved.

The size of the trace files can be reduced in a number of ways. One could use a compression library to write compressed data to the trace file. Alternatively, one could modify the event format to be more dynamic, making the average event smaller. However, these approaches run counter to the former inefficiency: execution time. Adding on compression or more complex encoding of events will further slow down the execution of the program.

8.2.2 Custom Instrumentation

As mentioned in section 4.2, the current instrumentation takes advantage of the pre-existing tool ThreadSanitizer to track read and write operations. This presents the issue that we are using part of ThreadSanitizer for something it was not meant to do, introducing potential problems, such as the one discussed in section 7.4. It could cause us to miss operations that affect the cache, and that we therefore want to record in our trace. We also do not control the development of TSAN, and future updates to it could cause compatibility issues.

This problem could be resolved by writing our own instrumentation tool for tracking reads and writes, possibly starting from a fork of TSAN. To an extent we already do custom tracking with MallocTracker. However, we currently still rely heavily on the function calls inserted by TSAN for our instrumentation.

8.2.3 Better Analysis of Heap Memory

Our current tool for analysing data stored on the heap has one major short coming. It can not track if multiple data types are stored in the memory returned by for instance a malloc operation. Say the developer requests 100 bytes of data on the heap using malloc, then uses the first 20 bytes to store a struct of type A, and the remaining 80 bytes for a struct of type B. In this case MallocTracker will detect that 100 bytes was requested and that a struct of type A was stored there, but misses information about struct B. This would cause issues when trying to use the reorder feature, as the simulator then has no way of discerning that struct B is present for 80 bytes, and instead assumes that struct A is present for all 100 bytes.

To solve this issue in general would require extensive analysis of the code, tracking address spaces and operations involving those addresses. If done correctly, this would provide more complete information about the layout of data, in turn giving more information back to the developer after running the simulation.

8.2.4 Default Configuration

Currently the simulator requires a configuration file in order to run. While it is not possible to have defaults for the trace file or type data file, we could add support for default values for

the configuration. This would make the simulator a little easier to use, reducing the number of required files to two.

8.3 Final Remarks

The Cacheray system is a success in that is able to simulate the cache interactions of a program with a decent level of correctness. It offers a re-order feature which can be useful in exploring the effects of structural changes to a program without needing to recompile the code. Hannes learned to type "quickly" too :)

We are however not entirely pleased with only reaching a decent level of correctness, and aspired to do better. However, instrumenting the code at the source code level carries with it some limitations we had not initially considered (such as the challenge of tracing function calls in external libraries). As such, this trace based approach serves a very niche function, mainly for analysing smaller programs that do not use a lot of external libraries. In this niche we consider Cacheray to be useful, but outside it less so.

References

[1] Daniel Pierre Bovet and Marco Cesati. *Understanding the Linux kernel.* O'Reilly, 2006.

[2] HADI BRAIS, RAJSHEKAR KALAYAPPAN, and PREETI RANJAN PANDA. A survey of cache simulators. *ACM Computing Surveys*, 53(1):1 – 32, 2020.

[3] Dwarf Debugging Information Committe. *DWARF Debugging Information Format Version 5*, 2017.

[4] The Valgrind Developers. *Cachegrind: a cache and branch-prediction profiler*, 2000-2020.

[5] The Valgrind Developers. *Valgrind User Manual*, 2000-2020.

[6] Michael J. Eager. *Introduction to the DWARF Debugging Format*, 2012.

[7] eliben. pyelftools. `https://github.com/eliben/pyelftools`, 2019.

[8] Free Software Foundation. *GNU Gzip*, 2018.

[9] P. Guo, Stephen McCamant, and Michael D. Ernst. Bridging the gap between binary and source analysis.

[10] John L. Hennessy and David A. Patterson. *Computer architecture : a quantitative approach.* Morgan Kaufmann, 2011.

[11] Mark D. Hill and Jan Edler. Dinero iv trace-driven uniprocessor cache simulator. `http://pages.cs.wisc.edu/~markhill/DineroIV/`, 1999.

[12] David J. DeWitt Hong-Tai Chou. An evaluation of buffer management strategies for relational database systems. 1985.

[13] ECMA International. The json data interchange syntax. Standard, Ecma International, Geneva, CH, December 2017.

[14] ISO/IEC. Programming languages - C. Standard, International Organization for Standardization, Geneva, CH, 2017.

[15] Ravi Iyer. On modeling and analyzing cache hierarchies using casper. In *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003.*, pages 182–187. IEEE, 2003.

[16] Andhi Janapsatya, Aleksandar Ignjatović, and Sri Parameswaran. Finding optimal l1 cache configuration for embedded systems. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 796–801, 2006.

[17] E. E. Johnson. Pdats ii: improved compression of address traces. In *1999 IEEE International Performance, Computing and Communications Conference (Cat. No.99CH36305)*, pages 72–78, 1999.

[18] Hui Kang and Jennifer L Wong. vcsimx86: a cache simulation framework for x86 virtualization hosts. *Stony Brook University*, 2013.

[19] Brian Wilson Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice-Hall software series. Prentice Hall, 1988. 2nd edition.

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

[21] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD dissertation, University of Cambridge, 2004.

[22] David A. Patterson and John L. Hennessy. *Computer organization and design : the hardware/software interface*. Elsevier Science & Technology, 2014.

[23] Nathan N. Sadler and Daniel J. Sorin. Choosing an error protection scheme for a microprocessor's l1 data cache. *the International Conference on Computer Design (ICCD)*, 2006.

[24] Julian Seward. *bzip2 and libbzip2, version 1.0.8*, 2019.

[25] sharkdp. hyperfine. `https://github.com/sharkdp/hyperfine`, 2020.

[26] Charles F. Shelor and Krishna M. Kavi. Moola: Multicore cache simulator. *30th International Conference on Computers and Their Applications (CATA-2015)*, 2015.

[27] Jonas Skeppstedt and Christian Söderberg. *Writing efficient C code : a thorough introduction*. Skeppberg, 2016.

[28] Atlassian support. Why does clover use source code instrumentation? `https://confluence.atlassian.com/clover/why-does-clover-use-source-code-instrumentation-79986998.html`.

[29] The Clang Team. *ThreadSanitizer*, 2007-2020.

[30] Miguel Ángel Vega Rodríguez, Juan Manuel Sánchez Pérez, and Juan Antonio Gómez Pulido. An educational tool for testing caches on symmetric multiprocessors. *Microprocessors and Microsystems*, 25(4):187–194, 2001.

[31] Josef Weidendorfer. Profiling as part of application development. `http://kcachegrind.sourceforge.net/html/Introduction.html`, 2013. Accessed: 2021-09-21.

EXAMENSARBETE Cacheray - A Trace Based Cache Simulator
STUDENTER Hannes Åström, Wilhelm Lundström
HANDLEDARE Jonas Skeppstedt (LTH), Kim Gräsman (Sandvine Inc.)
EXAMINATOR Flavius Gruian (LTH)

Cache simulering efter exekvering

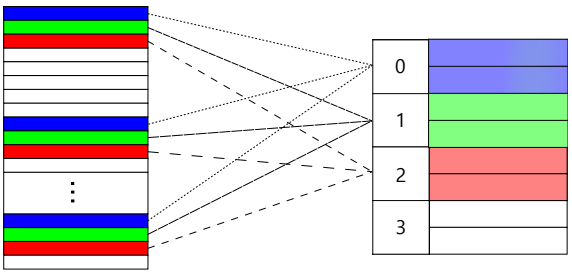
POPULÄRVETENSKAPLIG SAMMANFATTNING **Hannes Åström, Wilhelm Lundström**

Simulering av cacheminnet kan vara en nödvändighet för att få ut maximal prestanda och för att lösa svårhittade minnesineffektiviteter i program. Detta arbete tar fram ett mjukvaruverktyg som gör det möjligt att simulera program som inte kan simuleras direkt, samt simulera förändringar i deras struktur efter kompilering.

Processor cachen är en hårdvarukomponent som finns i praktiskt taget alla moderna datorer. Den agerar som ett mindre minne mellan processorn och minnet, men som är betydligt snabbare att nå. Dess nyttjande är dock ofta begränsat av hur svårt det är att se kopplingen mellan kod och cache. Man kan inte få direkt insyn i vilken information som faktiskt ligger i cache under ett programs exekvering.

För att komma runt detta är det inte helt ovanligt att använda sig av en så kallad *cache simulator*. Genom att använda dessa kan man, som namnet antyder, simulera vad som finns i cache under ett programs exekvering. Dock kan dessa simulatorer vara för krävande för att köra på väldigt simpla system. Detta går att lösa genom att avlasta själva simulationen till ett annat, kraftfullare system, och endast *spela in* programmets minnesoperationer.

I detta projekt har systemet *Cacheray* skapats. Systemet består av ett verktyg för att spela in läsningar och skrivningar från ett program till en så kallad *trace fil*. Denna fil kan sedan användas i *Cacheray simulatorn*. Simulatorn spelar upp dessa operationer och simulerar dem i en modellerad cache. När simulationen är klar får användaren en utskrift med information, såsom hur många gånger data behövde hämtas utanför cachen. Det finns också en möjlighet att ändra om vissa datastrukturer i programmet, direkt i simulatorn, utan att användaren behöver kompilera om och/eller exekvera programmet igen.



När *Cacheray* testades mot en etablerad cache simulator, *Cachegrind*, visade sig vissa likheter i deras resultat. Dock så förlitar sig *Cacheray* på en mindre mängd information än *Cachegrind*, som simulerar program på ett mer direkt vis. Därför är resultatet mellan dem, på många sätt olika.

Trots svagheter i datainsamlingsskedet så tror vi att Cacheray har ett visst användnings område vid tillfällen då en direkt simulation, en så kallad *on-line simulation*, inte kan göras. Till exempel, vid användning av system där mer än ett program inte kan köras samtidigt.