Fakultät für Informatik
Technische Universität München

Bachelorarbeit in Informatik

Development of a multicore cache simulator for performance analysis

Entwicklung eines Cachesimulators für Multicoreprozessoren zur Leistungsanalyse

Author:	Robert Franz
Supervisor:	Prof. Arndt Bode
Advisor:	Dr. Josef Weidendorfer
Submission date:	15.07.2008

I assure the single handed composition of this bachelor thesis only supported by declared resources.

Ich versichere, daß ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15.07.2008

Robert Franz

Contents

Abstract

In order to be able to utilise their computational potential, modern processors use caches. In these caches, data and instructions are be stored in. Caches enable a modern processor to access necessary data much more faster than awaiting the access to the really slow main memory. When using concurrent software on a computer system with more than one processor core, it is not possible to avoid accessing data that another processor core perhaps has already changed. If this memory access, caused by the processor, aims the cache, and data is already stored there, it is necessary to reload the data in order to assure coherent caches. This means the cache has to be synchronised with the main memory. If synchronising has to be done very often, this can affect the speed of the program running.

When developing parallel software, it can not be prevented that a situation like above can occur. It is important having access to a tool, that enables analysing and evaluating the problem. Furthermore, such a tool should deliver appropriate data, that can be worked with.

Part of this thesis is developing such a piece of software, as well as analysing, if it is possible to simulate this kind of situation. Furthermore, it is shown that the data produced by the simulator are exactly enough for improving multithreaded software.

Zusammenfassung

Um überhaupt ihre volle Leistungsfähigkeit ausspielen zu können, benötigen moderne Prozessoren Caches in denen die Daten und Instruktionen mit denen sie arbeiten, gelagert werden, um so einen schnelleren Zugriff auf den benötigten Speicherbereich zu ermöglichen, anstatt auf den im Vergleich zum Cache sehr langsamen Hauptspeicher warten zu müssen. Bei Verwendung von nebenläufiger Software auf Rechnersystemen mit mehreren Prozessorkernen kann man nicht vermeiden, daß es zu einem Speicherzugriff eines einzelnen Prozessorkernes kommt, der auf einen Speicherbereich abzielt der von einem der anderen Prozessorkern verändert worden ist. Wenn die notwendigen Daten des jeweiligen Kerns in seinem Cache bereits lagern, tritt die Situation auf, daß der gecachte Speicherbereich veraltet ist. In diesem Fall ist es notwendig, daß der entsprechende Bereich aktualisiert wird um die Kohärenz der Caches zu gewährleisten. Sofern solche Aktualisierungen häufig durchgeführt werden müssen, so kann dies dazu führen, daß diese Zugriffe zu einer relevanten Zunahme der Laufzeit der Software führen.

Bei der Entwicklung von nebenläufiger Software kann man nicht vermeiden, daß des zu Konstellationen kommt, die dem obigen Szenario entsprechen. Daher ist es wichtig ein entsprechendes Werkzeug an der Hand zu haben, mit dem sich die Situation analysieren und beurteilen lässt.

Die Entwicklung eines solches Werkzeuges ist Bestandteil dieser Arbeit. Desweitern wird gezeigt, daß die beschriebenen Situation hinreichend genau simuliert werden kann, um damit nebenläufige Software optimieren zu können.

Chapter 1

Introduction

Background

Today there is a development to multicore as well as manycore processors. This trend is reinforced by many `x86`-processors released during the last three years. At first, there where many dualcore processors. Intel started this development with a Pentium D, and now the Core2Duo. AMD on the other hand started a dualcore version of their Athlon processors. Now a lot of quadcore processors have been released. Intel made the first step with their dual-dualcore Core2Quads. AMD then brought their new Barcelona architecture. Before both of them, IBM started this trend in 2001 releasing dualcore PowerPCs. Later the Cell Broadband Engine was released. It has got eight supporting processors, that shall to the main part of the work. This supporting processors are optimised for streaming operations. Also processor manufacturers presented road maps of their new architectures showing multicore as well as manycore processors. Just rising the maximum frequency of the processor is not the solution at all, as there are thermal reasons as well as problems in increasing the frequency of each processor core.

Motivation

But when looking at all the software, most of them does not scale very good on multicore systems. Often many programs are able to use only one processor at a time, but most of the computers used today possess more than one core. But having a multicore system without software that can challenge such a current multicore system, such a multicore system does not make any sense without using more than one program at a time that needs a lot of performance. So it is necessary to improve the development of common software in order to be able to use the full potential of these multicore systems.

Another example are all the HPC systems that use processors with more and more cores. Currently more than 90 percent of the systems in listed in `top500.org` have 1k up to 16k processors. Recently those systems used processors with one core. But now these systems use multicore processors as well. So optimising software for this purpose is also necessary. When using software on such a system it is required, that no problem occurs that is based on using multicore processors.

One reason why there are problems is the fact, that there are problems using caches. Each modern processor has got a cache, that stores data more local to the processor. When a processor accesses data in the main memory, this takes a long time. But accessing the cache is much more faster. So a modern processor takes a big advantage in using caches, because the processor has not to wait as long as when accessing the main memory. When a processor does a lot of memory accesses without using a cache system, this significantly decreases performance. So writing software, in order to optimise cache usage, can improve performance. But when using a multiprocessor or a multicore system, all the caches have to be coherent. When optimising cache usage of sequential code, e.g. the hit rate when accessing the cache has to be improved. But when optimising the cache usage of multicore software, the way how to access data must be improved. When a number of processors try to access the same ressource, the caches have to be synchronised each time when the ressource has been modified. Thus synchronising very often significantly decreases performance in a multicore system.

For these reasons it is necessary, to have an appropriate tool that supports developing common software for multicore systems. But currently there are not enough tools available, that support recognising caching problems, concerning multicore systems effectively. So it is necessary to wirte a tool that supports developing a piece of software, that offers an optimised cache usage in multicore or multiprocessor systems.

Objective

There exists a tool called Valgrind. Valgrind is a runtime instrumentation framework. Valgrind catches all the memory accesses and gives the possibility to analyse these memory accesses. It is possible to write tools that base on Valgrind in order to analyse these memory accesses. The tool that is started by default is called Memcheck and recognises memory lacks. Based on Valgrind, there exists a tool called Callgrind (developed by Josef Weidendorfer). Callgrind in turn is based on the cache simulator Cachegrind. Cachegrind is also a tool based on Valgrind. Josef Weidendorfer works at Chair X[1] of the Department of Informatics at the Technische Universität München. So the idea came up to develop the tool based on Callgrind, that is able to cope problems with the cache

[1]Chair X: Computer Organisation; Parallel Computer Architecture, `http://www.lrr.in.tum.de/`

synchronisation, concerning a multicore system.

This tool must handle all the problems ocurring on a multicore system that affects caching. Furthermore all the handling required for simulating all the memory accesses is done by Valgrind. Therefore Callgrind is a good basis, in order to improve this tool by multicore features, as an cache simulator, that catches single threaded cache accesses. But in order to simplify the cache simulator, every cache line must have the same size, as the information about every cache line is stored in a central table. Furthermore currently only a write through cache is supported. Another limitation is the support of currently only eight cores. But in order to get valuable results this is enough.

To be able to understand every aspect of this thesis it essential to be aware of all the concepts used within. Cache coherency is the most important part [AB86] (See 2.2). Furthermore understanding how a cache works is necessary 2.1. But knowledge about all these processor architectures available, may also help. Maybe it is also a good idea to know what exactly Valgrind is and how it works.

Chapter 2

Background

This chapter handles all the stuff that is required to know, in order to be able to understand this thesis. Every modern central processing unit has a cache, that greatly improves performance. As a modern CPU could not reach the same performance as without cache, caches are so important for these CPUs. This chapter makes clear why caches are so important (See 2.1). A cache is an additional component between the main memory and the CPU. Normally all these caches are implemented onto the CPUs die. But there is not only one type of cache but there are numerous. This chapter also describes all these different types of caches as well as a number of strategies these caches use in order to their work (Please see 2.1.2 - 2.1.7). But today there are many multicore systems. Thus synchronisation between all these caches is required when using multithreaded software. So an important part is cache coherency that guarantees, that the will be no problem when using such a piece of software (See 2.2). Furthermore all other issues are handled, that affect using multithreaded software on a multicore or multiprocessor system (Section 2.2.1 - **??**).

To be able to understand every aspect of this thesis it essential to be aware of all the concepts used within. Cache coherency is the most important part [AB86] (See 2.2). Furthermore understanding how a cache works is necessary **??**. But knowledge about all these processor architectures available, may also help.

2.1 Caching

That it influences performance and data consistency on a large scale is the main reason why caching is so important. The first part describes why caching is so essential for modern microprocessors, then there is a part about different types of caches and their characteristics. Furthermore strategies in reading and writing data are handled, as well as replacing old cache lines and prefetching data. An overview about the structure of a cache is also present.

2.1.1 The Importance of Caching

When processor tries to access to data, it normally fetches them from main memory. But this access lasts very long, compared with a cache. So data already called stay stored in the cache memory. So the processor needs to wait the first time only. The next time it can benefit from the already cached data and does not need to await the main memory access. Good software reaches cache accesses instead of memory accesses of at least 95%.

Table 2.1: Memory Access Times
The following data are assumed with a CPU clocked at 2 GHz.

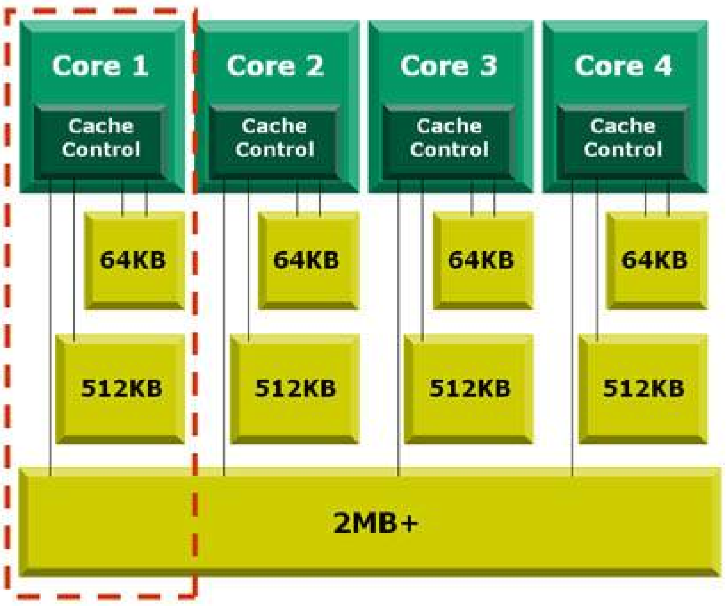
Data Level	**Access Time**	**Size**	**Bandwidth**
Register	1 cycle	4-128 register	as required
L1 Cache	1-4 cycles	16KiB-64KiB	
L2 Cache	10-40 cycles	128KiB-4MiB	
L3 Cache	more than L2	more than L2	
RAM	ca. 10ns, 50-200 cyc.	few MiB to many GiBs	
HDD	3-10ms, still more	many GiB up to TiB	50MB/s-150MB/s

In table 2.1 can be seen, that access times growth with the distance to the processor core, also does the size of each memory level. But the farther the distance to the processor the smaller the available bandwidth. When looking at the number of instructions a modern `x86` processor can cope with, memory bandwidth would decrease the maximum number of instructions, as the bandwidth of the memory bus can not deliver as many instructions as a processor can handle. E.g. a 2 GHz quadcore processor uses instructions with 4 Bytes length. This means a memory bandwidth of 8 GB/s per core is required and therefore 32 GB/s for one processor. Now imagine that each `x86` core normally is multiscalar. A current Intel processor can handle up to four instructions at a time. This would mean a maximum bandwidth of 128 GB/s is necessary only for instructions. Keep in mind, no data is transferred at this moment. No current processor architecture can deliver this amount of data over the memory bus system. Due to this fact, caches have a very big importance, because no available processor architecture could reach its maximum performance without caches, because it would not be possible to deliver as many instructions as required. Of course, this is a worst case scenario.

A cache memory has to be much more faster than main memory, as well as it can not store all the data. Because if it would be as big as the main memory, the memory would be so big, that fetching data would last longer and the effectiveness of the cache would be reduced. Another issue is the fact, that cache memory is compared to "normal" memory quite expensive. As it has to be fast as well as short access times, this raises its price. So it has to be small, not only by economical reasons, but by technical too. This leads to the position, that there were developed strategies to use cache more effectively. Caching data can be done by only one cache. But recently there came up cache hierarchies with up to three caches. The idea behind this was to use a first cache that is very small and very fast. Because this cache is the most expensive cache measured by costs per bit, there is a second cache, that is much more cheaper. The first cache is fed by the second cache that “precaches” the data before load into the first cache. Normally first and second level cache work at the same frequency as the the processor core it is connected to.

In modern multicore CPU architectures like AMD Barcelona there is also a third level cache [AMD07]. An overview about its cache hierarchy can be seen in image 2.1. Each core of the processor has two caches. The third one is a cache that can be used by every core of the processor. It is shared memory, so it can be used as cache for every cache of the processor. Furthermore the is no distinction between data and instruction cache. Another example is given by Intel that currently uses a double cache hierarchy in its Core2Duo processors. Normally these caches are named by their locality to core.

Figure 2.1: Cache Hierarchy - AMD Barcelona



The fastest one is directly next to the processing units and called `L1`. `L1`-Cache (first level cache). In the `x86`-architecture it exists in two versions. One for data, the other for instructions. Called `L1D` for data and `L1I` for instructions. The name of the second one, is following to the nomenclature `L2`-Cache. It caches data as well as instructions. The same play for the third level. But this third level cache then represents shared memory.

2.1.2 Associativity Cache

There are three types of caches. Each type has got its special task it should be preferred.

fully associative This type of cache can allocate each possible cache line when loading data from main memory. The main advantage of such an implementation is its great performance, compared to its size, but the really big disadvantage is the enormous complexity of its implementation. No current cache of a modern CPU is implemented this way. When the CPU wants to access to data, the cache tries to figure out if the required data is already stored in cache memory. In this case, each cache line can held each possible address of the whole system, this means, that every cache line has to be checked whether the required data lies in it. When having a cache memory that has a size of several mega bytes this procedure can last some time. Exactly this fact raises the cost implementation.

direct mapped In this case, the main memory is divided into blocks. These blocks are called sets. One cache line can held exactly one entry of a set of the main memory. This furthermore means, that only one address of the main memory in a set can be cached at a time. Thus it is very easy an cheap to implement, but performance could be better. When accessing to a certain address, only one cache line has to be checked whether the requested address is stored or not. This makes it really cheap but also not really effective.

n-way associative This is a combination of fully associative cache and direct mapped cache. One address of a set of the main memory can be stored in n possible cache lines. So n addresses of a set can be stored at once. It is not as complex as fully associative cache, but offers an improvement of performance because more than one address can be stored.

Table 2.2 shows the connection between cache type, number of sets and associativity. m is the number of cache lines. In order to get the scheme that is used in associativity cache please look to section 2.1.7 on page 18.

Table 2.2: Cache Types

Type	Sets	Associativity
direct mapped	m	1
n-way	$\frac{m}{n}$	n
fully associative	1	m

2.1.3 Event Classification

When a processor accesses to data, perhaps the data is cached. But maybe it is not. These both situations are called “cache-hit” and “cache-miss”. A cache-hit or cache-miss can occur on every cache level. It can be differentiated between each level of cache, that is hit or missed. Cache hits and misses are an essential part when looking to cache coherency, because each of them triggers another event that has to be handled. But now let’s have a look on how caching exactly works:

Cache-Hit In case of a cache-hit a processor wants to have some data and requests it. In case of a `L1D` the request is handled by the logic of the cache. There is a table the cache logic can look in, if the data is only stored in the main memory of in the `L1D` as well. In case of a cache-hit the logic finds the mirrored data of the main memory in the cache. So the processor can access these mirrored data instead of the same data in the main memory. Since this access is much more faster, the processor has not to wait as long as for the main memory to access the data and can continue operating.

Cache-Miss In this case the processor also wants to have some data and also requests the data. The logic of the cache reads the cache table and finds out, that the data needed is not available in the cache. The next step is a request for the data to the main memory. After a period of time the requested data from main memory arrive at the cache. In case there is enough free cache memory, the cache logic duplicates the data. One part it sends to the core the other it stores in the free memory. The next time there will not a cache-miss but a cache-hit. Of course this only occurs in case the data will not be overwritten by the logic. This case is handled in section 2.1.5 on page 17. Furthermore it is possible to classify Cache-Miss. The following four points describe each kind of a cache miss.

Cold Miss: This means, that some data is accessed the first time. So it is not yet stored in the cache system. The only way to reduce the number of cold misses is to use a prefetcher. How prefetching works is described in 2.1.6 on page 17.

Capacity Miss: Some Data was already stored in the cache, but it has been replaced by another data, because the cache was too small.

Conflict Miss: It is not possible to store some data in a set, as the number of blocks is limited. Thus a miss occurs. The only way to reduce the number of conflict misses is to increase the associativity of a cache system. In case of a fully associative cache, no conflict miss can occur.

Coherency Miss: The last possibility of a miss is coherency miss. When one processor writes some data, the cache line of another cache has to be invalidated as the other cache holds the same data. When then accessing these data by the other processor, a coherency miss occurs. This type of miss is the most important miss concerning multicore systems and so for this simulator.

Of course these examples work with every other cache than `L1D` as well. But there can occur a situation, that more than one processor wants to access the same data. This means, cached data can be outdated and have to be reloaded by the processor. In this case "Cache Coherency" is a very important issue. See section 2.2.

2.1.4 Writing Strategies

Another important issue is the way how to handle modified data after processing it.

Write-Back Using this method, data the processor has modified, are written back to cache first. After the process of writing back the data to the cache has finished, nothing else happens. This means that the cache's data are not equal to the main memory's. As cache and main memory are not consistent, a little trick is used to recognise this issue. A so called "Dirty-Bit" signals for each cache line that it is not synchronised with the main memory. When a part of the cache has to be replaced (See section 2.1.5) the data changed, have to be written back to the main memory. When there is a number of cache levels, a cache line is written back to the next level only. Why and when data has to be replaced see the same section.

Write-Through In this case, the processor writes the data directly to the main memory, ignoring the cache available. Normally a buffer is used for writing the data, since it is possible the processor has to wait. But the cache is updated when writing data, as well as when reading some data. When there is a number of caches in a row, the data is written to each cache level and to the main memory.

Assuming Write-Through as "normal" procedure, Write-Back grants the advantage of less transactions with the bus system. But it is more complex to implement. Furthermore when another processor requests some data these data have to be written back to main memory as well as each cache level, when using Write-Back.

2.1.5 Cache Line Replacement

After some time a CPU's cache is full, this means no more data can be stored in. Because anything like this is not acceptable, there has to be a strategy able to cope with this situation. This means that, selected by an algorithm, the cache writes back some data to the main memory. So it is possible to get new data into the cache. There are various algorithms coping with the problem. The most important:

Optimal This algorithms unloads all the information that will not be needed the longest time. This is the best algorithms. But unfortunately nobody knows what kind data this is. Only the programmer could know it, but he has no influence to the cache algorithm.

Least Recently Used Unloads all the data, that has not been used the longest time. Data very often used, normally are changed periodically, so these data will not be unloaded. This procedure takes its advantage by its simple implementation, as only a table has to be hold, that holds information which cache line was used least recently. A disadvantage is a medium hit rate when trying to throw out the right data.

Least Frequently Used The algorithm counts how often some data is needed and rejects the data least often used. This algorithm almost as simple to implement as "Least Recently Used", but it pays attention for the frequency, the cache line has been used in past.

2.1.6 Prefetching

Because the developers of modern processors want to improve the performance of their processors up to the top, they implemented an algorithm in their processor, they hope it can forecast the next required data. This forecasting is technically called "Prefetching". Imagine there is a data stream, a processors works with. A bundle of data that lies linear in the main memory. Furthermore imagine, these data are accessed sequentially. In such a case the prefetching can guess the data required next.

In reality it is common, that a modern processor can detect more than one stream at once. E.g. a program is working on some data, maybe a number of arrays, it reads data of a few arrays and then writes the data on another. This is a situation that needs more than one stream at a time. For example multiplying matrices using peano curves is such an example. [FGH07]

As prefetching does not bother this thesis, this topic is not further handled. But it is important to know how prefetching works, as it can improve a processor's performance.

2.1.7 Mapping

When caching data, the cache controller internally splits the memory address into three parts. These three parts are the byte address, set address and the tag address.

Cache Line Offset The cache line offset is the address of each byte within a cache line. The length of a cache line offset is $ln_2(n)$ bits, whereas n is the size of the each cache line. A cache does not care about these bits, because each of them lies within a cache line, the smallest unit of a cache. In case the CPU wants to access data these bits are used, but for caching as well as replacement and prefetching these bits are not required.

Set Memory is divided into a number of sets. The number of sets used within a cache: $\frac{s}{l \cdot a}$ whereas s is the size of the cache, l the size of a cache line and a the associativity of a cache. Each cache can hold a entries within one set at a time.

Tag The tag address is an element in main memory that is stored within a cache line. Its size is the size of a cache line. Normally there are additionally some tags like a bit if the current cache line is valid or not. E.g. in case of a cache flush, all the cache lines are invalidated at once. In this case each valid bit of the whole memory is set to false. Furthermore each cache has to displace cache entries that have not been used for some time. Thus each cache also has an entry for the age of a cache entry. As there is no better solution in practise the oldest entry is deleted when caching a new address. See 2.1.5.

2.2 Cache Coherency

The reason why it is so important is the fact, that no multicore computer system would work without a consistent memory. Instead it does not work deterministically. As modern processors need a cache to reach maximum performance, it is necessary, to guarantee that every of theses caches of the computer system is consistent. For example, one processor changes some data it normally writes back the data into its local cache. In this case there has to be an algorithm, in order to ensure, that every processor works with the correct set of data. Every cache of a processor has to be synchronised, to assure a correct working of every CPU. There are different ways to cope with synchronising the cache with the other processors. The technique of choice depends on how the memory is accessed the most time. In most scenarios **read**s occur more often than **write**s. As a simulator for `x86` architecture underlies the same requirements. Thus a cache coherency protocol has to be used, that has to be optimised for reading data.

There are various algorithms that can handle cache coherency. Some examples are MOSI, MSI, MOESI and MESI[1]. It is very important to use an algorithm that fits all needs, concerning the multicore cache simulator. In this case the decision was made in favour of MESI (See 2.2.2). These protocols are used to preserve the consistency of a cache line, relating to the main memory of a computer system. This is quite important, as it would not be a good idea to hope nothing bad could happen. It has to be guaranteed, so that no error can occur.

2.2.1 Cache Transactions

If a cache is not consistent any more, it obviously has to be synchronised. Altogether two kinds of synchronisation are possible:

Triggered by CPU A processor calculated something and therefore modified a cache line. So the main memory is not consistent to the current value of a cache line. The main memory has to be updated.

Triggered by Bus Not the processor, the cache works for, modified a cache line, but another processor. Furthermore it has written the correct value into main memory. So the cache must be updated, if a cache line holds data modified in main memory.

This two ways of synchronisation is local to <u>one</u> cache.

2.2.2 The MESI Protocol

The MESI protocol gives each cache line one of four states, normally hold by two bits. Each possible state represents one possible state that can be reached when accessing data. These states are very important when working with a multiprocessor or multicore system.[HPM03]

Let's have a look how MESI works. MESI is a protocol, that consists of four states. These four states are:

[1]Illinois protocol

- **M**odified
- **E**xclusive
- **S**hared
- **I**nvalid

Furthermore two different transitions are known:

direct When a processor core read or writes data, it is possible to change from one state to another state, reading or writing data. When one part of the program wants to access data, every time it has to be checked if a new state is reached.

indirect Only when another processor did some transaction on the memory bus, all data of a cache line has to be renewed. (Bus Snooping)

Figure 2.2 shows every transition from state to state. A dotted line means an indirect transition, each solid one a direct transition.

Modified Data descriped as "modified" are held in one cache only. This cache line can be read and written locally without accessing the bus system. In case of a "write-back" cache, data may are not written to main memory until required by another processor core.

Exclusive As modified, "exclusive" means, that data are stored in one cache only, but it has not modified at all. Data are exactly the same as in main memory. As data are stored in one cache only, there is no requirement for writing data to main memory, when they are stored exclusively in one cache. After modifying the cache line it has to be declared as modified. E.g. this state is used when one core locks some data.

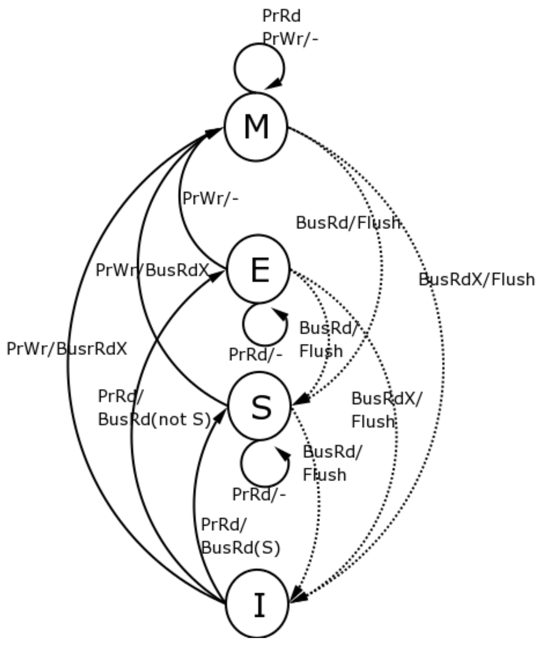
Shared In this case, the data are held in more than one cache. In case, a cache line is declared as "shared", this cache line is up to date. When writing on a shared cache line, changing data has to be signaled to the bus system. This procedure resembles write-through. In case of writing to the local cache, data has not be invalidated.

Invalid A cache line declared as "invalid", logically does not exist. Either the cache line is empty, or data stored in it are invalid. In case of trying to read the content of the cache line, the access returns a cache-miss. Furthermore the cache controller fills the line with the required data, after accessing the data in main memory. When writing data, this is handled as write-through.

Table 2.3: MESI States
The following table represents each state that is possible within the MESI Protocol.

State	Cache line	Main memory	Number of copies	Access
M	valid	invalid	single	cache
E	valid	valid	single	cache
S	valid	valid	multiple	cache & bus system
I	invalid	unknown	multiple	bus system

Figure 2.2: MESI



Concerning the multicore cache simulator there is to determine which states are necessary to be implemented in the simulator. When looking to section 4.3 on page 28, not every MESI state is required.

Chapter 3

Analysis and Considerations

When implementing a multicore cache simulator many questions came up and there are various issues to consider. Of course performance is an important fact, but also the requirements to multicore cache simulator. Furthermore the documentation of the multicore cache simulator is maybe also in fact that should not be undermined.

3.1 Performance

Each simulator has got the problem, that it cannot reach the same performance as a real computer system. Furthermore many other information is collected during executing the simulator. Due to the fact, that every memory access has to be logged, this is not a surprise. Many status information and events are documented. This leads to a situations that requires a simulator, that satisfies a minimum aspect of performant software. Every aspect of the software has to be performant at all. When collecting information about processing a piece of code, each piece of assembler code, a processor core has to execute has to be simulated. This means not only executing one command, but the command plus setting every status and variable that has to be collected. This also means collecting more information as required results in a low performant program. Maybe giving the user the choice, either having more performance or more detailed information about its concurrent program could be the right solution.

3.2 Time and Effort

An important fact is the question how many features shall be implemented in order to get a good working multicore cache simulator, delivering data that contains all required information in order to be able to use them effectively. An analysis what kind of features are required and why or why they should not be implemented can be read in 3.4. For sure it is a lot of time and work to spend . . .

This leads to the next paragraph. There are tools, that are able to serve as a basis for developing this tool. This would result in much less work that doing all the work by one person. Furthermore this means, that more features could be implemented instead of writing all the basic parts of the program. Another point is the fact, that an already existing tool reduces the chance of getting errors into the program, as it is already developed and tested.

3.3 Valgrind

Valgrind is a runtime instrumentation framework that supports developing analysis tools, that helps to develop software. In order to analyse software it is required to compile this software with `gcc` and debugging information. So it is possible to identify a problem, and furthermore it is possible to say which line of code is responsible for this problem.

```
gcc -g anything
```

Especially there is a tool based on Valgrind called Callgrind. Callgrind was developed by Josef Weidendorfer. It extends another Valgrind tool, called Cachegrind, by much more improved tracking features when executing code. In Callgrind currently exists a cache simulator based on Cachegrind. Nevertheless Callgrind is a separate tool, that also can be run by itself only. Also threads can be handled by this extension. But there is no analysis of problems concerning multicore systems.

3.4 Requirements to a Multicore Cache Simulator

When developing a multicore cache simulator it is necessary to know what kind of features are required. Features that can lead to a better understanding of a piece of software. When developing software, it can not be avoided producing errors that can not be anticipated. Problems that does not influence correctness but performance. Software that is correct maybe does not fit requirements, as it runs to with bad performance. It is necessary to know what kind of performance problems occur when running a piece of software. Thus it can be required to renew data stored in a cache when another processor has changed this piece of data.

3.4.1 Statistics

To get expressive data, it is necessary to log the right data. For example, it is interesting how many cases appear, the cache memory has to be refreshed, as the data in main memory has been changed. Also the number of bus transactions is very interesting, as well as which kind of bus transaction occurred. Furthermore it is interesting which kind of thread is affected by such a bus transaction. Maybe a special thread has got problems with cache transactions, or two threads fight each other. Another interesting fact is to get to know, with how many processors/cores this problem exceeds. If the problem occurs, when using two cores, this is worse than using e.g. eight cores. Furthermore an interesting matter can be the number of caches that have to be refreshed, as well as the exact number of each cache. Maybe the problem occurs only using a special number of cores an so on . . . Furthermore it is very interesting for a programmer to gain the line of code, a problem occurs. Thus compiling a piece of software with debugging information is really useful. Valgrind supports gaining the line of code, a problem occurs when processing a program.

Concerning logging all the data, when refreshing a cache should wrapped into a so called "MESI-Logger". In this case every bus transaction can be assigned to a MESI transaction. Examples are bus reads and bus writes when accessing data. The following part describes what kind of collecting information is interesting in detail.

3.4.1.1 Collecting cache behavior

It is important to know how many cache refreshes occur, when using a special piece of software. It is also good to know which processor caused the refresh. Maybe a piece of data is stored in a lot of caches, and in case of changing this data, every cache is affected by this change. This can help a programmer to find problems when accessing data structures or accessing a piece of data.

3.4.1.2 Collecting bus transactions

To get to know what kind of bus transaction occurred, as well as which processor causes the problem. E.g. a special part of a piece of software is really slow, because a tremendous number of bus transactions are executed. So it is a good idea to get to know how many transactions occur and that they occur. As the memory bus of a computer system is restricted, this can affect less performance, as the amount of data that is transferred over the bus system, exceeds its maximum transfer capacity.

3.4.1.3 Assuming program runtime

Another interesting fact would be to guess how long the runtime of a program would be in reality. Maybe it is possible to gain an estimation of the programs runtime. Some facts are known that can result in such an assumed runtime:

- The number of instructions, the program requires to run
- The number of cache hits and cache misses
- A counter how often data of cache line has to be reloaded, due to modified data

3.4.2 Performance

As each single assembler instruction is simulated, the whole simulator is really slow, as statistics has to be written down. Furthermore some extra space is required, but in case of the multicore cache simulator this should not be a problem, as only the cache is mapped into the consistency table 5.2.5.

3.4.3 Extensibility

As this extension of the current cache simulator, maybe shall extended in future, it is necessary to document the whole software as well as guarantee a good extensibility. This document e.g. clarifies the kind the whole program works.

3.5 Documentation

Of course documenting the simulator is an important matter. Nobody could work with a simulator that is not documented. Nobody would work with a simulator, nobody knows how it works and if it works correctly. Relating to the decision to use Valgrind as basis to implement the multicore cache simulator the question “how it works” is no problem any more, because it is open source. Together with this paper every question should be answered. The source code of the simulator can be seen in the appendix A. Every important line of code that was modified during working on the multicore cache simulator is printed in there.

Chapter 4

Developing the Multicore Cache Simulator

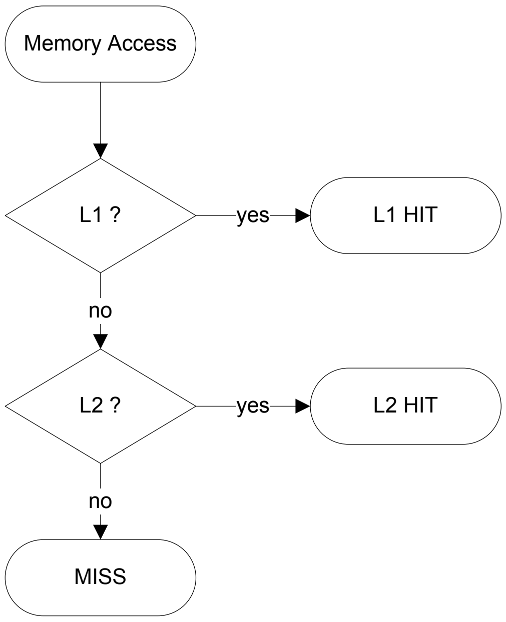
Before, or while implementing the multicore cache simulator it is necessary, to regard which parts are required for the simulator as well as how these parts can be used the most effective way. The simulator shall be based on the MESI cache coherence protocol, as well as on a write through memory.

At first let's have a look at the initial situation of the cache simulator implemented within Callgrind. There is a part that simulates a write through cache simulator. Handling each memory access works the following way: At first there is a check whether there is a L1 cache hit, then if there is a L2 cache hit. If no cache hits, a miss is returned. (See figure 4.1). But in case of the multicore cache simulator, there also has to be checked whether the cached data have been changed by another processor core, or not. Further actions depend on the fact if the requested data are already cached or not. There is no distinction between reading an writing data.

4.1 Threading

As Valgrind supports threading and the multicore cache simulator obviously is based on threads, the following question should be solved. Shall the multicore cache simulator simulate a real processor that handles threads or is such a simulator too complex to implement? Does it really matter? This would mean, moving a thread from one core to another core. Furthermore more than one thread could run on a processor core. Changing between threads on a single core must be handled. Such a solution would be really ugly and complex. Mapping each thread to a single core would be the simplest solution. Furthermore allowing only one thread per core also would decrease the complexity of the simulator. So in the multicore cache simulator a thread is mapped to a core, one to one.

Figure 4.1: Memory Access - Common



4.2 Checking cache line consistency

The most important thing when using more than one processor, is having all information consistent all the time. So there has to be a table that holds all information, required to decide if a cache line is valid or invalid. I decided to store this piece of information in a table called **consistency table**. This consistency table is comparable to directory based cache coherence protocols. In a cache, an address is split into three parts (See chapter 2.1.7). As the last part of the address, the byte address only references bytes within a cache line, this part of information is not required. For implementation it is required, that each cache line of every present cache has the same size. So there is required only one consistency table instead of one for each cache level. So an "Entry" stores all the information of all caches referring a piece of data in the main memory. An entry must hold at least the MESI state of each single cache, as well as the memory address that is cached.

4.3 Implementing MESI without M state (ESI)

As the multicore cache simulator shall be based on MESI, an essential step towards the simulator is to figure out which MESI states are actually used, and which transitions are essential, but also what kind of events have to be thrown. Having a look at MESI, and having a look at the write through cache system the following transitions can happen.

4.3.1 Transitions

At first each cache line is marked as invalid. When reading data, the first time it can change to shared or exclusive, depending on the fact if another cache already holds the same information. In case of an invalidation by another processor core, state I is reached. This means data is not cached in a CPU's cache system any longer. When looking at a write access of the local CPU, according to MESI, the M state is reached. Since this is a write through cache system, a write access implies changing the data in the main memory at once. So the M state is left right away. This means, when reaching the M state, it instantly changes to the E state, as it is the only cache that holds the information. The cache lines of each other core are invalidated at this moment. Furthermore this means, that it is possible to discard the `M` state, and instead doing one of the following transitions: E $\rightarrow$ E or S $\rightarrow$ E. This results in being able to delete every transition concerning the Modified state.

4.3.2 States

Below there is a description of all required states concerning the cache simulator.

Exclusive When one cache holds some data, and no other cache holds the same, this state is active. Furthermore when a processor core has modified some data, no other cache has stored the same information, so this state is active.

Shared Represents the same as in MESI. When at least two caches hold the same information at a time, the shared state occurs.

Invalid This state is used when a piece of memory is not stored within a cache system. The same when this piece of data was invalidated by another processor, caused by modifying some data.

Table 4.1: ESI States

The following table represents each state that is possible within a write through cache system and all the transitions that are possible:

from State	**to State**	**Local Transition**	**Bus Transition**
M	M	not required	not required
M	E	not required	not required
M	S	not required	not required
M	I	not required	not required
E	M	not required	not required
E	E	read or write	—
E	S	—	read
E	I	—	write
S	M	not required	not required
S	E	write	—
S	S	read	read
S	I	—	write
I	M	not required	not require
I	E	read or write	—
I	S	read	—
I	I	not cachable	not cacheabe

4.3.3 Conclusion

Taking every required state and every required transition, MESI is no more, because the Modified state is not reachable within a write through cache system. So only the three states Exclusive, Shared and Invalid remain. Also see diagram 4.2.

4.4 Accessing the memory

In contrast to the "normal" write through cache simulator, there has to be as distinction of a read access and a write access. When modifying data, the other caches that hold the has to be told to invalidate the cache line. But in case of a read access, the consistency table has to be told, data are now stored in another cache, too. And even in case of a write access the same procedure has to take place. So there is no possibility to do the same handling when reading and writing data. This is a fact that has to be noticed.

4.4.1 First- and Last Level Cache

Looking at the cache hierarchy, there is a cache level that is next to the CPU, and one that is next to main memory. A cache next to the CPU is called "First Level Cache" (FLC). On the other side, a cache next to main memory is called "Last Level Cache" (LLC). When reading and writing data, it is essential to notice which kind of cache level is used. Modifying the MESI state of a cache is only required when either a FLC or a LLC is present. On the one hand, when writing data, it is only necessary in case of a FLC, as in each other cache, the same modifications occur. On the other hand, when reading some information, only when using the LLC maybe there has to occur a change of the MESI state. When writing data, in case of write through this means, that the main memory is changed immediately. So the MESI state also changes right away. So even in the first cache level, the state has to be changed. In contrast to a write access, reading requires only a change, when new data is cached. Hence changing a cache line's state is only necessary on the last cache level.

4.4.2 Reading data

Figure 4.3 shows a flow chart that describes the operation method of the multicore cache simulator. When some data is read, at first there has to be a check, whether the requested data is already stored within cache memory. In case of a hit, at first the cache tag is updated, to ensure the right order within the cache memory. This is important in case of a n-way associative cache or a fully associative cache. After that, looking into the consistency table is required, as to check whether data have been changed by another processor core or not. If no change occurred a hit is returned. But in case, the cache line has been invalidated, because the cache line of another core was changed, a miss is thrown as well as the consistency table is told, to add the cache to the caches that hold this special piece of main memory. If there was a cache miss, the last entry of the cache is deleted and the requested data is stored within the cache. In every case, the position of the stored data, relating to the cache is the latest position. Furthermore the has to be checked if the current cache is a last level cache or not. In case it is a last level cache, the consistency has to be told that no cache line of the whole cache system of this processor core holds the data anymore. This handling is required for the cache line that is deleted out of the cache memory.

4.4.3 Writing data

Compared to reading data, writing data is quite similar. When looking at the flow chart 4.4 on page 34 there are only few differences to a read access. The crucial difference is when writing data using a first level cache. In this case, the entry that represents a cache line has to be updated. This means to invalidate each other cache line that holds the same data. Remember the difference between an invalid cache line and an invalid entry concerning MESI.

4.5 Getting Statistics

Without getting useful information there is no chance of being able to improve ones program in case of a problem concerning synchronisation. In order to get useful information, only a number of simple counters is necessary. These counters sum up each single event that is relevant. E.g. the number of bus transactions is some required data. In this case, simply on the right place in the code, the counter has to be incremented. After the simulated program stopped, there has to be a method that calculates all the relevant other data, like percentages and so on. In the initial cache simulator there is a piece of code, that sums up, all the information valuable for the normal cache simulator. For the multicore cache simulator simply the new stats have to be added. How these counters are implemented and why they are implemented can be seen in chapter 5 on page 35.

Figure 4.2: ESI

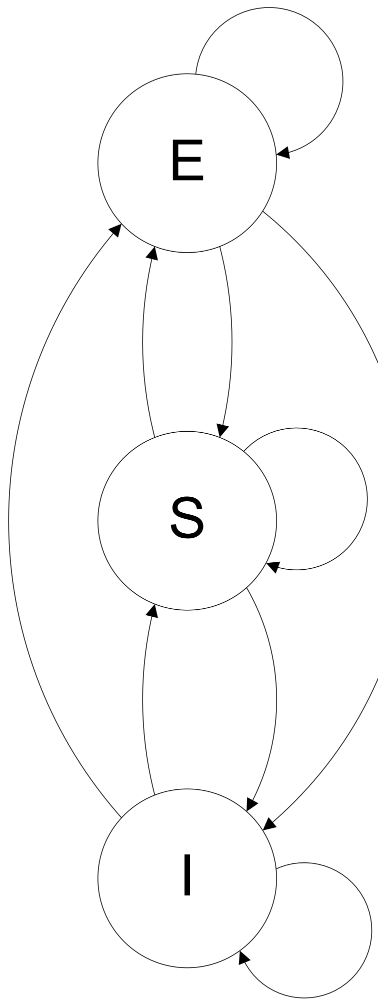


Figure 4.3: Memory Access - Read

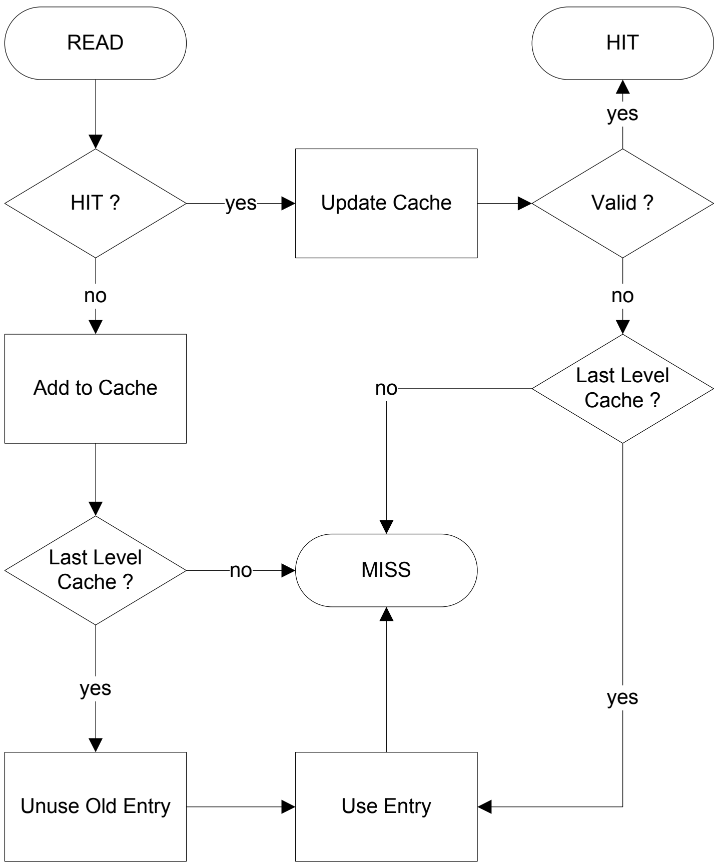
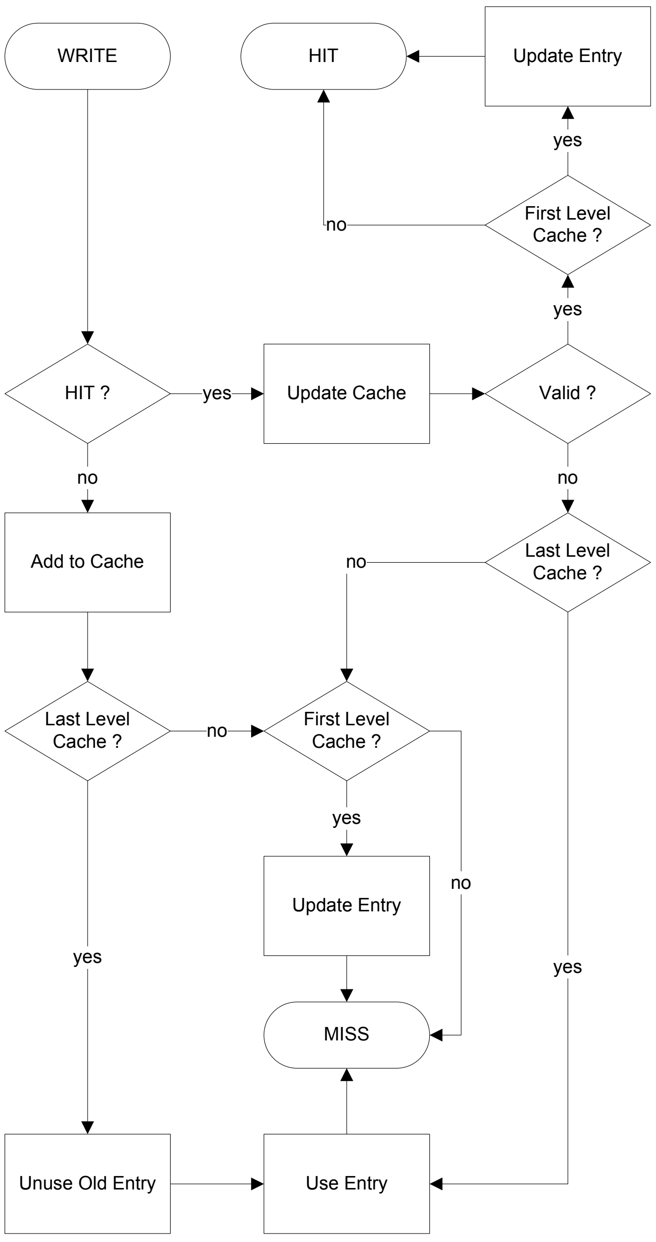


Figure 4.4: Memory Access - Write



Chapter 5

Implementing the Simulator

This part shows how the multicore cache simulator has been implemented in detail. So it is split into three main categories. At first an short overview over all the parameters affected by the multicore cache simulator (5.1). The next part (5.2) describes all the data structures used in order to implement the simulator. And the last part describes a number of methods used within the multicore cache simulator that ensure a correct working of the simulator (See 5.3).

5.1 Parameters

No command line program can work without parameters, so does the multicore cache simulator. The following parameters can be used: When starting a program, each thread

Table 5.1: Parameters of Callgrind extended for multicore

Command	**Value**	**Default**	**Description**
multicore	int	1	Sets the number of cores

is started on a virtual CPU. When setting the number of cores to `1`, only a single threaded program can run on the simulator. When using another number, at maximum this number of threads can run at a time. As for performance reasons is is not possible to modify the number of simulated cores at runtime. Furthermore a program may have at maximum as many threads as set by the parameter `multicore`. Currently there is a support for a maximum of eight threads at a time. So the parameter must be set to a value between **1** and **8**.

Nevertheless there are some parameters, used by Callgrind that cannot be set, or have to be set, so that the multicore extension of Callgrind can work probably. E.g. the “normal” cache simulator must be enabled. Other functions like hardware prefetching or a write-back cache system may not be enabled, as the multicore cache simulator only works in connection with a write-through cache system. Another option that cannot be used, is cache-use. There are some functions within Callgrind that base on cache-use. Furthermore it is required, that every cache line of all the caches has exactly the same size. The multicore cache simulator is implemented in a way, that demands such a limitation. So the parameters that assign a cache a non default configuration require the same size for each cache line. `I1`, `D1` and `D2` **must** have the same cache line size.

Table 5.2: Required Callgrind parameters

Command	Value	Default	Description
simulate-cache	string	yes	enables cache simulator
simulate-wb	string	no	enables write-back cache system
simulate-hwpref	string	no	enables hardware prefetching
cache-use	string	no	counts usage of cache blocks

5.2 Data Structures

The simulator uses many types of data structures. This part shows what they are used for, why they are used and what function they have. As the multicore cache simulator is integrated into the Valgrind tool Callgrind, Callgrind had only to be enhanced. This exactly means adding support for multicore processors, or in this case support for running more than one thread a time concurrently.

5.2.1 MESI

The following `enum` represents all possible and required states a cache line has. A description of all the MESI states can be found in 2.2.2.

Listing 5.1: MESI

```
typedef enum _MESI MESI;
enum _MESI {M,E,S,I};
```

5.2.2 Cachelevel

As it is required, to distinguish between the different cache levels, there has to be another `enum` that represents the level of a cache. It is required to know whether a cache is a first level cache (`FIRST`) or last level cache (`LAST`). In order to see why it is important to know to distinguish between first level cache and last level cache, please refer to 5.3.11.

Further cache levels are possible, if required. Maybe a level called `BETWEEN` is possible that describes a cache level that is neither a first level cache, nor a last level cache. Or `ONE`, when there exists one cache level only. But these states are only for further implementations. Currently only `FIRST` and `LAST` are used.

Listing 5.2: Cachelevel

```
typedef enum _Cachelevel Cachelevel;
enum _Cachelevel {FIRST,LAST};
```

5.2.3 Entry

This part describes an `Entry`, that represents all the states, a piece of memory has. A `MemState` describes a global state that is valid for all the caches used within a system. Only Shared and Invalid are described this way. This value is used for performance reasons, for example, not every cache has to be checked, whether it is valid or not, but only this one value. Furthermore the states of each cache are represented in `CacheStates`. Currently the multicore cache simulator is limited to eight cores.

Accessing MemState requires only one access instead of n accesses required when reading or writing CacheStates, whereas n is the number of cores used within the multicore cache simulator. As the maximum number of threads is limited to eight cores, there are only eight `CacheStates`. The data structure that requires the most memory in the multicore cache simulator is `Entry`. For each cache entry is requires $(n + 1) \cdot 4$ bytes of memory. As `Entry` is used many times in the consistency table, it is used very often. To see the maximum required memory please see 5.2.5.

Listing 5.3: Entry

```
typedef struct _entry Entry;
struct _entry
{
  MESI MemState;
  MESI CacheStates[8];
};
```

5.2.4 McHashNode

This one shows a node used for the hash table described in 5.2.5. The pointer `next` is used internally by the hash table, so the pointer is a required element. `key` is used the same way. But the standard node is extended by a pointer to an `Entry` (See 5.2.3). Each entry is stored in the hash table in order to be able to find an entry again when only knowing the memory address.

Listing 5.4: McHashNode

```
typedef struct _McHashNoder
{
  struct _McHashNode * next;
  UWord key;
  Entry* entry;
}
McHashNode;
```

5.2.5 Consistency Table

Because preserving data consistency of the caches is the most important fact. Thus a table that guarantees these consistency is necessary. A mapping between main memory and each cache memory is required. This mapping happens in the consistency table. It is very important to know how each data structure works together with each other. An overview therefore is required as well as an overview of each single data structure. It holds all the information used to check whether a cache line is valid or not. The first element is `cores`. It holds the maximum number of threads that can be simulated at a time. Internally, each thread is simulated on a singe core, so one thread means one core. The next one is an object called `Hash`. It describes a hash table that is defined within Valgrind and that is now used in order to find the right entry to an address. Furthermore there are counters that collect the number of events per processor core. One important issue of the consistency table is the fact how many memory is required to do all the consistency checks. As described in 5.2.3 a cache entry requires $(n + 1) \cdot 4$ bytes of memory, whereas n is the maximum number of cores. In case of the multicore cache simulator eight cores are possible. So one cache entry requires 36 bytes of memory. At most there are as many cache entries as cache lines exist. Imagine a cache system is completely inclusive. In case of a `L2` cache that has a size of 2 MiByte and a cache line size of 64 Bytes, the consistency table requires only 1179648 Bytes of Memory (1152 KiByte). $\frac{2 \cdot 1024^2}{64} \cdot 36$. As the caches used in the cache simulator are inclusive, and the size of a cache line in each cache level is the same, the maximum number of cache entries required is: $\frac{36 \cdot n}{l}$, whereas n is the size of the last level cache and l is the size of a cache line.

`READ[n]` counts the number of bus reads

`WRITE[n]` counts the number of bus writes

`INVALIDATE[n]` counts the number of invalidations per core

`SREAD[n]` counts the number of shared reads

`INVALIDATES[n][m]` describes the number of invalidations when writing data

As for performance reason, the ID of the current thread is resolved only one time. Then the ID of the current thread is written into the variable `CurThread`. Each time the current thread ID is required, this value can be read. Listing 5.5 shows the source code.

Listing 5.5: Consistency Table

```
typedef struct _consistencytable ct;
struct _consistencytable
{
  unsigned int cores;
  VgHashTable Hash;
  long READ[8];
  long WRITE[8];
  long INVALIDATE[8];
  long SREAD[8];
  long INVALIDATES[8][4];
  ThreadId CurThread;
} CT;
```

5.2.6 The Caches

Currently in Callgrind there only exist three caches. One first level instruction cache L1I, one first level data cache L1D and one second level data cache L2D. As the multicore cache simulator shall handle a number of cores, for each core an independent cache has to be simulated. This requires to implement an algorithm that handles these various caches when accessing these caches. The easiest way is to use an array of caches for each cache level. The number of caches must be the number of possible threads, running at a time. Accessing these cache array does not corrupt performance, so this will not be a problem. Only the number of cache structures that have to be handled means a complexity of $O(n)$. n is the maximum number of threads that can be handled by the multicore cache simulator. The maximum number of caches is assigned by the parameter **multicore** (See 5.1). Currently there is a limit to eight cores at a time. Furthermore the cache configuration was modified. The parameter `sets_bits` was added as it is required in order to the the number of bits, the address has to be shifted when generating the hash key for the entry. The part of Callgrind that generates all the values for the cache configuration, also generates the number of `sets_bits`. The following both listings show the code that is used for this modifications.

Listing 5.6: Caches

```
static cache_t2 mulI1[8], mulD1[8], mulL2[8];
```

Listing 5.7: Cache Configuration

```
typedef struct {
  Cachelevel cachelevel;

  ...

  int sets_bits;

  ...

} cache_t2;
```

5.3 Functions

Of course data structure are important, but without doing anything every data structure has no function. The following pages describe all the functions used for the multicore cache simulator.

5.3.1 Initialising the Consistency Table

Before starting simulating, the consistency table of the multicore cache simulator has to be initialised. For this purpose there is a function that does everything that is required to be able to start. The function is called `initCT`. The code can be read in listing 5.8. At first all the values are assigned. Then the counters are reset. Then a new hash table is created.

Listing 5.8: Initalise the conistency table

```
void initCT(unsigned int cores,unsigned int size)
{
  CT.cores=cores;
```

Table 5.3: Parameters required for `initCT`

Parameter	Value	Description
`cores`	`unsigned int`	Number of cores
`size`	`unsigned int`	Maximum number of Entrys required

```
  CT.size=size;
  int i;
  int j;
  for(i=0;i<4;i++)CT.READ[i]=0;
  for(i=0;i<4;i++)CT.WRITE[i]=0;
  for(i=0;i<4;i++)CT.INVALIDATE[i]=0;
  for(i=0;i<4;i++)CT.SREAD[i]=0;
  for(j=0;j<8;j++)for(i=0;i<4;i++)CT.INVALIDATES[j][i]=0;
  CT.Hash = VG_(HT_construct)("Addresses");
}
```

5.3.2 Initialising an Entry

Listing 5.9 shows the initialisation of an `Entry`. Before using an Entry within the consistency table, it is necessary to initialise the `Entry`. The function shown in the listing sets all the `CacheStates` to invalid. Furthermore the global state is set to invalid. The global state represents all caches.

Table 5.4: Parameters required for `initEntry`

Parameter	Value	Description
`tmp`	`Entry*`	Pointer to an Entry

Listing 5.9: Initalising an Entry

```
void initEntry(Entry* tmp)
{
  int i;
  for(i=0;i<CT.cores;i++)tmp->CacheStates[i]=I;
  tmp->MemState=I;
}
```

5.3.3 Activating an entry for a cache

When loading data, the first time the entry must know, that the active cache caches the data. The global state `MemState` has to be modified if required, as well as as the local `CacheStates`. This function sets a new `MemState` and `CacheStates` if required. It handles whether an `Entry` is used by only one cache or by a number of. Furthermore it sets an `Entry` to shared if more than one cache uses this `Entry`.

Table 5.5: Parameters required for `useEntry`

Parameter	**Value**	**Description**
`tmp`	`Entry*`	Pointer to an Entry

Listing 5.10: Activate entry usage

```
void useEntry(Entry* tmp)
{
  if(tmp->MemState==E)
  {
    if(tmp->CacheStates[CT.CurThread]!=E)tmp->MemState=S;
    tmp->CacheStates[CT.CurThread]=E;
    return;
  }
  if(tmp->MemState==I)tmp->MemState=E;
  tmp->CacheStates[CT.CurThread]=E;
}
```

5.3.4 Deactivating an entry for a cache

When a cache line is replaced by another, these data are not used any longer by the currently active cache. So the usage of the `Entry` concerning this cache has to be modified. The functions shown in listing 5.11 handles that the data the entry represents. are not required any more. At first the `CacheStates` for the active thread is set to inactive. Furthermore there is a check whether there are at least two other caches that hold the information after the active cache has replaced the cache line. If there is only one cache that holds the same information, the global state is set to exclusive. If no other cache holds the information, the `Entry` is not required any more. Thus it is deleted by this function.

Listing 5.11: Activate entry usage

Table 5.6: Parameters required for `unuseEntry`

Parameter	Value	Description
`tmp`	`Entry*`	Pointer to an Entry

```
void unuseEntry(Entry* tmp,Addr a)
{
  tmp->CacheStates[CT.CurThread]=I;

   if(tmp->MemState==S)
  {
    int i;
    int number=0;
    for(i=0;i<CT.cores;i++)if(tmp->CacheStates[i]==I)number++;
    if(CT.cores-1==number)tmp->MemState=E;
    return;
  }
  VG_(HT_remove)(CT.Hash,a);
  VG_(free)(tmp);
}
```

5.3.5 Adding an entry to the hash table

When using an entry, this entry has to be added to the hash table. E.g. in case an entry is not associated to a cache line, this entry has to be added to the hash table. Thus a new hash node has to be generated. After that the `key` and the pointer to the `Entry` is assigned. Then the node is added to the hash table. The hash table used in this part of the code is a part of Valgrind. It is located in part of Valgrind called coregrind. Coregrind holds many basic functions that are essential for Valgrind and its tools. It is required to give an address that later is assigned to the key.

Table 5.7: Parameters required for `useEntry`

Parameter	Value	Description
`tmp`	`Addr`	Address as key value for hash table
`tmp2`	`Entry*`	Pointer to an Entry

Listing 5.12: Add address to hash table

```
inline int addAddressToCT(Addr tmp, Entry* tmp2)
{
  McHashNode* newNode=VG_(malloc)(sizeof(McHashNode));
  newNode->key=tmp;
  newNode->entry=tmp2;
  VG_(HT_add_node)(CT.Hash,newNode);
}
```

5.3.6 Check ThreadID

The multicore cache simulator supports a number of threads at the same time. But there is only a maximum number of threads that can be handled simultaneously. If the maximum number of cores is exceeded, an error message is returned to the console. It is possible to get the ID of the current thread by using a function, but the ID already is assigned to `CT.CurThread`.

Listing 5.13: Add address to hash table

```
inline void checkValidThread()
{
  if(CT.CurThread>=CT.cores)
  {
    VG_(message)(Vg_UserMsg,"Currently␣it␣is␣not␣possible␣to␣use␣more
        ␣than␣8␣threads␣at␣a␣time.");
    VG_(message)(Vg_UserMsg,"Aborted!");
    VG_(exit)(1);
  }
}
```

5.3.7 Reading Data

Every time when the simulator reads some data, it checks whether the data is cached or not. Furthermore it checks whether it is cached within the `L1` cache or the `L2` cache. In case of the multicore cache simulator it also has to check if the read data are consistent. So it has to look into the consistency table (See 5.2.5) and get the `Entry`, that represents the consistency information concerning the cache line. This method also gets some of the information required for printing statistic information after running a program. E.g. shared reads as well as bus reads are stored. Normally, this functions is called in the last cache level only. So each read access is a bus transaction.

Listing 5.14: readData

Table 5.8: Parameters required for `readData`

Parameter	Value	Description
`tmp`	`Entry*`	Pointer to an Entry

```
inline readData(Entry* tmp)
{
  int i;
  CT.READ[CT.CurThread]++;
  if(tmp->CacheStates[CT.CurThread]==E)CT.SREAD[CT.CurThread]--;
  for(i=0;i<CT.cores;i++)
  {
    if(tmp->CacheStates[i]==E)
    {
      CT.SREAD[i]++;
    }
  }
}
```

5.3.8 WritingData

As well as the function `readData`, `writeData` stores information concerning statistics that are printed after simulating. This method changes the entry of some data in the main memory that is stored in at least one cache of the multicore cache simulator. This function checks, whether another cache uses the same data and then this function invalidates this data. Furthermore the information for all entries is set to exclusive, as there is only one cache, that holds the right information. This setting is made in `MemState` within the entry. Also the `CacheStates` of the currently active thread is set to exclusive, as it is the only cache that holds the information.

Table 5.9: Parameters required for `writeData`

Parameter	Value	Description
`tmp`	`Entry*`	Pointer to an Entry

Listing 5.15: writeData

```
inline writeData(Entry* tmp)
{
  int i;
  int n=0;
  CT.WRITE[CT.CurThread]++;
  if(tmp->CacheStates[CT.CurThread]==E)
  {
    CT.INVALIDATE[CT.CurThread]--;
    n--;
  }
  for(i=0;i<CT.cores;i++)
  {
    if(tmp->CacheStates[i]==E)
    {
      CT.INVALIDATE[i]++;
      n++;
      tmp->CacheStates[i]=I;
    }
  }
  tmp->CacheStates[CT.CurThread]=E;
  tmp->MemState=E;
```

The next part describes getting the statistics that is printed after the program that was simulated has finished. This part describes the number of invalidations per write access. There is a distinction based on the number of invalidations per write access. In the code before, the counters are incremented for all relevant threads.

```
  if(n==1)
  {
    CT.INVALIDATES[CT.CurThread][0]++;
    return;
  }
  if(n==2)
  {
    CT.INVALIDATES[CT.CurThread][1]++;
    return;
  }
  if(n==3 || n==4)
  {
    CT.INVALIDATES[CT.CurThread][2]++;
    return;
  }
  if(n>4)
  {
    CT.INVALIDATES[CT.CurThread][3]++;
    return;
  }
}
```

5.3.9 Doing a memory access

Before doing all the operations when accessing a cache line, there is a memory access. In Valgrind there exists a function pointer to three functions. One for reading instructions, one for reading data and the last one in order to write data. Each memory access can be different. The same is in the multicore cache simulator. These three functions can be seen in the listings 5.16, 5.17 and 5.18. The first operation is to get the ID of the current thread running. As the simulator supports only a number of threads there is a check that validates, if the current thread is within this maximum thread border. Then each of them calls a function called `cachesim_mulref`. This functions checks whether one or more than one cache lines are affected when doing an operation. At first there is a check if there is a `L1`-Hit, then if there is a `L2`-Hit. If no cache hits, a miss is thrown. This part is also used in the legacy cache simulator. But in this case it is distinguished if a cache line is read or written. But to see what `cachesim_mulref` requires, please see 5.3.10. Furthermore in case of an instructions an instructions counter is incremented. This counter stores the number of instructions per core.

Table 5.10: Parameters required for a memory access
This table shows the parameters for each of the three functions

Parameter	Value	Description
`a`	`Addr`	Address to be called
`size`	`UChar`	Number of bytes that are accessed

When reading instructions a counter is incremented that counts the number of instructions per thread.

Listing 5.16: Reading Instructions

```
static
CacheModelResult cachesim_I1_mulRead(Addr a, UChar size)
{
  CT.CurThread=VG_(get_running_tid)()-1;
  checkValidThread();
  threadcount[CT.CurThread]++;

  if ( cachesim_mulref( &(mulI1[CT.CurThread]), a, size,CT.Hash,1) ==
       Hit ) return L1_Hit;
  if ( cachesim_mulref( &(mulL2[CT.CurThread]), a, size,CT.Hash,1) ==
       Hit ) return L2_Hit;
  return MemAccess;
}
```

Listing 5.17: Reading Data

```
static
CacheModelResult cachesim_D1_mulRead(Addr a, UChar size)
{
  CT.CurThread=VG_(get_running_tid)()-1;
  checkValidThread();

  if ( cachesim_mulref( &(mulD1[CT.CurThread]), a, size,CT.Hash,1) ==
       Hit ) return L1_Hit;
  if ( cachesim_mulref( &(mulL2[CT.CurThread]), a, size,CT.Hash,1) ==
       Hit ) return L2_Hit;
  return MemAccess;
}
```

Listing 5.18: Writing Data

```
static
CacheModelResult cachesim_D1_mulWrite(Addr a, UChar size)
{
  CT.CurThread=VG_(get_running_tid)()-1;
  checkValidThread();

  if ( cachesim_mulref( &(mulD1[CT.CurThread]), a, size,CT.Hash,0) ==
       Hit ) return L1_Hit;
  if ( cachesim_mulref( &(mulL2[CT.CurThread]), a, size,CT.Hash,0) ==
       Hit ) return L2_Hit;
  return MemAccess;
}
```

5.3.10 Accessing a cache line

When doing a memory access, it is possible that more than one cache line is affected. Furthermore it is required to gain some useful information required when accessing a cache line. So the correct set and tag is required. If the memory access is straddled about two cache lines, a second cache line access is required. The function `cachesim_mulsetref` requires the correct address for the cache line access. So this address has to be submitted. If there are two cache line accesses, this address is modified. In case, the size of an access is bigger than a cache line, and it straddles about more than one cache line an error message is thrown. Furthermore the address of the first byte of each cache line is generated and stored within `a`. This address is required when accessing the hash table. Please read 5.3.11 for this purpose.

Table 5.11: Parameters required for a accessing a cache line

Parameter	**Value**	**Description**
`c`	`cache_t2`	Cache configuration
`a`	`Addr`	Address to be called
`size`	`UChar`	Number of bytes that are accessed
`hash`	`VgHashTable`	Required hash table
`read`	`int`	Read access or not

Listing 5.19: cachesim_mulref

```
static CacheResult cachesim_mulref(cache_t2* c, Addr a, UChar size,
    VgHashTable Hash, int read)
{
  UInt  set1 = ( a         >> c->line_size_bits) & (c->sets_min_1);
  UInt  set2 = ((a+size-1) >> c->line_size_bits) & (c->sets_min_1);
  UWord tag  = a >> c->tag_shift;
  if (set1 == set2)   return cachesim_mulsetref(c, set1, tag, a, Hash
      , read);
  else if (((set1 + 1) & (c->sets-1)) == set2)
  {
    UWord tag2  = (a+size-1) >> c->tag_shift;
    CacheResult res1 =  cachesim_mulsetref(c, set1, tag, a, Hash,
        read);
    CacheResult res2 =  cachesim_mulsetref(c, set2, tag2, a+c->
        line_size, Hash, read);
    return ((res1 == Miss) || (res2 == Miss)) ? Miss : Hit;
  }
  else
  {
    VG_(printf)("addr: %lx  size: %u  sets: %d %d", a, size, set1,
        set2);
    VG_(tool_panic)("item␣straddles␣more␣than␣two␣cache␣sets");
  }
  return Hit;
}
```

5.3.11 Handling all the Caching

The function described in here, is the most complex function within the multicore cache simulator. The code is implemented in a way as described in 4.4.2 and 4.4.3. Basically, the code is based on the code used in the “old” cache simulator. But there are some drastic modifications. These modifications concern the checks whether a cache hit or a cache miss occurs. The following listing 5.20 shows exactly this part of code, that handles all the cache accesses.

Table 5.12: Parameters required for `cachesim_mulsetref`

Parameter	Value	Description
`c`	`cache_t2`	Cache configuration
`set_no`	`UInt`	Required parameter for the standard cache simulator
`a`	`Addr`	Address of the memory access
`Hash`	`VgHashTable`	The hash table that has to be accessed
`read`	`int`	Checks whether a read access or a write access

Listing 5.20: cachesim_mulsetref

```
CacheResult cachesim_mulsetref(cache_t2* c, UInt set_no, UWord tag,
   Addr a, VgHashTable Hash, int read)
{
  Addr add = a >> c->line_size_bits;
  Entry* tmp;
  McHashNode* node=VG_(HT_lookup)(CT.Hash,add);
```

In the upper part, the address is modified in a way, that the hash table can handle is without any problems. The hash table handles only the addresses of the first byte of a cache line. If every possible byte would be stored in the consistency table, this would cause another problem. For every access of a cache line, no check is required this way, but by storing every address. So when accessing another address, but in the same cache line this handling is covered by this part of the code. Furthermore there is a lookup if there already exists an `Entry`.

```
  int i, j;
  UWord *set;
  set = &(c->tags[set_no << c->assoc_bits]);

  if(node==NULL)
  {
    tmp=VG_(malloc)(sizeof(Entry));
    initEntry(tmp);
    addAddressToCT(add,tmp);
  }
  else tmp=node->entry;
```

There is check whether an entry already exists or not. If not, a new `Entry` is generated and then added to the hash table. Otherwise the returned entry is assigned to the pointer. To find out what all the functions used in this part of code do, please see 5.3.5 for `addAddressToCT` and 5.3.2 for `initEntry`.

```
  if (tag == set[0])
  {
    if(tmp->CacheStates[CT.CurThread]==E)
    {
      if(read==0)if(c->cachelevel==FIRST)writeData(tmp);
      return Hit;
    }
    if(c->cachelevel==LAST)useEntry(tmp);
    if(read==0)if(c->cachelevel==FIRST)writeData(tmp);
    if(read==1)if(c->cachelevel==LAST)readData(tmp);
    return Miss;
  }
```

If the memory access hits the first entry of the cache memory, all the processing is done. No rearranging of the cache set is required. If the cache line holds valid data a cache hit is returned. If there is a write access, and the cache is the first level, all the processing when writing data is also done. If there is a cache miss, it signals that it is now valid. Furthermore the same processing is done for writing data. Direct mapped cache requires only this part of code in order to check if there is a hit or not. A description of `writeData` can be found in 5.3.8, `readData` in 5.3.7. For `useEntry` please refer to 5.3.3 on page 42.

```
  for (i = 1; i < c->assoc; i++)
  {
    if (tag == set[i]) {
      for (j = i; j > 0; j--)
```

```
            {
              set[j] = set[j - 1];
            }
            set[0] = tag;
            if(tmp->CacheStates[CT.CurThread]==E)
            {
              if(read==0) if(c->cachelevel==FIRST) writeData(tmp);
              return Hit;
            }
            if(c->cachelevel==LAST) useEntry(tmp);
            if(read==0) if(c->cachelevel==FIRST) writeData(tmp);
            if(read==1) if(c->cachelevel==LAST) readData(tmp);
            return Miss;
          }
        }
```

This part is called when there was no match in the first cache entry. In such a case it is tested whether one of the other cache line holds the required data. Associativity of the cache requires to test as many cache tags as big as associativity is. When the correct one is found, the same processing is done as in the piece of code before. In case of a cache miss, the same handling is done, too.

```
if(c->cachelevel==LAST)
{
  useEntry(tmp);
  add = (set[c->assoc-1] << c->sets_bits) + set_no;
  node=VG_(HT_lookup)(CT.Hash,add);
  if(node!=NULL)
  {
    tmp=node->entry;
    unuseEntry(tmp,add);
  }
}
```

If the last cache level is affected, the `Entry` is modified. Furthermore the entry that represents the data that are replaced by the new data, has to be modified, too. To see what `unuseEntry` does, please look to section 5.3.4 on page 42.

```
for (j = c->assoc - 1; j > 0; j--)
{
  set[j] = set[j - 1];
}
```

```
  set[0] = tag;
  if(read==0)if(c->cachelevel==FIRST)writeData(tmp);
  if(read==1)if(c->cachelevel==LAST)readData(tmp);
  return Miss;
}
```

The last part of this function simply reorders the set of the cache. Furthermore all the required processing when reading and writing data is done.

5.3.12 Printing the Logs

In order to be able to work with the results, generated by the cache simulator, it is essential to sum up all the collected information and print them onto the monitor. This function does all the work. The following information is printed:

- The number of instructions handled by each core.
- The number of invalidations, concerning modified data.
- All the reads, when other caches already hold the same information.
- The number of each bus transaction, concerning the MESI protocol.
- How many invalidations occur when writing data

The code used within the cache simulator can be seen in listing 5.21. At first all the processing is done, than the number of instructions per core is printed. After that some calculations to get a summary is done. The last point is printing the number of invalidations per write access.

Listing 5.21: Printing the logs

```
  if(CLG_(clo).cores>1)
  {
    // Printing all the information got before
  }
```

5.3.13 Capturing Data

The multicore cache simulator collects some data. The upper sections describe how these data is collected. But in order to give an overview of all these sections, this part is a summary of all these pieces of code that are affected.

In `readData` (5.14 line 10) the counter in incremented, in order to signal a shared read. But only if more than one cored holds the same data at a time. But the same piece of code generates a read bus transaction on line 5. Furthermore the code in `writeData` (5.15 line 5) signals, that a bus write happens. If any other core holds the same data, these cache lines are invalidated in line 8 and 15. Beginning from line 22, the number of invalidations are handled, that happen when writing data. Between one, two, more than two and more than four invalidations per write access is differentiated.

Chapter 6

Results

This chapter handles all the results, got by testing the multicore cache simulator. Furthermore it shows how exactly the multicore cache simulator really works. In order to get usable data, some programs are required, in order to test the simulator. Another important point is the performance of the multicore cache simulator. This chapter also handles this topic compared to the “normal” cache simulator implemented in Callgrind. Another point are the differences between a real computer and the multicore cache simulator. Why they exist and if there could be a solution how to solve thus differences, is also handled in this chapter.

6.1 Testing routines

The most important part, is to figure out what kind of programs are required to be tested. So there has to be an analysis what kind of problems and algorithms has to be used for testing the multicore cache simulator. Furthermore the same tests are required to figure out differences between a real computer system an the multicore cache simulator. The second important part is to get some programs that help to figure out how good the multicore cache simulator works.

6.1.1 Programs

As this is a **multicore** cache simulator, apparently only tests should be done that are useful to test the multicore features of the extended version of Callgrind. As the normal cache simulator already works fine, no more tests are required to test these features again. In order to get some concurrent software, there is a number of solutions that can be used to implement the test programs. Various solutions to automatically generate concurrent software is available. One of them is MPI [MPI08], another OpenMP [Ope08]. A big advantage of OpenMP is the fact, that it automatically generates concurrent software, only by giving the compiler some instructions how the code can be used and what conditions have to be met. These compiler directives make it possible to easily generate concurrent software automatically. Especially on a multicore system this is the right way in order to improve performance.

6.1.1.1 Verify Correctness as singlecore simulator

In order to see if the multicore cache simulator works without any problems, of course it should be compared to the "old" simulator. So there shall be tests, that compare the new one with the old one in order to guarantee that is works fine with single threaded programs as well. But be aware of the results: Valgrind calls programs and libraries in the background. So the number of instructions may differ from program to program. One program that does not seem to have this problem is `date`. `date` simply returns the current date, as well as time and time zone. So I tested this program. Another program that seems to be great for testing is `df`. `df` returns the amount of space used on all mounted volumes, free space and the name of each volume. This program also seems to produce in every run the same results. These programs shall check, whether there are differences between the implementation of the "old" and the "new" cache simulator. As a lot of code of the old simulator has adopted, normally in these parts of the code there shouldn't be errors. But new parts could affect the old parts to generate errors. As there are many routines added, as well as only the core parts of the old simulator is used in the new simulator. Thus a problem could appear, that some code is called a wrong way.

6.1.1.2 Reader Writer Test

This small program simulates various threads writing on the same data. Another situation that is simulated, is writing various data on the same cache line. In such a case the cache line has to be reloaded after each modification into the local cache of the CPU that uses the data. So the cache has to reload the whole cache line, because one piece of the cache line was modified. Even when the current processor needs an other piece of data. In case of "False Sharing", different bytes of a cache line are used, but within one cache line. The program in listing 6.1 is such an example. Every thread tries to write on a variable called abc. But in this program every thread tries to write the same data. Maybe this program delivers quite nonsense as result. But concerning the all the caches in the multicore cache system this really stresses the caches. In a multicore cache system, every other cache line that holds the same data has to be reloaded. When looking at the source code, line 8 holds the following command: `omp_set_num_threads(n);`. Depending on the number of threads that shall be simulated this parameter can be changed. In the end there is a number of routines that are compiled and executed. This test shall show the number of events that occur when each thread is writing on one variable. Especially the number of invalidations is interesting. But all the other events collected maybe are also be interesting.

Listing 6.1: Reader Writer Problem

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main(void)
6  {
7    int abc=0;
8    omp_set_num_threads(n);
9    #pragma omp parallel for shared(abc)
10   for(int i=0;i<1000000;i++)
11   {
12     abc++;
13     #pragma omp barrier
14   }
15   return EXIT_SUCCESS;
16 }
```

6.1.1.3 Performance

As the performance of this simulator is an important fact, there must be some tests in order to determine the performance of the multicore cache simulator. In this case, measuring the time for executing a program natively, in the normal cache simulator and the multicore cache simulator is a good idea. So some tests are run in order to test the performance. Three programs where run: `date`, `df` and `RWP`. The last one with each number of threads.

6.2 Running the tests

6.2.1 Verify Correctness as singlecore simulator

6.2.1.1 `date`

In order to verify the correctness the following tests were run:

```
./valgrind --tool=callgrind --multicore=8 date
```

In order to get the results of the multicore cache simulator, the results can be seen in 6.2. Furthermore numerous tests with the old cache simulator were run:

```
./valgrind --tool=callgrind --simulate-cache=yes date
```

As it can be seen in 6.1 and 6.2, these both results are exactly the same. The version of Valgrind, that is called with `-multicore=8` uses the new multicore cache simulator, the other program the old one. Both tests were run many times. But the result was every time the same. The number of `I1` misses and `L2i` misses is the same. The same is with `D1` and `L2d`.

6.2.1.2 `df`

The second program is `df`. `df` prints the size and the already used memory on each mounted device.

```
./valgrind --tool=callgrind --multicore=8 df
```

The upper line is used for the multicore cache simulator, the lower for the standard single core simulator.

```
./valgrind --tool=callgrind --simulate-cache=yes df
```

Comparing these results with the results of `date` in 6.2.1.1, the result is the same. Both outputs (See 6.3 and 6.4) are the same. This results confirm the result as the conclusion is the same.

6.2.2 Reader Writer Test

In this section, describes the results, generated by the multicore cache simulator. For this purpose, the program is used, that is described on page 56. The following listings (6.5 - 6.9) show the results that are generated by the multicore cache simulator.

Figure 6.1: date - singlecore cache simulator

```
==8336== I   refs:      2,437,584
==8336== I1  misses:        1,673
==8336== L2i misses:        1,483
==8336== I1  miss rate:       0.6%
==8336== L2i miss rate:       0.6%
==8336==
==8336== D   refs:        972,785  (708,283 rd + 264,502 wr)
==8336== D1  misses:        3,392  (  2,481 rd +     911 wr)
==8336== L2d misses:        2,486  (  1,655 rd +     831 wr)
==8336== D1  miss rate:       0.3% (    0.3%   +     0.3%  )
==8336== L2d miss rate:       0.2% (    0.2%   +     0.3%  )
==8336==
==8336== L2 refs:           5,065  (  4,154 rd +     911 wr)
==8336== L2 misses:         3,969  (  3,138 rd +     831 wr)
==8336== L2 miss rate:        0.1% (    0.0%   +     0.3%  )
```

Figure 6.2: date - multicore cache simulator

```
==8345== I   refs:      2,437,584
==8345== I1  misses:        1,673
==8345== L2i misses:        1,483
==8345== I1  miss rate:       0.6%
==8345== L2i miss rate:       0.6%
==8345==
==8345== D   refs:        972,785  (708,283 rd + 264,502 wr)
==8345== D1  misses:        3,392  (  2,481 rd +     911 wr)
==8345== L2d misses:        2,486  (  1,655 rd +     831 wr)
==8345== D1  miss rate:       0.3% (    0.3%   +     0.3%  )
==8345== L2d miss rate:       0.2% (    0.2%   +     0.3%  )
==8345==
==8345== L2 refs:           5,065  (  4,154 rd +     911 wr)
==8345== L2 misses:         3,969  (  3,138 rd +     831 wr)
==8345== L2 miss rate:        0.1% (    0.0%   +     0.3%  )
```

Figure 6.3: df - singlecore cache simulator

```
==20989== I   refs:      2,741,288
==20989== I1  misses:        1,986
==20989== L2i misses:        1,682
==20989== I1  miss rate:       0.7%
==20989== L2i miss rate:       0.6%
==20989==
==20989== D   refs:      1,090,175  (793,067 rd + 297,108 wr)
==20989== D1  misses:        3,036  (  2,173 rd +     863 wr)
==20989== L2d misses:        2,355  (  1,565 rd +     790 wr)
==20989== D1  miss rate:       0.2% (    0.2%   +     0.2%  )
==20989== L2d miss rate:       0.2% (    0.1%   +     0.2%  )
==20989==
==20989== L2 refs:           5,022  (  4,159 rd +     863 wr)
==20989== L2 misses:         4,037  (  3,247 rd +     790 wr)
==20989== L2 miss rate:        0.1% (    0.0%   +     0.2%  )
```

Figure 6.4: df - multicore cache simulator

```
==20976== I   refs:      2,741,288
==20976== I1  misses:        1,986
==20976== L2i misses:        1,682
==20976== I1  miss rate:       0.7%
==20976== L2i miss rate:       0.6%
==20976==
==20976== D   refs:      1,090,175  (793,067 rd + 297,108 wr)
==20976== D1  misses:        3,036  (  2,173 rd +     863 wr)
==20976== L2d misses:        2,355  (  1,565 rd +     790 wr)
==20976== D1  miss rate:       0.2% (    0.2%   +     0.2%  )
==20976== L2d miss rate:       0.2% (    0.1%   +     0.2%  )
==20976==
==20976== L2 refs:           5,022  (  4,159 rd +     863 wr)
==20976== L2 misses:         4,037  (  3,247 rd +     790 wr)
==20976== L2 miss rate:        0.1% (    0.0%   +     0.2%  )
```

When looking at the dualcore simulation 6.5, it can be seen, that about one million invalidations occur. Furthermore these invalidations invalidate only one other core. When than looking at the `L2d` misses, about one million misses occur when reading data. These misses are the one million misses, affected by settting the variable to another value. But the barrrier used in OpenMP seems to produce another misses, too. When writing data, Valgrind handles this as a read access followed by a write access. Thus the number of `L2d` misses when writing data is not affected, as a read access happens before. So writing to invalid data, ends in a read miss. When reaching the barrier of OpenMP, all other caches have to be invalidated. This way, the number for invalidations in 6.6, 6.7 and 6.8 can be explained. In 6.6 each third access reaches the border. So, the number of **two** invalidations is roughly exactly the third part of a million. The same is in 6.7 and 6.8. But in these cases, it is the fourth part and the eighth part of a million. All the other invalidations are a result of the operations required for the barrier. Another result that can be seen is the fact, that the work that has to be done in order to synchronise each thread increases by the number of cores. When using more cores, apparently more instructions are required in order to do synchronisation between all the cores. Another aspect that is recognisable, is the number of instructions per core. Thread 0 is the leading thread. It has got a few instructions more than each other thread. For each other thread the number of instructions is roughly the same. Thread 0 starts all the other thread and also calculates the "problem" by itself. The number of cache misses concerning instructions is so good as a very small loop is executed. Every instruction is cached in each `I1`. Furthermore when summing up the number of invalidations per write access and the number of invalidations that affects each cache, the result is exactly the same. Both values have to be the same in order to be correct.

6.2.3 Performance

In order to get compareable results, all the tests were run on a singecore system. Valgrind runs single threaded, too.

6.2.3.1 `date`

In order to test the performance, the following test runs were executed:

```
time date

time valgrind --tool=callgrind --simulate-cache=yes date

time ./valgrind --tool=callgrind --multicore=8 date
```

Figure 6.5: Reader Writer Problem - 2 cores

```
==22629== I   refs:      117,168,242
==22629== I1  misses:          1,175
==22629== L2i misses:          1,120
==22629== I1  miss rate:         0.0%
==22629== L2i miss rate:         0.0%
==22629==
==22629== D   refs:       60,696,770  (33,487,637 rd + 27,209,133 wr)
==22629== D1  misses:      2,508,981  ( 1,007,305 rd +  1,501,676 wr)
==22629== L2d misses:      1,004,924  ( 1,004,082 rd +        842 wr)
==22629== D1  miss rate:         4.1% (       3.0%   +        5.5%  )
==22629== L2d miss rate:         1.6% (       2.9%   +        0.0%  )
==22629==
==22629== L2 refs:         2,510,156  ( 1,008,480 rd +  1,501,676 wr)
==22629== L2 misses:       1,006,044  ( 1,005,202 rd +        842 wr)
==22629== L2 miss rate:          0.5% (       0.6%   +        0.0%  )
==22629==
==22629== Multicore Cache Simulator
==22629== Thread Stats: Instructions
==22629== Thread 0:59080720
==22629== Thread 1:58087522
==22629== Bus Transactions:
==22629== Thread 0: Reads:146185; Writes: 13706781; Shared Reads: 859012;
Invalidation: 1141244
==22629== Thread 1: Reads:859031; Writes: 13502365; Shared Reads: 141013;
Invalidation: 1359245
==22629== Summary:  Reads: 1005216; Writes: 27209146; Shared Reads:1000025;
Invalidation: 2500489
==22629==
==22629== Number of invalidations per write access
==22629== 1;2;<5;>4
==22629== Thread 0: 1359245,0,0,0
==22629== Thread 1: 1141244,0,0,0
```

Figure 6.6: Reader Writer Problem - 3 cores

```
==10628== I   refs:      127,665,345
==10628== I1  misses:          1,208
==10628== L2i misses:          1,153
==10628== I1  miss rate:         0.0%
==10628== L2i miss rate:         0.0%
==10628==
==10628== D   refs:       66,02 8,808  (36,487,012 rd + 29,541,796 wr)
==10628== D1  misses:      3,008,743  ( 1,340,617 rd +  1,668,126 wr)
==10628== L2d misses:      1,338,256  ( 1,337,390 rd +        866 wr)
==10628== D1  miss rate:         4.5% (       3.6%   +        5.6%  )
==10628== L2d miss rate:         2.0% (       3.6%   +        0.0%  )
==10628==
==10628== L2 refs:         3,009,951  ( 1,341,825 rd +  1,668,126 wr)
==10628== L2 misses:       1,339,409  ( 1,338,543 rd +        866 wr)
==10628== L2 miss rate:          0.6% (       0.8%   +        0.0%  )
==10628==
==10628== Multicore Cache Simulator
==10628== Thread Stats: Instructions
==10628== Thread 0:43473194
==10628== Thread 1:42101810
==10628== Thread 2:42090341
==10628== Bus Transactions:
==10628== Thread 0: Reads:234119; Writes: 9937165; Shared Reads: 646575;
Invalidation: 1000057
==10628== Thread 1: Reads:552227; Writes: 9803437; Shared Reads: 510251;
Invalidation: 1000088
==10628== Thread 2: Reads:552211; Writes: 9801199; Shared Reads: 509904;
Invalidation: 1000050
==10628== Summary:  Reads: 1338557; Writes: 29541801; Shared Reads:1666730;
Invalidation: 3000195
==10628==
==10628== Number of invalidations per write access
==10628== 1;2;<5;>4
==10628== Thread 0: 771114,104398,0,0
==10628== Thread 1: 781285,114535,0,0
==10628== Thread 2: 781122,114404,0,0
```

Figure 6.7: Reader Writer Problem - 4 cores

```
==13734== I   refs:      132,917,961
==13734== I1  misses:          1,241
==13734== L2i misses:          1,186
==13734== I1  miss rate:         0.0%
==13734== L2i miss rate:         0.0%
==13734==
==13734== D   refs:       68,696,915  (37,987,746 rd + 30,709,169 wr)
==13734== D1  misses:      3,259,033  ( 1,507,358 rd +  1,751,675 wr)
==13734== L2d misses:      1,505,020  ( 1,504,130 rd +        890 wr)
==13734== D1  miss rate:         4.7% (       3.9%   +        5.7%  )
==13734== L2d miss rate:         2.1% (       3.9%   +        0.0%  )
==13734==
==13734== L2 refs:         3,260,274  ( 1,508,599 rd +  1,751,675 wr)
==13734== L2 misses:       1,506,206  ( 1,505,316 rd +        890 wr)
==13734== L2 miss rate:          0.7% (       0.8%   +        0.0%  )
==13734==
==13734== Multicore Cache Simulator
==13734== Thread Stats: Instructions
==13734== Thread 0:34193432
==13734== Thread 1:32918420
==13734== Thread 2:32920591
==13734== Thread 3:32885518
==13734== Bus Transactions:
==13734== Thread 0: Reads:196996; Writes: 7764630; Shared Reads: 597517;
Invalidation: 807424
==13734== Thread 1: Reads:435502; Writes: 7650724; Shared Reads: 554033;
Invalidation: 813903
==13734== Thread 2: Reads:436057; Writes: 7651259; Shared Reads: 551164;
Invalidation: 814836
==13734== Thread 3: Reads:436775; Writes: 7642569; Shared Reads: 547525;
Invalidation: 814297
==13734== Summary:  Reads: 1505330; Writes: 30709182; Shared Reads:2250239;
Invalidation: 3250460
==13734==
==13734== Number of invalidations per write access
==13734== 1;2;<5;>4
==13734== Thread 0: 615600,0,58187,0
==13734== Thread 1: 628559,0,64659,0
==13734== Thread 2: 628773,0,63938,0
==13734== Thread 3: 627519,0,63219,0
```

Figure 6.8: Reader Writer Problem - 8 cores - Part 1

```
==11306== I   refs:      140,804,893
==11306== I1  misses:          1,373
==11306== L2i misses:          1,318
==11306== I1  miss rate:         0.0%
==11306== L2i miss rate:         0.0%
==11306==
==11306== D   refs:       72,703,459  (40,241,090 rd + 32,462,369 wr)
==11306== D1  misses:      3,634,995  ( 1,757,390 rd +  1,877,605 wr)
==11306== L2d misses:      1,755,134  ( 1,754,148 rd +        986 wr)
==11306== D1  miss rate:         4.9% (       4.3%   +        5.7%  )
==11306== L2d miss rate:         2.4% (       4.3%   +        0.0%  )
==11306==
==11306== L2 refs:         3,636,368  ( 1,758,763 rd +  1,877,605 wr)
==11306== L2 misses:       1,756,452  ( 1,755,466 rd +        986 wr)
==11306== L2 miss rate:          0.8% (       0.9%   +        0.0%  )
==11306==
==11306== Multicore Cache Simulator
==11306== Thread Stats: Instructions
==11306== Thread 0:18913262
==11306== Thread 1:17411489
==11306== Thread 2:17419461
==11306== Thread 3:17421096
==11306== Thread 4:17369685
==11306== Thread 5:17421984
==11306== Thread 6:17446223
==11306== Thread 7:17401693
==11306== Bus Transactions:
==11306== Thread 0: Reads:113223; Writes: 4208223; Shared Reads: 551199;
Invalidation: 448046
==11306== Thread 1: Reads:234511; Writes: 4035640; Shared Reads: 543774;
Invalidation: 453595
==11306== Thread 2: Reads:234674; Writes: 4037809; Shared Reads: 548067;
Invalidation: 454361
==11306== Thread 3: Reads:234758; Writes: 4038131; Shared Reads: 548596;
Invalidation: 454495
==11306== Thread 4: Reads:234279; Writes: 4026468; Shared Reads: 540909;
Invalidation: 452352
==11306== Thread 5: Reads:234670; Writes: 4038398; Shared Reads: 549506;
Invalidation: 454428
==11306== Thread 6: Reads:235007; Writes: 4043821; Shared Reads: 549973;
Invalidation: 455544
```

When looking at the test results, natively the program runs quite fast. Valgrind counts 2 441 210 instructions when executing `date`. When executing Callgrind, there is a runtime of 0.952 seconds. The multicore cache simulator requires 1.15 seconds. These values are average values about a number of runs.

6.2.3.2 `df`

In order to get the performance, the following three commands are executed:

```
time df

time valgrind --tool=callgrind --simulate-cache=yes df

time ./valgrind --tool=callgrind --multicore=8 df
```

When running Valgrind 2 743 443 where executed. In the first case the required time is as little as the system time counter limits the result. In the second case the average time is about 1.131 seconds. In the third case 1.396 seconds. This huge difference occurs, because there Callgrind has to be started and afterwards all the collected data have to be analysed. The same is for the multicore cache simulator.

6.2.3.3 `RWP`

The last program that was run is `RWP`. Depending on the fact if two, three, four or eight threads where run, the number of instructions differs. From 116 169 356 instructions, when doing a dual threaded run and about 139 856 094 instructions when running with eight threads. Table 6.1 show the results, gained from running the program. In order to run the tests, the following instructions where run:

```
time ./RWPx

time valgrind --tool=callgrind --simulate-cache=yes ./RWPx

time ./valgrind --tool=callgrind --multicore=8 ./RWPx
```

6.3 Conclusion

When looking at all the results, it is possible to say, that the multicore cache simulator works quite well. Singlecore operations seem to run exactly the same as the cache simulator in Callgrind. Furthermore all the results when running multicore programs seem

Table 6.1: Results running `RWPx`

Runtime in seconds.

Program	**Native**	**Callgrind**	**Multicore**
RWP2	2.571	25.15	35.98
RWP3	2.908	28.23	41.06
RWP4	3.010	29.97	41.56
RWP8	3.961	32.36	45.12

clear. Why these results occur, can be explained (See 6.2.1 and 6.2.2). Furthermore when looking at the results concerning the runtime of Valgrind, in my opinion, performance is quite well. Of course the runtime is bigger than those of the "old" cache simulator, but many other instructions have to be run. In case, the programs, that have been run in order to gain the results, the average runtime of the multicore cache simulator is about 138% to 143% compared with Callgrind. Compared with natively executing the program the simulator is about 12 to 14 times as slow. Of course these results are gained by a program, which behaviour is quite bad concerning cache coherency synchronisation.

Figure 6.9: Reader Writer Problem - 8 cores - Part 2

```
==11306== Thread 7: Reads:234358; Writes: 4033884; Shared Reads: 544045;
Invalidation: 453488
==11306== Summary:  Reads: 1755480; Writes: 32462374; Shared Reads:4376069;
Invalidation: 3626309
==11306==
==11306== Number of invalidations per write access
==11306== 1;2;<5;>4
==11306== Thread 0: 339986,0,0,16961
==11306== Thread 1: 344258,0,0,15666
==11306== Thread 2: 344708,0,0,15348
==11306== Thread 3: 344758,0,0,15263
==11306== Thread 4: 343091,0,0,15740
==11306== Thread 5: 344775,0,0,15347
==11306== Thread 6: 345557,0,0,15014
==11306== Thread 7: 344155,0,0,15664
```

Chapter 7

Future

After getting all the results, the following questions maybe can be answered: Can be done any more? Are there further possibilities? What kind of improvements can implemented?

Implement the simulator a threaded way This could greatly improve the performance of the simulator. But the cache simulator in Valgrind is implemented in a serial way. So maybe it will be difficult.

Commonly increase performance Another chance is to read the whole code and try to improve the performance of the simulator. For sure there are pieces of code, that can improve the performance.

Add improved reporting features In order to be able to get more results, or to be able to get other also interesting results, it could be a good idea to improve the number of counters, within the multicore cache simulator.

Recognising False Sharing Based on the multicore cache simulator, it is easily possible to improve this simulator. So analysing if false sharing occurred could be a good idea for further implementations.

Implement write back Furthermore implementing write back can be a good idea, as the current multicore cache simulator is implemented as a write through cache system. But this would only change the result of the number of bus transactions. Maybe it is possible to "simulate" a write back, based on the current write through.

Implement more than two cache levels All new processor architectures own more than two cache levels. So it could be a good idea to support e.g. three cache levels. But using three cache levels maybe requires support for a shared memory cache system. For this idea please look the next point.

Add a shared memory cache system Currently each processor has its private cache system. There is no shared cache used within the multicore cache simulator, as well as in the old cache simulator. As all the processors currently available use a shared memory cache system, the results of the simulator maybe can be improved.

Add code line support Currently, all the data collected in the multicore cache simulator, can not be assigned to a cache line. By replacing the variables, that count the number of a value, only by a function call that assigns as special counter to a code line, such a support is added.

Gain results compared to a real machine As of time reasons, no results could be gained, in order to compare the multicore cache simulator with a real machine. For sure it is a good idea to do this.

Summary

Taking everything into account, this multicore cache simulator is able to simulate a computer's multicore cache system. A progammer is possible to recognise a problem concerning synchronisation between each single core of a computer system.

Of course this simulator is not perfectly exact at all, but it delivers good results. But simulating a modern multicore or multiprocessor system exactly is not possible without spending a lot of time. Every new architechture has got different features concerning the cache hierarchy. E.g. the current AMD Barcelona is able to directly communicate between each `L1` cache. Implementing this procedure would definately increase the amount of work to spend enormously. So this multicore cache simulator is compared to a real multicore cache system quite simple. But in order to gain valuable results this simulator is good enough. Furthermore this is a multicore cache simulator and not a multicore cache emulator. The increased runtime of the simulator is caused by the many additional instructions that is required to simulate all the multicore operations and get valuable information about it.

Bibliography

[AB86] James Archibald and Jean-Loup Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 4(4):273–298, 1986.

[AMD07] AMD. *Family 10h AMD Phenom^{TM} Processor Product Data Sheet.* Advanced Micro Devices, revision 3.00 edition, November 2007.

[FGH07] Robert Franz, Stephan Günther, and Alexander Heinecke. Hardware-nahe implementierung schneller kernel für matrixoperationen. Studienarbeit/sep/idp, Institut für Informatik, TU München, May 2007.

[HPM03] Klaus Dembowski Hans-Peter Messmer. *PC-Hardwarebuch.* Addison-Wesley, 2003.

[MPI08] The homepage of mpi. http://www-unix.mcs.anl.gov/mpi/, 2008.

[Ope08] The homepage of openmp. http://openmp.org, 2008.

[Wik08] Wikipedia. Binary prefix — wikipedia, the free encyclopedia, 2008. [Online; accessed 12-April-2008].

List of Tables

List of Figures

Listings

Index

Glossary

Binary Prefix	Special identifier when not using 10^n but 2^n. E.g. KiByte $2^{10} = 1024$ instead of $10^3 = 1000$ [Wik08]
Cache-Hit	When requesting data, it can be found in the caches memory
Cache-Miss	When requesting data, it can not be found in the caches memory
Callgrind	A tool within Valgrind for profiling call graphs
EiByte	See “Binary Prefix”
First Level Cache	A cache next to CPU
FLC	See “First Level Cache”
GiByte	See “Binary Prefix”
KiByte	See “Binary Prefix”
L1D	L1D-cache, first level data cache
L1I	L1I-cache, first level instruction cache
Last Level Cache	A cache next to main memory
LLC	See “Last Level Cache”
LRU	Least Recently Used, a page replacement algorithm
MESI	Also known as Illinois Protocol, a cache coherency protocol
MiByte	See “Binary Prefix”
PiByte	See “Binary Prefix”
TiByte	See “Binary Prefix”
Valgrind	Valgrind is an award-winning instrumentation framework for building dynamic analysis tools

Appendix A

Source Code

This appendix contains all the files modified during working on this bachelor's thesis. For the sake of completeness, the whole source code that has been modified, is listed in here. Printing all the code would require ca. 100 additional pages. Because this really would exceed the size of this thesis I decided not to do it this way. So, instead of printing the whole code on a huge number of pages, the code is provided on a CD-ROM. Furthermore the code can be found on my homepage.

`http://www.robert-franz.com`

Appendix B

GPL

As well as Valgrind and Callgrind, the code implemented for this thesis is under GNU General Public License. The whole GNU General Public License can be found when following the this address:

`http://www.gnu.org/licenses/gpl.html`