Cache Behaviour in Multi-/Many-Core Environment

N-Core Configurable Cache Simulator
Performance Evaluation

Magnus Øren

Thesis submitted for the degree of
Master in Programming and Network
60 credits

Department of Informatics
Faculty of Mathematics and Natural Sciences

University of Oslo

May 2019
Cache Behaviour in a Multi-/Many-Core Environment: N-Core Configurable Cache Simulator Performance Evaluation

Magnus Øren
© Magnus Øren

2019

Cache Behaviour in a Multi-/Many-core Environment: N-Core Configurable Cache Simulator Performance Evaluation

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo
Abstract

The rapid increase in the number of processors demands quicker and more reliant data availability to avoid extensive memory accesses. Caches hierarchies are most important in an efficient n-core system. However, more processors often imply more complexity in cache design. This thesis investigates various cache configurations in single-, multi- and many-core environments with a cache simulator developed for this thesis. The cache simulator is a highly configurable n-core simulator implemented with the MESIF cache coherency protocol. Over various configurations, some of the results include findings such as rapid increase in the occurrence of thrashing increasing with the number of cores and the relationship between compulsory misses, coherence misses and block size. Furthermore, the thesis investigates how various configurations can reduce the number of false sharing misses. The results are indicating a possible solution to the problem, but at the cost of an increasing number of compulsory misses.
Acknowledgements

I would like to thank my supervisors; Einar Broch Johnsen and Violet Ka I Pun at the University of Oslo for assisting me in this thesis. I would like to thank them for a great, constructive and educational process.

Thanks to friends and family for patience throughout the period working on this thesis.
3.1.3 Memory designs ........................................................................................................ 20
3.1.4 Thrashing .................................................................................................................. 21
3.2 Caches: Closing the gap between memory and CPU performance ............................. 22
3.2.1 Locality of reference ................................................................................................. 22
3.2.2 Characteristics of caches .......................................................................................... 23
3.2.3 Capacity, block size and associativity ....................................................................... 24
3.2.4 Cache misses: 3C’s ..................................................................................................... 24
3.3 Replacement policies ................................................................................................... 25
3.3.1 Least-recently used ................................................................................................. 25
3.3.2 Most recently used .................................................................................................... 25
3.3.3 Pseudo LRU ............................................................................................................. 25
3.3.4 Random replacement ............................................................................................... 26
3.3.5 Least frequently used ............................................................................................... 26
3.3.6 How caches work: The basics .................................................................................. 26
3.4 Multi-level caches ....................................................................................................... 27
3.4.1 Inclusion policy ......................................................................................................... 27
3.4.2 Intuition on inclusion policies .................................................................................. 28
3.5 Caches in a multicore environment ............................................................................. 29
3.5.1 Memory consistency ................................................................................................. 29
3.5.2 Cache coherency ...................................................................................................... 30
3.5.3 Coherency Models .................................................................................................... 31
3.5.4 Coherence misses: The 4th C .................................................................................. 31
3.6 Cache issues, risks and events .................................................................................... 32
3.6.1 Cache thrashing also known as Cache line ping-ponging ...................................... 32
3.6.2 True sharing ............................................................................................................. 32
3.6.3 False sharing ............................................................................................................ 32
3.7 Cache coherency protocol .......................................................................................... 33
3.7.1 MSI based protocol ................................................................................................. 33
3.7.2 Other known coherency protocols ........................................................................... 36
3.7.3 How caches work: Synchronization in caches ......................................................... 36
4 Optimizing Cache behaviour ......................................................................................... 39
4.1 Configurations ............................................................................................................. 39
4.1.1 Impact of capacity, block size and associativity ....................................................... 39
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.1</td>
<td>Input</td>
<td>56</td>
</tr>
<tr>
<td>5.2.2</td>
<td>ISA: x86</td>
<td>56</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Output</td>
<td>56</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Design of simulated architecture</td>
<td>57</td>
</tr>
<tr>
<td>5.3</td>
<td>Creating an abstract model of cache behavior</td>
<td>57</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Cache mechanism</td>
<td>57</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Memory addressing model</td>
<td>61</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Processors and classifications</td>
<td>61</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Main memory</td>
<td>62</td>
</tr>
<tr>
<td>5.3.5</td>
<td>Cache</td>
<td>63</td>
</tr>
<tr>
<td>5.3.6</td>
<td>Registers</td>
<td>65</td>
</tr>
<tr>
<td>5.4</td>
<td>Running a simulation</td>
<td>66</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Reading from file and constructing instructions</td>
<td>66</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Translating from virtual address to physical address</td>
<td>67</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Branches and JMP</td>
<td>67</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Pre-execution stage</td>
<td>67</td>
</tr>
<tr>
<td>5.4.5</td>
<td>Execution stage</td>
<td>68</td>
</tr>
<tr>
<td>5.4.6</td>
<td>Affinity</td>
<td>71</td>
</tr>
<tr>
<td>5.4.7</td>
<td>Random process generation</td>
<td>72</td>
</tr>
<tr>
<td>5.4.8</td>
<td>The bus</td>
<td>72</td>
</tr>
<tr>
<td>5.4.9</td>
<td>Main controller</td>
<td>72</td>
</tr>
<tr>
<td>5.5</td>
<td>Problems, issues and limitations</td>
<td>74</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Scheduler</td>
<td>74</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Instruction length</td>
<td>74</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Memory constraints</td>
<td>74</td>
</tr>
<tr>
<td>6</td>
<td>Optimal cache configuration: The results</td>
<td>75</td>
</tr>
<tr>
<td>6.1</td>
<td>The basics: A view on single-level cache behavior</td>
<td>75</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Preliminary assumptions for single-processor</td>
<td>75</td>
</tr>
<tr>
<td>6.1.2</td>
<td>The single-processor</td>
<td>75</td>
</tr>
<tr>
<td>6.2</td>
<td>Adding more processors: Adding parallelism</td>
<td>81</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Preliminary assumptions for multi-processor</td>
<td>81</td>
</tr>
<tr>
<td>6.2.2</td>
<td>The Multi-processor: Results</td>
<td>81</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Preliminary assumptions for many-processor</td>
<td>87</td>
</tr>
</tbody>
</table>
6.3 Moving to many-core architectures ................................................. 87
   6.3.1 The Many-processor: Results .................................................. 87
   6.3.2 Discussion .............................................................................. 94
6.4 Adding another cache: Near-optimal solution for single-processors ........ 94
   6.4.1 Expectations ........................................................................... 94
   6.4.2 Single-processors: Results ...................................................... 94
6.5 Adding another cache: Near-optimal solution for parallel computing .......... 96
   6.5.1 Expectations ........................................................................... 96
   6.5.2 Many-core: Are multi-level caches efficient? ............................ 97
6.6 Discussion .................................................................................... 101
   6.6.1 Single-level setup .................................................................... 101
   6.6.2 Partial block invalidation ........................................................ 101
   6.6.3 Multi-level setup ..................................................................... 101
   6.6.4 Many-core performance: The declining parallelization factor ....... 102
   6.6.5 Comparing results to the VTune Amplifier 2019 ....................... 103
7 Conclusion and further works ................................................................ 105
   7.1 The performance of a single-level cache in single-processors ............ 105
      7.1.1 Optimal configuration ............................................................ 105
      7.1.2 Enabling prefetching mechanism .......................................... 105
      7.1.3 Share of memory access instructions ..................................... 106
   7.2 Multi-processors: Cache performance in a single-level cache ............ 106
      7.2.1 The 4th C: Sharing misses and thrashing .................................. 106
      7.2.2 Reducing false sharing .......................................................... 106
      7.2.3 Limitations to reduced block size model with MESIF protocol .... 107
      7.2.4 An increase of compulsory misses ........................................ 107
   7.3 Many-processors: Performance evaluation of a single-level cache ....... 107
      7.3.1 Little increase in memory access .......................................... 107
      7.3.2 Relative usage of processors ................................................. 107
      7.3.3 Evaluating performance ....................................................... 108
   7.4 Multi-level cache in single-processors: A comparison of inclusion policies 108
      7.4.1 Multi-level versus single-level caches .................................... 108
   7.5 Multi-level cache in many-processors: Review and limitation .......... 109
      7.5.1 The inverse ........................................................................ 109
    XIII
List of figures

Figure 1: x86 instruction ................................................................. 14
Figure 2: 32-bit register structure .................................................. 18
Figure 3: 64-bit register structure .................................................. 19
Figure 4: Graph colouring ............................................................... 20
Figure 5: NUMA architecture ......................................................... 21
Figure 6: Address ........................................................................... 23
Figure 7: Cache ............................................................................... 24
Figure 8: R/W cache simple ............................................................. 27
Figure 9: Inclusive cache mechanism .............................................. 28
Figure 10: MSI-protocol state diagram .......................................... 34
Figure 11: MESI state diagram ....................................................... 35
Figure 12: Write miss .................................................................... 37
Figure 13: LRU queue .................................................................... 52
Figure 14: Simulated architecture ................................................. 57
Figure 15: Exclusive cache .............................................................. 58
Figure 16: MESIF protocol ............................................................. 60
Figure 17: Segmented memory model ............................................. 61
Figure 18: System architecture ....................................................... 73
List of tables

Table 1: Loop tiling results explanation .................................................................................. 44
Table 2: Single-core and single-level NINE cache ................................................................. 76
Table 3: Single-core and single-level inclusive cache ............................................................ 76
Table 4: Single-core and single-level exclusive cache ......................................................... 77
Table 5: Single-core and single-level NINE cache with prefetcher .................................... 77
Table 6: Single-core and single-level Inclusive cache with prefetcher ............................... 78
Table 7: Single-core and single-level exclusive cache with prefetcher ............................. 78
Table 8: Assumptions for hit time, miss penalty and base CPI ........................................... 79
Table 9: Increased block size ................................................................................................ 79
Table 10: Increased capacity ................................................................................................ 80
Table 11: Increased associativity .......................................................................................... 80
Table 12: Enabling prefetcher .............................................................................................. 81
Table 13: Dual-core, single-level cache with near-optimal configuration ......................... 82
Table 14: Quad-core single-level cache with near-optimal configuration ......................... 82
Table 15: 16-core, single-level cache with near-optimal configuration ............................... 83
Table 16: Reducing block size to reduce the occurrence of false sharing ......................... 84
Table 17: Eliminating false sharing ...................................................................................... 84
Table 18: ollydbg2.exe on near-optimal configuration ......................................................... 85
Table 19: iexplorer.exe on near-optimal configuration ....................................................... 86
Table 20: photodirector.exe on near-optimal configurations ............................................... 86
Table 21: 32-core, single level cache .................................................................................. 88
Table 22: 128-core, single level cache ................................................................................. 88
Table 23: 512-core, single level cache ................................................................................ 89
Table 24: Performance results ............................................................................................. 91
Table 25: Single-processor, inclusive multi-level cache ..................................................... 94
Table 26: Single-processor, exclusive multi-level cache ...................................................... 95
Table 27: Single-processor, NINE multi-level cache ........................................................... 96
Table 28: 32-core, inclusive multi-level cache .................................................................... 97
Table 29: 128-core, inclusive multi-level cache .................................................................. 97
Table 30: 512-core, inclusive multi-level cache .................................................................. 98
Table 31: Decreasing L1 size over 32-cores ....................................................................... 98
Table 32: Increasing L2 capacity cache over 32-cores ....................................................... 99
Table 33: 32-Core, exclusive multi-level cache ................................................................... 99
Table 34: 32-Core, exclusive multi-level cache with reduced capacity ............................ 100
Table 35: 32-Core, exclusive multi-level cache reduced capacity and increase associativity 100
List of graphs

Graph 1: The relationship between coherence and compulsory misses per block size........ 85
Graph 2: Comparison of test programs .............................................................................. 87
Graph 3: Relative CPU usage.......................................................................................... 89
Graph 4: The evolution of P and S over n-processors for firefox.exe ................................ 91
Graph 5: Speed-up over n-processors firefox.exe ............................................................. 93
Graph 6: Parallelization factor for all test programs on 32-cores ........................................ 102
Graph 7: P and S for test programs with separate registers ............................................. 103
List of code examples

Code 1: Loop unrolling ........................................................................................................... 42
Code 2: Software pipelining ..................................................................................................... 42
Code 3: No loop tiling .................................................................................................................. 43
Code 4: Loop tiling ..................................................................................................................... 43
Code 5: Loop interchange ......................................................................................................... 44
Code 6: Loop fusion .................................................................................................................... 45
Code 7: Loop fission ................................................................................................................... 46
Code 8: Peer cache hit ............................................................................................................... 64
Code 9: Execution section A/B .................................................................................................. 70
Code 10: Section C ..................................................................................................................... 71
1 Introduction

1.1 General overview

Modern day computing is parallel. The number of processors is increasing, thus potentially increasing performance. In a homogenous multi-core environment, most common in general-purpose machine, responsibility for each task is shared between all processors. This requires shared memory models and memory structure as well to store private data for quicker access. Implementing caches is traditional approach to this requirement.

1.1.1 Caches

A cache is a memory unit located on the processor or close to it. The purpose of a cache is to provide quicker access to data. An increase in the number of processors is not enough to increase performance alone. Since main memory is located farther from the processor and relies on an omnibus to transfer data, it is often the bottleneck in sense of performance. The implementation of caches partially solves these issues. Although not as quick to access as registers, caches drastically improve performance. To further increase performance, multiple levels of caches can be implemented. To minimize main memory accesses, a second, third or even fourth level of cache can be implemented. A main memory access can be very time consuming, thus each level should have access times lower than the main memory has.

Caches can mainly be configured in three ways. The configurations of a cache relate to capacity, block size and associativity. Capacity is the size of the cache and can be further divided to sets. Each set can then be divided to cache blocks. The configuration block size refers to the size of a cache block. Block size is normally less than capacity. The number of blocks is decided by capacity, number of sets and associativity. The latter describes the number of possible ways to store a piece of data. There can be more than one cache in a computer and it is called a multilevel cache. Each of these caches can have different configurations. However, in the case of \( n \)-level caches, the \( n \)-level cache (\( L_n \)) is at least as large as the upper-level cache (\( L_{n-1} \)) is.

Both single- and multi-processor often implements multilevel caches. In the case of multi-processors, the caches can be shared between each processor or private to each processor. Often multi-processors implement both shared and private caches. Usually upper-level caches are shared, and lower-level caches are private (non-shared).

Capacity, block size and associativity that are optimal for a single-processor may not be optimal in a multi-processor environment. More processors imply more complex systems to increase performance. Increased complexity can often be related to performance and often can increase latency. Especially in systems with multiple processors and multiple levels of cache
both shared and non-shared. However, the advantage with more caches is the decreased need for main memory access, thus decreasing memory bandwidth.

Multi-processor with multiple caches each has the benefit of increased capacity, without the disadvantages of one very large cache. A large cache will normally have increased latency and depending associativity, increased access times. Smaller, but more caches will benefit from the capacity combined and it may increase performance.

1.1.2 Motivation: The evolution of computers

As the number of processors increase for each new generation of CPUs, the general-purpose computer is becoming vastly more powerful. The next step in the evolution of computers is most likely many-core computers. Many-core is not a standard term, but in this thesis, it will be referred to as a computer with more than 16-processors. With the potential increase in speed, an optimized cache hierarchy can help improve performance further. Worst case scenario, caches becomes a bottleneck like main memory is to computers in general. If programs are poorly optimized for more processors, then it reduces the need for more processors. However, if a program can be subdivided into smaller tasks, it can benefit more from multiple cores. It does not always rely on developer, but also how programs are optimized e.g. a through a compiler.

In basic terms, cache behaviour is related to performance and the efficiency of a cache. A poorly configured cache behaves differently from an optimized cache. This can be measured in the number of cache **hits and misses**. Whenever a referenced piece of data is in the cache, it is called a cache hit. The opposite is called a cache miss and refers to the scenario where the referenced data is not in the cache and it forces the system to check other levels of caches and/or peer caches, or it must fetch the data from main memory.

One of the goals is to understand cache behaviour in a many-core environment compared to a single-core environment. This knowledge can be used to find optimal configuration and the optimal number of processors for a given program.

1.1.3 Optimal configurations

The main goal of the project is to be able to determine the optimal cache configuration of a given program in a $n$-core environment. The results can be utilized in code production and it can help e.g. programmers to optimize their code regarding caches. Moreover, the results can help further understand the mechanism and behaviours of a cache in multi- and many-core environment.
1.1.4 Optimal number of processors

To ascertain an optimal configuration, it is important to determine the number of cores required for a program to run. If the granularity of the program and tasks is the number of processes that can run in parallel, then the caches should be configured accordingly.

1.2 The problem

1.2.1 The evolution of cache memory

The recent trend is that caches become larger as larger programs need more memory space. There are several factors influencing performance. However, there is a reason why caches are smaller than main memory. Thus, there is a limit to how large they can become. Perhaps more caches are the solution rather than capacity. In a many-core environment, there are more concerns to address. If a processor is required to fetch data from main memory, there are many unknowns. It would be interesting to identify exactly what the problems related to cache behaviour are, and how cache behaviour relates to the configurations of the cache systems in an \( n \)-core environments.

1.2.2 How efficient are caches in environments with \( n \)-processors?

Cache behaviour in a many-core environment is of recent relevance. The added complexity and parallelism are bound to influence performance. Complexity can relate to coherence and consistency in the memory hierarchy which may degrade performance. Parallelism is the ability to subdivide tasks into smaller task, which may increase performance. Coherence issues such as sharing, and overhead can negate the influence of a parallelism. In many-core caches there is inter-cache communication which provide some latency, but less than main memory access does. Moreover, inter-cache communication reduces the need for memory bandwidth and this induce better performance.

Configurations adds complexity and mostly when concerning associativity. In a multi-/many-core environment, how does this influence cache misses? How does capacity and block size influence cache misses? Is there are correlation between sharing and misses? If so, how to reduce the occurrence of these misses. What is the optimal configuration for single program in a many-/multi-core environment, if it is already compiled and optimized?

Many programs utilize parallelism, but some are also developed for a single processor. How can these programs be subdivided to smaller tasks and does it improve performance? Is there a difference in performance from a single processor to many-processor architectures? Naturally, a parallel computation is faster than a non-parallel computation when a task can be subdivided into smaller tasks, but what about cache behaviour? Does cache behaviour impact performance in such way that surpasses the impact of added parallelism? To what degree does
cache behaviour influence performance in an $n$-core environment? To understand these problems, a simplified yet powerful method has been developed.

### 1.2.3 Method

To achieve an approximation of the cache behaviour for a single program, a simulator (Øren, 2019) is developed for this thesis. The simulator is flexible and can simulate many scenarios and configurations.

Some of the main features are:

1. Cache behaviour simulation in an n-core environment
2. Multi-level non-shared caches
3. Prefetching
4. Advanced cache coherency protocol simulation
5. Parser for x86 instructions
6. Main memory simulation
7. Out-of-order inspired execution
8. Easy-to-use graphical user interface

The simulator is not only a cache simulator, but also a program simulator.

### 1.2.4 How can the method identify and help understand the cache behavior?

A $n$-core simulator based on a simplified model of a computer, provides the necessary environment to identify the problems. The key to understanding the problems is analysis of the data provided by the simulator. It would be interesting to understand the mechanism that causes cache behavioural problems. Also, it is interesting to understand the problems from an abstract point of view and to compare the results to concrete data from other sources. Are the simulations able to depict cache environments with great accuracy?

### 1.3 Related works

There is done much research on this topic for single- and multi-processors, but due to the recency of general-purpose many-core machines there is little research done in this field.

#### 1.3.1 Sniper

The Sniper simulator is project developed in Belgium. It is an accurate x86 many-core simulator with many features. It is based on an interval-core model which makes it faster than many other simulators which use a more mechanistic approach. The scope of the sniper simulation is huge. Among the features is a cache simulator which focus on both shared and
private caches. Sniper simulator’s cache is highly configurable (Carlson, Heirman, Eeckhout, & Hur, 2019).

1.3.2 VTune Amplifier 2019

This is a tracing program developed by Intel and it is not a simulator. It presents accurate results based on real-time data. In addition to examining memory bandwidth it can provide accurate measurements of core usage and a general performance metrics. In VTune AM, every procedure call is measured. Therefore, the level of detail provided by this program can to a high degree help the development of programs (Intel, 2019).

1.4 Chapter overview

Chapter 2: An in-dept insight to single-, multi- and many-core architectures. The most important performance metrics; Amdahl’s law and Gustafson – Barsis’s law as well as Pollack’s rule. In addition to this there is comparison between single- and multi-core architecture, the general-purpose machine versus the special purpose machine.

Chapter 3: Key memory components: Main memory and registers, key concepts such as cache coherency and memory consistency, use of replacement policies and a description of how caches work.

Chapter 4: Many techniques and technologies are utilized to optimize caches from a hardware and software perspective. This chapter explains many key optimization techniques from compiler techniques, virtual memory to hardware technologies such as adaptive caches. A more in-depth view on replacement policies.

Chapter 5: This chapter describe the methodology and a description the simulator. An in-depth insight to the features and techniques utilized to develop the simulator, as well as problems and issues related to the development.

Chapter 6: The results are presented with a discussion and comparison are made between single-, multi- and many-core environments. Moreover, the performance metrics are presented.

Chapter 7: After results are presented in chapter 6, conclusion and further work continues in chapter 7.
2 Processor architectures: Single-, multi- and many-core

Keywords: An in-depth view on computer architecture. Pollack’s rule, Amdahl’s law, Gustafson – Barsis’s law. Parallel computing. Instruction Set Architectures (ISA).

2.1 The processor

Parallel computing has existed for decades, but modern advances allowed integration in general-purpose computers. As the core number increases, the level of parallelism should increase as well. However, in some cases the single-core processor is more suited for certain tasks.

2.1.1 Single-core processor

Uniprocessors can be called single core processors, however single core is newer term, which refers to the number of CPU on the microchip. This is what Flynn’s taxonomy describes as a SISD. Since a single processor can only process instructions sequentially, the single core computer is reliant on clock frequency to improve performance. To measure the performance of single core computers, Pollack’s rule is best applied. This rule which can be described mathematically as:

\[ P = \sqrt{N} \]

where \( P \) = performance and \( N \) = number of transistors in the area, is an approximation based on empirical studies. This approximation is an indication that a doubling of transistor will result in a nearly 40% increase in performance (Borkar, 2011). In example:

\[ 1 \approx \sqrt{1} \text{ to } 1.4 \approx \sqrt{2} \text{ to } 2 = \sqrt{4}. \]

This example illustrates that to double performance the number of transistors must be four times the initial number of transistors. To further extent to this notion, with respect to Moore’s law it would take at least 36 months to double performance. Additionally, increasing clock frequency can further increase performance by 40% (Borkar, 2011). This metric alone does not describe speedup in a multi-/many-core environment. This requires different metrics.
2.1.2 Multi-core processor

A microchip with more than one core is called a multicore processor. This type of computer can execute instructions in parallel, which with respect to performance, this can be very advantageous. In Flynn’s Taxonomy the typical classification of a multi-core computer is MIMD (Flynn, 1972), which more precisely can be derived to SPMD or MPMD as an extension of the taxonomy. However, it also has the capabilities of a SIMD. In some tasks multiple processor can work on the same data set in parallel. In example, vector computing (Barlas, 2014). A multi-core computer can have different properties, such as homogenous cores or heterogenous cores.

A multicore computer often has homogenous cores. Homogenous cores imply identical specifications such as process power, memory design and ISA. This often implements symmetric multi-processing (SMP) and it is the current trend. An asymmetric multi-processing machine (AMP) is machine where cores have non-identical functionality e.g. different responsibilities, I/O attachments. As opposed to SMP, AMP can both utilize homogenous and heterogenous cores. As an example, an AMP can run different OS, on different cores. There is a clear distinction between AMP and heterogenous cores. A heterogenous core refers to the difference in various defining aspects of a processor. As a non-identical core, it may differ in processing power, cache design, memory model and ISA (Cowart & Knittel, 2000; Mittal, Balas, Hermanth, & Kumar, 2018)

A metric with respect to performance is Amdahl’s law (Amdahl, 1967). This metric describes performance through latency. In the context of performance, it is very useful: It is derived from notion of:

\[ T_{after} = \frac{T_{improved}}{Speedup} + T_{unaffected} \]

which can be rearranged to express:

\[ S = \frac{old\ execution\ time}{new\ execution\ time} = \frac{T_1}{T_2} \]

\[ T_1 = T_{serial} + T_{parallel} \times n \]

Where \( T_{serial} = I_{serial} \)

if the number of instructions is equal to time consumed.

And

\[ (T_{parallel} \times n) = I_{parallel} \]
If the number of the instructions is the time that would be consumed executing the instruction in serial.

\[ T_2 = I_{\text{serial}} + \left( \frac{I_{\text{parallel}}}{n} \right) \]

\( n \) is average use of CPUs in parallel and \( I_{\text{serial}} \) is the number of instructions executed in serial and \( I_{\text{parallel}} \) is the number of instructions executed in parallel

Therefore:

\[ S = \frac{(I_{\text{serial}} + I_{\text{parallel}})}{I_{\text{serial}} + \left( \frac{I_{\text{parallel}}}{n} \right)} \]

Assuming:

\[ I_{\text{serial}} + I_{\text{parallel}} = 1 \]

And

\[ (1 - p) + p = 1 \]

In this, \( p \) is the parallel fraction and \( (1 - p) \) is the serial fraction

Then:

\[ S = \frac{(1 - p) + p}{(1 - p) + \frac{p}{n}} \]

The end formula is equal to Amdahl’s law is:

\[ S = \frac{1}{(1 - p) + \frac{p}{n}} \]

In this formula \( n \) is the number of processor or the speedup with respect parallel execution.

**Amdahl’s law** (Amdahl, 1967) describes performance in a parallel system. However, the law assumes constant small workloads. In a real-world computer, workloads can be larger. However, even if it is not accurate, **Amdahl’s law** is correct. Also, overhead is ignored, and estimates are too optimistic for parallel problems while being too pessimistic for less parallel and sequential problems.

A more precise and accurate estimate is the **Gustafson – Barsis’s** (Gustafson, 1988). It is comparable to **Amdahl’s law**, but solves the issues with large workloads. It describes a scalable theoretical speed-up in latency. Gustafson’s law assumes \( N \) processors as Amdahl’s law does. Moreover, it also ignores overhead, which consequently can achieve an overestimate of speed-up (Padua, 2011).
Gustafson – Barsis’s:

Assume that the time $s$ spent in serial execution is:

$$s = \frac{I_{\text{serial}}}{I_{\text{serial}} + \frac{I_{\text{parallel}}}{n}}$$

Then:

$$(1 - s) = \frac{I_{\text{parallel}}/n}{I_{\text{serial}} + I_{\text{parallel}}/n}$$

Substitute $I_{\text{serial}} and I_{\text{parallel}} with (1 - p)$ and $p$, respectively.

Then:

$$I_{\text{parallel}} = (1 - s) \times \left( (1 - p) + \frac{p}{n} \right) \times n$$

And

$$I_{\text{serial}} = s \times ((1 - p) + \frac{p}{n})$$

Insert both the expression in to Amdahl’s Law:

$$S_{\text{scaled}} = \frac{(s \times \left( (1 - p) + \frac{p}{n} \right) + ((1 - s) \times \left( (1 - p) + \frac{p}{n} \right)) \times n}{(1 - p) + \frac{p}{n}}$$

Can become:

$$S_{\text{scaled}} = (s + (1 - s) \times n) \times \frac{(1 - p) + \frac{p}{n}}{(1 - p) + \frac{p}{n}}$$

$$= (s + (1 - s) \times n) \times 1$$

Then scaled speed-up is:

$$S_{\text{scaled}} = s + (1 - s) \times n$$

Notice how this formula just as easy could be derived from

$$T_1 = T_{\text{serial}} + T_{\text{parallel}} \times n$$

And:

$$T_2 = T_{\text{serial}} + T_{\text{parallel}}$$
When calculating speedup $S_{scaled}$, then

$$S_{scaled} = \frac{(T_{serial} + T_{parallel} \times n)}{T_{Serial} + T_{parallel}}$$

Assume:

$$T_{Serial} + T_{Parallel} = 1$$

Then:

$$S_{scaled} = T_{serial} + T_{parallel} \times n$$

This can be written as:

$$S_{scaled} = (1 - p) + p \times n$$

Note that $p$ in Gustafson – Barsis’s law is a bit different from $p$ in Amdahl’s law.

These formula transformations are done with help from different sources (Quinn, 2004) and from the paper proposing this metric in 1988, by John L. Gustafson. The last attempt to find $S_{scaled}$ is based on the preliminary assumptions for Amdahl’s laws.

The Gustafson – Barsis’s (Gustafson, 1988) assumes that a more powerful computer must be used to solve larger problems. Therefore, the parallel portion of a program is increasingly larger for larger programs (Padua, 2011).

### 2.1.3 Many-core processor

It is not quite defined what many core processors is, other than several cores exceeding 10s 100s or 1000s of cores or even millions of cores. It is vaguely defined. However, many problems relate to this. Since it can be assumed that complexity will increase with the number of cores, many issues that also currently exist with multi-core and less will grow. Many issues related to memory, cache and software development will be prominent.
2.2 Advantages of parallel computing: Performance differences between single- and multi-core computing.

The effect of parallel computing is reliant on the granularity of the tasks that are executed. Even if a multi-core system is faster than a single-core system, if the program is less fine-grained, the tasks may only be executed sequentially.

2.2.1 Parallel Computing

It is quite common that multicore computer relies on a shared memory model. Since fine-grained parallelism often have high communication and synchronization overhead, a shared memory model can reduce communication overhead. There is often is only one main memory unit in addition to shared caches. In a parallel system, granularity of a program decides the level of parallelism. In case of fine-grained (granularity) parallelism, synchronization and communication is essential (Mandiviwala, Ramachandran, & Knobe, 2008). This often provided by a bus or multiple buses. Parallel and concurrent tasks are often mapped by a scheduler and is often handled automatically by a system.

Synchronisation is important in a multicore system. To avoid contention between processes, synchronisation is enforced by mutex or atomic statements. As with fine-grained parallelism synchronisation overhead can be large and to reduce synchronisation, atomicity is important. The consequence of not implementing synchronisation methods is race conditions. Implementing mutex in critical sections and ensure that a program is executed in a valid order reduces the risks of incorrect results (Andrews, 2000).

2.2.2 A comparison between single- and multi-core computing

It is a great advantage to be able to execute instructions, tasks and data transfers in parallel. A single core computer is very reliant on clock frequency and number of transistors for increased performance. However, this is not true for all cases. A single core computer is concerned with less complex systems and depending on granularity of tasks might perform just as good as a multi core system on an individual level. The single core computer is not equally reliant on complex systems, shared memory, synchronisation and cache protocols to maintain a decent performance. One problem associated with single core computers is heat emissions. If clock frequency is doubled, the heat emissions might increase with a factor of eight. This is one of the reasons why multi-core is more suited for increasing performance. Because multi core systems can perform to a high degree on lower clock frequency, power consumption can be vastly reduced and therefore heat emissions might be decreased. It has been shown that if processor speed is reduced by 20%, the power consumption is reduced by 50% and performance is only reduced by 13%. A dual-core architecture can increase
performance by 73%, at same time decrease power consumption and be comparable to single core processor with 20% reduced clock speed (Ross, 2008).

Another aspect of multicore computing is software development. Since the parallelism is very much dependant on granularity and can be complex, not all program tasks can be run in parallel. When deciding between an architecture for a device, the granularity of the tasks performed is important. A single core processor might be better suited for one specific task, while multi-core processor might be suitable if there is more granularity. Some examples are smart phones and PCs. Smart phone and PCs are often concerned with a great number task at once. Because of this multicore processing is more beneficial. A single core processor would stagger and often must have a higher clock frequency to be comparable in terms of performance. In the case of higher clock frequency, it would mean a higher power consumption which would not be beneficial for devices powered by batteries (Johnson & Omosehinmi, 2015).

It has been shown that there is currently a threshold in how many cores that is required for a system to be performing optimally. When $p$, the portion of parallel execution is 0,9 the optimal number of processors has been shown to be 81 (Sato, Mori, Yano, & Hayashida, 2012). When exceeding this number, performance is degrading, and it has been shown that single core processors can perform just as well. This might be because current architecture is too reliant on sharing resources and increased complexity and latency. However, the programs tested might not be optimized to run on such high number of cores. Therefore, it is difficult to determine the threshold.

There are many ways to optimize a single core processor. It can be done by multithreading and concurrency. This might give an impression of parallelism, even if it is not.

### 2.3 Instruction Set Architecture

An overview of different ISAs with most focus on x86 architecture. However, ARM-64 and IA-64 is also mentioned. The main difference, advantageous and disadvantageous between them.

#### 2.3.1 ARM-64: A RISC Technology

Being a RISC technology, ARM-64 is used in smart phones and tablets. Since ARM-64 require fewer transistors it is ideal for smaller devices. Consequently, the power consumption, heat emission and cost are reduced. Moreover, the instructions are of fixed length. This is also beneficial if there is less time consumed calculating length of each instruction and much easier to optimize performance-wise.
2.3.2 The most common ISA x86

The most common ISA is x86. Different from ARM-64 it is a CISC architecture. The main reason why this architecture is the most common is because of backwards compatibility. Compatibility with programs is important and the ability to be able to run as many programs as possible is important.

The x86 ISA instructions are of variable length. The instructions can be as wide as \(15B\) and as small as \(1B\). It depends on type of instructions.

Figure 1: x86 instruction

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Opcode</th>
<th>MODR/M</th>
<th>SIB</th>
<th>Displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Byte</td>
<td>1 Byte</td>
<td>1 Byte</td>
<td>1 Byte</td>
<td>1-4 Byte</td>
</tr>
</tbody>
</table>

**Description:** A possible instruction in the x86 ISA. It requires memory access for R/W operations.

The instruction in Figure 1 is just an example of what the different codes can be. **Prefix** is the first possible bytes, if included in the instruction. The length can vary between 0 and 4 bytes. Examples of prefix-bytes are LOCK and REPEAT. An operation is repeated \(n\) times, if prefix is a REPEAT type. LOCK is related to ensuring exclusive access to a shared memory location. The operation itself is called an **opcode**.

**Opcode** is the main instruction. In x86 there several 100s of opcodes. From data movement, arithmetic to control operations, as well as other types. Some common types are JMP (changes program counter, jumps to a specific location), LOOP (loops section of code), MOV (moves data between locations), ADD (addition), SUB (subtraction) and LEA (loads effective address). An opcode can be 1 to 3 bytes. The opcodes can specify memory location (register) implicit or it can be explicitly expressed by a **MODR/M** byte.

Specifying operands which may be a register or a reference to a memory address, is the idea with **MODR/M** byte. MOV operations often includes a **MODR/M** byte. The MOV operation often require a memory access and to find the reference to that address, **MODR/M** is required usually.

**Example:**

```assembly
MOV EAX, [EBX]
```

This can be understood as move data from location in \([EBX]\) to EAX where EAX is a register and \([EBX]\) contains the reference to the memory address. Since the address in the memory location is effective and incomplete, the **SIB**-byte is a scaling byte, which the address is multiplied with. As **MODR/M**, **SIB** can be 0 to 1 byte. Furthermore, both types can require
an additional sequence of bytes called displacement. This is a value added to the address. Depending on the value of MODR/M or SIB, the displacement bytes can be 0 to 4 bytes.

**Example with SIB and displacement:**

MOV EAX, -4[EBX+FFH]

In addition to these parts of instructions there can be an immediate, which is a constant. Also, there memory addresses, relative addresses, pointers and offsets can be expresses in an instruction. Relative addresses are often combined with JMP and LOOP operations. Related to branching and changes to the program counter, the JMP instruction can be conditional or unconditional.

**Example:**

JB A0H

This instruction JMP if flag is below a certain value, and it is a conditional JMP operation. Other JMP operation can be JMP if not zero (JNZ) (Intel, 2016; Lejska, 2007; Stallings, 2013).
3 Memory designs: An in-depth view on main memory, registers and caches.

Keywords: Main memory, register, register allocation, caches, memory consistency, replacement polices and memory related issues.

3.1 Basic memory units

The traditional von Neumann architecture is basically a processor, control unit, main memory and external storage. This combined with I/O mechanism and registers, the von Neumann architecture was the most important computer models. Today, this model is still used with some additions.

3.1.1 Main memory

Since main memory is volatile, it can only function with power supply. Because of this main memory typically contains data from active processes. Most often main memory is random-access, which indicates that locations in the memory can be accessed in a non-sequential manner. As result a location can be accessed within the same time frame as any other location (Stallings, 2013).

As main memory is often word addressable, each cell contains a multiple of one byte (Mano & Kime, 2008). Some computers are byte-addressable, meaning 8-bits space per address to store data. A condition for main memory to be byte addressable is that the cell size is less than bus width, this is also the case for word addressable memories. Since a word is not defined as a standard unit, it is loosely defined, a word can be as small as one byte. Often a word is defined as 16-, 32- or 64-bits. A word is often proportional with memory address width and bus width. If a 32-bit machine with a 32-bit long address can address 4GB of data, a word-addressable memory can address 16GB of data if a word is 16-bit. The disadvantage is the extraction of one byte. The advantage of byte-addressable is the ability to manipulate smaller units more easily. An address in byte addressable scheme typically use two cells to store 16-bits, while an address in the word-addressable scheme only requires one cell.

A memory address is often 32- or 64-bit, thus there are $2^{32}$ or $2^{64}$ memory addresses. Because of size of a 64-bit address, many general-purpose machines are not able utilize all addresses at once. This is probably why 128-bit addresses are not common in general-purpose computers.
3.1.2 Registers

The processors each have their own set of registers. This is closest storage unit to the processor. Moreover, it is the fastest way to access data. Unfortunately, registers are too small to store large sets of data. Therefore, only current working sets are allocated to registers. There are different types of registers. A counter is a special register which transition through a pre-determined sequence of states. A counter is a register, but it is also differentiated from a regular register. A counter can be used as the **program counter** as it can be load and stored, used as a stack or incremented. Another difference is that a register can accumulate data through logical and arithmetic operations. Aside from counters, other registers are defined as memory buffer register (MBR), memory address register (MAR), instruction register (IR), instruction buffer register (IBR), accumulator (AC) and multiplier/quotient (MQ) (Mano & Kime, 2008; Stallings, 2013).

The smallest units that a register in a **byte addressable** memory scheme can store is most commonly one byte, although a \( n \)-bit register can store as little as 1-bit. Furthermore, it is common that the processor process larger sets of data, which is why there are many larger registers as well. Many new technologies have 64- to 256-bit registers (Intel, 2016). These registers of various width can be different in type. A type is specified from the information it can store and most types have a purpose, where general purpose can store both addresses and data and special purpose registers hold a program state. Special purpose register is where the program counter is located. Register are typically designed as the smaller registers are a subset of larger registers (Intel, 2016; Stallings, 2013).

Figure 2: 32-bit register structure

Description: An x86 32-bit general purpose register. EAX includes AL, AH and AX.
Figure 3: 64-bit register structure

<table>
<thead>
<tr>
<th></th>
<th>RAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-Bit</td>
<td></td>
</tr>
<tr>
<td>32-Bit</td>
<td>EAX</td>
</tr>
<tr>
<td>16-Bit</td>
<td>AX</td>
</tr>
<tr>
<td>8-Bit</td>
<td>AL, AH</td>
</tr>
</tbody>
</table>

Description: An x86 64-bit general purpose register. A bit different representation from Figure 2, but the structure is equal. RAX is the combination of all previous register from the bottom-up. The sum of the lower levels equals the next level.

The registers are often a part of a larger register in x86 architecture, as illustrated in Figure 2 and Figure 3. There are many more registers e.g. MMX, SSE and FPU registers. The register must be assigned values that are most fitted. This is done by register allocation. Register allocation can be divided into four parts (Koes & Goldstein, 2009):

1. Move insertion: The movement of register value from one register to another.
2. Spilling: The movement of register values to a memory location
3. Assignment: The assignment of register values
4. Coalescing: Register values are limited to one or few registers in a lifetime

Some issues are related to register allocation. These issues are solved in compile time and one of the techniques used is a NP-complete problem, known as graph-colouring. In this technique, each node in the graph is assigned a colour. As in Figure 4, the colour must be different than the colour of the adjacent nodes. If a register, which also is assigned a colour, is of the same colour, the value of the node may be assigned to that register (Stallings, 2013).
3.1.3 Memory designs

In shared memory systems there is requirement of communication between different processors. How communication is accomplished is often implicit because of conventional access instructions. Since logical addresses can be read by other threads, it will provide better throughput for multiprocessors. A block in main memory can be accessed by a multitude of CPUs in parallel system. To maintain that certain conditions to avoid redundancy and to maintain communication, a memory design decides the rule and circumstances of a shared memory access.

Uniform memory access (UMA) is design where all processors shared the same data uniformly. This design is comparable to a single processor’s memory access Therefore access time is equal for all locations. Conversely, the non-uniform memory access (NUMA) is different. Since NUMA do not share the same data uniformly, access times are relative to the location of the data. Therefore local-memory provides faster access than non-local memory. There is no restriction for private caches in UMA nor in NUMA. However, to reduce the number of memory accesses, the NUMA presents better performance than UMA does.
Another difference is that UMA utilizes a single memory controller, while NUMA uses multiple memory controller (Tanenbaum, 2009). Time-sharing, the concept of where a processors time is shared among more than one user simultaneously, and general-purpose applications are most suitable for UMA. NUMA is more suitable for real-time and time-critical applications. Real time is concept that refers to constraint that a system is subject to (Tanenbaum, 2009).

A third model is cache-only memory access (COMA). As it refers to the replacement of main memory with caches. It is like NUMA, because of the attachment of shared data to all processors. However, it differs in the absence of a main memory. Each cache memory is utilized the as main memory is in NUMA. There is no static memory (Hagersten, Landin, & Haridi, 1993).

Figure 5: NUMA architecture

In Figure 5, the distance between the memory units are illustrated with the bus between main memory and processors. Each processor is interconnected, where the bus is required to fetch data in peer caches.

3.1.4 Thrashing

When an excessive number of conflict misses occurs, it can be caused by an inefficient replacement policy (section 3.3). When an evicted entry is the entry needed closest in to future and the replacement policy continuously evict the entry needed closest in to the future, thrashing has occurred. Since, the right entries are not in the cache in time, thrashing implies reduced temporal locality (subsection 3.2.1). Moreover, if there is no victim cache, the consequence is further performance degradation (Denning, 1968, 1983).

The main memory is a bottleneck for performance. Through the evolution of computers, processor speed has evolved at a high rate only to increasing the gap in performance between processors speed and memory speed. One of the solutions to this problem is caches.
3.2 Caches: Closing the gap between memory and CPU performance

Implementing caches has increased performance. There are two main reasons, spatial and temporal locality. This is related to locality of reference.

3.2.1 Locality of reference

There are two concepts that are important in the utilization of caches, temporal and spatial locality. When repeatedly accessing the same data from the same memory address, performance degrades if data is only located in main memory. However, if the data resides in a cache it is beneficial to overall performance. This is what is called temporal locality. It is the availability of data that is repeatedly used over a certain time (Culler, Singh, & Gupta, 1999). A good example is a loop that access the same locations repeatedly. Another example is the natural order of a program. However, one threat to temporal locality is conditional branching, which changes the program counter so that data residing in the cache may no longer be needed. Nonetheless, if the data most often accessed is in the cache, overall performance is increased (Culler et al., 1999).

Spatial locality is the notion that if a memory addresses is accessed, it is probable that nearby location also will be accessed soon. Moreover, spatial locality can refer to accessing data locations sequentially (Stallings, 2013). There are other factors such as granularity and distance as well. They are important since the further from the target data is, the more complicated and larger it is. Because latency and start-up cost are increasing with distance, the data-sets must increase in size to compensate (Culler et al., 1999).

To exploit temporal locality organisation and structure is important. In some cases when concerning local and non-local data. If nonlocal data residing only in main memory is to be compared, the latency can be reduced by comparing each data set in main memory to each cache data set, instead of comparing each data-set in the local cache to the main memory data sets. The first option reduces the number bus transfers, while the latter option increases the number of bus transfers. The relationship between locality and communication is also very important (Culler et al., 1999).

If a memory location is accessed the neighbouring data is also likely to be required soon. Suppose a loop is iterating \( n \) times, during each iteration \( m \) instructions are executed. Since many the instructions most likely includes a reference to addresses which are adjacent to the requested address, the adjacent can also be fetched. These adjacent addresses can be mapped to the adjacent blocks or within the same block that is currently in use, depending on address. This is an example of spatial locality (Mano & Kime, 2008).
3.2.2 Characteristics of caches

A cache is a storage unit both on and off the CPU and it exploits locality of reference. The cache itself is a set or several sets that is further dived into subsets. These subsets are known as blocks or line. The number of blocks is determined by the capacity and block size. Furthermore, associativity determines the number of subsets within a set and the number of sets are determined by:

\[ N = \frac{C}{s \times a} \]

where

\( C \) is capacity, \( N \) is number of sets, \( s \) is block size and \( a \) is associativity.

The capacity of cache is much smaller than the capacity of main memory, however depending on the three factors beforementioned, it is much faster than main memory is.

Each block contains a word. In that sense a cache is word addressable. Each word in a memory block, has corresponding residence in the cache which it can be mapped to. The location can be found by dividing the address to three parts. Tag is the identifier, index is the set and offset are location within a block or set. These parts are calculated from the memory address. An offset is calculated by

\[ offset = a \mod b \]

where \( a = \) address and \( b = \) block size. In a similar manner the index is calculated by

\[ index = \frac{a}{b} \mod N \]

where \( a = \) address, \( b = \) block size and \( N = \) the sets length.

Furthermore, depending on different configurations tag, index and offset length varies. Offset length is calculated by \( l = \log_2 b \) where \( b \) is block size and \( l \) is offset length. Index length is as previously shown \( N \) and the tag is the remaining bits.

**Figure 6: Address**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
</table>

**Description:** This a description of how an address is split to the parts tag, index and offset.

In Figure 6, a memory address is split in to three parts; tag; index; offset. Tag is the identifier, index is which the block is in and offset is the location within the block.
Lines, blocks and slots refer to the same set of data in a cache, while entry is harder to define. It can be single pieces of data or data-set; however, an entry is referred to in this thesis as active data in a cache.

### 3.2.3 Capacity, block size and associativity

There are multiple configurations of a cache. Based on three main parameters, the efficiency of a cache will vary. Larger capacities are often slower than smaller capacities are. However, other parameters also influence this such as associativity. There are three main types of associativity. A direct mapped cache is cache where data maps to exactly one location in the cache. A \( n \)-way set associative cache is a cache where the number of alternatives to map is equal to \( n \). A direct mapped cache could be defined as 1-way set associative cache. The last type is a fully-associative cache, where very memory address can map to any of the sets. This implies that there is \( n! \) ways to store data.

The last parameter is block size as seen in Figure 7. This parameter is the size of each block. While capacity and block size do not directly influence performance the same way associativity does, the relationship between all parameters influence the number of cache misses (subsection 3.2.4).

**Figure 7: Cache**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>10101</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

**Description:** This is a direct-mapped cache with 32B capacity and 8B block size. The address length is 8-bits.

### 3.2.4 Cache misses: 3C’s

A cache misses is a failed attempt to complete an operation in the cache that leads to further actions to ensure availability of the data requested or actions to relocate data to other caches or main memory. Every operation in the cache that induce a delay can most often be called a cache miss.

It possible to produce a miss on two distinct operations; read and write. Furthermore, there are at least three subcategories. The compulsory miss refers to so-called cold miss. This type is occurring when cache line does not contain the entry and never have contained it in a program’s lifetime. Usually this miss occurs at the very beginning of a program’s lifetime.

A conflict miss occurs when the cache line that is written to contains other sets of data. This leads to a conflict where one entry must be evicted from the cache. Conflict misses occurs in direct-mapped and set-associative caches, while capacity misses occur in a fully-associative
cache. Since capacity misses refer to a full cache, it occurs in a fully-associative cache. On conflict misses and capacity misses, the cache must evict an entry and it is done through a replacement policy (Culler et al., 1999).

### 3.3 Replacement policies

Deciding what entry to be evicted can be based on several conditions. The ideal evictee is the entry that is to be used furthest in the future (FIFT). This is very difficult to achieve, but several close-to-ideal approaches exist.

#### 3.3.1 Least-recently used

An evicted entry should ideally be an entry that is not required in soon by the processor. How to decide which entry to evict can be done in various manners. Least recently used (LRU) is probably the most common algorithm in this context. The LRU evicts the block that has been accessed furthest back in the past, the least recent access, and replaces it with the new entry. The evicted entry is then often written back to memory. The algorithm requires information about accesses. Often there is a counter which helps indicate which entry was accesses most recently and least recently (Rochange, Sainrat, & Uhrig, 2014). However, this can also be done by time stamps and in that case, it is called a time-ware LRU (Bilal & Kang, 2014).

#### 3.3.2 Most recently used

Most recently used (MRU) is another policy, which works in the opposite way as LRU. As LRU works best when it is likely that most recently accessed entries are require soon, while when the opposite is true, then MRU is useful. In context of looping through a set of instructions, the MRU might be useful. The MRU keeps the older entries, while newer entries are discarded. Since it is the older entries will be used before the newer entries the next iteration, it is better performance wise to keep the old entries (Rochange et al., 2014).

#### 3.3.3 Pseudo LRU

These algorithms can be effective, however when concerning higher associativity there can be cache miss issues. A pseudo-LRU can be implemented. This algorithm utilizes approximations instead of exact measurements of age. In example, pseudo-LRU can utilize a tree structure with one flag bit. The flag bit indicates which direction to traverse. If flag bit is 1, then traverse to the right child. If flag bit is 0 then traverse to the left child. This requires \( N - 1 \) bits where \( N \) is the associativity. This means that searching with pseudo-LRU is average case \( O(h - 1) \) where \( h \) is the height of the tree. While with searching, LRU is worst case \( O(1) \), it is just as efficient as pseudo-LRU, but only when associativity < 4. On insertion and deletion, LRU is \( O(1) \) (Sarma & Govindarajan, 2003), while pseudo-LRU is
$O(b)$ where $b$ is the number of flag bits because every time a change occurs, the flag bits must be reset.

### 3.3.4 Random replacement

Another technique used is **random replacement**, this algorithm pseudo-randomly evicts and entry. The time complexity is $O(1)$. A **random replacement** algorithm uses no history of access and base the eviction purely on random (Flynn, 1995).

### 3.3.5 Least frequently used

Least frequently used (LFU) is often used when the system keeps a history of the number of accesses per block. Typically, there is a counter that for each access increments. However, this is problem, since newer entries that are likely to be used soon will be discarded because older entries have a greater counter value. This algorithm is $O(\log_2 n)$ in time complexity (Lee et al., 2001).

### 3.3.6 How caches work: The basics

When retrieving a cache block, the set with the corresponding index is scanned for **blocks** with the corresponding **tags**. If there is a **block** with a **tag** match, a **hit** occurs. If the set does not contain that **block**, it is a **miss**. When a read miss occurs, the next level of cache or main memory must be checked. Consequently, there is increased latency and potential use of memory bandwidth. If **conflict miss** occur, the **replacement algorithm** must evict an **entry** so that the new **entry** can be stored in the cache.
Description: R/W operations in a single-level cache, in a single-core environment.

The illustration in Figure 8, show how each memory address maps to a cache location. From, last two number in the address specifies the location within a block. Assume that this is a direct mapped cache with 8B capacity and 4B block size. Each set contains one way to store a piece of data.

3.4 Multi-level caches

To further exploit locality of reference, there is often multiple levels of cache. There is usually 3 to 4 levels of caches connected to the processors. The last-level cache (LLC) is often shared.

3.4.1 Inclusion policy

With multilevel caches there are multiple options to how to share data between the levels. An inclusive cache is where $L_1$ is included in $L_2$. When all cache blocks present in the higher-level cache are also present in the lower-level cache the cache is inclusive. An exclusive cache is where $L_1$ and $L_2$ does not contain the same elements. A multi-level cache hierarchy where all cache blocks present in the higher-level cache is not present in a lower-level cache is called an exclusive cache. There is a third option called none-inclusive-none-exclusive (NINE) cache. This is case where the cache is neither exclusive nor inclusive (Jacob, Ng, & Wang, 2010).
Figure 9: Inclusive cache mechanism

**Description:** Inclusive property guarantees the existence of X in L₂ if it is in L₁. This is 2-way associative inclusive cache.

In Figure 9, X is in L₁ and L₂, when the cache misses on Y, it is moved to L₁ from L₂. Since this is 2-way set associative cache, X is moved down one level. When Z is written to L₁, it must be written to L₂ as well, thus evicting X as the least recently used and X is written back to main memory.

### 3.4.2 Intuition on inclusion policies

There several disadvantages with all policies. Whereas an inclusive cache’s capacity is determined by the largest capacity cache, whereas exclusive cache’s capacity is the combine capacity of the two or more caches. One of the conditions of inclusion policy is higher associativity in lower-level caches than in higher-level caches. If this condition is not present, there will be many conflict-misses with the inclusion property. Moreover, an exclusive policy does not have that requirement and does not waste space like inclusive cache. Performance issues arises when a higher-level cache is forced to be smaller than the lower-level, if not lower-level would be filled with duplicate data.

Since it is easier to re-fetch data on a miss, an inclusive cache is reliable. An exclusive cache fetches data from main memory on cache miss. This causes cache misses to be heavily
penalized. A NINE do not maintain exclusivity or inclusivity, but it may hold the same cache line between each level, or it may not.

Inclusive caches do have easy state transitions, since there potentially fewer unique cache blocks than there is in an exclusive cache. An exclusive cache must swap cache lines between caches when the cache line is evicted. It has been shown that relatively small $L_2$ cache, exclusive caches perform better than inclusive caches (Zheng, Davis, & Jordan, 2004). Because of larger capacities, the bus traffic and latency are largely reduced. Also, an exclusive cache with higher associativity will have this effect.

### 3.5 Caches in a multicore environment

Adding processors adds more complexity. Shared memory models often utilize complex methods of guaranteeing the return of a correct result. There are to main concepts of guaranteeing correct results, memory consistency and cache coherence.

#### 3.5.1 Memory consistency

In single core environment, shared memory access is not an issue. This is one of the advantages of single core processors. However, in a multi core environment, main memory and often LLCs are shared. Because of this, data in all caches and main memory must be consistent. The guarantee the that data is the same for each run of an action regardless of source of request. This refers to processors seeing the same data or same set of data and the order of each operation. In a multicore environment the result of any operation must be predictable.

There many models of consistency. They can be defined by the degree of synchronisation that occurs. Some are strong such as the strict consistency model. In this case, every operation on set of data requires be seen instantaneously by all processors. This model is currently impossible to implement in current technologies. A less theoretical model is the sequential model. It is weaker model than the strict consistency model (Mosberger, 1993).

**Definition:** A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program (Adve & Gharachorloo, 1995; Lamport, 1979).

Since it does not require changes to be immediately seen, it is more contemporary. The result of any operation must be seen in an order that is equal to the order if the execution occurred sequentially. This concept can be seen in out-of-order execution where the result is ordered by the program order. This is the case of sequential consistency.
There are many models such as processor consistency model and cache consistency model. The first mentioned refers to the case where all processor must be consistent when write operations are finished on the same locations. However, when processor is writing to different locations consistency is not required. The condition that all writes to the same location must be seen in sequential order is maintained in this model. The cache consistency model is weaker than the process consistency model. Nevertheless, the condition that all write operation to the same memory location are performed in sequential order is maintained (Mosberger, 1993).

Consistency is not to be confused with coherency which is another property of a shared memory system. Cache coherency is how a system concerns the validity of data. Data throughout as system can be different, even on same location, and the system must ensure that this data propagated throughout the system. Moreover, it is important that main memory is coherent with the caches. Cache blocks in a coherent system have states. These states are specified by a cache coherency protocol. In these states the validity of data, the cleanliness and availability of it are expressed (Culler et al., 1999).

### 3.5.2 Cache coherency

There are two requirements for cache coherence, write propagation and transaction serializations. When changing a cache line in one processor, the content must be propagated to the other peer caches. This is called write propagation. Transaction serialization is that read and write operation to a single memory location must be seen by all processors in the same order. The execution can be parallel, but it must have the results equal to a serial execution (Culler et al., 1999).

More formally:

**Coherency is based in two conditions that must be fulfilled** (Patterson & Hennessy, 2008):

1. *In a read made by a processor P to a location X follows a write to same processor P to the X, with no writes to X by another processor occurring between the write and read instructions made by P, must always return the value written by P.*

2. *In a read made by Processor P1 to a location X that follows a write by another processor P2 to X, with no other write to X by any other processor occurring between the two accesses, and with the read and write being sufficiently separated, X must always return the value written by P2.*

The first condition refers to the notion on always returning the expected result and it is like consistency. The second conditions describe the validity of the content of a memory location. A processor that writes to a location followed by another write operation must clarify which write is currently valid. If the first processor reads the old value after the second processor has written the new value, then the cache is incoherent. These conditions are enough to satisfy the write propagation requirement, but not the transaction serialization requirement.
To fulfil this requirement an additional condition must be met (Patterson & Hennessy, 2008):

- **Writes to the same locations must be sequenced, if location \( X \) received two different values \( A \) and \( B \) from any two processors, the order of which they are read must be \( A \), then \( B \) and not \( B \) and \( A \).**

An alternative definition of coherency is through sequential consistency model. Coherency is achieved when all R/W operations on a single memory location are executed in an order that corresponds with the program order.

The processor consistency model is weaker than the sequential consistency model.

### 3.5.3 Coherency Models

To achieve coherency in a practical sense there must be implemented techniques that allow processors to receive information about changes. This can be done through a bus. Through a technique called snooping, a bus can notice where a change is made and deploy special actions to ensure coherency. There are two common techniques of snooping, write-invalidate and write-update. When a change is completed, a signal must broadcast to the other processors that either the cache line is invalid for all other caches or simply update the caches. The advantage of write-invalidate is speed. Since updates only occurs on requirement bus traffic is smaller which means latency is reduced. Write update is not as common as write-invalidate in a cache coherency protocol.

Directory-based cache-coherence is a different technique from a bus-snooping. It utilizes directories that works as filter to manage caches. Since it is a centralized system, the shared data is overseen in directories attached to each processor.

Since snooping has less scalability, directory-based cache coherence is preferred in large-scale machine designs (Zhang, 2010). Bus snooping requires high bandwidth with larger scale. Moreover, with increased number of cores, the contention for one shared resource such as the bus, it is not desirable to utilize a bus. A broadcast of invalidate blocks is demanding performance wise, since all processors must commit. One very important problem is that a consequence of inter-processor contention of address, is ping-ponging which is discussed in subsection 3.6.1.

### 3.5.4 Coherence misses: The 4th C

**Coherence** misses occur whenever a processor is updated, and all other corresponding cache blocks connected to other CPUs are invalidated. Then when another CPU reads from this block, it will miss. This type of miss is related to sharing, both **true** (subsection 3.6.2) and **false sharing** (subsection 3.6.3) and can cause cache block ping-ponging.
3.6 Cache issues, risks and events

Sharing comes at a cost. There are certain risks and inevitabilities. True sharing and false sharing can both occur. While true sharing is inevitable and necessary, it may occur even if a cache is infinitely large, false sharing is unnecessary. The problem that can arise from sharing is thrashing.

3.6.1 Cache thrashing also known as Cache line ping-ponging

When a cache line is transferred between multiple cores in rapid succession, it is called ping-ponging. The effect of ping-ponging is performance degradation. Since memory bandwidth is used when data must be fetched for every time there is invalidation invocation. Suppose two processor reads and writes respectively to the same block, every time the entry is updated, the other processor has an invalid cache line. When this processor attempts to read, it is required to get the data from another source. This requires bandwidth and increase latency. Moreover, when this happens in a rapid succession it has a major impact on performance. This can happen with both true sharing and false sharing which are part of the 4th C, coherence misses.

3.6.2 True sharing

When two processors read and writes to the same word. Suppose two processor read and write to the same word, the other process is forced to reload the cache line. As solution to this problem is increase block size thus increasing spatial locality.

3.6.3 False sharing

False sharing occurs when at least two processor, with one or more processors that perform a read operation, while one other processor performs a write operation, all on the same cache line. However, the write operation will cause an invalidation signal to be broadcast to all processors, causing all the data in the cache line to be invalidated. If the processor writing operated on a different word than processors reading required, it is called false sharing.

In example:

Suppose processor $P_1$ writes to location $X$ and processor $P_2$ reads from location $Y$, where location $X$ and $Y$ is in the same cache block $B$. When $P_1$ writes, an invalidation signal will be broadcast. $B$ will be invalidated, even if data from location $Y$ is unaltered and in a sense valid. This causes $P_2$ to produce a cache miss and $P_2$ must be updated form either main memory or another cache.
This is a huge problem if it occurs in a tight loop. Because, the processor repeatedly must fetch the valid data. It has been shown that decreasing block size reduces the number of misses caused by false sharing. Comparable to decreased block size, is subblock invalidation or partial block invalidation. In these techniques, a block is subdivided to smaller parts where each part contains a valid state. Moreover, each word may contain a valid state, which close to eliminates false sharing. A disadvantage with this solution is that there is a need for at least one more bit per block to represent the valid state of each word (Culler et al., 1999).

### 3.7 Cache coherency protocol

A cache coherence protocol is the appliance of coherence in a multiprocessor system. A condition for coherency is that different values on shared data is never seen. Snooping based systems and directory-based systems are two of the main types. Furthermore, there are many different implementations of cache coherency protocols and to further optimize the coherency system more states are usually added. The snoopy protocols of write-invalidation are most common in multicore genera-purpose machines.

#### 3.7.1 MSI based protocol

A classic write-invalidation scheme is MSI (Modified, Shared and Invalid). The invalid state refers to the validity of the data in a cache block. The modified refers to an exclusive state. This means that there is only one copy of that entry in the cache. Also, modified state is a dirty state, which refers to the availability in main memory. As opposed to clean state where the cache block data is in main memory, a dirty state is when the data does not exist in main memory or it is different from the data in main memory. The shared state refers to multiple copies of the cache block throughout the cache system. A peer cache also holds a copy of the referenced data. The shared is a clean state. Because the shared state is a clean state, it can be discarded at any time. If the cache replacement policy evicts the block, it is not required any further action to guarantee the data’s existence in the memory hierarchy. If a cache block is in the S-state and the cache block is overwritten, it transitions to M-state and all other copies of the cache block in peer caches are invalidated.

In Figure 10, it is shown how the states transitions from one to another. The M-state can transition to S, I or M. When a M-state transition to S-state, a coherence miss has occurred, and the data referenced must attempted located in a peer cache. If a copy is in a peer cache, it is in the M-state. The state must then transition to the S-state. However, it is also in a dirty state, so it must be written back to memory to transition to a clean state. If the referenced data it not in one or more peer caches, the data must be fetched from main memory. Invalidation occurs when a cache block is written to or overwritten. The cache block in the peer caches, either in the S-states or the M-state transition to the I-state (Culler et al., 1999).
**Description:** One of the most basic write-invalidate cache coherency protocol. There exist only one exclusive state (M) and it is dirty. Shared (S) is clean.

The cache coherency protocols are becoming more complex, each new protocol adds a new state. The next step in the evolution of MSI is **modified, exclusive, shared and invalidate (MESI)**. The added state is exclusive (E). The E-state refers to availability of the data in a cache block throughout the cache system. As with modified, there is only one copy of it between all the peer caches. However, it is not a dirty state. When a **miss** occurs, and the system is forced to fetch the referenced data from the main memory, the cache block state transition to the E-state. Naturally, since the main source of the referenced data is main memory, the data is clean. In Figure 11, an exclusive state can only transition from invalidate-state on a read miss or from the exclusive state on a read hit (Culler et al., 1999).
Description: The Mesi protocol adds an exclusive clean state (E).

Compared to the MSI in Figure 10, the MESI, in Figure 11, distinguish from writes from memory and write from registers. The advantage of an exclusive clean state is that can be discarded at any time without being written back to memory.

The current most applied cache protocols are MESIF and MOESI. The MESIF (Figure 16) protocol adds an extra shared state with the same properties as a regular shared state in addition to the responsibility of sharing on read miss called forwarding state (F). Both S- and F- are uncertain states, meaning that they are not necessarily are descriptive of the actual availability of the referenced data. When a shared or forwarding state is discarded due to eviction, if there at that point in time only exist one copy of the referenced data, the F- or S-state does not transition to E-state. Since, that action requires additional use of shared resources e.g. bus broadcasting, it would increase the overhead. The MOESI cache protocol adds an owned state (O). This state is like MEISF protocol’s forwarding state, but it is not clean. A cache with a block in owned is responsible of sharing the content with other peer caches. Because, shared copies often originate from the owned copy, the shared state can be dirty in MOESI protocol. Moreover, owned state block has both read and write privileges to that cache block, while blocks in shared state is read-only.
3.7.2 Other known coherency protocols

Based on the write-update protocol, the Dragon protocol (Culler et al., 1999) propagates updated data directly from write. This is efficiently done by read operation by other processors to the cache line. There are four states. Exclusive-clean (E) and modified (M) are exclusive states, but respectively clean and dirty. In addition to this there are two shared states, shared-clean (Sc) and share-modified (Sm). Sm-state like forwarding state in the MESIF protocol where the Sm-state is the last one to write to the block, but it is different as Sm-state is not clean state.

An advantage with write-update protocols such as Dragon, is that cache line ping-ponging is not present as opposed to write-invalidate. Because of this, the write-update based protocols may seem like the better alternative. However, the overhead is much larger. Moreover, due to synchronisation techniques write-invalidate policies are preferred.

3.7.3 How caches work: Synchronization in caches

A cache is synchronisation is related to coherence and consistency between caches and main memory. If an entry is invalid, it must be updated with the correct value. Coherency protocols are designed to ensure that synchronisation and the same time conditions for consistency must be fulfilled. In the context of caches, synchronisation refer to the point where data is updated. Most typically, they are updated when a read miss occurs.

Many cache coherency protocols implement a S-state. Coherency protocols such as MESIF implements an additional shared state forwarding (F). Whichever block in the state F is responsible of sharing the block on a miss. On a read miss a bus signals the other CPUs that data must be updated, and whichever is in state F copies the block, changes state to S and passes F to written block. If no block is in F, M, or E, then data must be fetched from main memory.
Description: Write miss dual-core non-shared cache system.

In Figure 12, the cache coherency protocol must fetch data from main memory and write exclusively to $L_1$. It can be assumed that it is an exclusive cache. This illustration is a preliminary to the project design. The system does not have shared caches.
4 Optimizing Cache behaviour

Keywords: Reducing cache misses through various technique and technologies. Cache configurations, compile-time techniques, adaptive caches, optimal replacement policies such as ARC.

4.1 Configurations

Configurations in this thesis refer to capacity, block size and associativity as mentioned in subsection 1.1.1 and 3.2.3. Increasing capacity, block size and associativity all have their advantages and disadvantages. The next section will provide insight in the pros and cons of altering configurations.

4.1.1 Impact of capacity, block size and associativity

Optimizing performance often involves trade-offs such as increasing complexity, greater cost and longer access times. A sequential search is more complex than a direct-access search. While time complexity for sequential search is worst case $O(n)$, time complexity for direct-access is constant. That implies that direct-access is faster, but less complex, has lower cost and decreased access times. Associativity tends to be increasingly more complex, costlier and have slower access times with higher associativity. However, with higher associativity is the number of cache misses is reduced. A fully associative cache has fewer misses than a direct-mapped cache, if capacity and block size are equal. Since a fully associative cache have $N!$ ways to fill the blocks, while a direct-mapped cache has only one way to store a line in each set. Because of slower access times, the penalties for cache misses are greatly increased in fully associative caches. When calculating:

$$CPU_{time} = \sum_{i+1}^{n} (CPI_i \times IC_i)$$

Where $CPI_i$ is cycles per instruction for $i$, and $IC_i$ is the instruction count. $CPU_{time}$ is total cycle time. CPI can be put in context with average memory access time and present execution time. there is absolutely no guarantee that a fully associative cache is better performing.
Moreover,

1. 

\[ AMAT = H + (MR \times (AMP)) \]

2. 

\[ H_1 + (MR_1 \times (H_2 + MR_2 \times AMP_2)) \]

Where:

1. \( H \) is hit time, \( MR \) miss rate and \( AMP \) is miss penalty.
2. \( H_1 \) is hit time for \( L_1 \) and \( H_2 \) is hit time for \( L_2 \). This formula is defined recursively to cover all the levels in the cache hierarchy.

Average memory access time shows that fully associative caches often have slower hit time, but more hits and larger miss penalty but fewer misses. Conversely, a direct-mapped cache has fast hit times, fewer hits, a greater number of misses, but smaller miss penalty. This is naturally difficult to show without any concrete data. However, in theory this is how it relates. Since both are different, a better alternative could be a set-associative cache. It has been shown that a 2-way set associative cache can have the almost the same hit time as a direct mapped cache, but with greater hit rate. Because of this, many \( L_1 \) caches are 2-way set associative. An 8-way set associative cache is also shown to have an almost equal miss rate as fully associative cache, but with smaller penalties. There are other parameters affecting performance such as block size.

When block size increases, the number of compulsory miss decreases. However, the number of misses caused by false sharing will increase. Conversely, because of fewer, but larger sets the number of true sharing misses will be reduced. In any case there is chance of cache line ping-ponging if coherency model is write-invalidating. When increasing block size, prefetching can be implemented. This will increase spatial locality and further reduce the number of compulsory misses. Since compulsory misses are caused by empty set in the initial phase of a program, the prefetching allows to completely utilize the full width of the block and prefetches the next memory block, and by all likeliness include data that is soon to be required. Block size in general influence the number of cold misses by reducing the number of total blocks.

Capacity naturally affects the number of capacity misses in a fully associative cache. In a direct-mapped cache the miss rate does not differ greatly as capacity has no effect. However, it is not certain that due to an increase in blocks, capacity can increase the occurrence of compulsory misses as well (Culler et al., 1999; Stallings, 2013).
4.1.2 Prefetching

While prefetching might be proven useful, it also provides disadvantages. Prefetched data might not be used. If prefetch occurs after the program counter is changed, it will have caused an unnecessary delay. The delay refers to use of memory bandwidth. However, a prefetcher can be very efficient. Even a simple \( i + 1 \) stride prefetcher improve, which prefetches the next memory block and write it to the adjacent block \( i + 1 \), can improve performance greatly. However, many prefetchers are more sophisticated. There are software and hardware prefetchers. A hardware prefetcher can interpret patterns, but those patterns only include number of misses and hits. A software prefetcher of adds PREFETCH instruction to the code. This is done by a compiler.

4.2 Compiler and software techniques

The cache behavior is not always dependent on hardware. In many cases sub-optimal code causes caches to perform poorly. Luckily there are a few techniques to improve cache performance. These techniques exploit locality of reference and many of them are used in compilers and executed in compile-time. Other techniques can be used by developers to optimize the code with respect to caches.

4.2.1 Loop unrolling

This is a compiler technique involves decreasing the number of iterations in loop. If loop is poorly optimized, it can run more times than necessary. Because of that the loop overhead is increased. To reduce the loop overhead as seen in Code 1, loop unrolling increases the number of operations done each iteration. This potentially increases parallelism. Moreover, it decreases the number of control hazards. Since the number of branches taken is vastly reduced (Cardoso, Coutinho, & Diniz, 2017).
4.2.2 Software pipelining

In machines with a great amount for pipelining and instruction level parallelism, software pipelining is a useful technique when it is desirable to execute more code in a clock cycle. When executing code in a loop it is desirable to remove dependencies, so that a code section can parallelized. This is done by unrolling the loops. The re-arranging the order how the code, such that it choose code from other iterations of the loop. When re-arranging the order, atomicity should be preserved. All operations are synchronous, so that no processors interfere with an operation. The complexity is NP-complete in general (Allan, Jones, & Allan, 1995; Hatanaka & Bagherzadeh, 2012; Lam, 1988).

**Code 2:** Software pipelining

```java
public static void softwarePipeliningComparison() {
    int[] A = {1, 2, 3, 4};
    int[] B = {1, 2, 3, 4};
    //Sequential execution
    A[0] = A[0] + B[0];
    // end
    //1st Parallel execution: 2 processors or threads
    //end
}
```

**Description:** A sequence of operations are split based on dependency between them and run in parallel.
4.2.3 Loop tiling

Also known as loop tiling is technique where cache blocks required for each iteration is kept in the cache and is not evicted. This is a method of ensuring that working sets are kept in cache and exploits spatial and temporal locality. The technique involves transforming the loop space into smaller chunks that are not constrained by the cache capacity. There are two types fixed and parameterized loop tiling. The first involves a fixed tile size while the latter have a variable tile size. The technique involves adding additional loop for each loop, which serve as the inner loops. Outer loops are called tiling loops (inter-tile) and inner loops are block loops (intra-tile). This can reduce the number of memory loads if done write. Atomicity is important in loop-tilling to reduce synchronisation cost. However, if the atomicity constraint is removed, the internal parallelism is exploited by overlapping computation with communication on multiprocessors (Cardoso et al., 2017).

Code 3: No loop tiling

```java
public static void noLoopTiling()
{
    int[][] A= {{1,2,3,4},[5,6,7,8],[9,10,11,12],[13,14,15,16]};
    int[] B= {1,2,3,4};

    for(int a=0;a<4; a++)
        for(int b=0;b<4;b++)
            A[a][b] = A[a][b]*B[b];
}
```

**Description:** A nested loop executed sub-optimally with respect to memory locations. The total number of iterations is 16.

Code 4: Loop tiling

```java
public static void loopTiling()
{
    int[][] A= {{1,2,3,4},[5,6,7,8],[9,10,11,12],[13,14,15,16]};
    int[] B= {1,2,3,4};
    int iTot=4;
    int jTot=4;
    int iStep=2;
    int jStep=2;

    for (int i = 0; i < iTot; i+=iStep) //Iterates iTot/2 times
    {
        for (int j = 0; j < jTot; j+=jStep)
            for (int b = j; b < Math.min(j+jStep,jTot); b++) //Max: 2
                for (int a = i; a < Math.min(i+iStep,iTot); a++) [ //Max: 2
                    A[b][a]= A[b][a]*B[a];
                ]
    }
}
```

**Description:** Adding loop unrolling to the example in Code 3, there are 2 more loops added. The concept is to create blocks, which are more efficient memory wise. The total number of iterations are 16.

As seen in Code 4, there are two additional loops. Purpose is the take advantage of locality of reference. To further describe the algorithm, it can be described in:
Table 1: Loop tiling results explanation

<table>
<thead>
<tr>
<th>Indexes (i)</th>
<th>Indexes (j)</th>
<th>A[b][a]</th>
<th>Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 0</td>
<td>b = 0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>a = 1</td>
<td>b = 0</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>a = 0</td>
<td>b = 1</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>a = 1</td>
<td>b = 1</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>a = 0</td>
<td>b = 2</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>a = 1</td>
<td>b = 2</td>
<td>20</td>
<td>6</td>
</tr>
<tr>
<td>a = 0</td>
<td>b = 3</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>a = 1</td>
<td>b = 3</td>
<td>28</td>
<td>8</td>
</tr>
<tr>
<td>a = 2</td>
<td>b = 0</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>a = 3</td>
<td>b = 0</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>a = 2</td>
<td>b = 1</td>
<td>21</td>
<td>11</td>
</tr>
<tr>
<td>a = 3</td>
<td>b = 1</td>
<td>32</td>
<td>12</td>
</tr>
<tr>
<td>a = 2</td>
<td>b = 2</td>
<td>33</td>
<td>13</td>
</tr>
<tr>
<td>a = 3</td>
<td>b = 2</td>
<td>48</td>
<td>14</td>
</tr>
<tr>
<td>a = 2</td>
<td>b = 3</td>
<td>45</td>
<td>15</td>
</tr>
<tr>
<td>a = 3</td>
<td>b = 3</td>
<td>64</td>
<td>16</td>
</tr>
</tbody>
</table>

**Description:** The results is equal to the results in Code 3. However, each loop works as a block to exploit locality.

### 4.2.4 Loop interchange

When concerning a nested loop, the order of loops can interchange, by changing the order of which they occur. The usage if interchange must be carefully considered and compilers are of great importance when optimizing with loop interchange. Since not all languages order elements in the same manner, a compiler can detect inefficient code and rearrange the order. Not all nested loop can be or should be re-ordered. Since there are potential dependencies between the variables and order of which the occur. Therefore, there must be a dependency analysis (Cardoso et al., 2017; Srikant & Shankar, 2007).

**Code 5: Loop interchange**

```java
public static void loopInterchange() {
    int[][] A = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13, 14, 15, 16}};
    //before loop interchange
    for(int i=0;i<4;i++)
        for(int j=0;j<20;j++)
    //after loop interchange
    for(int i=0;i<4;i++)
        for(int j=0;j<16;j++)
}
```

**Description:** In some cases, it more advantageous to swap loops. This can be done if locality can be exploited.
4.2.5 Loop fusion

Concerning adjacent loops, they can be combined. If through the iterations, the loops access the same locations it can be increase performance through computational density and smaller loop overhead, while improving software pipelining. This is done by a compiler and provides more opportunities for instruction-level parallelism. However, it does not necessarily induce better performance due increased computational weight in some cases (Cardoso et al., 2017; Srikant & Shankar, 2007).

Code 6: Loop fusion

```java
public static void loopFusion()
{
    int[] A = {1,2,3,4};
    int[] B = {1,2,3,4};
    //No loop fusion
    for(int i=0;i<4;i++)
    for(int i=0;i<4;i++)
        B[i]=B[i]+1;
    //loop fusion
    for(int i=0;i<4;i++)
    {
        B[i]=B[i]+1;
    }
}
```

Description: To save the number of total loop iteration, these two loops can be merged.

4.2.6 Loop fission

In the case of high computational density, splitting a loop to two loops can be beneficial. Since the computational density can have an impact on software pipelining, loop fission can increase performance. If a loop includes conditions as seen in Code 7, the conditions can be inserted to separated loop. This might increase software pipelining efficiency. The consequence of too heavy computational density can be unnecessary registers spills. A drawback of implementing loop fission is that the compiler might fusion the loop with another loop (Cardoso et al., 2017; Kennedy, 2001; Srikant & Shankar, 2007).
Description: As opposed to the loop fusion in Code 6, loop fission splits a loop in two. Often if the original loop contains a conditional statement.

4.3 Hardware techniques

A very efficient way to increase hit rate is to increase the number to of caches. As size of programs grow, the need for more memory increases accordingly. Adding another level of cache or even adding cache for evictees can impact performance. However, there is relationship between size, complexity and speed.

4.3.1 Multi-level caches: Optimal configurations

Added one or more cache can prove to increase performance. Because of high bandwidth, low latency and fast access times, the introducing multiple levels is favourable. When a miss occurs, there is greater likeliness to find the entry in another cache regardless of inclusion policy. Multiple levels allow the highest level to be smaller and as result more efficient. The relationship between capacity, block size and associativity are of great importance when designing a cache hierarchy. When $L_1$ is of any size $L_2$ should be at least as large to be equal the number of hits. Larger $L_2$ provides better hit rates, but when $L_2$ is at one point $n$ times larger the size of $L_1$, the hit rate growth stagnates. Therefore, an overly large $L_2$ is not as beneficial due to higher latencies and lesser growth of hit rate (Stallings, 2013).
Associativity increase hit rate with increased capacity. However, complexity increases with higher associativity. In the context of inclusion policies, the associativity in $L_2$ must be at least twice as the associativity in $L_1$ in an inclusive cache. Since associativity is higher, capacity should be increased to allow more blocks especially in fully associative caches. The drawback is latency and slower access times.

4.3.2 Unified vs split caches

Introducing another cache on any level so that there are two caches on the same level is called a split cache. The two caches are different regarding content as one cache mostly contain data, while the other cache mostly contain instructions. When a processor attempts to fetch an instruction it first attempts to fetch it from the instruction cache, and the opposite for data. The advantage is in instruction-level parallelism. Because there is less contention between the fetch and decode stages and the execution stage, this can be quite efficient. Because of less competition between fetching and execution stages, there would be fewer conflict misses. Moreover, a split cache eliminates the performance degrading effect when a fetch operation blocks the execution of an instruction. This is an example of structural hazard. Moreover, since the instructions and data are stored different location, prefetching can be implemented more efficiently (Stallings, 2013).

The advantage of a unified cache is that it has a higher hit rate than split caches due to balancing the occurrence of instructions and data automatically. Based on execution pattern, the cache tends to contain more instructions if the execution pattern involves more instruction fetches and vice versa. Another advantage is reduced complexity of a system, also more space for only one cache (Stallings, 2013).

4.3.3 Dynamic split cache

If a compiler partitions a cache into two part, one for data and another for instructions and this is done at runtime is called a dynamic split cache. Normally a split cache is rigid, strict and is decided at design time. However, a dynamic split cache is a cache which can adapt to a working set of instructions and data (Juan, Royo, & Navarro, 1995).

4.3.4 Victim caches

Whenever a block is evicted, it can be moved to a victim cache. This is often a fully associative cache that contains entries that are valid, but not “wanted”. The benefit is clear, as previously required data may be useful soon. To further optimize a cache design, prediction logic may be used to select the most likely candidate for placement in the victim cache. The other discarded entries are written back to main memory. The original idea was to implement a victim cache to reduce direct-mapped $L_1$ caches (Jouppi, 1990). However, last-level caches are often victim caches.
4.4 Reducing false sharing

Configurations can decrease the number of false sharing misses. It is known that smaller block sizes reduce the occurrence of false sharing. Reduced block size also increases overhead and the number of true sharing misses. How can false sharing be reduced and at the same time limit impact on performance?

4.4.1 Partial block invalidations and subblock coherence: Reducing false sharing

Misses caused by false sharing can be avoided by implementing states per word or subdivide blocks to smaller subblocks and implement an invalid/valid state. This will have the same effect as reducing block sizes. The partial block invalidation scheme extends the write-invalidation scheme and each complete block can transition between the states M, S and O. The subblock is either valid or invalid. This scheme involves $R$ invalidation blocks with $b$ size where $R \times b = B$ where $B$ is the total block size. Consequently, this requires $R$ bits and since the valid bit in the complete block is changed to stale, there is additional 1-bit. Therefore, $R + 1$ bits are needed. The condition for the address block transitions to the stale state is when O-state is reset and at least one valid bit is reset (Chen & Dubois, 1993).

It has been shown that partial blocks have fewer misses caused by false sharing than a cache block equal to the partial block in size. However, the study was conducted on small data-sets rather than large data-sets. Since false-sharing is a problem in fine-grained parallelism, the data-sets were kept small (Chen & Dubois, 1993).

4.4.2 Subblock coherence

Subblock coherence is an evolution of the partial block invalidation scheme. This scheme focuses on dynamic subblock coherence. Since false sharing occurs when block size is larger than one byte, a fixed size larger than one byte only reduces the problem and not eliminates it. However, when the size of a subblock can dynamically change, further optimizing of performance occurs. Since smaller cache blocks reduces false sharing, but increases compulsory misses, the dynamic approach can have an impact on performance (Kadiyala & Bhuyan, 1995).

4.4.3 Combining it with modern cache coherency protocols

In the subblock coherence protocol, the states are associated with the subblocks. The result is an increased number of bits required. If combined with the MESIF protocol, additional 5-bits per subblock is needed. There is an increased complexity and the trade-off is locality of reference to reduction of misses caused by false sharing.
It is worth noting that this method does not reduce the ping-pong effect from true-sharing. Because of decreased block size, this may increase the number of true-sharing coherence misses. Moreover, the exploitation of spatial locality is reduced.

4.5 Memory abstractions

Main memory and caches are often not large enough for storing all a program’s or multiple programs data and instructions. Solving this problem is an abstraction technique called virtual memory.

4.5.1 Virtual memory

As an extension of main memory, the virtual memory provides larger capacities. Virtual is translates to “close to something without actual being it”. The word virtual in virtual memory can imply a logical or abstract representation of a physical unit. An emulator has a virtual property, thus virtual memory is an emulation of main memory or a cache. Since, the virtual memory is in the HDD, the capacity can be substantially larger than the capacity of main memory.

Performance degradation can be a problem, but there are many techniques to ensure high efficiency. The TLB is an example of such technique or technology that reduces time consumption. The TLB includes both virtual and physical addresses and maps them correspondingly. Since the TLB contribute to faster loop-ups, one of the drawbacks is power consumption.

A TLB is often of high associativity and sometimes fully associative. Since fully associative caches often have low miss rates, but high miss penalties. The problem is when a miss occurs, and the system must launch the fault handler to handle the exception. The virtual memory uses the paging model (Tanenbaum, 2009).
4.5.2 Paging

As a common memory model, the paging model allow more versatility than a flat model scheme. Each memory address is composed by a page number and an offset called a frame. The page is mapped to the main memory either direct-mapped or with \( n \) associativity (Stallings, 2013; Tanenbaum, 2009).

4.6 Issues in VM and how to solve them

Many of the issues related to caches are also related to virtual memory. Thrashing and collision are the main issues in a virtual memory.

4.6.1 Thrashing: Recurring problem

Thrashing is a problem not only associated with caches, but also virtual memory. When a repeatedly swapping entries in virtual memory with physical memory locations because of memory constraints occur, it is called thrashing. Since, swapping is more time consuming than actual commutations, this is a performance degrading event. To solve this the number of missing pages can be reduced and to avoid page fault by increasing main memory capacity or swap size (Denning, 1968; Stallings, 2013; Tanenbaum, 2009).

4.6.2 Collisions

When a physical address mapped to a virtual address, a collision may occur between pages. This resolved by a paging scheme called page colouring. Different colour maps to different addresses. If the colour is equal for physical and virtual addresses, the system can guarantee that here are no homonyms and there is no need for aliasing.

4.7 Algorithms, adaptiveness and ideals:

One interesting application of machine learning is in prefetching and replacement policies. ARC is well known example of a replacement policy that is combination of two or more replacement policies.

4.7.1 Optimizing replacement policies

The ideal replacement policy is in the case of an evicted entry being the entry to be used furthest into the future (FITF). Therefore, the last entry to be last evicted is the entry that is needed nearest into the future. This replacement strategy will always choose the correct entry. Since, the correct entry is always chosen, the number of misses will be vastly reduced. Two
conditions are needed to fulfil this ideal case. Since the number of misses are to be reduced, and to maximize the utilization of the FIFT-replacement strategy, the cache must be fully associative (Hassidim, 2010). In addition, the algorithm must be a cache-oblivious algorithm. While a cache-aware algorithm is optimized regarding the parameters of cache, parameters are not known in a cache-oblivious algorithm. Nevertheless, a cache-aware algorithm can be more efficient for a single problem. Whether a cache-oblivious algorithm is more efficient in general is not known. The problems that arises from not knowing the parameters are in general is uncertainty regarding data. Because of this, the cache-aware algorithm is often more efficient, but it must be adapted to a single case, while cache-oblivious algorithms do not require tuning (Frigo, Leiserson, Prokop, & Ramachandran, 2004).

LRU, MRU, MFU and LFU have both disadvantages, although the LRU is a close approximation of the FIFT policy. The MRU works well when iterating through a loop. A combination of the MRU and LRU could be closer to the ideal FIFT-policy. By implementing a dynamic replacement policy, which based on heuristics can determine when to use MRU and when to use LRU. The ARC algorithm is such an attempt to implement policy at most as difficult to implement as LRU. However, this policy combines both LRU and MFU/LFU. The implementation cost is comparable to the implementation of the LRU. There is slight increase in overhead and meta-data because of storing history of all evictees (Megiddo & Modha, 2004).
The Figure 13 is an illustration of the order of the entries with respect to recency. The topmost object is the first object and it is not to be evicted if required. The last entry is the least recently used entry. Whenever an operation is performed on a cache the order may change.

### 4.7.2 Adaptive caches

The ability to reconfigure cache parameters for each program so that the configuration is ideal for a given program, could improve performance. Nevertheless, the problem that arises is parallelism. When programs and task are run in parallel or concurrently, the ideal configuration would possibly change for each program and task, thus increasing overhead and latency. Because of this an adaptive cache must adapt to a multitude of programs, where the optimal configuration must be optimal for running all programs in parallel. Another solution would be to dedicate processors to each program. However, this would possibly limit parallelism where all programs may not exploit the potential of parallelism.

An example of an adaptive cache is a **Smart Cache**. This cache was tested in a multi-core environment with decent results. However, multicore architectures were not the focus area. The Smart Cache could reconfigure close to ideal configuration for each program tested (Sundararajan, Jones, & Topham, 2012).

A dynamic cache is type of adaptive cache. As opposed to static-caches, which are configurated before runtime, a dynamic cache can reconfigure within a certain time interval in runtime. Nevertheless, a problem occurs when deciding what meta-data to base
reconfiguration parameters on. Solely basing on cache misses can be inaccurate, therefore more advanced techniques must be implemented. Consequently, overhead may potentially increase. Another pitfall is when rapid changes occurs. When reconfigurations occur in rapid succession, it can potentially degrade performance. Since the reconfiguration process has a slight impact on CPU cycles, rapid succession will degrade performance greatly (Sundararajan et al., 2012).
5 Methodology

Keywords: Cache simulator, decision, limitation, issues, description. Black box, memory model, coherency protocol, scheduler.

5.1 Approaches

5.1.1 Black box

The implementation of the simulation is carried-out with a black-box strategy. To ensure that any program can be run, the black-box approach allows a more general approach. Consequently, any executable file can be run, ideally. However, there are certain limitations to the simulator such as file size. The input file must be an .EXE file. Also, the file size of test programs should not exceed 1MB.

5.1.2 The white box

Alternatively, the program could have been a white box simulator. But since the program only would work for a certain class of languages, it would limit the flexibility of the simulator. An option was to develop a compiler for a specific language. Since it would be integrated with the compiler, more accurate and detailed result could be extracted. It would be easier to determine which parts of a given code is inefficient. Because the results are more explicit further optimizing could be done. However, the disadvantage is that the simulator would only work for one language.

5.1.3 Chosen programming language

Since the simulator is complex, Java was a best choice. With regards to familiarity and knowledge, it was the most natural choice. Other languages were considered, but without knowing the scale of the program difficult to decide the language. However, there may be other languages better suited for this type of program.

Since Java is an object-oriented language, it has many features that are desirable when developing programs with many data structures. Because the language contains a feature called garbage collector (GC), there is no need to cleaning the program manually. Also, features such as memory allocation decreases difficulty and code size. The GC handles all data-structures, variables and objects that are not in use and collects the unused elements.

The Java version used is JDK-8.
5.1.4 Issues with programming language

The choice of language as it may have obvious at the start, there were certain problems with scalability. Since, the program generates thousands or even millions of objects, the overhead is very large. Java has some memory constraints. However, within reasonable parameters, the language provides good performance. To limit the effects of this programs, the program should be carefully about usage of object and remove unused objects and variables. Also, the garbage collector (GC) in Java is manually invoked between each simulation. Since, the GC does not respond fast enough.

The garbage collector in Java is not always efficient when dealing with large scale programs, thus it consumes much CPU-time, and at it can end in time-out.

5.2 The simulator: Overview

5.2.1 Input

The input must be a .EXE file. The simulator reads the input, byte per byte. Constructs instructions based on the input. Moreover, the instructions are processed according to the ISA used. While instructions are available through streaming of a program, data is not. Data is randomly generated. Any register value, memory value and address are randomly generated.

5.2.2 ISA: x86

The x86 instruction set was the most obvious choice for two reasons, availability of information and common usage. Since x86 is still commonly used, even if there are more suited alternative ISAs, it was a natural choice. Due to small resource availability, other ISAs were not natural to choose. Alternatives like IA-64 and PowerPC did have as much resource available, but the quality of resources was less. As a resource of x86 ISA, a web page containing x86 instructions was helpful (Lejska, 2007), and it contains almost a full set of instructions with descriptions of each instructions.

5.2.3 Output

The output is cache behaviour data. Each miss and each type of miss is counted. It is written to the an excel-file and it is displayed in a graphical user interface. Other metrics such as parallel fraction and speed-up (in section 2.1.2) is also presented in the metrics. For speed-up, the program assumes the total number of instructions as execution time in one processor. The parallel fraction is then found by rearranging Amdahl’s law.
5.2.4 Design of simulated architecture

Figure 14: Simulated architecture

Description: Architecture simulated

The architecture is potential multi-level non-shared cache system. In Figure 14, the distance between processors may indicate a NUMA memory design. The bus is utilized by all processors.

5.3 Creating an abstract model of cache behavior

5.3.1 Cache mechanism

The simulator allows simulation of multiple levels of caches, but no shared caches. The focus is solely on behaviour in the non-shared caches. The inclusion policies are NINE, inclusive and exclusive. Comparing results from various inclusion policies can be useful to understand the different advantages and disadvantages for each policy.
Figure 15: Exclusive cache

**Description:** Exclusive cache mechanism

The Figure 15 displays the scenario where $X$ and $Y$ swap places when there is a cache miss in $L_1$. Then $Z$ is written to $L_1$ and $X$ is evicted from $L_2$ and written back to main memory.

The NINE-policy exploits capacity of both caches, while being less complex than an exclusive cache. Moreover, an exclusive cache must always fetch from main memory on misses, while the NINE policy does not necessarily require that, thus less bandwidth is occupied. Also, NINE-policies sometimes fulfil the condition of inclusion and therefore it have some of the benefits of an inclusive cache.

The inclusive cache writes modified (M) to all levels of cache. It is simpler to implement than an exclusive cache and provides advantages on miss. Since it is likely that a discarded block
in $L_1$, is still located in $L_2$, the benefit of an inclusive cache is very large. Naturally, the number of hits is counted on all levels of cache when writing modified.

An exclusive cache writes modified to only the first level cache. On read miss it also writes exclusive to the first level. This provides great capacity. However, the miss penalty on coherence misses is large. The exclusive cache, benefits from relatively small difference in capacity between the levels of cache. The utilization of an exclusive cache is most efficient as a victim cache.

The coherence protocol utilized is a version of **MESIF**. When a block is first written to and it is not from main memory, all versions of that block are invalidated and the block transitions to modified state (M) and dirty state. The block written to resides only in $L_1$. If miss occurs on a read operation and data is required to be fetched from main memory, every version of that block is invalidated, and the block updated transition to exclusive state. All levels are written to, so that the data is inclusive. If there is a read miss occurs and the inter-cache transfer locate a block in forwarding state, the forward-method is invoked. This method sends the block to all CPUs and inserts the block in the caches.

When a read miss occurs, all caches to all CPUs are checked, but only if the data does not exist in any of the cache levels to the CPU in use. The blocks are checked to contain the necessary data and if the block is found, the state is updated to shared and clean in that block, while the state of the block written to transition to forward (F), shared (S), and clean states. If the block is M, the block is written-back, then state change to S. Because the block is written back to memory as well, it is clean.
In Figure 16, the state’s transitions caused by various events. As mentioned in the section 3.5.3, MESIF is an extension of MSI and MESI. The forwarding state (F), refers to a shared state with sharing responsibilities. The M-state transition to the F-state and S-state, if a peer cache contains the referenced data. If a cache is in the E-state, the data is from main memory and it is clean. F- and S-states are uncertain states, which means that they are uncertain in terms of correctness in the system. When a block in the F- or S-state is removed due to LRU-replacement, the state of the other shared block remains in S or F, even if there is only one copy of them in the system.

Since the block size can be as small as $1B$, tests can be conducted to investigate the behaviour of subblock invalidation policy or partial-block invalidations. Each block will still include the five states (MESIF) and it can be shown to what extent false sharing is present by comparing different block sizes to the smallest possible block size.

Another policy implemented is subblock invalidation protocol combined with MESIF. Each subblock is the size of a word and have can be modified, shared and invalidated. This policy eliminates false sharing. Since the occurrence of true sharing may be increased, cache line ping-ponging may be present.
5.3.2 Memory addressing model

When choosing a memory addressing model there were several choices. The most natural choice would be to follow the x86 models, a segmented address model or paged memory model. However, segmented memory model is to specific and the paged memory model is too complicated, it was not necessary for the end solution to have such detail. The chosen model for the simulator incorporates a flat memory addressing model. This is the most basic model, but it serves the purpose. A flat memory model has 16-, 32, 64-bit addresses, which are incremented by one for every new address. This is also called linear memory.

Alternatively, a segmented memory as in Figure 17, was the most common model in the x86 before paging became more utilized. A segment model categorizes data by purpose. There original segments were data-, code-, stack- and extras-segment. Since there were four segments, additional 4-bits are added to the address. If the address width is 16-bits, then in a segmented memory model it is 20-bits.

Figure 17: Segmented memory model

<table>
<thead>
<tr>
<th>Segment</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>OxA</td>
<td>Ox0AF</td>
</tr>
</tbody>
</table>

Description: An example of a 16-Bit address where 4-Bits are segment and 12-Bits is offset.

Aside from being a flat memory model. The memory access is like the NUMA design, but there is a single bus and it is structured like a crossbar. That is more of an UMA design. However, there multiple cache controllers, but only one memory controller. Distance between local and non-local memory matters. All processors have at least one cache. The memory access design is incorporating elements form NUMA and UMA.

5.3.3 Processors and classifications

The simulator is simulating a SPMD with some options of increasing and decreasing parallelism. Since SPDM is common with the NUMA design, this combination best describes a modern computer. If the cores are homogeneous, the simulator describes a general-purpose multi-processor machine. Since there caches are not shared, the cores are loose-coupled.
5.3.4 Main memory

Represented by a hash table, main memory, store data identified by string key that corresponds to the memory address. The addresses are hexadecimal, but also binary representation is stored in the hash table. Each element in the hash table is a Memory.java object which contain data and addresses. The hash table solution is preferred before arrays and lists. Since there are many constraints regarding array sizes, arrays are not advantageous for this design. If each index in an array represents an address, the array can only represent $2^{32}$ addresses. Moreover, typically an integer can represent indexes from $-2^{31} + 1$ to $2^{31} - 1$ objects. This is also constrained by hardware on test machine, so the actual range is $-2^{27} + 1$ to $2^{27} - 1$. Another problem that arises is when address width exceeds 32-bits e.g. 64-bits. Where addresses become too wide to be represented, other methods must be utilized to represent main memory.

An array with a size of $2^n$ where index does not correspond with address could possibly be the solution. However, the array would have to be search sequentially and worst-case complexity would be $O(n)$. A hash table provides the same problem when multiple keys are hashed to the same value, but since that occurs rarely, the time complexity is in general $O(1)$.

A list has complexity $O(n)$ when searching and $O(1)$ when inserting, providing that there is no replacement of objects. If an element is needed to be updated, complexity is worst-case $O(n)$. However, lists are not restricted by the same constraints as arrays as the size is not predefined. The size can exceed the JVMs memory available and cause an “out-of-memory exception”.

Like list, a tree structure’s size does not need to be predefined. An attempt was made to create a tree structure based on address where each node was either true or false. The last node in each branch would contain the data. The complexity would be $O(h)$ where $h$ is height, but since all addresses maps the same and the corresponding data always will be located on the same addresses, the complexity is $O(1)$. This solution is comparable to the hash table regarding complexity, but since it requires larger overhead, it is not optimal. The problem is that larger overhead and more object cause GC to consume more CPU-time and eventually time-out. The idea was that the system would support wider addresses than 64-bits.

A hash table also support wider address width than 64-bit addresses in theory. It is not paramount to allow this as 64-bit address width has not fully been exploited yet. Since the Memory.java object contains a short-array and Boolean-array corresponds to the hexadecimal address and the binary address, the address width can be larger than 64-bits.

The next addresses to be utilized is in the MainMemory.java. This is the address counter. It contains the first address of the next data to be inserted. It is different from the program counter.
5.3.5 Cache

The complex structure of cache can be represented by arrays and hash maps. This is the most accurate abstraction of cache. Since each set, block and word can easily be accessed directly with integer values, an array provides the simplest yet best structure. As memory addresses are both hexadecimal a binary, the tag index and offset it calculated only form the binary address. While index and offset are transformed to integer values, the tag remains a binary and string. Since tags often are larger than index width and the offset width combined, it may exceed 32-bits and to still be able to support 128-bit address width, it is kept as a binary. Alternatively, the tag could be a long value. It may not be optimal to keep addresses on a binary form. Because the Boolean-datatype are at least 8-bits in size since and a binary address with width 64-bits would reserve at least $8 \times 64 = 512$-bits which equals 64 bytes and in addition to headers, it equals $64 + \text{headersize}$ of memory space. A byte-array would be more advantageous to use. Since byte-arrays reserves less memory space than Boolean-arrays.

Located in the hardware group the Cache.java includes Index.java objects in an HashMap. The HashMap is called set and represents the set. By default, the sets are null, and it is required to declare a new Index.java object the first time it is accessed. The length of the Index.java array is calculated by:

$$N = \frac{\text{capacity}}{\text{block size} \times \text{associativity}}$$

regardless of associativity. A direct-mapped cache is of $\text{associativity} = 1$, and fully associative caches is of:

$$\text{capacity} = \text{block size} \times \text{associativity} \times n_{\text{sets}},$$

thus

$$\text{associativity} \times 1 = \frac{\text{capacity}}{\text{block size}}$$
in a fully associative cache.

Furthermore, the number of blocks and the block offset is calculated in the Cache.java class. The Index.java class includes LRU-policy.

The Slot.java class represents a block. The number of blocks per set is determined by associativity and block offset width is determined by block size. If all the slots are occupied, the Index.java class invokes the LRU-policy. The policy is enforced by incrementing a recency queue for each slot. The last object in the queue will be evicted, and the new entry is inserted first in the queue, the queue is a linked list. This method does not require all slots to be checked, which is of $O(1)$ complexity. The slot is a HashMap. Since the tag is a String as well as a Boolean-array, the tag (String) is the key of HashMap. This results in very efficient read-method. Reading from the slot is $O(1)$. However, if the HashMap does not contain the key, the read method returns NULL.
The Offset.java does not include an LRU method. If there is need to simulate partial block invalidations (Chen & Dubois, 1993), the block size can be set to \(1B\). This is practical when investigating false sharing mechanisms. Since evictions occurs on each individual word, there can be as many evictions as the length of the data required in the read operation. An offset.java object includes an Entry.java object where data is stored.

The LRU-policy is one of the most common policies. The method in the simulator approximates the optimal solution. A more sophisticated replacement policy could be added, such policies as ARC and more machine learning policies.

The coherence protocol used is MESIF. The applied version of the protocol, forwards on read misses. It forwards to all caches on the first level. The write-back is completed on modified only. If a block is in state E or F, no write-back is required. When writing F, it can happen in three different scenarios.

The block exists in another cache and:

1. It has state M or
2. It has state E or
3. It has state F

The entry is fetched from that cache. The fetched block transition to the S state, and the new block transition to F.

Code 8: Peer cache hit

```java
for(Slot s:slots)
{
    valid_slot=s;
    if(valid_slot.is_modified())
    {
        valid_slot.set_modified(false);
        valid_slot.set_dirty(false);
        short[] temp= valid_slot.get_all_data();
        bus.write_to_memory(start_hex_addr, start_bin_addr,temp);
    }
    else if(valid_slot.is_exclusive())
    {
        valid_slot.set_exclusive(false);
        valid_slot.set_shared(true);
    }
    else if(valid_slot.is_forwarding())
    {
        valid_slot.set_forwarding(false);
        valid_slot.set_shared(true);
        bus.flush(valid_slot,start_bin_addr,start_hex_addr);
    }
}
```

**Description:** Logic for dealing with states in peer cache hit
In Code 8, when valid blocks are found in peer caches, the peer caches must change state to shared. However, some states are required to perform an action such as M-state or F-state blocks. These blocks must either forward (F: called flush in the program) or write-back to memory (M).

An alternative design to cache was considered. Like main memory, a tree-structure was considered. However, because of simplicity arrays proved to be a better suit. Hash tables can also be used. Because, hash-tables can be ordered by recency if the key is recency, the bottom most entry is the least-recently used. Moreover, a HashMap can be sorted by tag, and an LRU-queue can be used to reference the least recently used entry.

### 5.3.6 Registers

Arrays are used to represent registers. In contrast to real-world registers the smaller register is not a part of the larger registers. In example, AX is not part of EAX and there is no coherency between them. There are in total two register classes based on REX-bit being 1 or 0. Moreover, there is an extra register for FPU-operation. The registers are represented by a 2D Register.java-arrays. Nevertheless, the size of the data does not matter. However, if there is an MMX register it contains 256 bytes in real-world and 256B in the simulator. The values are randomly generated, but they are not important. When examining behaviour which register accessed is more important.

For simplicity sake, and to reduce overhead and number of objects there is only one Register_x86.java class with registers limited to one. Since there would be a great number of registers for many cores, it is not desirable regarding performance and memory concerns. In example, 1000 cores would mean 3000 registers and at least 1000 caches of C capacity. Consequently, the results of the simulation become an approximation of the real-world results.

Some architectures have registers as large as 32-bytes, but the availability of resources and the base of architecture describes a register with maximum of 16-bytes. Both MMX and SSE registers are included. In addition, there are many types of registers. Some registers are general-purpose, control and debug or MMX and SSE. A purpose refers to the application of the data. A general-purpose register can contain data, that can be used in many different operations. This may include arithmetic or logic operation as well as memory operation. In the implementation of the simulator, the purposes are specified by an index. Each index is based on a series of numbers. When the indexes are parsed, the type of register is known. The index is assigned in the instruction length method in the FileReader.java. The purposes describe the application of the data stored whether it is pure data or memory addresses. The program counter is not a part of these registers, as it is in the FileReader.java-class and is different from the address counter.
5.4 Running a simulation

5.4.1 Reading from file and constructing instructions

The program counter is the index of an array list instead of a memory address. The array list is the compilation of bytes that results in instructions, and it operates as an instruction register or a memory. Since, the instructions are not stored in the cache alongside the data, it is a split cache. The choice of not including a unified cache is that the split cache is more common recently. Moreover, since misses are only counted on data entries, the array list deviation from a normal cache structure does not matter.

A program is read by a by a byte reader, then for each byte it inserted in to the array list to preserve the natural order of the program. When the reader is finished, the array list is accessed to obtain enough bytes to compose an instruction. The first problem to be solved is instruction length. Since there is no way to determine length by interpreting the first byte unless it is an empty operation (NOOP), multiple bytes must be read to determine the length. If the first byte is PREFIX, then there is at least two bytes of instruction. There might be as many as four PREFIX bytes, so there must be a loop that iterates at most four times through the PREFIX-checks. After the prefixes have been determined, the next byte is an OPCODE. Like with prefixes, opcodes are hard coded in the FilerReader.java class. There are primary opcodes from 00H to FFH and secondary opcodes from 00H to FFH. If there is a secondary OPCODE, then in general the primary OPCODE is 0FH. There can even be a tertiary OPCODE. Because of the structure of the set of opcodes, a pattern is very difficult to extract. Especially regarding MODR/M bytes, additionally. When the correct opcodes are found, the next byte may be MODR/M, but it can also be an immediate (a constant added to the value in an operand), relative address, pointer or offset. Therefore, through the opcodes the presence of a MODR/M byte must accounted for. For each opcode, if there is a MODR/M, an immediate, a pointer, an offset, a relative address or memory address (often called real address), the cases are registered. Still, the length cannot be determined without decoding the MODR/M byte. Since MODR/M determines whether there is a SIB byte or displacement bytes, it is decoded by subdividing it into MOD, REG and RM. If MOD < 3, then there is a memory address involved. If MOD > 0 but MOD < 3 , then there are displacement bytes. Additionally, there may be a SIB byte as well. After this is determined and there are no immediate bytes, the length is calculated.

The alternative to hard coding all opcodes, was to extract them from an xml. This provides more flexibility. Moreover, the FileReader.java would require less code However, because of time constraints, this was not a priority.

In the logic that involves opcodes, all memory and data transfer operations are registered. This is useful when later executing the instructions. The instructions are passed over to the CPUController.java and further processed.
5.4.2 Translating from virtual address to physical address

When the instruction is passed over to the CPUController.java, a process must be created. When this occurs, a memory address is generated, the values from the registers are extracted and the logical address called an AbstractAddress_x86.java is created. This logical address is mapped to a memory address and a value from the registers. Also, the instructions are included. The logical address is a composition of MOD, destination-register value, displacements, relative address, pointer, offset or immediate.

Example:

```
01H: {FFH: FFH}
```

The example shows an address with MODR/M byte and register-value. However, this combination of values can be much longer. Typically, there is displacement or immediate values as well. The example shows a type of virtual address. It is how the cache simulator understands if an existing memory address is read or written to. The location of this so-called virtual address is in the AbstractAddress_x86.java object where each of the virtual addresses are mapped to physical addresses.

The abstract address is used to identify parameters so that there are no duplicate addresses. Since there is a chance that two or more instruction involves the same values, MOD, displacements etc. to ensure that these values refer to the same memory address, abstract addresses are compared.

5.4.3 Branches and JMP

When all processes are created, the list of processes in the natural order is passed over to generate branches. If an instruction includes a relative address, then it is a jump (JMP) operation. The JMP operation may be either unconditional or conditional, however the algorithm does not discriminate between the two types. Therefore, all JMP operations are set to be conditional. There is circa a 50% chance of branching. This algorithm could be more precise, when distinguishing between conditional and unconditional. However, it may not make much difference since simulations can be done up to 10 times over a single configuration. The solution as it is creates a sense of non-determinism.

5.4.4 Pre-execution stage

After generating branches, the processes continue to the multi/many-core simulation. This algorithm attempts to create a sense of parallelism as it executes processes in the same time interval. A CPUProcess.java object is removed from the processes list. There are two schedulers. In the first scheduler, registers and memory locations must be available before entering execution stage. The second scheduler only requires memory address to be available. If these conditions are met, the process object is inserted into the execution list. The scheduler
with both registers and memory address dependencies, scheduler A, must prevent registers to be accessed before it is appropriate. This is done by blocking access to the registers, if one of the operands are not available for the process to use. This is done to prevent premature read operations of registers, if there is a process writing to the registers before. This model assumes that there is only one set of register and it can be relevant in thread execution. The other scheduler, scheduler B, prevents only premature read operation on memory addresses. This can be useful in executing processes with more than one processor. As an example of the dependencies in scheduler A and B, consider three (six) memory operations:

1. Read $M_1$, then write to $M_2$
2. Read $M_2$, then write to $M_3$
3. Read $M_3$, then write to $M_4$

In this case the first operation can be executed while the second must wait. If the third is executed, then it will contain an incorrect value. Because of dependencies between them, $M_3$ is depending on $M_2$ to be finished. Because of this, $M_3$ and $M_4$ must be blocked to ensure correct result. In the real-world, this can be stall performance as it is called a data hazard. Because in the case of hazards, the system must prevent the execution by inserting a NOOP instruction. In the real-world, this is solved by bubbling the pipeline. Moreover, dependencies can be resolved in compile-time. Also, out-of-order execution prevents this. Since, results are written back in the natural order of instructions, there are less hazards.

### 5.4.5 Execution stage

Since the algorithm in the simulation allow many tasks to be executed in parallel or concurrently, the granularity of the algorithm is fine-grained. However, the actual algorithm is a sequential algorithm and because of this, atomicity conditions are fulfilled when the algorithm performs. Furthermore, race conditions and deadlocks are avoided as all processes finish and when a process cannot be finished it is inserted in a wait list. This wait list is ordered so that older processes are priority, and newer and later processes are less prioritized.

When the loop iterates, it is most important that all processes are executed, and no processes are lost. In simulation time following scenario has been observed.

$$n_{processes} = r_{remaining} + f_{finished}$$

This is always true through the loop iterations. The loop will iterate as long the lists of processes, processes in wait and processes to be executed are not empty. The elements of these lists are the remaining number of processes. The number of finished processes is a counter. It assumed that all processes later added to the $n_{processes}$ are also added $r_{remaining}$.

The first processes are prioritized, and it is the oldest process. To decide seniority, it is the process that:

1. Is oldest according to the natural order
2. Has the most attempts to execute
In the case of generating new processes, they inherit the seniority from the parent process. Because of this, the number of instructions increase throughout the execution.

When the processes are passed over to the execution stage, the logic distinguish between various of data movement operations. If the operation is not a data movement operation, it does nothing. Since values in the registers and main memory are random, there is no requirement of consistency. When executed, the results are queued and sorted by the order of sequence to preserve the natural order of the program. Since the scheduler is loosely based on out-of-order execution, the results are added to a list and action are performed when every process in a single iteration has been executed. Another observation has been made and it includes the number of executions per iteration. In the beginning of each iteration:

1. 
   \[ n_{\text{exec}} = 0 \]

   Then, after pre-execution phase the execution queue is:

2. 
   \[ n_{\text{exec}} > 0 \]

   After execution, the queue has been emptied and it is:

3. 
   \[ n_{\text{exe}} = 0 \]

The schedulers are designed to always execute one or more processes. Even if not all processors or threads are utilized. In scheduler A, the operands are always available at the beginning of the execution-phase. Then they are made unavailable for the remainder of a single iteration. It is assumed that an iteration is one cycle. Therefore, cycle time is the sum of all cycles run in parallel and in serial.
Description: Process execution: memory check.

In Code 9, memory locations are checked if unlocked and unblocked. The other algorithm includes register checks. The memory checks are done to ensure correctness. If a memory location is written to and the next process is read, the next process must wait. The memory location is also blocked.
Description: The execution and simulation of a n-core environment

The execution code in Code 10, is done sequentially. Every finished process will free memory location. Some processes are child processes of other processes. These are given the parent’s CPU number.

5.4.6 Affinity

When the results are finishing, the cache is accessed, and R/W operation can be performed. When a cache is written to all blocks with the same tag are invalidated. When there many processors, the MESIF protocol may be inefficient. Since, the number of invalidation invocations are greatly increased, the performance degrades. This is the reason why write-invalidate is not preferred for large-scale computing. Too much latency associated with bus broadcasting given the occupation of shared resource. Because of this, an alternative solution
is presented. Each process and the associated abstract address are assigned to one or more processors, but not all processors. When assigning CPUs, the abstract address contains a CPU number of $-1$. Then all the process can be assigned to any CPU. When a CPU is assigned, the address is added to a HashMap, containing an object, which includes an overview of processors that can operate on the address. Next time the address is relevant, the process can only continue if the CPUs is available or be assigned new processors. If a CPU is not available, the process will be queued.

This method is used to decrease latency in invalidation. Since some memory addresses are private to a single cache, there can possibly be fewer broadcasts to invalidate. This method is purely abstract as may not apply to real-world scenarios. However, in the approximation and abstraction of a computer the method can provide interesting results.

The scheduler is a type of affinity process scheduler. Certain memory locations can be bound to one or more CPUs.

**5.4.7 Random process generation**

Some instructions are CALL instructions. The system generates a procedure from a set of existing instructions on a CALL operation. This results in more randomness and generates more interesting results. The procedure is given the CPU number the process that generated the task. A LOOP instruction behaves like CALL, but with a random iteration value. This adds some realism. The simulator does not act on any prefixes.

**5.4.8 The bus**

The bus controls all data flow from the cache controllers to the main controller. All communication between caches and main memory flows through the bus. The Bus.java a class that is implemented through an interface (IBus.java) on almost all layers of the program. The bus handles broadcasts and invokes the invalidate block method. Moreover, when the invalidate methods is invoked, all CPUs. invalidate the block in their cache. The bus is in general a class of methods and the purpose is connecting Bus.java and the main controller.

**5.4.9 Main controller**

Main memory is connected to the other layers through the MainController.java and Bus.java. All memory traffic and all methods in main memory is invoked in MainMemory.java. The challenge was to make MainController.java available for the Bus.java class and vice versa. This is done by implementing an interface (IMc.java). To avoid co-dependency and circular dependency between the controller classes three interfaces containing controller was implemented, thus avoiding passing a reference to this object to the next layer.
Figure 18: System architecture

Description: System architecture with respect to Java classes

In Figure 18, the classes are ordered by the location of the class relative to each other. The classes are rectangles and interfaces are circles. The graphical user interface is a set of classes, thus just represented by an ellipse. The most essential classes are shown here. Note that bus and main controller is available for multiple classes and the arrow indicates the data flow. The circles indicate interfaces.

The GUI is developed with JavaFX. With JavaFX, layout can be easily developed through drag-and-drop. The layout is described in an XML-file. The GUI is simple to use. Input values for caches are 0 to 11 for capacity and 0 to \( \log_2(\text{capacity}) \) for block size. The associativity multiplied with block size cannot be larger than capacity.

The GUI displays hits, misses and miss rates. In addition to this, performance metrics such as speed-up and parallel fraction are displayed.
5.5 Problems, issues and limitations

5.5.1 Scheduler

Aside from finding the right data structures, performance was an issue. The processor code in CPUController.java was slow. Because of this, a re-write of the code occurred in the next iteration. Simplifying the lists and the logic used in the processor simulator was re-done. The while-loop in the previous iteration only considered one size, and there was additionally one inner loop. The wait-list logic was the performance degrading and instead checking the while-loop in beginning of each iteration, the wait-list moves the elements back to the processes-list. This way needless search is not required. The schedulers are inspired by out-of-order execution but is more simplified. One scheduler assumes shared register space, and the other scheduler assumes that each CPU implements a private register space. The scheduler must wait for an operand to be free before it can execute which is done by locking memory or register locations.

5.5.2 Instruction length

The x86 instruction set is large and complex and as with CISC, the instruction length is not fixed. Many possible solutions were considered, but the hard coding of each opcode was the best solution at the time. Problems related to the ISA in general is the lack of a consistent patterns in order of opcodes. A rule of thumb, as opcode number increase, more high capacity registers are used. Moreover, since the ISA is often extended, the pattern disappears, and it becomes more difficult to implement.

5.5.3 Memory constraints

Simulating more than 1000 cores was at a point impossible due to memory constraints in JVM. With a list or tree structure, an out-of-memory exception would occur. This was fixed by reducing the number of objects and introducing a hash table as main memory. Furthermore, the hash table provides at least as good performance wise as the tree structure when reducing the number of cores.

Note that the program cannot simulate 1000 cores with very large cache capacities. It can be wise to test on small capacities. It is not known what the limit is, however exceeding millions of objects will cause out-of-memory exception or GC will time-out. Constraints are added in the GUI. There is also the ability to simulate without a main memory component. This is due to performance issues and memory. The data will be randomly generated.
6 Optimal cache configuration: The results

Different configuration with different inclusion policies and multiple non-shared caches. Mainly one test program is presented for the comparison of different results. The performance metrics can be used to define the granularity of programs in addition to reviewing performance. Furthermore, this chapter includes discussions about configurations and comparison with related studies. The tests are conducted with main memory simulation.

6.1 The basics: A view on single-level cache behavior

To understand the results and to be able to compare with other results, the optimal configuration must be found for one level caches. The first tests are conducted on a single processor, then multiple processors are added. The difference in results may indicate the impact of sharing and it is expected that there will be a substantial number of coherences misses in the multiple processor simulations. The single-processor simulation will provide results that indicates the optimal configuration and the same configurations will be used in multiprocessor to understand if the configuration is optimal for both cases. Additionally, a stride prefetcher will eventually be enabled to show the difference in results.

6.1.1 Preliminary assumptions for single-processor

Consider a single-processor a SISD machine with one level of cache, a main memory and an omnibus. Misses and hits are counted per compulsory miss on read and per conflict/capacity miss write modified. They are not counted per conflict/capacity miss events write exclusive. The cache and main memory are word-addressable, where a word is one byte.

6.1.2 The single-processor

The first tests are conducted on small capacity and block size with variable associativity. As capacity and block size increases, the simulation will provide result that faithful to reality. Moreover, increased associativity will provide better hit rates as well as increased capacity and block size.
In the three following tables, a comparison over the same configurations with different inclusion policies is conducted.

**Table 2: Single-core and single-level NINE cache**

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>1</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>OFF</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>NINE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>1024</td>
<td>16</td>
<td>Direct-mapped</td>
<td>27065</td>
<td>5373</td>
<td>16,57</td>
</tr>
</tbody>
</table>

**Table 3: Single-core and single-level inclusive cache**

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>1</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>OFF</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>Inclusive</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>1024</td>
<td>16</td>
<td>Direct-mapped</td>
<td>26991</td>
<td>5353</td>
<td>16,56</td>
</tr>
</tbody>
</table>
Table 4: Single-core and single-level exclusive cache

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>1</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>OFF</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>Exclusive</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>1024</td>
<td>16</td>
<td>Direct-mapped</td>
<td>27151</td>
<td>5369</td>
<td>16.51</td>
</tr>
</tbody>
</table>

Comments: As expected, the miss ratio does not vary over different inclusion policies with this configuration. A direct-mapped cache with 1KB capacity and 16B block size provides. The results in tables: Table 2, Table 3 and Table 4 show an high miss rate. It is near optimal to achieve miss rates less than or equal to 5%.

If a prefetcher is enabled, the test will yield better results. The next three tables will show the results with a prefetcher. No changes aside from that.

Table 5: Single-core and single-level NINE cache with prefetcher

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>1</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>NINE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>1024</td>
<td>16</td>
<td>Direct-mapped</td>
<td>30887</td>
<td>5458</td>
<td>15.02</td>
</tr>
</tbody>
</table>
Table 6: Single-core and single-level Inclusive cache with prefetcher

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>1</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>Inclusive</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>1024</td>
<td>16</td>
<td>Direct-mapped</td>
<td>30620</td>
<td>5343</td>
<td>14,86</td>
</tr>
</tbody>
</table>

Table 7: Single-core and single-level exclusive cache with prefetcher

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>1</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>Exclusive</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>1024</td>
<td>16</td>
<td>Direct-mapped</td>
<td>30790</td>
<td>5401</td>
<td>14,93</td>
</tr>
</tbody>
</table>

Comments and comparison: The simple prefetcher have a clear impact on results. Comparing Table 2 and Table 5, Table 3 and, Table 6 and Table 4 and Table 7, there is approximately a 1,5% reduction in miss rates. If 1 ns equals one clock cycle and assuming one hit is one cycle and one miss results in a ten cycles penalty, the performance increase with 1,5 cycles per cache access. Furthermore, 97% of all instructions the test program require memory access (data references). Comparing AMAT for prefetcher ON and OFF, the result between the different inclusion policies are the roughly the same and assuming:
Table 8: Assumptions for hit time, miss penalty and base CPI

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Hit time</td>
<td>1 cycle</td>
</tr>
<tr>
<td>Miss penalty</td>
<td>100 cycles</td>
</tr>
<tr>
<td>Base CPI</td>
<td>1</td>
</tr>
</tbody>
</table>

1. Prefetch OFF

\[ AMAT = 1 + (0,165 \times 100) = 17,5 \text{ cycles} \]

2. Prefetch ON

\[ AMAT = 1 + (0,15 \times 100) = 16 \text{ cycles} \]

Unfortunately, the instruction cache miss rate is ignored. However, there is a slight improvement in performance on the test configuration in Table 2 to Table 7 when prefetcher is enabled.

Types of misses: Compulsory misses are the prevalent type with approximately 74% (4050 of 5500) of the total misses, while the remaining share of misses are conflict misses.

If capacity and block size is increased, it is expected that miss rates will be decreased. An increase in block size will reduce compulsory misses. The following three following tables will present results based on different capacity, block size and associativity.

Table 9: Increased block size

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>1</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>OFF</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>NINE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>1024</td>
<td>64</td>
<td>Direct-mapped</td>
<td>15115</td>
<td>2553</td>
<td>14,45</td>
</tr>
</tbody>
</table>

Comments: The miss rate is reduced by approximately 2% with an increase in block size. The number of compulsory misses is in this test: 2252, which is reduction of approximately 55%. However, the share of compulsory misses in the total number of misses are 0,89% (2252 of 2553), which implies that the number of conflict misses are reduce as well.

Increasing capacity will further decrease the number of misses.
Table 10: Increased capacity

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>1</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>OFF</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>NINE</td>
</tr>
<tr>
<td>Cache</td>
<td>Capacity in bytes</td>
</tr>
<tr>
<td>L1</td>
<td>65536</td>
</tr>
</tbody>
</table>

Comments: The number of compulsory misses is now 100% of the total misses. Conflict misses has been completely reduced. The miss rate is close to target of 5%.

To further reduce cache misses, associativity can be increased. Most commonly the associativity reduces the number of conflict misses. Since, the number of conflict misses are zero, it may not have a great impact on the results.

Table 11: Increased associativity

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>1</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>OFF</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>NINE</td>
</tr>
<tr>
<td>Cache</td>
<td>Capacity in bytes</td>
</tr>
<tr>
<td>L1</td>
<td>65536</td>
</tr>
</tbody>
</table>

Comments As expected based on the results from Table 10, the increase of associativity has little impact on cache performance for this test program at 64KB capacity and 64B block size.

To accomplish miss rate of less than 5%, prefetcher can be enabled. This will have an impact on compulsory misses.
Table 12 Enabling prefetcher

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>1</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>NINE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>65536</td>
<td>64</td>
<td>2</td>
<td>17616</td>
<td>786</td>
<td>4.28</td>
</tr>
</tbody>
</table>

Comments: The number of compulsory misses is reduced by 23% when enabling prefetcher.

\[ AMAT = 1 + (0.043 \times 100) = 5.3 \text{ cycles} \]

6.2 Adding more processors: Adding parallelism

Multi-core environment, the first part of n-processor simulations.

6.2.1 Preliminary assumptions for multi-processor

Consider a multi-processor a SPMD machine with one level of cache, a main memory and an omnibus. Assume a SMP machine. Additionally, consider a shared memory model like NUMA. Misses and hits are counted per compulsory miss on read and per conflict/capacity miss write modified. They are not counted per conflict/capacity miss events write exclusive. Whenever an entry is in a peer cache, then the number of misses is reduced by one. Because, it is not a main memory access and there is no use of memory bandwidth. However, it can be assumed that there is in increased latency.

6.2.2 The Multi-processor: Results

The configurations from the single-processor test will be used in the multi-processor. In addition to the 3 C’s the 4th C will be present. It is excepted that cache thrashing and sharing will impact results. Is there a notable difference in results from single-core to multi-core?
Table 13: Dual-core, single-level cache with near-optimal configuration

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>2</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>NINE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>65536</td>
<td>64</td>
<td>2</td>
<td>17288</td>
<td>928</td>
<td>5.1</td>
</tr>
</tbody>
</table>

**Comments:** Compared to the results from Table 12, there is a slight increase in misses. This is caused by synchronisation issues between the peer caches. 100% of the misses are compulsory misses. There are no coherency misses with two processors in this test.

Increasing the number of processors to 4, may increase the number of coherence misses.

Table 14: Quad-core single-level cache with near-optimal configuration

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>4</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>NINE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>65536</td>
<td>64</td>
<td>2</td>
<td>17340</td>
<td>989</td>
<td>5.4</td>
</tr>
</tbody>
</table>

**Comments:** The number of misses is slightly increased with a doubling in the number of processors. There are in total 1000 misses, but only 989 of them results in memory access. 99% of the total misses are compulsory misses and the remaining percentage is coherence misses. This can be because of false or true sharing. However, with such small number of coherence misses, cache line thrashing is not prevalent in this test.

Further increasing the number of processors most probably increase the likelihood of cache thrashing.
Table 15: 16-core, single-level cache with near-optimal configuration

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>16</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>NINE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>65536</td>
<td>64</td>
<td>2</td>
<td>15634</td>
<td>821</td>
<td>4.99</td>
</tr>
</tbody>
</table>

Comments: A small decrease in misses compared to the results in Table 13 and Table 14. The actual number of misses are 3291, where 1659 misses are compulsory, and 1632 misses are coherence misses. Since, misses are subtracted when a block is in a peer cache, the actual number of misses that requires memory accesses is; 821. A quadrupling in the number of processors increases the number of coherences misses by a large factor. It can be assumed that many of these misses are caused by false-sharing. The likelihood of false sharing occurrences increases with larger block size. Since a larger block can contain more data, there is a chance that the two or more processors reference the same block, but different locations in the block. Then, when one processor writes to the block, the cache block is invalidated in the peer caches. This forces the peer cache to reload the block and it increase latency in the system. When this occurs repeatedly cache thrashing occurs and most likely, based on the numbers of coherence misses, cache thrashing is most probably very prevalent.

To investigate the number of false sharing occurrences, block size can be reduced. This however increases the number true sharing occurrences and does not reduce the prevalence of cache thrashing. It is difficult to assess the actual number of false sharing instances without providing a means to count it. The next tables will present results with small block size to reduce false sharing. The inspiration is partial block invalidations (Chen & Dubois, 1993) and subblock coherence (Kadiyala & Bhuyan, 1995).
Table 16: Reducing block size to reduce the occurrence of false sharing

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>16</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>NINE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>65536</td>
<td>16</td>
<td>2</td>
<td>28256</td>
<td>1998</td>
<td>6,61</td>
</tr>
</tbody>
</table>

Comments: A decrease in block size results in larger miss ratio. The total number of misses is 5619 where the number of compulsory misses is 4736 and the number of coherence misses is 883. Compared to the results from Table 15 there is reduction close to 50% in coherence misses. It can be assumed that many false sharing misses are the difference in results. Note that a decrease in block size increases the number of compulsory misses.

Further block size can be decreased. To completely remove the occurrence of false sharing, block size can be reduced to $1B$. One disadvantage such small block size is larger overhead.

Table 17: Eliminating false sharing

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>16</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>NINE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>65536</td>
<td>1</td>
<td>2</td>
<td>241701</td>
<td>21090</td>
<td>8,03</td>
</tr>
</tbody>
</table>

Comments: Decreasing block size to $1B$, completely abolish false sharing. However, the occurrence of true sharing is increased. The number of misses is 43472, but memory access is only required in 21090 of them. The number of compulsory misses is 39593 and the number of coherence misses is 3879. Approximately 9% are coherence misses. Compared to the results from Table 15 and Table 16, where the share is approximately 50% and 16% respectively, the number of false sharing instances can be between a much as 50% of all coherence misses in $64B$ block size configuration to zero in a $1B$ block size configuration.
Graph 1: The relationship between coherence and compulsory misses per block size

Description: An increase in block size in a 64KB cache, with 16 processors decreases the difference between compulsory and coherence misses. False sharing probably is probably the cause of increase in coherence misses.

To show the relationship between coherence and compulsory misses, Graph 1 show a clear increase in coherence misses, most likely false sharing misses at increasing block size. The compulsory misses are reduced at per increasing block size. Thrashing is more likely to occur with larger block size because of the presence of both false and true sharing.

Table 18: ollydbg2.exe on near-optimal configuration

<table>
<thead>
<tr>
<th>Program</th>
<th>ollydbg2.exe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>16</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>NINE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>65536</td>
<td>64</td>
<td>2</td>
<td>31361</td>
<td>1834</td>
<td>5.53</td>
</tr>
</tbody>
</table>

Comments: This program has miss on about 3800 blocks, but the actual number of main memory accesses is 1834. About 50% of the 3800 misses are coherence misses; the remaining share is compulsory misses. The program utilizes all processors.
Table 19: iexplorer.exe on near-optimal configuration

<table>
<thead>
<tr>
<th>Program</th>
<th>iexplorer.exe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>16</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>NINE</td>
</tr>
<tr>
<td>Cache</td>
<td>Capacity in bytes</td>
</tr>
<tr>
<td>L1</td>
<td>65536</td>
</tr>
</tbody>
</table>

Comments: Compared to firefox.exe and ollydbg2.exe, iexplorer.exe is cache-friendly. There are 2132 misses in total of which 396 results in main memory accesses. There are 1277 coherence misses and 855 compulsory misses. The program is more coarse-grained than both ollydbg2.exe and firefox.exe and utilizes 14 of 16 processors. There are more dependencies.

Table 20: photodirector.exe on near-optimal configurations

<table>
<thead>
<tr>
<th>Program</th>
<th>photodirector3.exe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>16</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>NINE</td>
</tr>
<tr>
<td>Cache</td>
<td>Capacity in bytes</td>
</tr>
<tr>
<td>L1</td>
<td>65536</td>
</tr>
</tbody>
</table>

Comments: The number of total misses is 3674 where 766 results in main memory accesses. The number of coherence and compulsory misses are 2133 and 1541, respectively. It can utilize at least 16 processors.
Description: The miss ratios for each of the test programs with a 2-way set associative, 64KB capacity and 64B block size cache.

To further show that different programs perform differently, Graph 2 presents comparable results done with 64KB capacity, 64B block size and 2-way set-associativity.

6.2.3 Preliminary assumptions for many-processor

The same assumptions that are made for multi-core are still valid.

6.3 Moving to many-core architectures

Many-core architecture is defined as an architecture with more than 16-processors in this thesis. See subsection 2.1.3.

6.3.1 The Many-processor: Results

The configurations from single- and multi-core is still used. Will there be an increase in misses? Will false sharing be more prevalent, and will the caches be subject to cache thrashing to larger degree than with multi-core simulations. How many CPUs are used per simulation.

The three next tables will show the results from 32,128 and 512 processors. This is the max number of processes tested on this thesis.
Table 21: 32-core, single level cache

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>32</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>NINE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>65536</td>
<td>64</td>
<td>2</td>
<td>15382</td>
<td>876</td>
<td>5,39</td>
</tr>
</tbody>
</table>

**Comments:** An increase in processors have virtually no change in miss ratio. Compared to the results in Table 15, there is a slight increase in the number of compulsory misses. The number of compulsory and coherency misses are 2145 and 1330, respectively. This perhaps due to randomness in the simulator. The number of processors utilized is 17 of 32 processors.

Increasing the number processors will not significantly provide better performance. Since firefox.exe is not able to run on all processors. There may be an increase in number of utilized processors, but the relative increase to the total number of processors will be negligible.

Table 22: 128-core, single level cache

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>128</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>NINE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>65536</td>
<td>64</td>
<td>2</td>
<td>14536</td>
<td>808</td>
<td>5,27</td>
</tr>
</tbody>
</table>

**Comments:** There is almost no difference in number of misses (main memory accesses) compared to the results in Table 21. However, the total number of misses is 3259 compulsory misses and 1410 coherence misses; 4669. As expected, the program does not utilize all the processors. The number of processors used is 31 of 128.

Increasing the number of processors further, will probably increase the number processors used. It the relative value of used processor over total number of processors will probably be smaller than in the results from Table 21 and Table 22.
Table 23: 512-core, single level cache

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>512</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>NINE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>65536</td>
<td>64</td>
<td>2</td>
<td>13766</td>
<td>775</td>
<td>5.33</td>
</tr>
</tbody>
</table>

Comments: The miss rate is approximately the same as in Table 21 and Table 22. However, there is 775 memory accesses of a total of 5428 misses, where 4089 are compulsory misses and 1339 are coherence misses. A noticeable increase in compulsory misses. This is most probably due to a naïve scheduler. There are more processors that miss on first reference. The total number of processors used is 75 of 512.

Graph 3: Relative CPU usage

Description: The number increase in efficiency is smaller for a higher number of processors.

The Graph 3 shows how little gain there is from adding more processors based on the most processor running at the same time.

Can Amdahl’s law or Gustafson – Barsis’s law show the difference in speedup with the results available?
Finding $p$:

A method of finding $p$ is to rearrange Amdahl’s law to express (see section 2.1.2):

$$p = \left(1 - \left(\frac{1}{5}\right)\right) \times \left(\frac{n}{(n-1)}\right)$$

For $n \geq 2$

Assume:

$$S_{latency} = \frac{T_{serial} + T_{parallel} \times n}{T_{Serial} + T_{Parallel}}$$

As shown in section 2.1.2, where $n$ is the average number of processors in parallel execution. $T_{serial}$ and $T_{parallel}$ is time for serial and parallel execution, respectively. As assumed in section 2.1.2, in the part about Amdahl’s law, it is assumed that $T_{serial} + T_{parallel} \times n = 1$.

Therefore, $T_{parallel} \times n = I_{parallel}$ and $T_{serial} = I_{serial}$ where $I_{serial}$ is the number of instructions executed in serial and $I_{parallel}$ is the number of instructions run in parallel. Then, Amdahl’s formula can be expressed as:

$$S_{latency} = \frac{I_{serial} + I_{parallel}}{I_{serial} + \frac{I_{parallel}}{n}}$$

Then assume that:

$$I_{Serial} + I_{Parallel} = 1$$
**Graph 4: The evolution of P and S over n-processors for firefox.exe**

![Graph 4: P versus S](image)

**Description:** firefox.exe: p is decreasing with more processors, but s is increasing. This is because the with more processors, the program cannot fully utilize the potential of n-cores. Many processors are not used. Thus, parallelization is decreased relative to processor usage.

From the next results will focus on Amdahl’s law and Gustafson – Barsis’s law. As seen in Graph 4, p and s does not improve a lot when using average CPU utilized in parallel.

First the results represent when blocking register when writing to the registers. This will naturally decrease the level of parallelization.

**Table 24: Performance results**

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Relative usage of processors (Max number of processors in use)</th>
<th>Portion of program run in parallel (p)</th>
<th>Average use of processors per cycle (for parallel execution)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0,53</td>
<td>0,63</td>
<td>3,32</td>
</tr>
<tr>
<td>128</td>
<td>0,24</td>
<td>0,61</td>
<td>3,32</td>
</tr>
<tr>
<td>512</td>
<td>0,15</td>
<td>0,61</td>
<td>3,31</td>
</tr>
</tbody>
</table>

Ignoring workload and if the average use of processors is approximately 4, there is a speed-up by a factor of 4.

The next test will include Amdahl’s law. The assumptions are that all processors are used and p is calculated with respect to the total number of processors. The parallel fraction used is the parallel fraction seen in Table 24.
Thus, with Amdahl’s law:

1. 32 processors:
   \[ S_{\text{latency}} = \frac{1}{(1 - 0.63) + \frac{0.63}{32}} = 2.56 \]

2. 128 processors:
   \[ S_{\text{latency}} = \frac{1}{(1 - 0.61) + \frac{0.61}{128}} = 2.53 \]

3. 512 processors:
   \[ S_{\text{latency}} = \frac{1}{(1 - 0.61) + \frac{0.61}{512}} = 2.56 \]

Since the average use of processors is roughly 3,30 for 32, 128 and 512 processors, the difference is small between. However, there is a slight difference performance with the different number of processors. Furthermore, it is assumed that workload is equal per instruction.

Next, calculation done with Gustafson – Barsis’s law. Note that \( p \) is calculated differently and instead of the total number of processors, the average number of processors is used. It is assumed that registers are no shared.

And with Gustafson – Barsis’s law:

1. 32 processors:
   \[ S_{\text{scaled}} = 3.43 + (1 - 0.99) \times (1 - 3.43) = 3.41 \]

2. 128 processors:
   \[ S_{\text{scaled}} = 3.45 + (1 - 0.97) \times (1 - 3.45) = 3.38 \]

3. 512 processors:
   \[ S_{\text{scaled}} = 3.71 + (1 - 0.89) \times (1 - 3.71) = 3.41 \]

Since, the calculations make different assumptions it is difficult to compare. Since, registers was assumed to be shared in the last Amdahl’s law calculation. If registers are assumed to be non-shared and the assumptions are equal. How will the result be? The next calculations will show Amdahl’s law based on the same assumptions as with Gustafson – Barsis’s law.
With Amdahl’s law:

1. 

\[ S_{\text{latency}} = \frac{1}{(1 - 0.998) + \frac{0.998}{3.45}} = 3.43 \]

2. 

\[ S_{\text{latency}} = \frac{1}{(1 - 0.991) + \frac{0.991}{3.45}} = 3.39 \]

3. 

\[ S_{\text{latency}} = \frac{1}{(1 - 0.97) + \frac{0.97}{3.71}} = 3.42 \]

If the results from 1, 2 and 3 are optimal for this level of parallelism, there is a decrease in the speed-up. Due to a low-level of parallelism. Note that these calculations are meant to show the difference in increasing the number of processors for the test program.

Graph 5: Speed-up over n-processors firefox.exe

Graph 5.5: There is little actual gain in running firefox.exe on 128 – 512 processors. Even 32 processors may be excessive.

The comparison between number of processors in Graph 5 show that there is little difference in performance. It must be the granularity of the program. Unfortunately, the tests have not been conducted on embarrassingly parallel programs. The assumptions for speed-up are conducted with shared register. If it was assumed that each CPU have their own set of registers, the result would be higher, but the difference between each scenario approximately the same.
6.3.2 Discussion

Since \( p \) is different from each of the simulations on 32, 128 and 512 processors, but the average number of processors in use is roughly the same, firefox.exe is not optimized for many-core architectures. The difference in \( p \) is mainly due to the maximum number of processors that can execute instructions at the same time.

6.4 Adding another cache: Near-optimal solution for single-processors

Multi-level caches normally increase performance and improves hit rates in \( L_1 \).

6.4.1 Expectations

When adding another cache, inclusion policies will have an impact on results. The first goal is to find a near optimal configuration for a single-processor. The assumptions from the tests in the previous subchapters are still valid. In addition, one other cache will be added. The misses are counted equally for all policies. It is expected that inclusive caches will yield better results for \( L_2 \) than the other policies will. Since, \( L_2 \) in inclusive caches counts hits on write modified, the number of hits will be larger as the other policies do not write to \( L_2 \) on write modified.

6.4.2 Single-processors: Results

Table 25: Single-processor, inclusive multi-level cache

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>1</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>Inclusive</td>
</tr>
<tr>
<td>Cache</td>
<td>Capacity in Bytes</td>
</tr>
<tr>
<td>L1</td>
<td>65536</td>
</tr>
<tr>
<td>L2</td>
<td>1048576</td>
</tr>
</tbody>
</table>

Comments: Compared to results from Table 12, the \( L_1 \) miss rates are roughly the same. However, the average memory access time is obviously smaller if miss penalty for \( L_1 \) are now 10 cycles, \( L_2 \) miss penalties are 100 cycles and \( L_2 \) hit time is 5 cycles.

\[
AMAT = 1 + (0.042 \times (5 + 0.054 \times 100)) = 1.44 \text{ cycles}
\]
Compared to the average memory access time calculated from the miss rate in Table 12, the performance is improved by a factor of $\sim 3.7$. Will the other inclusive policies yield as good improvement as an inclusive cache.

An exclusive cache will most probably yield as good results in $L_1$ as a single-level cache. Moreover, the exclusive cache benefits from smaller difference between the cache levels. The capacity should be smaller than the inclusive cache capacity. Since the efficiency of $L_2$ should be more than zero, the capacity is scaled down by 50% in $L_1$ and 69% in $L_2$. This is done because some preliminary results showed less efficient $L_2$ cache, if the capacity in $L_1$ is too large. It is done to justify the use of a multilevel cache.

Table 26: Single-processor, exclusive multi-level cache

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>1</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>Exclusive</td>
</tr>
<tr>
<td>Cache</td>
<td>Capacity in Bytes</td>
</tr>
<tr>
<td>L1</td>
<td>32768</td>
</tr>
<tr>
<td>L2</td>
<td>32768</td>
</tr>
</tbody>
</table>

**Comments:** Compared to results from Table 12, the $L_1$ miss rates are roughly the same.

\[
AMAT = 1 + \left( 0.045 \times (5 + 0.642 \times 100) \right) = 4,11 \text{ cycles}
\]

An exclusive cache yields 1.3 times better results than a single-level cache. However, this is not on the same configurations as the configuration used in to produce the miss rate in Table 12. The $L_2$ is not fully utilized and exclusive caches are best suited for victim caches.

A NINE-cache may produce the same results in $L_1$ as the other types of cache. However, $L_2$ may a bit larger in miss rates than the other types of cache’s results. The optimal configuration for is perhaps $L_1$ capacity is equal to the optimal exclusive cache $L_1$ capacity and $L_2$ capacity equal to optimal inclusive cache $L_2$ capacity.
Table 27: Single-processor, NINE multi-level cache

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>1</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>NINE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in Bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>32768</td>
<td>64</td>
<td>2</td>
<td>16983</td>
<td>599</td>
<td>3,41</td>
</tr>
<tr>
<td>L2</td>
<td>1048576</td>
<td>64</td>
<td>8</td>
<td>117</td>
<td>482</td>
<td>80,47</td>
</tr>
</tbody>
</table>

Comments: The miss rate in $L_1$ is much smaller than the $L_1$ miss rate from Table 26. The miss rate in $L_2$ is much larger than the $L_2$ miss rate from Table 25. Average memory access time will be calculated.

\[ AMAT = 1 + \left( 0.034 \times (5 + 0.804 \times 100) \right) = 3.90 \text{ cycles} \]

The performance is better than the exclusive cache, but worse than the inclusive cache. If the results are comparable, the NINE-cache has an improvement in performance of a ~1.7 times the average memory access time calculated from miss rate in Table 12.

The question that arises is how these results will change by adding more processors. How efficient will the added cache be? The multi-processors will not be tested, since the results in subsections 6.2 and 6.2.3 are near equal.

### 6.5 Adding another cache: Near-optimal solution for parallel computing

Many-core, multilevel caches provides great potential in improving performance.

#### 6.5.1 Expectations

It is expected that $L_2$ cache will be less efficient for exclusive, NINE and inclusive caches. The results from 5.1 will probably be better. However, it is expected to be required to adjust the configurations to the parallel execution. The average number of processors will be the same as with the number of processors.
6.5.2 Many-core: Are multi-level caches efficient?

Table 28: 32-core, inclusive multi-level cache

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>32</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>Inclusive</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in Bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>65536</td>
<td>64</td>
<td>2</td>
<td>14478</td>
<td>977</td>
<td>6.33</td>
</tr>
<tr>
<td>L2</td>
<td>1048576</td>
<td>64</td>
<td>8</td>
<td>10750</td>
<td>775</td>
<td>6.73</td>
</tr>
</tbody>
</table>

Comments: Compared to the results in Table 24, there is an increase in the number of misses.

Table 29: 128-core, inclusive multi-level cache

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>128</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>Inclusive</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in Bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>65536</td>
<td>64</td>
<td>2</td>
<td>12863</td>
<td>975</td>
<td>7.05</td>
</tr>
<tr>
<td>L2</td>
<td>1048576</td>
<td>64</td>
<td>8</td>
<td>9959</td>
<td>780</td>
<td>7.27</td>
</tr>
</tbody>
</table>

Comments: The increase in miss ratio continues. But $L_1$ misses are reduced compared to the results in Table 28.
Table 30: 512-core, inclusive multi-level cache

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>512</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>Inclusive</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in Bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>65536</td>
<td>64</td>
<td>2</td>
<td>12156</td>
<td>904</td>
<td>6,93</td>
</tr>
<tr>
<td>L2</td>
<td>1048576</td>
<td>64</td>
<td>8</td>
<td>9399</td>
<td>753</td>
<td>7,42</td>
</tr>
</tbody>
</table>

**Comments:** The increase in miss changes little from the previous two tables. But $L_1$ misses are reduced from the results in Table 29.

Table 31: Decreasing L1 size over 32-cores

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>32</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>Inclusive</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in Bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>32768</td>
<td>64</td>
<td>2</td>
<td>13860</td>
<td>885</td>
<td>6,01</td>
</tr>
<tr>
<td>L2</td>
<td>1048576</td>
<td>64</td>
<td>8</td>
<td>10336</td>
<td>560</td>
<td>5,14</td>
</tr>
</tbody>
</table>

**Comments:** Compared to the results in Table 28, the miss rate is reduced by nearly $\sim 0,28\%$ in $L_1$ and $\sim 1,5\%$ in $L_2$. Cache performance is 1,61 from the results in Table 31 versus 1,74 in from the results in Table 28.
Table 32: Increasing L2 capacity cache over 32-cores

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
<th>Number of CPUs</th>
<th>32</th>
<th>Number of simulations</th>
<th>1</th>
<th>Prefetch</th>
<th>ON</th>
<th>Inclusion policy</th>
<th>Inclusive</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in Bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>32768</td>
<td>64</td>
<td>2</td>
<td>13774</td>
<td>820</td>
<td>5,62</td>
</tr>
<tr>
<td>L2</td>
<td>2097152</td>
<td>64</td>
<td>8</td>
<td>10097</td>
<td>540</td>
<td>5,08</td>
</tr>
</tbody>
</table>

Comments: Increasing difference in capacity, clearly have a great impact on results. Notice that \(L_2\) have a smaller miss rate than \(L_1\) have. Since hits are counted on write modified, the miss ratio in \(L_2\) is better than in exclusive- and NINE-caches.

The three next results following will include 32-processors with 2-level exclusive cache. The configuration will be altered to be closer to optimal configuration.

Table 33: 32-Core, exclusive multi-level cache

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
<th>Number of CPUs</th>
<th>32</th>
<th>Number of simulations</th>
<th>1</th>
<th>Prefetch</th>
<th>ON</th>
<th>Inclusion policy</th>
<th>Exclusive</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in Bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>65536</td>
<td>64</td>
<td>2</td>
<td>15761</td>
<td>853</td>
<td>5,14</td>
</tr>
<tr>
<td>L2</td>
<td>65536</td>
<td>64</td>
<td>2</td>
<td>0</td>
<td>1745</td>
<td>100</td>
</tr>
</tbody>
</table>

Comments: The \(L_2\) cache is useless with this configuration. The average memory access time is comparable with a single-level cache.

What if capacity is decreased? Will an exclusive cache yield better results given the environment?

The capacity will be decreased to 32\(KB\) in the next table.
Table 34: 32-Core, exclusive multi-level cache with reduced capacity

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>32</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>Exclusive</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in Bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>32768</td>
<td>64</td>
<td>2</td>
<td>15229</td>
<td>565</td>
<td>3,58</td>
</tr>
<tr>
<td>L2</td>
<td>32768</td>
<td>64</td>
<td>2</td>
<td>50</td>
<td>1565</td>
<td>96,91</td>
</tr>
</tbody>
</table>

**Comments:** Decreasing capacity provides great $L_1$ results. However, the $L_2$ miss rate is still too large.

The results can still be improved. What if associativity was increased by a factor of 4.

Table 35: 32-Core, exclusive multi-level cache reduced capacity and increase associativity

<table>
<thead>
<tr>
<th>Program</th>
<th>firefox.exe (TOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>32</td>
</tr>
<tr>
<td>Number of simulations</td>
<td>1</td>
</tr>
<tr>
<td>Prefetch</td>
<td>ON</td>
</tr>
<tr>
<td>Inclusion policy</td>
<td>Exclusive</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Capacity in Bytes</th>
<th>Block size in bytes</th>
<th>Associativity</th>
<th>Hits</th>
<th>Misses</th>
<th>Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>32768</td>
<td>64</td>
<td>8</td>
<td>15110</td>
<td>477</td>
<td>3,07</td>
</tr>
<tr>
<td>L2</td>
<td>32768</td>
<td>64</td>
<td>8</td>
<td>11</td>
<td>1329</td>
<td>99,18</td>
</tr>
</tbody>
</table>

**Comments:** Increasing associativity improves provides great $L_1$ results.


6.6 Discussion

When reviewing the results, there are a few distinctions. Some results are different from related studies. However, this may due to different protocols. Most of the results from a single-level cache scenario, correlates with results from related studies. The many-core scenarios are more difficult to correlate.

6.6.1 Single-level setup

The single-level setup correlates well with prior studies (Culler et al., 1999). It behaves in a close-to-reality fashion. The block size, capacity and associativity provide great results. Also, the prefetcher yields good performance.

6.6.2 Partial block invalidation

The results are not equal to the results from the study of partial block invalidation (Chen & Dubois, 1993). However, it may be different because of a different cache coherency protocol. The large overhead and the large number of compulsory misses does not advocate partial block invalidation inspired tests. The question remains if false sharing misses are more performance degrading than a huge number of compulsory misses is. If false sharing misses were counted in the simulator, it would be easier to assume the degree of false sharing misses and thrashing. Even if miss ratio in Table 17 was ~3% more than the miss ratio in Table 15, it may be more inter-cache communication with a larger block size, due to thrashing and false sharing. The inter-cache communication increase latency and another question is how much slowdown is caused by it? A compulsory miss often requires memory access, thus increase memory access times. Coherence misses may imply that there is valid copy somewhere among the peer caches, thus increases latency. Intuitively, there is less gain in decreasing block size because memory access times increase compared to the increase latency for coherence misses. However, it depends on system architecture and coherency protocol.

6.6.3 Multi-level setup

The multi-level behaviour for inclusive caches is as expected. However, the NINE and exclusive cache yields poor $L_2$ performance. This may be due to the method of counting. Moreover, it also may depend on how the main test program behaves. It may be the case that exclusive property is best suited for victim caches (Jouppi, 1990). Since the penalty for missing in $L_1$ on coherence misses are greater than in an inclusive cache, but equal on compulsory misses, the exclusive cache is great as the last resort.
6.6.4 Many-core performance: The declining parallelization factor

Reviewing the performance, the main test program was of a more coarse-grained than fine-grained. This correlates with the notion that not all improvements in hardware is necessarily benefit performance. The factor $p$ is declining for an increasing number of processors. The serial part is increasing with a larger number of processors. This may be that the program is not developed to utilize that many cores.

Graph 6: Parallelization factor for all test programs on 32-cores

The parallelization factor is small in all test programs as seen in Graph 6. For an embarrassingly parallel program it is close to 1. The tests in Graph 6 are done with a shared register model.

If registers are separated, one for each CPU, the parallel fraction and speedup for each test on 32 cores program are:
Graph 7: P and S for test programs with separate registers

Description: The results from performance test for all test programs

In Graph 7, compared to Graph 6, the results are very different. The relative difference in \( p \) between each of the programs, are almost equal to the parallel fractions in Graph 6. This can imply that memory access was a much greater factor in deciding the program’s granularity. Here \( p \) is calculated with the total number of processors.

6.6.5 Comparing results to the VTune Amplifier 2019

The VTune AM shows an actual statistic and can accurately measure performance. Related to the parallelization factor, VTune AM presents a small utilization of all cores. This is naturally in the context of the computer the program is run on. The machine used to run the simulation is an 8-core core i7. Still the main test program does not utilize all the cores. The advantage of a tracing program is the factual performance measurements. In this context, the number of cache misses are much larger. Probably due to an accurate execution of instructions; REPEAT, LOOP and CALL. The last LLC miss rate is small, but it is probably a shared cache. This project did not cover shared caches, which may yield worse results than an actual computer would.
7 Conclusion and further works

The conclusion derived from the results from chapter 6. Some of the topics are discussed in chapter 4. This is largely based on the findings through simulation of various programs.

7.1 The performance of a single-level cache in single-processors

Review of single-level caches in a single-core environment.

7.1.1 Optimal configuration

Naturally, there is no difference between the different type of caches with respect to the various inclusion policies. Thus, the results are roughly equal. The impact of cache size, block size and associativity are faithful to reality. The increased block size reduces the number of compulsory misses, while associativity influences the number of conflict misses. A 2-way set associative cache 64KB cache completely reduces the number of conflict-misses to zero. As this is in an isolated environment, where only one program is running, the results are not quite comparable to actual real-life cache performance. Consider, multiple programs running, and configuration is required to be adjusted to the increasing number of programs. However, the problem is scalable, so the 64KB cache is the space required for the test programs to run near-optimal with respect to cache performance. An increase in associativity will also be required to address issues with conflict misses with more programs running.

7.1.2 Enabling prefetching mechanism

A simple prefetcher increased performance with approximately 1,5 cycle per instruction on average in direct-mapped 1KB cache with 16B block size and 1,9 cycles per instruction on average in a 2-way set associative cache 64KB cache with 64B block size. A more advanced prefetcher may yield better results. When prefetcher is enabled the number of compulsory misses was further reduced. Nevertheless, the disadvantage with a prefetcher is the use of memory bandwidth. Especially if there is CALL, LOOP or JMP instruction, the program counter is not changed to reference the adjacent address, most likely. In this case the prefetcher fetches useless data that is likely to be evicted soon. However, since the performance is improved overall, it can be assumed that the number of such instances is small.
7.1.3 Share of memory access instructions

The test programs provided a high number of memory access instructions. This may because the programs are initialized and the largest portion available to the simulator is function, variable and constant declaration.

7.2 Multi-processors: Cache performance in a single-level cache


7.2.1 The 4th C: Sharing misses and thrashing

Increasing the number of processes to 2 processors did not affect the number of coherence misses in the tests. However, when increasing the number of processors to 4, coherence misses increased, but only with an insignificant number. Evidently, there was no occurrence of cache thrashing or it did not impact as much as expected on this number of processors. Further, the increase in processors to 16, provided a significant number of coherence misses. Of the ~1600 coherence misses, there is probably a great share of false sharing. It is hard to distinguish the number of false sharing misses from true-sharing, but cache thrashing is prevalent. The programs are not optimized for such level parallelism, and the performance degradation is significant.

7.2.2 Reducing false sharing

A known technique to reduce the number of false-sharing instances is to reduce the block size. The disadvantage in reducing block size is the increase in true-sharing instances, thus it does not remove the prevalence of cache thrashing. The partial block invalidation (Chen & Dubois, 1993) and subblock invalidation schemes (Kadiyala & Bhuyan, 1995) was the inspiration to conduct such experiment. When reducing the block size to 16B, the coherence misses was reduced by 50%, the share of coherence misses in overall number of misses were 16%. Evidently, the number of false-sharing instances is much smaller, but it is still difficult to assess the actual number of false-sharing instances. To completely remove false-sharing, the block size was reduced to 1B, as the cache is word-addressable this is the smallest block size possible. The results show a great increase in total misses. The share of coherence misses was approximately 9%. None of the coherence misses were caused by false-sharing. This implies that the share of false-sharing can be as much as 50% of the total number of misses. However, if assumed that in the 16B example:

\[0,09 \times 5619 = \sim 506 \text{ misses}\]
Thus, the number of false-sharing misses is $\sim 377$ in the $16B$ example. Similarly, in the $64B$ example the number of false-sharing misses is $\sim 1336$. Naturally, this is assumed that the percentage of true-sharing instances is constant.

### 7.2.3 Limitations to reduced block size model with MESIF protocol

The disadvantage of reducing block size is an increase in total misses and especially compulsory misses. An increase in overall misses probably have a larger impact on performance than the number of false sharing instances. Moreover, the overhead is larger with MESIF with smaller block size. Each cache block requires 5 bits for coherence.

### 7.2.4 An increase of compulsory misses

As the number of processors is increased, the likelihood of a compulsory miss to occur is greater. However, this can be solved by a stricter affinity scheduler. Assigning certain addresses to processors can stall performance, but it will reduce the number of compulsory misses.

### 7.3 Many-processors: Performance evaluation of a single-level cache

A review of single-level cache in a many core environment.

#### 7.3.1 Little increase in memory access

The increased in number of processors does not impact the number of memory accesses significantly, however it increases the bus traffic. Because, more inter-cache communication is needed, the number of misses that does not require memory access increase. This causes the system to stall as latency increases. However, a more centralized approach is more suited for many-core processing e.g. directory-based coherency protocol. Because of this optimal cache configuration may be smaller cache capacity than $64KB$ for the test program. However, it depends on the scheduler.

#### 7.3.2 Relative usage of processors

When the number of processors is increased, the test program’s relative usage of processors decreased. The test program is not optimized for many-core executions and there are few parts of the execution that require more than 3 to 4 processors. The relative usage relates to the difference in efficiency of execution. It is relying on the maximum number of processors in use at one time. This metric may be inaccurate, since the average processor is many fewer
than the maximum number of processors in use. A better metric may be average processors used per cycle in relation with the total number of processors:

\[ u_{\text{relative}} = \frac{n_{\text{avg}}}{n_{\text{total}}} \]

### 7.3.3 Evaluating performance

The test program averages few processors per cycle. For `firefox.exe`, the average was 3.33 for 32, 128 and 512 processors with the shared register model. Because of this the performance is was not substantially better from 32 to 128 to 512 processors. However, the factor \( p \) did not change much with more processors. The speed-up was greater for a many-core computer than a multi-core computer with one shared register. Assuming non-shared registers, the difference between each case was also small. This strengthen the indication that the test programs are not developed to maximize the potential of \( n \)-processors.

### 7.4 Multi-level cache in single-processors: A comparison of inclusion policies

A review on multi-level caches in a single-core environment.

#### 7.4.1 Multi-level versus singe-level caches

The inclusive cache yields the best results for \( L_2 \) caches. This is due to write-hits on all levels of cache regardless of state. The inclusive cache requires a greater difference between \( L_1 \) and \( L_2 \) in capacity and associativity, to avoid waste of space. The ~3.7 times better performance compared to a single-level cache is huge. The exclusive and NINE caches only yielded a ~1.5 times better performance. Moreover, the exclusive cache requires \( L_1 \) and \( L_2 \) to be equal in configurations to achieve better performance. The exclusive cache is better suited as a victim cache. The NINE cache yielded a better performance with \( L_1 \) configurations equal to exclusive cache’s \( L_1 \) configuration and \( L_2 \) configuration equal to the inclusive cache’s \( L_2 \) configuration.
7.5 Multi-level cache in many-processors: Review and limitation

A review on multi-level caches in a single-core environment. Limitations of the simulator.

7.5.1 The inverse

Smaller caches yield better results than a large exclusive cache. The smaller difference between $L_1$ and $L_2$ is desirable. The inclusive cache produced decent results but benefited from a greater difference between caches. Capacity and associativity were key in producing better results.

7.6 Developing a cache simulator

7.6.1 The advantages in method

The simulator is fast and can simulate a program in a matter of seconds. As the maximum number of simulations is 10 per run, enough data can be collected to get representable results. Since results from the simulations are not equal, but consistent and does not deviate much per simulation, there is no need to simulate more than 10 times. The configurability and number of options provides many opportunities in research and the platform can be extended to implement many new additions.

7.6.2 Live to learn

The development of the cache simulator gives a greater insight in actual architecture and there plenty of opportunities to learn new aspects of computers. Even if it is a simplified model and it assumes memory consistency, the main goal is to extract representable cache behaviour data. Through the process of development, several key aspects of computer architecture have been uncovered. Insights in system behaviour, performance metrics, schedulers, x86 ISA and caches is now strengthened. Even if there were at some points some difficulties, the overall experience is great.

7.6.3 Limitations

The simulator can share register to save space, because of memory constraints related to the language and actual memory usage of the simulator. Since, there are certain constraints, A parallel execution can at max equal the number of registers. This may limit the result’s accuracy in terms of CPU usage. An alternative solution can be developed to ensure, increase accuracy for CPU usage. Some of the performance measurements was done in with no regard
for register. However, the results were not impressively better. The coarse-grained parallelism the program exhibits also prevents the results from becoming clear. It is difficult to conclude if the results are representable. Most of the results were found with a shared register. Because of this, the results differ a bit in miss rates. Also, if memory is disabled, the results yields better. However, comparing different configs is not affected because the cache behaves the same way, but with bit more generous results with non-shared register and random data on read from memory.

Another limitation is space and memory. Limiting the number of max simulations to 10 and disabling main memory helps on performance.

7.7 Future works

7.7.1 White box: Compiler solution

Since it would be more useful to use for a programmer or computer scientist, a compiler-ready simulator would give more accurate results and more precise feedback. This could really improve parallelism and help programmers to develop more cache-aware programs. Since the future most likely will include many-core computers, there are many multi-core computer issues that needs to be solved to create better method of developing for parallel computing. Implemented with a compiler, the results can help overcome these issues and especially if the compiler can optimize code to any computer, it can increase performance of any program. The disadvantage with compiler implementation is that it must be implemented per language e.g. Java or C/C++, thus the workload is great. Going from black box to white box simulation also will give the programmer precise feedback.

To do this, writing a compiler with optimization techniques such as loop unrolling, software pipelining and loop tiling will also be advantageous in development of programs, since some optimizing in compile time can negate an effects of optimized pre-compile time code, it can be valuable to receive knowledge about this. Some techniques such as loop splitting can have unwanted to effects, if the code has already implemented this. Developing a compiler with an integrated simulator can really improve performance and parallelism.

7.7.2 Black box: Additions

Implement ARM-64 and IA-64 ISAs can really improve the versatility of the simulator. In example, with smart phones and other battery dependent devices, the optimal cache configuration is not only important in terms of performance, but also power consumption. Since many smart phones include ARM-64 ISA, being able to run applications in the simulator can really help reduce power consumption and increase battery life.

Other additions could be to introduce a unified cache simulation and receive a more precise results and to achieve instruction cache misses. In terms of architecture it can be improved to
be more like NUMA, adding a more complex bus. However, this does not affect the miss ratio. More results such as AMAT and CPI can also be included to present more results. Clock cycles and time consumption approximates can be added. All this to improve the simulator and to evolve it to a computer simulator. More accurate simulation of registers and processors. Even adding a virtual memory and paging. Also extend on file types, improve from .EXE files and add .BIN files.

Implement different cache coherency protocol such as MOESI, MESI, MSI, Berkley and Dragon to achieve results of the optimal protocol. More configuration and even being able to generate a protocol form a state diagram could be possibilities to develop the program further. Generating a cache coherency policy, would imply to develop some type of template and adapt to a state diagram.

Adding a shared cache to further improve results. A shared cache could work as a victim cache exclusive to the caches on the other levels. In addition to more replacement strategies, more adaptative policies perhaps like ARC or just simple single replacements policy such as LFU, MRU, MFU pseudo-LRU etc.

There are many possibilities, perhaps a compiler solution is more useful, but certainly the are many other interesting directions to traverse.
Bibliography


