TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

OLLI RANTALA

DYNAMIC ANALYSIS OF CACHE MEMORY USAGE

Master of Science Thesis

Examiner: prof. Karri Palovuori
Examiner and topic approved by the Faculty Council of Computing and Electrical Engineering on 13th January 2016

ABSTRACT

OLLI RANTALA: Dynamic Analysis of Cache Memory Usage
Tampere University of Technology
Master of Science Thesis, 48 pages, 1 Appendix page
May 2016
Master's Degree Program in Electrical Engineering
Major: Embedded Systems
Examiner: Professor Karri Palovuori

Keywords: ARM, cache, Cachegrind, LTE, optimization

Cache memory in processors is used to store temporary copies of the data and instructions a running program uses. Cache usage increases the program execution performance because reading data from and writing data to the cache is faster than using the system's main memory. Even though cache memories are too small to hold all the data the program needs, due to the temporal and spatial locality properties of programs they improve the memory access times of the system. A cache hit is an event that happens when the data a program needs is found in the processor's cache and a cache miss an event where the data is not found in the cache. When a cache miss happens, the processor has to use the system's main memory and wait for the read or write operation to finish. The processor hardware controls the cache usage but the programmer can improve it with proper design of the source code.

Cachegrind is a tool that is used to perform dynamic cache usage analysis. When a client program is run under Cachegrind's control, the tool counts the amount of cache misses that take place during the execution of the program. Cachegrind pinpoints the exact places in the program's source code where cache misses are happening and the programmer can redesign the code based on the results to decrease the amount of misses. The aim of this thesis is to study if Cachegrind can be used to analyze the cache usage of a Long Term Evolution (LTE) base station software.

When Cachegrind was used on the base station device, the client program crashed at startup. This happened because Cachegrind causes changes in the execution of the client. As a result, the analysis was conducted using a host environment that runs the same software on a normal computer. Another tool called Callgrind was also used. Callgrind extends the functionality of Cachegrind by offering many additional features that were helpful in the analysis. The results were used to make changes in the source code and several thousand cache misses were removed from the software.

TIIVISTELMÄ

OLLI RANTALA: Välimuistin käytön dynaaminen analyysi
Tampereen teknillinen yliopisto
Diplomityö, 48 sivua, 1 liitesivu
Toukokuu 2016
Sähkötekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Sulautetut järjestelmät
Tarkastaja: Professori Karri Palovuori

Avainsanat: ARM, välimuisti, Cachegrind, LTE, optimisointi

Prosessorien välimuistiin tallennetaan tilapäisiä kopioita suoritettavan ohjelman käskyistä ja datasta. Välimuistin käyttö parantaa ohjelman suorituskykyä, koska datan lukeminen ja kirjoittaminen välimuistiin on nopeampaa kuin järjestelmän päämuistin käyttö. Vaikka välimuistit ovat liian pieniä kaiken ohjelman tarvitseman datan tallentamiseen, ohjelmien ajallisen ja tilallisen paikallisuuden vuoksi ne parantavat järjestelmän muistinkäsittelyaikaa. Välimuistiosuma on tapahtuma, jossa ohjelman tarvitsema data löytyy välimuistista ja välimuistihuti tapahtuma, jossa data ei löydy välimuistista. Kun huti tapahtuu, prosessorin täytyy käyttää järjestelmän päämuistia ja odottaa luku- tai kirjoitusoperaation valmistumista. Prosessorin laitteisto ohjaa välimuistin käyttöä, mutta ohjelmoija voi parantaa sitä asianmukaisella lähdekoodin suunnittelulla.

Cachegrind on työkalu, jota käytetään välimuistin käytön dynaamiseen analysoimiseen. Kun asiakasohjelma suoritetaan Cachegrindin hallinnassa, työkalu laskee ohjelman suorituksen aikana tapahtuvien välimuistihutien määrän. Cachegrind osoittaa ohjelman lähdekoodista kohdat, joissa hudit tapahtuvat ja ohjelmoija voi näiden tulosten perusteella suunnitella koodin uudelleen hutien määrän vähentämiseksi. Tämän diplomityön tavoite on tutkia voiko Cachegrindia käyttää Long Term Evolution (LTE) tukiaseman ohjelmiston välimuistin käytön analysoimiseen.

Kun Cachegrindia käytettiin tukiasemalaitteella, asiakasohjelma kaatui käynnistettäessä. Tämä tapahtui, koska Cachegrind aiheuttaa muutoksia asiakasohjelman suorituksessa. Tästä syystä analyysi suoritettiin käyttämällä ympäristöä, joka ajaa samaa ohjelmistoa normaalilla tietokoneella. Myös toista työkalua nimeltä Callgrind käytettiin analyysissa. Callgrind laajentaa Cachegrindin toimintaa tarjoamalla muutamia lisäominaisuuksia, jotka olivat hyödyllisiä analyysissa. Saatuja tuloksia käytettiin muutoksien tekemiseen lähdekoodiin ja useita tuhansia huteja poistettiin ohjelmistosta.

PREFACE

This Master’s thesis work was done for Nokia Oyj to study the usage of Cachegrind in LTE layer 2 software optimization. I would like to thank my co-workers in LTE UP Feature Team 9 for their help and guidance during the work. A special thanks to my supervisor Petri Laine, Ari-Pekka Taskila and Dr. Jorma Taramaa for their input and instructions.

Oulu, 17.5.2016

Olli Rantala

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

3GPP	Third Generation Partnership Project
AL	Access Line
ARM	Advanced RISC Machine
BL	Bit Line
bps	Bits per second
C	Capacitor
CI	Continuous Integration
CISC	Complex Instruction Set Computing
CPU	Central Processing Unit
D1mr	First level cache data read miss
D1mw	First level cache data write miss
DL	Data Line
DLdmr	Last level cache data read miss with write-back
DLdmw	Last level cache data write miss with write-back
DLmr	Last level cache data read miss
DLmw	Last level cache data write miss
Dr	Data read
DSP	Digital Signal Processor
Dw	Data write
eNB	Evolved Node B
I1mr	First level cache instruction fetch miss
IC	Integrated Circuit
ILdmr	Last level cache instruction fetch miss with write-back
ILmr	Last level cache instruction fetch miss
IP	Internet Packet
Ir	Instruction read
L1	First level cache
L2	Second level cache
L3	Third level cache
LRU	Least Recently Used
LTE	Long Term Evolution
M	Transistor
MME	Mobility Management Entity
OSI model	Open Systems Interconnection model
P-GW	Packet Data Network Gateway
PID	Program Identifier
RISC	Reduced Instruction Set Computing
S-GW	Serving Gateway
SUT	System Under Test
TLB	Translation Lookaside Buffer
V_{dd}	Power Supply Voltage
VoIP	Voice over IP
VoLTE	Voice over LTE
WL	Word Access Line

C	Capacitance
e	Euler’s number
h	Hit rate
h_1	L1 hit rate
h_2	L2 hit rate
R	Resistance
Q_0	Initial charge
t	Time
t_{av}	Average memory access time
t_{cache}	Cache memory access time
t_{L1}	L1 access time
t_{L2}	L2 access time
t_{main}	Main memory access time

1. INTRODUCTION

In the recent years there has been an increasing demand for wireless high speed mobile broadband networks. People are using their smart phones to transfer voice, video and data over different kinds of radio access technologies. Because of this increasing consumer demand, higher capacity and greater quality wireless networks needed to be developed. One of these mobile communication technologies is called Long Term Evolution (LTE). [1, pp. xiii-13; 2, pp. xv-xviii]

1.1 Aim and Scope

LTE has been standardized by the Third Generation Partnership Project (3GPP) but the technical details of the devices that are used to create the network are up to the telecommunications equipment manufacturers. These devices are embedded systems whose performance can be optimized to increase the overall performance of the network. By developing faster and better optimized equipment one manufacturer, such as Nokia, can gain a marketing advantage over its competitors. [2 pp. xv-119]

Software developers can use profilers and error checkers to improve the quality of their software [3]. The purpose of this thesis is to study if one such program and its toolset can be used to help optimize the source code of an LTE base station device. Specifically a tool called Cachegrind is used to analyze the cache memory usage of the device dynamically. Secondary goal is to create an easy-to-use method of executing the cache usage analysis on the device in an existing build and test environment so that continuous testing can be done.

As processor have become faster and faster the performance of programs is often limited by memory access. The processor has to read instructions and data from and write data to a main memory but because memory access is slow the processor has to wait for these actions to complete. Processor cache is used to combat this problem. Reading from and writing to cache is much faster than normal memory access. However, optimal cache usage is not achieved automatically but requires some help from the programmer. Also, in a large software project it is not easy to find out which parts of the source code are causing bad cache usage and a tool such as Cachegrind can be used to ease the task. [4] Therefore cache usage analysis was chosen as the topic of this thesis.

1.2 Structure of the Thesis

In Chapter 2 different processor architectures are discussed. This chapter also explains how cache memory usage can affect the performance of a device and what kind of optimizations can be done. Chapter 3 introduces some mobile network basics and terminology. It also elaborates on the LTE software and hardware technicalities on a common level without going into company specific design solutions. Chapter 4 presents the tools used in the test environment. This includes the test and continuous integration frameworks. Chapter 5 describes Cachegrind usage and some alternatives to Cachegrind are also briefly examined. In Chapter 6 the actual implementation solutions are listed. The chapter also includes the results of the analysis and some methods that were used to improve the cache usage. Chapter 7 discusses about the accuracy of the results. It also examines if further improvements in the analysis might be possible. Chapter 8 sums up the thesis by discussing about the conclusions that were made.

2. PROCESSORS

In this chapter the purpose and need for cache memory are explained. First, the general design of processors and memory types are introduced and then the details of how cache memory works is talked about. In the last part of the chapter some optimization techniques that can be used to improve cache usage are listed.

Processor is an electronic circuit and it is the part of a system, a computer or an embedded device, that executes the instructions of a program to perform a specified task. It fetches the instructions and data that are needed to execute the calculations from the memory of the system. The processor also controls the input devices, output devices and other components of the system. The term central processing unit (CPU) is often used to refer to a processor. A processor runs at some clock frequency and executes the program instructions sequentially. Executing one instruction usually takes one clock cycle. [5; 6]

2.1 History

In the early days of computers the performances of the different parts were much closer each other than today. For example, the main memory and network interfaces were able to provide data at the same speed as the processor. This situation changed in the early 1990s when the hardware developers concentrated on the optimization of individual components after the basic structure of computers was stabilized. Especially mass storage and memory subsystems were falling behind of the other components in their performance. This is not because faster memory could not be built but because using fast memory is not economical. Main memory that is as fast as current CPUs is much more expensive than memory that is normally used today. Also, implementing a large memory system with very low latency is difficult. The memory would have to be near the CPU to negate any signal delays but other devices also use the same memory. A large memory system has a big footprint and fitting all that memory close to the CPU is near impossible. Besides the actual memory modules, some room is also required for the signal paths and memory controllers. [4] High performance CPUs need cooling to keep them within safe operating temperatures. All the extra components near the CPU would also cause thermal issues. [7]

Most modern CPUs are microprocessors, which means that they are contained on a single integrated circuit (IC). If the IC contains memory, peripheral interfaces and other components in addition to the CPU it is called a microcontroller or a system on a chip. Multi-core processors contain two or more CPUs called cores on a single IC. [5]

2.2 Memory Types

Digital systems, such as processors, operate using logical states. All data is presented by logical states called bits, ones and zeroes, that the processor can store, manipulate and use in calculations. These bits are stored in memory cells. The memory cells have primarily two different kind of implementations: dynamic and static. Dynamic memory is cheap but slow and static memory is fast but expensive. [4; 8]

The basic structure of a dynamic memory cell is shown in Figure 1. It consists of one transistor M and a capacitor C. The capacitor holds the state of the cell and the transistor is used as a guard to access the state. The access line AL is raised to read the state of the cell. This either causes current to flow to the data line DL or not, depending on the charge of the capacitor. Writing to the cell is done by setting DL to the value that is written and raising AL long enough for the capacitor to charge or drain. The use of a capacitor causes a few problems. First of all, every read action causes the capacitor to lose charge. Therefore, every read action has to be followed by an operation to recharge the capacitor. This is done automatically but requires additional energy and time. Secondly, the capacitor slowly leaks charge even when it is not accessed so its state has to be constantly refreshed. Third problem is that charging and draining the capacitor does not happen instantaneously. These operations are characterized by equations 1 and 2. The resistance R of the circuit and capacitance C of the capacitor affect the time the operations require. An empty capacitor is fully charged after a time of $5RC$ seconds. The same value applies to discharging as well. The simple structure of the dynamic cell allows manufacturing cheap memory that can store large amounts of data. [4]

$$Q_{Charge}(t) = Q_0(1 - e^{-t/RC}) \tag{1}$$

$$Q_{Discharge}(t) = Q_0 e^{-t/RC} \tag{2}$$

$Q_0 = initial\ charge$

$e \approx 2.718\ (Euler's\ number)$

$t = time$

$R = resistance$

$C = capacitance$

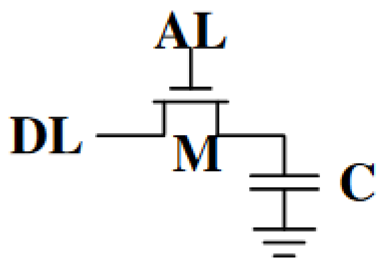

Figure 1. *Dynamic memory cell [4].*

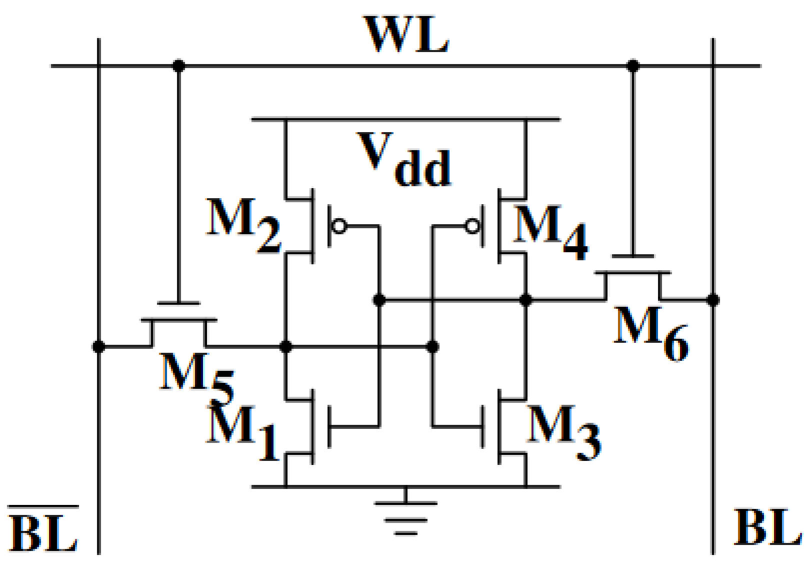

Figure 2. *Static memory cell [4].*

One possible static memory cell structure is shown in Figure 2 but different implementations are also used. The basis of the cell is made of the four transistors M_1 to M_4 that form two cross-coupled inverters. The inverters have two stable states that represent the logical zero and one respectively. The state is stable as long as power on the power supply line V_{dd} is available. To read the state of the cell the word access line WL is raised. After this the state can be immediately read from the bit lines BL and $\overline{BL}$. Writing is done by setting the bit lines to the desired state first and then raising WL. A static memory cell has none of the problems caused by the capacitor in a dynamic cell. However, each cell requires six transistors and a power supply line. This structure makes manufacturing static memory more difficult and more expensive than dynamic memory. [4]

2.3 Architectures

CPUs have two different design architectures that are called Harvard and Von-Neumann. Von-Neumann architecture CPUs share a single common bus for fetching both instructions and data. Therefore program instructions and data are stored in a common main memory. Harvard architecture, on the other hand, has separate busses for instructions and data which allows transfers to be performed simultaneously on both busses. The data memory can be read and written while the instruction memory is accessed. The separate busses also allow one instruction to be executed while another one is fetched (pre-fetching). Theoretically the pre-fetch allows faster execution than Von-Neumann architecture but it also requires more complex IC design. [9; 10, pp. 21-23]

CPUs can also be classified by the instruction set they use. Currently the two most common instruction set architectures are ARM (Advanced Reduced Instruction Set Computing Machine or Advanced RISC Machine) and x86 which uses the Complex Instruction Set Computing (CISC) design. The difference between RISC and CISC is the complexity of the instructions they offer. RISC instructions are smaller single operations the CPU can perform while CISC instructions can require several smaller steps. Therefore RISC instructions are typically executed in one CPU cycle while CISC instructions can take several cycles to complete. ARM processors consume less power than x86 processors and for this reason they are more commonly used in battery powered devices such as mobile phones. x86 CPUs are used mainly in desktop and laptop computers where more processing power is required. Both architectures have 32-bit and 64-bit variants. [11-13]

Special purpose processors designed specifically for the computational needs of embedded audio, video and communications applications are called Digital Signal Processors (DSPs). They implement algorithms that are used in signal processing in hardware. This architecture makes signal processing applications faster than on a general purpose CPU where the speed of execution depends primarily on the clock speed of the CPU. [10, pp. 21-23]

2.4 Cache

As mentioned earlier fast memory is expensive. However, small amounts of fast static memory can be used without a significant cost increase. A processor usually has a small amount of registers on the IC which can be accessed without delay but cannot hold much data. Cache memory is static memory that is used to store temporary copies of the data in the dynamic main memory. Its size is significantly smaller than the main memory size so the entire program cannot usually be loaded in cache. However, because program code has spatial and temporal locality the small cache memory can be used to speed up the overall memory access of the program. Spatial locality means that data segments near each other in the memory are referenced close together in time. Temporal locality, on the other hand, means that recently accessed data segments are accessed again in a short period of time. If these data segments are loaded in the cache the CPU can access them faster. Cache memory usage and administration is handled automatically by the CPU hardware. [3; 4; 14]

Modern processors can have multiple levels of cache. A small cache called level 1 (L1) is usually placed on the same IC near the CPU to ensure best possible performance. Furthermore, the L1 cache is often split into two parts. One half of the cache is dedicated for storing instructions only and the other half is used for data. This separation is done on the Harvard CPU architecture so that both busses can be fed at the same time, which speeds up the execution. The L1 cache contains copies of the data in a second level cache (L2) which is larger than L1. In this case the L2 cache is inclusive. Some processors have exclusive caches, which means each cache stores unique entries. L2 is slower than L1 but still much faster than the external main memory. Also, L2 is not split in separate parts for data and instructions like L1. Additional cache levels work the same way. On multi-core CPUs each core can have its own low level caches while the higher level caches are shared between all the cores. [3; 14; 15] The structure of a three level cache is shown in Figure 3. The first two cache levels are shown to be on the same IC with the CPU and the third level is off chip.

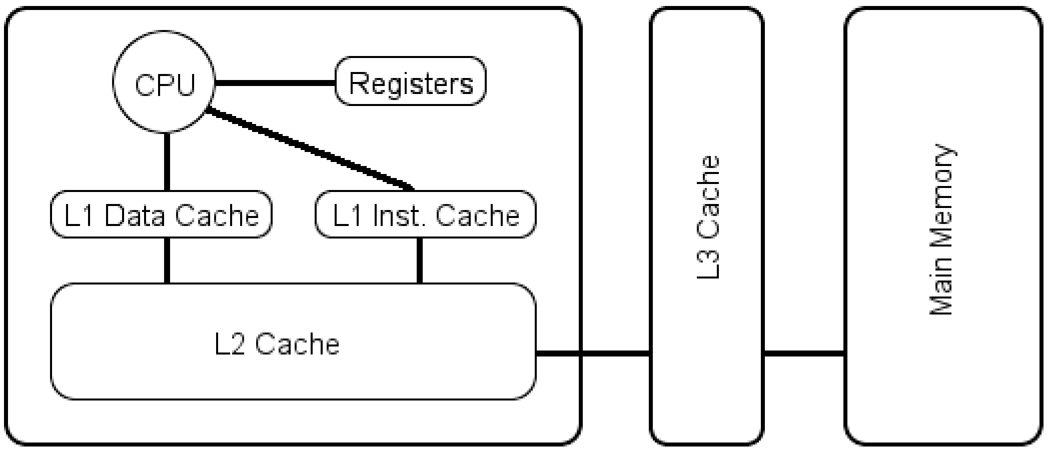

***Figure 3**. Processor with three levels of cache.*

The data within a cache is stored in fixed size blocks called cache lines. A cache line holds a copy of a continuous block of data in the main memory. The data is transferred from the main memory to the cache in blocks which are smaller than the cache line size. A cache hit happens when the processor requests data and the data is found in a cache line. If the data is not found in a cache line it has to be retrieved from a higher level cache, the main memory or from a mass storage device. This is called a cache miss. [3; 4; 14]

When a block of data is copied in a cache line an old value is often overwritten. Therefore an important question is which old value should be replaced to maximize performance. The simplest implementation is direct-mapped cache where a memory block can be placed in exactly one cache line. However, for most programs this solution has a disadvantage where some cache lines are used heavily while some others are hardly used at all or remain empty. Another implementation is fully associative cache where a memory block can be placed in any cache line. This approach is usable only for small cache sizes. A combination of these two is called set associative cache where a memory block can be placed in a set of cache lines. For example, if a cache is 24-way set associative it means there are 24 cache lines where a single memory block can be copied. Most modern CPUs have set-associative caches. [3; 4]

The selection within a set has its own logic called replacement policy which is used when all the cache lines of a set are already in use when a new memory block is fetched and needs to be inserted in the cache. There are multiple different replacement policies, such as least recently used (LRU), first in-first out and random. Writing in the cache lines also requires a policy to choose when the main memory is updated. In write-through policy when a cache line is written the CPU immediately writes the entry in the main memory. This implementation is simple but not very fast. A more efficient policy is called write-back which only marks the written cache lines as dirty. When the cache entries are dropped from the cache at some point in the future the CPU writes the dirty lines in the main memory. The write-back policy causes a problem in multi-core processors where different cores could see different memory content because of their separate caches. This problem can be solved with a coherency protocol but the details are not important in the context of this work. The write-though and write-back methods are commonly called write hit policies. Furthermore, the write policies are divided in two categories based on whether a memory block is stored in the cache after a write operation miss. These methods are commonly called write miss policies. In a no-allocate memory system data is written directly to the main memory and a subsequent read of the same data would cause a read miss. In write-allocate system the written data is stored to the cache. A few other write policies also exist. [4; 16]

2.4.1 Impact on Performance

According to the Valgrind homepage an L1 miss will typically cost about 10 cycles and a last level cache miss can cost about 200 cycles [15]. It is easy to see that if several of these misses happen during the execution of a large program then the processor spends a lot of time doing nothing but waiting for data.

A simple equation can be used to calculate the average memory access time. If we let h be the cache hit rate then $1-h$ is the cache miss rate. The average memory access time for one-level cache system can then be written as in equation 3 where t_{cache} is the cache memory access time and t_{main} is the main memory access time.

$$t_{av} = ht_{cache} + (1 - h)t_{main} \quad (3)$$

If we take a two-level cache system and let h_1 be the first level hit rate and h_2 the second level hit rate the same access time can be calculated by equation 4. The two cache levels have different access times which are presented in the equation with the variables t_{L1} and t_{L2}.

$$t_{av} = h_1 t_{L1} + (h_2 - h_1)t_{L2} + (1 - h_2)t_{main} \quad (4)$$

Given that the main memory access time is around 100 nanoseconds and cache access time only a few nanoseconds, the difference between the best-case and worst-case scenarios is substantial. [14]

Instruction cache is less important than data cache for a few reasons. First, the amount of code needed depends on the complexity of the problem the program is made to solve. Secondly, compilers are good at generating optimized code. Thirdly, program flow is much more predictable than data access patterns and modern CPUs are good at detecting patterns which helps with instruction pre-fetching. In addition, code always has good spatial and temporal locality. [15]

2.4.2 Optimization Strategies

Cache misses are divided in three different types based on why the miss happened. A compulsory miss occurs when data is first read and it is therefore difficult to avoid. A capacity miss is caused by a too large working set, which means the cache is not big enough to store all the needed data blocks. In this case the data blocks are evicted from the cache to make room for new blocks but later the evicted blocks need to be retrieved again. The third type is a conflict miss which happens when two data blocks are mapped to the same cache line. This type can be further divided in self-interference and cross-interference. When an element of an array causes a conflict miss with another element of the same array it is called self-interference. In cross-interference the elements are from different arrays. [3; 14]

There are a few common practices for optimizing the program code for data access. These methods transform the code mainly to improve spatial or temporal locality without changing the numerical results of the computations. One of the simplest data access optimization techniques is loop interchange. This method changes the order of nested loops in array-based computations. Consider that we have a matrix operation as shown in Figure 4 and that the a-array is stored in memory in row-major order (two array elements are stored adjacent in memory if their second indices are consecutive numbers). However, in the left part of the figure the code accesses the array in a column-wise order. Because of this the preloaded data in the cache line (represented by the grey colour in the access patterns in the figure) is not reused if the array is too big to fit entirely in the cache. The situation changes when the loops are interchanged. The same effect can also be achieved by transposing the array. [3]

```
double sum;
double a[n][n];
// Original loop nest:
for (j = 0; j < n; ++j) {
   for (i = 0; i < n; ++i) {
      sum += a[i][j];
   }
}
```

```
double sum;
double a[n][n];
// Interchanged loop nest:
for (i = 0; i < n; ++i) {
   for (j = 0; j < n; ++j) {
      sum += a[i][j];
   }
}
```

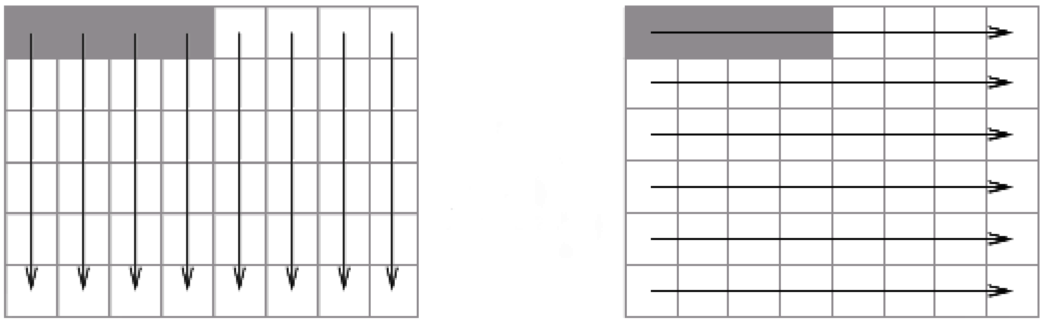

Figure 4. *Loop interchange code and access patterns [3].*

Another transformation is called loop fusion where adjacent loops that have the same iteration space traversal are combined into a single loop. The fused loop contains more instructions in its body and therefore increases instruction level parallelism. Also, since only one loop is executed this method decreases the total loop overhead. Loop fusion also improves data locality. [3; 17]

Loop blocking is a transformation that adds more loops to a loop nest. This method is illustrated in Figure 5 and Figure 6. It is used to improve data locality by enhancing the reuse of data in the cache. The memory domain of the problem is split into smaller chunks that are small enough to fit entirely in the cache. Consider the original case without loop blocking in Figure 5. If the size of one row in the A-array is big enough then each access to the B-array will always generate a cache miss. This can be avoided if the loop is blocked with respect to the cache size. The size parameter is chosen so that the combined size of the blocked chunks of the arrays is smaller than the cache size as shown in Figure 6. [3; 18]

```
double A[MAX][MAX];
double B[MAX][MAX];
// Original code:
for (i = 0; i < MAX; ++i) {
   for (j = 0; j < MAX; ++j) {
      A[i][j] = A[i][j] + B[i][j];
   }
}
```

```
double A[MAX][MAX];
double B[MAX][MAX];
// Loop blocked code:
for (i = 0; i < MAX; i += size) {
   for (j = 0; j < MAX; j += size) {
      for (ii = i; ii < i + size; ++ii) {
         for (jj = j; jj < j + size; ++jj) {
            A[ii][jj] = A[ii][jj] + B[ii][jj];
         }
      }
   }
}
```

Figure 5. *Loop blocking code sample [18].*

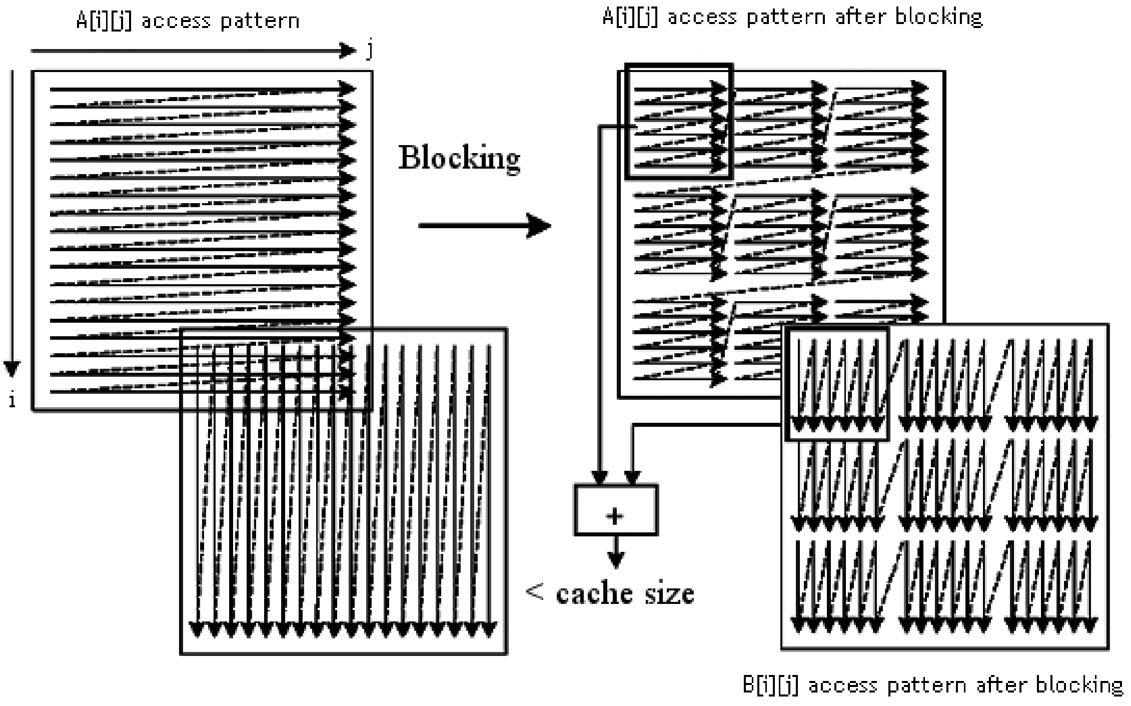
A[i][j] access pattern
j
i
Blocking
A[i][j] access pattern after blocking
+
< cache size
B[i][j] access pattern after blocking

Figure 6. *Loop blocking access pattern [18].*

Software pre-fetching is a technique where a special instruction is given to the processor to tell it to load data from the main memory to cache before the data is actually needed. If the data is requested early enough the penalty of compulsory and capacity misses not covered by loop transformations can be avoided. Pre-fetching instructions can be added manually by the programmer and automatically by the compiler. Because the pre-fetching instructions have to be executed by the CPU, this causes some overhead in the execution time. Some processors implement hardware pre-fetching which means the processors has special hardware that monitors the memory access patterns and tries to predict the data needed in the future. Hardware pre-fetching does not require any changes from the programmer although some design decisions affect its performance. Pre-fetching works only if the data access pattern is predicted correctly. If the pre-fetched data replaces data that is still needed, the cache miss rates will increase. Unnecessary pre-fetching should be avoided also because it uses the memory bandwidth. [3; 4]

Data layout optimizations are intended to improve the spatial locality of a code. They modify how data structures and variables are arranged in memory. Array padding is a data layout optimization method that adds unused variables in the code to avoid self or cross-interference. Consider the example case in Figure 7. The two arrays are accessed in an alternating manner and if they are mapped to the same cache lines a high number of cache conflict misses are introduced. When the first element of the a-array is read, the element and possibly the subsequent array elements are loaded in a cache line. If the first element of the b-array is mapped to the same cache line, the a-array elements will be replaced and no cache reuse can happen. The padding modifies the offset of the second array so that both arrays are mapped to different cache lines. The size of the padding depends on the cache size, the mapping scheme of the cache, the cache line size, the cache associativity and the data access pattern of the code. Typical padding sizes are multiples of the cache line size. The disadvantage of array padding is that extra memory is needed for the padding. [3]

```
//Original code:
double a[1024];
double b[1024];
for (i = 0; i < 1024; ++i) {
      sum += a[i] * b[i];
}
```

```
//After applying inter-array padding
double a[1024];
double pad[x];
double b[1024];
for (i = 0; i < 1024; ++i) {
      sum += a[i] * b[i];
}
```

Figure 7. *Array padding.*

Array merging combines arrays that are often accessed together into a single data structure. This can reduce the number of cross-interference misses for scenarios with large arrays and alternating access patterns. [3] This layout transformation is depicted in Figure 8. Originally the Nth elements of the arrays are always separated by the size of the arrays in memory. After merging they are contiguous in memory which means they have better spatial locality.

```
//Original data structure:
double a[1024];
double b[1024];
```

```
//Array merging using structure:
struct S {
  double a;
  double b;
} ab[1024];
```

Figure 8. *Array merging.*

Data alignment also affects cache usage. When a CPU reads from and writes to memory it does it in chunks of certain size. Compilers align data to memory addresses that are multiples of this size because it improves the memory access performance. Every basic data type has an alignment that is mandated by the processor architecture. For structures that typically contain several different data types, the compiler tries to maintain alignment by inserting unused memory between the elements. The extra memory can cause a structure that seems to fit in one cache line to actually require two lines. The order of the elements inside a structure affects the alignment. This is illustrated in Figure 9. If we assume that the word size is 8, integer size 4 and long integer size 8 bytes the data structure size seems to be 64 bytes. This is also the cache line size of the ARM processors introduced earlier, which means this structure should fit in a single cache line. However, the compiler will add 4 bytes of extra memory after both of the integers (a and b) to meet the alignment requirements and as a result the structure size increases to 72 bytes. By reordering the structure as shown on the right side of the figure the size is reduced to 64 bytes. As a general rule of thumb, proper reordering can be done by arranging the members in a decreasing alignment requirement order. [4; 19; 20]

```
//Original data structure:
struct foo {
  int a;
  long fill[7];
  int b;
};
```

```
//Reordered structure:
struct foo {
  long fill[7];
  int a;
  int b;
};
```

Figure 9. *Structure reordering.*

The alignment of structures can also be forced with some compilers. The gcc-compiler, for example, supports defining special attributes for variables. The aligned-attribute can be used to define the minimum alignment of a variable or structure. By combining small objects inside a structure and forcing it to be allocated at the beginning of a cache line by setting the alignment equal to the cache line size, it is possible to guarantee that each object is loaded into the cache as soon as any one of them is accessed. If these objects are often used together this improves the spatial cache locality and the performance of the program. [4; 19; 21]

Data structures usually contain elements that logically belong together. However, sometimes it might be possible to improve cache usage by separating these elements in different structures. Consider a case where a structure is too big to fit in a single cache line but contains elements some of which are used often and others that are not. In this case separating the often and rarely used parts improves cache locality at the expense of increasing the program complexity. This method is called hot/cold splitting. [4; 19; 22]

Alignment is not the only thing to consider when ordering the elements inside a structure. As mentioned earlier, when a cache miss happens the memory blocks that contain the data are retrieved from the main memory in small blocks to the cache. If the data that was requested is not in the first blocks that are retrieved, this causes more delay. For this reason the element that is most likely to be referenced first should also be the first element inside the structure. Furthermore, when accessing the elements, and the order of the access is not dictated by the situation, they should be used in the same order that they are defined in the structure.

The standard libraries that are part of a programming language are often highly optimized. Using the functions the language offers instead of writing your own implementations usually results in better performance. For example, using the memset-function to set the values of a big C-language array instead of setting the values one-by-one in a loop results in better cache usage. [4]

For instruction cache usage compiler optimizations are often enough. Modern compilers can do several different kinds of code transformations that improve performance. This includes methods such as loop transformations, memory access transformations, partial evaluation and redundancy elimination. However, many of these optimizations are not related to improving cache usage. The code size obviously affects the instruction cache performance. Some compilers, such as gcc, support enabling code size optimizations. With gcc this is done with the –Os compiler option. This can improve performance especially in cases where other optimizations are not possible. [4; 23]

3. MOBILE NETWORKS

This chapter provides a brief overview of mobile phone networks. Especially the LTE network is talked about. This includes shortly describing the functionality and hardware of the system that is under analysis in this work. The specifics are intentionally left short because they are company confidential information.

A mobile network is a telecommunication network where the last link, connection between the user equipment and a base station, is wireless. The base station is connected to the core network which is used to route the data. The most common mobile network today is the mobile phone network. Mobile phone networks are distributed over large areas of land by using cells. Each cell is served by one or more base stations. [1, pp. xiii-13; 24] This structure is depicted in Figure 10 below.

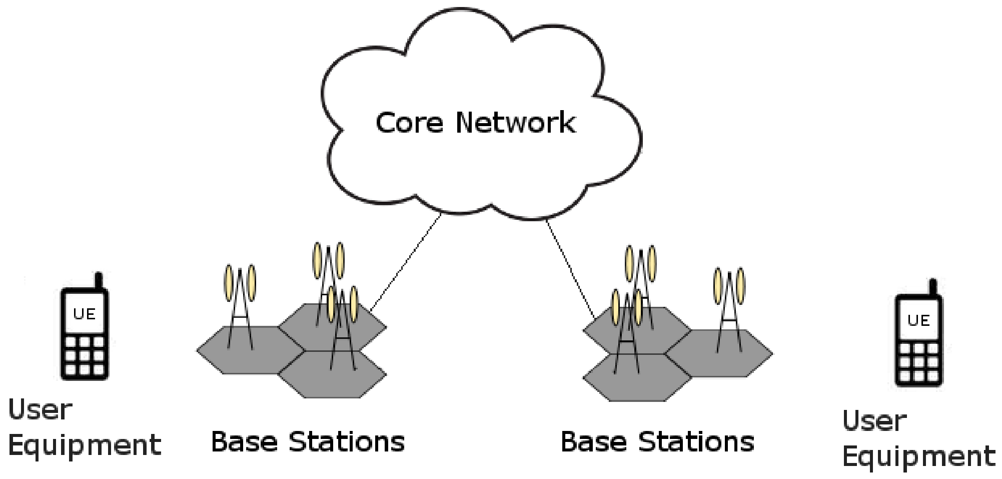

Figure 10. *Mobile phone network structure.*

3.1 Background

Mobile communication technologies are divided into generations, the first one being the analog mobile radio systems of the 1980s. This technology is usually referred to as 1G. In the beginning of the 1990s 1G was followed by 2G which introduced the first digital mobile systems and later by 3G which included the first systems capable of handling broadband data. LTE is often called 4G although many consider that the first release of LTE (release 8 in 2008) was 3.9G and the true step to 4G was taken in 2011 with release 10, which is also referred to as LTE-Advanced.

The peak data transfer rate has gone from the few kbps (bits per second) of 2G to several Mbps of 3G and finally close to 1 Gbps speeds of 4G. Besides the data rate other driving forces behind the development of mobile networks are latency and capacity. Interactive services such as gaming and web browsing require low latency. Capacity shortage causes degradation of the quality of service for the end-users. These three parameters have influenced the development of LTE. [1, pp. xiii-13]

3.2 LTE

In simple terms, the LTE architecture consists of only two nodes in the user-plane: a base station and a core network gateway as shown in Figure 10. Figure 11 details this architecture a bit more. First of all, LTE is an Internet Packet (IP) based network. Voice services such as phone calls are made using Voice Over LTE (VoLTE) which is an implementation of Voice Over IP (VoIP) in 3GPP LTE standards. The core network, Evolved packet core, treats voice just as other IP-based applications although with strict performance and operational requirements. All mobility management functions are part of the Mobility Management Entity (MME). The Serving Gateway (S-GW) routes and forwards user data packets and the Packet Data Network Gateway (P-GW) provides connectivity from the user equipment to external packet data networks like the Internet. The access network consists essentially of only the base station, the Evolved Node B (eNB), which provides connections to the user equipment. ENBs typically reside near the actual radio antennas and they are distributed throughout the entire networks coverage area. Each of the network elements is connected by standardized logical interfaces in order to allow multi-vendor interoperability. This means network operators can purchase different network elements from different manufacturers. [2, pp. xv-119; 25]

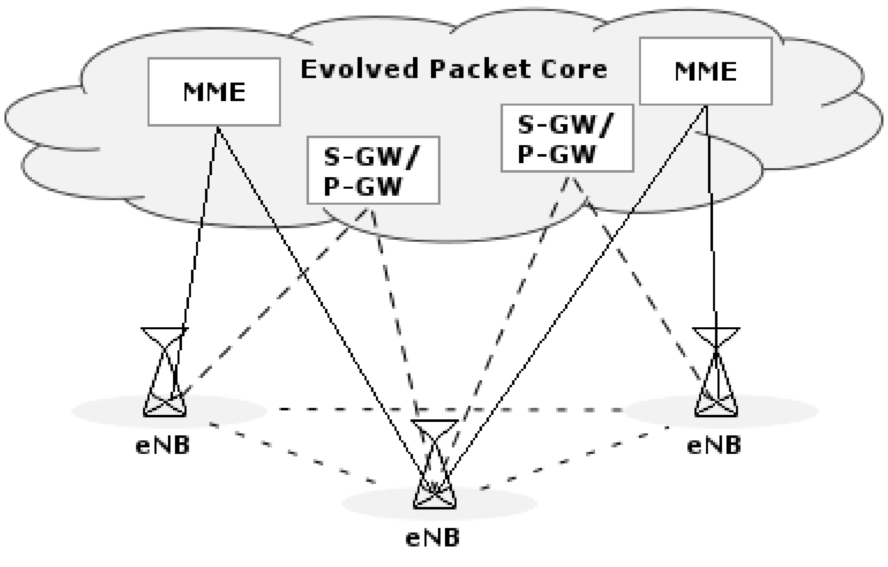

Figure 11. *LTE architecture.*

3.2.1 Hardware

The eNB device that is under analysis in this work has an ARM Cortex-A15 processor. It is a 32-bit processor with ARMv7-A instruction set architecture and has four CPU cores. Each core has 32 kilobytes of separate 2-way set-associative data and instruction L1 cache memory. The L1 cache uses LRU replacement policy.

The 16-way set-associative four-megabyte inclusive L2 cache is shared between all cores. Actually, the L2 cache has a configurable size but in this application the full size is used. It has a hardware pre-fetcher that can be disabled by the user. The processor also supports software pre-fetching. The L2 uses random cache replacement policy and write-back hit policy. Both caches have a line size of 64 bytes and can use different write miss policies. [16]

3.2.2 Software under Analysis

The software implements the LTE layer 2 functionality of the eNB. The term layer 2 refers to the Open Systems Interconnection model’s (OSI model) data link layer that, for example, defines the protocol to establish and terminate a connection between two physically connected devices. The software runs in a Linux operating system user space. One core of the CPU is reserved for operating system processes, the others are polling events at a 100 % CPU load.

The eNB is in control of all radio related functions in its own coverage area. It acts as a layer 2 bridge between the user equipment and the core network by being the termination point of all the radio protocols towards the user equipment and by relaying data between the radio connection and the IP based core network. For this reason, the eNB software performs ciphering and deciphering of the data and IP header compression and decompression to avoid sending the same header data repeatedly. The software is also responsible for many control plane functions such as allocating resources based on requests, prioritizing and scheduling traffic according to required quality of service and monitoring resource usage. In addition, it controls and analyzes the radio signal level measurements carried out by the user equipment, makes similar measurements itself and based on them makes decisions, for example, to handover a user to another base station. When a new user requests connection to the network the eNB is responsible for routing this request to the MME that previously served that user or selecting a new MME if the previous MME is absent. A single eNB can be connected to multiple MMEs and S-GWs but a user can be served by only one MME and S-GW at a time. For this reason the eNB software also has to keep track of the associations between users, MMEs and S-GWs.

The IP based flat architecture of LTE places a lot of pressure on the eNB. On a functional level the software has to follow the 3GPP standards but performance-wise it has to support thousands of users, each with varying data rates. The high number of users per eNB, the high data throughput and the low latency require the software to be optimal and well performing. [25]

4. TEST ENVIRONMENT

This chapter introduces the tools that are used in the automatic testing of the eNB software. The most important of these is Robot Framework that is used to test the functionality of the eNB software. The cache usage analysis was performed by running the framework test cases with the cache analysis tools.

4.1 Robot Framework

Robot Framework is a test automation framework for acceptance testing. It uses the keyword-driven testing approach and has a simple test data syntax. The framework is operating system and application independent. Its core is implemented in the Python programming language and its testing capabilities can be extended by test libraries written in Python or Java. The framework is open source software as are most of the libraries and tools it can utilize.

The modularity of the framework is depicted in Figure 12. It uses keywords defined in the test libraries to interact with the system under test (SUT). The framework itself does not have to know anything about how the SUT operates. After a test run the framework produces a report and a log that provide an extensive look into what the system did, which keywords failed (if any) and why they failed.

Figure 13 represents a simple Robot Framework test suite with two test cases. The suite uses a syntax where different reserved words, variables, arguments and other definitions are separated by two or more space characters. The suite loads keywords and variables from other files that are defined under the settings section at the top of the suite. It also defines the setup and teardown keywords that are executed for each test case. When the suite is run the framework executes all the test cases. Running a single test case from a suite is also possible. [26; 27]

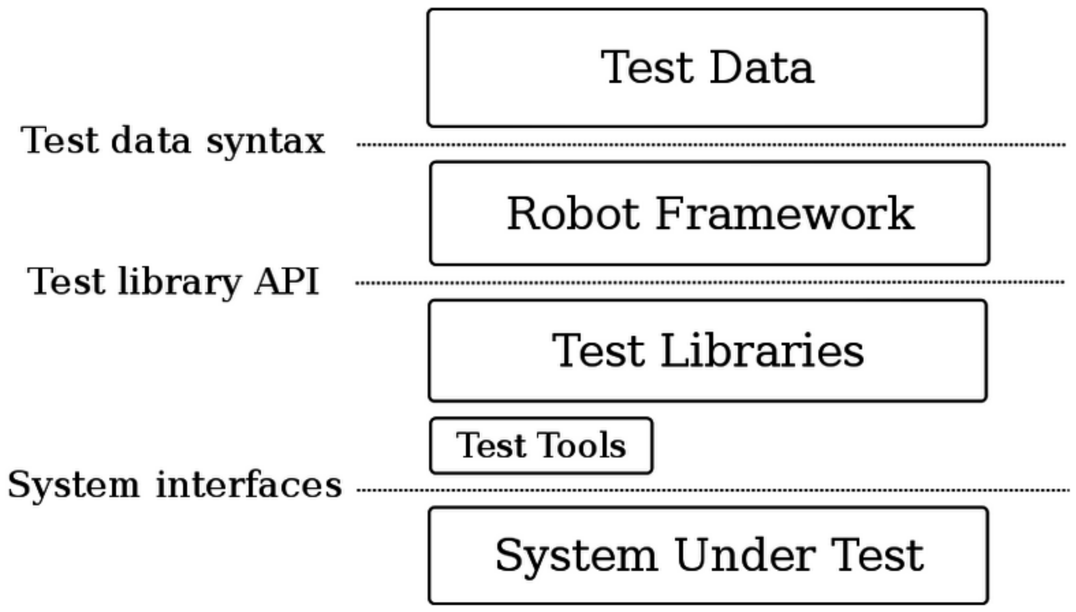

Figure 12. *Robot Framework modularity [26].*

```
*** Settings ***
Resource                ../test_setup_and_teardown.txt
Resource                 ../basic_data_template.txt
Variables                ../basic_data_users.py

Test Template           Basic data transfer template
Test Setup              Setup basic data transfer
Test Teardown           Teardown basic data transfer

*** Test Cases ***      User_group_descriptions    Duration    Ciphering
One user scheduled      ${basic_data_case0}        15          ${disabled}
12 users scheduled      ${basic_data_case12}       15          ${enabled}

*** Keywords ***
Setup basic data transfer
    L2 test setup with number of cells     2

Teardown basic data transfer
    L2 generic test teardown
```

Figure 13. *Test suite example.*

4.2 Jenkins

In software development, continuous integration (CI) is a practice of merging all developer working copies to a shared mainline several times a day. The aim of CI is to prevent integration problems by running automated compilation tasks and tests to verify the software's proper status. [28; 29] In the layer 2 software CI is implemented using Jenkins which runs jobs such as build tests and Robot Framework test cases.

Jenkins jobs are triggered when a developer commits changes in the version control system. If some job fails the committer can be notified by an automatic e-mail. Each job has its own web page which can be accessed using a web browser. On the page one can access, for example, the console output of the job or the Robot Framework logs. Jenkins can also generate graphs that can be used to see the progress made in the development. [27]

4.3 Manual Testing

The environment also supports running tests manually. This is done with a bash script that, for example, reserves the target hardware, uploads binaries to the target, reboots the target and starts the tests. After a test run it copies the results back to the user. The script supports several different command line parameters that are used to define the target hardware, which tests to run and other options. The Jenkins jobs use the same script to run tests.

5. VALGRIND AND RELATED TOOLS

This chapter introduces the tools that were used to execute the cache analysis. The usage and functionality of the tools are explained. In addition, some other programs that are helpful in examining the results of these tools are listed. Finally, the chapter introduces some other tools that were not used in this work but can be used to analyze cache usage.

Valgrind is an open source instrumentation framework for building dynamic analysis tools. Dynamic analysis means that the program observes an executing program and takes live measurements. Static analysis, on the other hand, only examines the source code or binary file and predicts behavior. Valgrind was originally intended for debugging and profiling Linux programs on the x86 processor architecture but now supports several other platforms as well, including Linux on ARM. It is completely free software and the program source code can be downloaded from the software's home page. [15; 30]

Using Valgrind does not require any changes in the program that is analyzed. However, the program and any supporting libraries it uses should be compiled with debugging information. The debugging information helps Valgrind point out the exact place in the source code where a certain event that was examined took place. Valgrind takes full control of the program before it starts and runs it in a synthetic CPU provided by the Valgrind core. The core grafts itself into the client process and recompiles its machine code. The compilation involves first disassembling the code into an intermediate representation and then converting it back to the original machine code. The intermediate representation is used by the tools Valgrind offers to add their own instrumentation and analysis code. This disassembling, instrumentation and analysis causes the execution of the client program to slow down. [15]

5.1 Cachegrind

Cachegrind is a cache profiler that is part of Valgrind's toolset. It is used to detect cache read and write misses for instructions and data separately. It can also be used to detect branch instruction and misprediction counts. Cachegrind gets these results by simulating the cache hierarchy and branch predictor of a processor. Although some processors have multiple levels of cache Cachegrind analyses only the first and last level cache usage since information collected on them is enough to get useful results.

Cachegrind analysis is launched from a Linux terminal using the following syntax: *valgrind --tool=cachegrind program*. The tool parameter tells Valgrind that Cachegrind should be executed. After that the call of the client program and all possible parameters to that program are listed as if it was executed normally from the terminal. The branch prediction analysis with Cachegrind is executed if *--branch-sim=yes* parameter is passed to Valgrind.

By default Valgrind, and consequently Cachegrind, analyzes only the process that is defined at the command line. If the program creates new processes, for example by using the fork-exec technique, the child processes will not be included in the analysis. Tracing the child processes can be forced with a command line parameter *--trace-children=yes* and uninteresting executable branches can be ignored with *--trace-children-skip=program1,program2*. Skipping a process means also that any children that the process forks will also be excluded from the analysis.

Cachegrind can try detecting the processor cache properties automatically but they can also be defined by the user. Because of this it is possible to simulate the cache usage of a processor on a different machine instead of running Valgrind natively on the real hardware. For this purpose the following command line options can be used (the cache and line sizes are given in bytes):

--I1=<size>,<associativity>,<line size>	(level 1 instruction cache)
--D1=<size>,<associativity>,<line size>	(level 1 data cache)
--LL=<size>,<associativity>,<line size>	(last-level cache)

The execution of Cachegrind produces a summary of the results and a log-file. An example of a typical summary is shown in Figure 14. Each line of the summary starts with a number surrounded by two equal signs. The number is the process identifier (PID) assigned to the program by the operating system. This same number is used in the name of the log-file which by default is dumped in a file called cachegrind.out.<PID>. The PID is used so that when Cachegrind traces into child processes there is one output file for each process. The log-file name can also be defined by the user with the following command line parameter: *--cachegrind-out-file=<filename>*. Rest of the output includes the number of instruction fetches made (I refs), L1-cache and last level cache instruction misses (I1 misses and LLi misses) and their miss rates. This is followed by the same numbers for data (D refs, D1 misses, LLd misses) but also included are numbers for reads and writes separately (rd, wr). At the end of the output combined results for the last level cache are shown. The number of instruction fetches is also the number of instructions executed, which can be useful information for profiling a program on its own right.

```
[orantala@ouling09 cg_tests]$ valgrind --tool=cachegrind ./slow_loop
==878== Cachegrind, a cache and branch-prediction profiler
==878== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==878== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==878== Command: ./slow_loop
==878==
--878-- warning: L3 cache found, using its data for the LL simulation.
--878-- warning: specified LL cache: line_size 64  assoc 16  total_size 12,582,912
--878-- warning: simulated LL cache: line_size 64  assoc 24  total_size 12,582,912
==878==
==878== I   refs:      2,519,190,335
==878== I1  misses:              757
==878== LLi misses:              740
==878== I1  miss rate:          0.00%
==878== LLi miss rate:          0.00%
==878==
==878== D   refs:      1,175,203,708  (1,007,229,048 rd   + 167,974,660 wr)
==878== D1  misses:      167,780,563  (        7,265 rd   + 167,773,298 wr)
==878== LLd misses:      167,777,578  (        4,561 rd   + 167,773,017 wr)
==878== D1  miss rate:          14.3% (          0.0%     +        99.9%  )
==878== LLd miss rate:          14.3% (          0.0%     +        99.9%  )
==878==
==878== LL refs:         167,781,320  (        8,022 rd   + 167,773,298 wr)
==878== LL misses:       167,778,318  (        5,301 rd   + 167,773,017 wr)
==878== LL miss rate:            4.5% (          0.0%     +        99.9%  )
[orantala@ouling09 cg_tests]$
```

Figure 14. *An example summary output of Cachegrind.*

The log-file is a human readable text file but an easier-to-read format can be made by executing the cg_annotate command. Cg_annotate is a Perl script that ships with Valgrind and it produces an output as shown in the example in Appendix A. First, the output includes information about the cache properties that were used in the analysis, which events were recorded and some other properties that were used to create the results. This is followed by the program and function-by-function level statistics, which are similar to that in the summary output. The functions are identified by a file:function pair except in case the name cannot be determined from the debugging information three question marks are shown. If a column contains a dot it means the function never performed that event. Furthermore, the source code can be annotated line-by-line by giving the *--auto=yes* command line option to cg_annotate or similarly by giving the source code file names and paths as parameters. The output in Appendix A shows that most misses happen in the loop-function when the array is filled with values. By changing the order of the indices i and j (loop interchange) and running the test again, the number of misses reported by Cachegrind is decreased to 1/16th of the original. The performance improvement gained by this change is also noticeable when running the program normally. The improved version finishes almost immediately while the original version needs several seconds to complete.

Cg_annotate can be given a sort option, which causes it to arrange the function level statistics based on user preference. The function level statistics can then be used to easily see which functions are causing the most cache misses and the details of that function can be examined in the line-by-line level statistics. Every source code file or line is not included in the results but only those that count for most of the event that was used to sort the results. By default this event is the number of instruction fetches made. Full path and file name precedes every source code file that is used and gaps in a single source code file are marked with line numbers. In the annotated source code a dot is used to mark an event not applicable for that specific line. This is useful information for knowing which events cannot happen and which events can but did not.

In addition to cg_annotate Cachegrind offers two other similar tools that can be used to analyze the results. Cg_merge can combine the log-files of multiple runs and cg_diff does the opposite, it subtracts the results of one log-file from another. Cg_merge is useful for combining the results from two or more runs of a program when different code paths cannot be executed in a single run. Cg_diff can be used to compare the results between different versions, for example after optimizations have been made. Cg_merge combines the results all the way to the line-by-line level statistics but cg_diff works only on the function level. [15]

5.1.1 Cache Simulation Details

As mentioned before, Cachegrind simulates a two level cache where the LL is inclusive. It uses LRU replacement policy and write-allocate miss policy which means that when a write miss occurs the block written is brought into the L1 data cache. Instructions that modify a memory location, such as *inc* and *dec*, are counted as doing only a read. This is because the read guarantees that the memory block is already in the cache and therefore a write miss cannot happen. References that straddle two cache lines are treated as follows:

- Both blocks hitting is counted as one hit.
- One block hitting and the other missing is counted as one miss.
- Both blocks missing is counted as one miss.

Cachegrind also has some shortcomings that can warp the results from the actual cache usage. It does not count for the operating system or other program activity which are also using the cache. Valgrind serializes the execution of threads and schedules them differently from how they would run natively. This can warp the results of threaded programs. It can only count events that are visible on the instruction level because it does not simulate any additional hardware such as a Translation Lookaside Buffer (TLB), which is a cache that memory management hardware uses to speed up virtual address translation. However, the results are still accurate enough to be useful in cache usage analysis and performance optimizations. [15]

5.1.2 Acting on the Results

Cachegrind gives a lot of information about the program that is analyzed but it is not always easy to see what to do with that information. In general, the global program level statistics are not practical unless you have multiple different versions of the program. In that case you might use them to identify the best versions. [15]

The function level statistics are more useful as they help to identify the places that are causing the most event counts. However, inlining can make these counts misleading. In C++, for example, the *inline* keyword is an indicator to the compiler that inline substitution of a function is preferred over generating a function call. It means that the function call is replaced with the code of the function body by the compiler. This avoids overhead caused by the function call such as copying arguments and retrieving the return value. The keyword is non-binding in C++ meaning that the compiler can inline functions that are not marked to be inlined and not inline functions that are. If a function is always inlined all the event counts will be attributed to the functions it is inlined into. However, in the line-by-line annotations you will see event counts that belong to the inlined function. This is why the line-by-line statistics are the most helpful. [15; 31]

It might be useful to first take a look at the functions and lines that are causing high instruction fetch counts. This does not include any cache usage information but helps to identify bottlenecks in the program. Next the high L1 and LL data read and write miss counts should be used to identify the places in the source code where cache usage optimizations might be made. [4; 15] Optimizations can be made with the techniques described in Chapter 2.

5.2 Callgrind

Callgrind is an extension to Cachegrind, which can be used to get all the data Cachegrind does and the function call graphs of the program under analysis. This means that it collects information about the program's function call chains and counts how many times functions are called. Callgrind produces similar output prints and files as Cachegrind. It uses the same cache simulation as Cachegrind but also offers some additional profiling options such as collecting cache line usage statistics and the ability to turn the event counters on and off at any point in the code. Because Callgrind collects function call statistics, it can report inclusive cache miss counts. Inclusive count is the cost of a function itself and all the functions it called. Cachegrind can count only self-cost which is the cost of a function itself. Because Callgrind collects more information than Cachegrind it slows down the execution of the client program even more. Callgrind's ability to detect function calls and returns does not currently work well on the ARM processors.

Controlling the Callgrind event collection can be done in a few different ways. The first option is changing the state programmatically by adding macros in the client program source code. This requires including the Callgrind header file in the source code and adding the Valgrind source code directory in the compilers search path. The second option is using a command line tool called callgrind_control. The event collection can first be turned off at startup with the command line option *--instr-atstart=no* and then turned on at any point in time in the future while the client program is running. The third option is using the *--toggle-collect=function* option. This command line parameter to Callgrind toggles the event collection state while entering and leaving the specified function. Valgrind and the client program can also be run in the gdb-debugger and the event collection state can be changed from the gdb command line. [15]

5.3 Other Tools

Valgrind's toolset includes some other programs that can be used similarly to Cachegrind and Callgrind to debug and optimize a program. Memcheck is a tool that can detect memory management related problems such as memory leaks and illegal memory accesses. Massif is a heap profiler that produces information about which parts of a program make the most memory allocations. Helgrind is a debugger for multithreaded programs and it can be used to detect data races. Some other tools are also available. [15]

KCachegrind is a visualization tool for Cachegrind and Callgrind output that can be used instead of the command line based tools such as cg_annotate. It is not included with Valgrind but has to be installed separately. KCachegrind is best used with Callgrind to see a graphical representation of the function call graphs of a program but it is also useful for Cachegrind output. For example, the user interface offers an easy way to sort the results by any of the collected parameters or group them based on the source file or C++ class. It also offers the ability to add new event types that are based on the existing events. Last level miss sum and cycle estimation, for example, are new events that are present in KCachegrind by default. With Callgrind output KCachegrind can be used to analyze instruction level cache usage statistics. This requires running Callgrind with the parameter *--dump-instr=yes*. [15; 32]

Eclipse is an integrated development environment that supports several different programming languages. It has a graphical user interface plugin that enables integrating Valgrind into its C/C++ development tools. The plugin supports Cachegrind and allows viewing Cachegrind results in a graphical view. In the view double-clicking a file, function or line in Cachegrind output will open the corresponding source code file and place the cursor on the appropriate location. [33]

5.4 Alternative Cache Profilers

DynamoRIO is an open source dynamic binary instrumentation tool framework. It includes a tool called drcachesim that is similar to Cachegrind but is still under development. Drcachesim can produce only program level cache usage statistics. It can simulate

TLBs unlike Cachegrind and also has better support for multi-core CPUs and threads. For example, drcachesim can simulate separate L1 caches for each CPU core while Cachegrind simulates only a single L1 cache and CPU core. DynamiRIO supports 32-bit x86, 64-bit x86 and ARM processors on Windows, Linux and Android operating systems. [15; 34]

OProfile is another open source tool that can be used to analyze cache misses. It is a statistical profiler that uses the native hardware event counter registers to collect the data and causes very little performance overhead. [35] The Linux Perf tool and Intel's VTune Amplifier XE can do similar cache usage analysis as OProfile. They do profiling by collecting samples from the event counters while the program that is analyzed is running. [36; 37] Perf is already used in performance analysis at Nokia. It records samples at certain intervals, which can be configured by the user. For example, if L1 cache misses are monitored and the sample rate is set to 1000, Perf records a sample after 1000 L1 cache misses. The sample is the line in code where the process was when the sample was recorded. Because the results are based on statistical probability, they are not as accurate as the results given by Cachegrind. [27] CriticalBlue's Prism Technology Platform is a commercial product which supports dynamic binary analysis. It can measure data and instruction cache usage efficiency. [38]

Pahole (Poke-a-hole) is an open source program that can be used to perform static analysis on binaries. It can detect the size of data structures, how many cache lines they use and the holes caused because of the compiler aligning the data elements to the word size of the CPU. Moreover, it can suggest ways to reorder the elements to fill the holes, optimize bit fields and combine padding and holes. These optimizations reduce the memory footprint of the data structures and can therefore improve cache usage because more data can fit in the cache and cache lines. [4; 39]

6. IMPLEMENTATION

This chapter talks about the details of implementing the cache usage analysis which includes compiling Valgrind, starting the system under Valgrind and running appropriate Robot Framework test cases. The results that were obtained are also listed in this chapter. Finally, practical difficulties that were encountered during this process are discussed.

6.1 Compilation

A binary distribution of Valgrind can be installed on some Linux distributions without the need to compile it from the source code. However, the compilation process of the ARM binaries is detailed here because a precompiled version was not available. The latest version of Valgrind at the time of this work was version 3.11.0 which was also used in this work. The ARM binaries were cross-compiled on a 64-bit x86 Linux machine.

Valgrind source uses the Autotools build system. This means that the first step of compiling Valgrind is running a configuration script simply called configure that scans the system for certain information and creates the makefile that is actually used to compile the binaries. The cross-compilation required setting some environment variables so that the configuration script could find the correct toolchain binaries such as the compiler and linker. These variables are listed in Figure 15, which shows the contents of a script that was used to execute the configuration. The actual toolchain path was omitted from the figure because it was very long. The variables were set using the *export* keyword to ensure that they were visible in child processes [40]. Two parameters were passed to the configure script. The *host* parameter defines the target architecture, which in this case was ARMv7, and the *prefix* parameter sets the installation path. [15; 41]

```
#!/bin/sh
export PATH=$PATH:<toolchain_path>
export CROSS_COMPILE=arm-cortexa15-linux-gnueabihf-
export CC=${CROSS_COMPILE}gcc
export CPP=${CROSS_COMPILE}cpp
export CXX=${CROSS_COMPILE}g++
export LD=${CROSS_COMPILE}ld
export AR=${CROSS_COMPILE}ar
./configure --host=armv7-linux --prefix=/tmp/valgrind-arm
```

Figure 15. *Preparing Valgrind cross-compilation.*

After the configuration script had finished running, Valgrind compilation was done by executing the *make*-command in the Valgrind directory. This required setting the toolchain path the same way as the configuration script. The final step was running the *make install* command which copied the binaries and libraries in the directory defined by the *prefix* option that was given to the configuration script.

6.2 Preparing the Environment

Before running any tests, the Valgrind binaries had to be transferred to the target device. This was done using the secure copy program, scp, which copies files between hosts on a network [42]. To make the process easier all the binaries were first added in a single tarball archive. The Linux terminal commands of these steps are listed below:

```
tar -cvjf valgrind-arm.tar.bz2 valgrind-arm
```

```
scp valgrind-arm.tar.bz2 <user>@<hostname>:/home/user/dest
```

The first command creates an archive called valgrind-arm.tar.bz2 from the contents of a folder called valgrind-arm. The second command sends the archive to the target machine in the path that is defined after the colon. The username and hostname have to be correct for this to work. On the target machine the files in the archive can be extracted using the following command:

```
tar -xf valgrind-arm.tar.bz2
```

The files were actually extracted in /tmp/valgrind-arm because that is the same path that was given to the configuration script before compiling the binaries. This way the binaries can find all the Valgrind libraries they need. The target device terminal was used through a Secure Shell connection by using the ssh-command:

```
ssh <user>@<hostname>
```

6.2.1 System Startup with Valgrind

Because of how Valgrind works the eNB software had to be started under its control. The device was accessed using an ssh-connection and all the eNB processes were first killed leaving only the operating system processes running. The killed processes could be restarted with an existing Linux shell script. The command for running Valgrind was added in the script at the appropriate location. Because the system runs multiple processes the option to trace child processes had to be given to Valgrind.

There were several problems with the system startup under Valgrind's control. The details of these problems are talked about in Chapter 6.5. Due to these problems and time constraints, the cache usage analysis was run in a host environment instead of the real hardware environment. However, Cachegrind was tested on the device by running some commands that the operating system offers with it. Cachegrind worked fine on the ARM processor, the problems were caused by the combination of Cachegrind and the eNB software.

6.2.2 Host Environment

The host environment is a 32-bit x86 executable Linux program that simulates the real layer 2 environment. It is a non-real time process unlike the system on the actual hardware but uses the same layer 2 implementation source code. The purpose of the host environment is to ease the testing of small changes in the software. Testing on the real hardware requires reserving the appropriate device for the job. Since several developers and the Jenkins jobs can be running tests at the same time this means that a lot of hardware is required. In addition, uploading the binaries, rebooting and transferring the results takes some time. Therefore using the host environment to run the tests makes things easier. At the time of this work the host environment was still under development. For this reason, a DSP hardware platform was used in the environment simulation.

Even though the DSP and the ARM platforms have many differences a lot of the same code is still executed in both devices. In addition, since Cachegrind can be given the cache properties that it should use in the simulation, cache usage results could be obtained this way. The accuracy of these results is discussed in Chapter 7.

At startup the host environment creates six separate processes that handle different parts of the simulation. Some of these processes are not interesting in the context of this work because they simulate systems that are not actually part of the eNB. This includes, for example, data generator and user equipment simulation. The actual interesting parts are running in one process.

6.3 Test Execution

Three new parameters were added to the bash script that is used to run the Robot Framework test cases:

--cachegrind

--callgrind

--valgrind-args

These parameters are supported only on the host environment. The first two options run cache usage analysis using Cachegrind and Callgrind respectively. The third parameter can be used to give additional arguments to Valgrind and the two tools. Callgrind was added because it helps separating the uninteresting parts from the analysis. The highest number of cache misses happen during the system setup which can be ignored by controlling the event collection status of Callgrind. Callgrind's function call statistics and inclusive costs can be used in examining the program behavior and identifying bottlenecks. The inclusive costs are also helpful when comparing the results of different test runs if function calls are added or removed in the cache usage optimizations.

The command that was used to run Cachegrind is as follows:

valgrind --log-file=valgrind.log.%p --tool=cachegrind --I1=32768,2,64 --D1=32768,2,64 --LL=4194304,16,64

The first parameter tells Valgrind to write all of its output in a file. The percentage sign followed by the letter p configures the filename to end with the PID of the program [15]. This was done because otherwise the Valgrind output would be mixed with the Robot Framework test case prints. Cachegrind writes its log-file normally. The cache properties that are defined here are the same that the ARM processor introduced in Chapter 2 has. The L1 cache memory sizes were converted to bytes by multiplying the size in kilobytes with 1024 and the LL by multiplying the size in megabytes with 1024^2. The cache properties were followed by the executable name and all the necessary parameters to start the host environment. These commands were added in a Robot Framework library written with the Python programming language. The Robot test cases use this library to start the host environment.

The command for running Callgrind was almost the same. In addition to the different tool name two additional parameters were used: *--intr-atstart=no* and *--cache-sim=yes*. The first option was used in order to skip the cache usage event collection at the system startup which includes several setup routines that are not interesting for performance optimization. The second option enables the cache usage analysis although technically this option is not needed because when the simulated cache properties are defined in the command cache simulation is executed automatically. [15]

Eight different test cases were used to perform the cache usage analysis and all of them simulate transferring data between the eNB and user equipment. The test cases use different features of the software. These tests were chosen because they worked in the host environment with minimal changes and because data transfer is a critical part of the system. Cache usage improvements in the data transfer source code have a direct impact on the transfer speeds of the system. The Valgrind results of the test cases were examined using KCachegrind. In KCachegrind the data read and write miss event counts were used to go through the results and find the places where optimizations could be made. The command line tools were also used to help analyzing the results.

Because the instrumentation was initially turned off when running the analysis with Callgrind it had to be turned on later at the appropriate location. The following commands were used to control the instrumentation status:

Run callgrind_control -i on

Run callgrind_control -i off

All the test cases were in the same test suite and used the same setup and teardown keywords. The first command was added in the setup keyword and the second in the teardown. The run-command is a built-in keyword in the Robot Framework that allows executing commands in the operating system [26]. It was used to execute the callgrind_control command line tool which can be used to toggle the Callgrind instrumentation state [15].

The execution of the test cases under Valgrind takes a relatively long time. This is not only because Valgrind slows down the execution speed but because delays had to be added in a couple of places in order to make the tests pass and get the cache usage results. A delay of 90 seconds was added at the startup of the host environment. This ensures that the host environment is ready before running any tests. In addition, the transmission duration of the test cases was increased. Originally, the test cases used durations ranging from 1 to 5 seconds. First they were changed to 15 seconds because the tests were timing out and failing under Valgrind. Finally, they were increased to 60 seconds to minimize the effect of the system setup in the analysis.

6.4 Results

The first thing that was noticed after running the tests a few times is that sequential runs of the tests produced different cache usage results even though no changes were made in the source code. The program level event counts of two runs are listed in Table 1. The table shows that the differences can be quite high, several millions in absolute values and around 1-3 % in most events. Cachegrind is very sensitive and according to the Valgrind homepage, small changes such as different file names and paths can affect the results. However, the differences seen here are too significant to be caused by such changes. Besides the paths and file names were not changed in this case except for a few log files that the system creates. A significant difference is seen between the Ir event counts, which means that the first test run executed more instructions than the other. This difference is most probably caused by the variable length packets and network latencies of the test environment. The data packet sizes and transfer rates are not constant which causes the system to execute different amounts of code between different runs. This also causes the other event counts to fluctuate. In addition, Callgrind's results showed that the function call counts fluctuated. This is one more reason for using Callgrind in the analysis. If the function call counts between two sequential test runs are the same or very close each other than the cache miss counts are also very similar. When changes are made in the client program's source code, the function call counts can be used with the cache miss counts to compare the results.

Table 1. *Fluctuations in the results.*

	Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	D1mw	DLmw
Run 1	20 460 174 383	94 120 523	100 906	7 318 052 337	69 696 966	268 815	5 308 248 187	25 589 278	16 366 074
Run 2	20 046 564 829	91 993 205	101 118	7 168 376 011	67 818 104	268 753	5 205 883 231	25 349 280	16 348 501
Diff.	413 609 554	2 127 318	-212	149 676 326	1 878 862	62	102 364 956	239 998	17 573
%	2,02 %	2,26 %	-0,21 %	2,05 %	2,70 %	0,02 %	1,93 %	0,94 %	0,11 %

Because the event counts can fluctuate so much the tests were always run three times to verify the results. KCachegrind was used to combine all the Cachegrind or Callgrind results and the function level values were used to compare the results between test runs before and after optimizations were made. The results of a few different optimizations are listed in Table 2. The first column of the table shows which event was optimized, the second column which optimization method was used, the next two show the event count numbers and the last column the difference between the numbers before and after optimizations. The numbers were taken from the combined function level results in KCachegrind and each value was divided by three to get an average.

Table 2. *Optimization results.*

Event	Optimization	Originally	Optimized	Difference
L1 Data Read	Data splitting	17 085	73	17 012
LL Data Write	Array merge	17 364	15 453	1 911
LL Data Read	Data splitting	855	0	855
L1 Data Write	Loop to memset	1 017	735	282

Cachegrind and Callgrind report the program level miss ratios for the different cache levels and with equation 4 the changes in the average memory access time could be compared. However, because of the fluctuations in the results it makes no sense to use the equation. In addition, because the changes in the miss event counts after the optimizations are so small compared to the overall program level counts the differences between the memory access times calculated with the equation would be very small.

A couple of the actual changes that were made in the code are detailed in the following paragraphs. The examples presented here do not depict the actual code. For example, the variable names and amounts were changed but the examples still present the same ideas that were used.

Figure 16 shows the changes that were made to get the improvement listed in line 2 of the optimization results. Originally the code contained two arrays of the same size that were used to store two different data structures. All the members of these structures were combined into a single structure, which means that only one array was needed to store the same data. The members were reordered so that the structure size was minimized. This decreased the LL data write misses reported by Cachegrind almost by a couple thousand. As explained in Chapter 2 the array merge increased spatial locality in the program. However, in the code the arrays were not used in a manner that would cause cross-interference. In this case the improvement is the result of a decreased amount of compulsory cache misses. When the first member of the merged structure is referenced in the code, the other members are loaded in the cache as well. In the original code the same work was done separately for the two structures.

```
//Original code:
struct S1 {
  uint32_t a;
  uint16_t b;
  uint16_t c;
  bool d;
};

struct S1 {
  uint32_t f;
  uint16_t g;
};

S1 s1_storage[SIZE];
S2 s2_storage[SIZE];
```

```
//Improved code:
struct S1S2 {
  uint32_t a;
  uint16_t b;
  uint16_t c;

  uint32_t f;
  uint16_t g;

  bool d;
};

S1S2 storage[SIZE];
```

Figure 16. *Optimization 1.*

Figure 17 shows another change that is listed in the last line of the table. The loop that zeroes the values of the two arrays was changed to two memset functions. Even though the loop was relatively short, this decreased the L1 data write misses by almost 300. The cache misses in the original implementation were apparently caused by cross-interference in the loop and the memset functions solved this problem. This result was given by Callgrind.

```
//Original code:                   //Improved code:
for(int i = 0; i < MAX; ++i) {     memset(array1, 0, MAX);
  array1[i] = 0;                   memset(array2, 0, MAX);
  array2[i] = 0;
}
```

Figure 17. *Optimization 2.*

The changes were also tested using a real hardware device. A data transfer capacity test case was run on the DSP device because the DSP processor offers an option to track the amount of CPU stall cycles that are related to the cache misses. The CPU stalls when it has to wait for the memory operations to finish. [43] The amount of stalls decreased after the optimizations, which means that the changes that were made based on the results from the host environment were positive. The changes were not tested on the ARM device because it has no way to check if the optimizations had any impact. Because only a few changes were made in the source code, they probably do not have enough impact on the transfer speeds of the system to be visible in tests.

6.5 Practical Difficulties

The Valgrind home page claims that support for the ARMv7-A instruction set is essentially complete. However, when Valgrind was used on the ARM device it terminated the program because it encountered an unhandled instruction and the analysis was not completed. Because Valgrind is a dynamic instrumentation tool, it has to know every instruction the client program uses. In total three missing instructions were implemented in the Valgrind source code to fix this issue. One of the implementations is shown in Figure 18. The unhandled instruction is mrc, which is used to read a coprocessor register to an ARM register [9]. The other instructions were similar rarely used commands.

```
// mrc      15, 0, r0, cr14, cr0, {0}
// read CP15 CNTFRQ register (Counter Frequency Register)
if ((INSN0(15,0) == 0xEE1E) && (INSN1(11,0) == 0x0F10)) {
    UInt rD = INSN1(15,12);
    if (!isBadRegT(rD)) {
         putIRegT(rD, IRExpr_Get(OFFB_CNTFRQ, Ity_I32), condT);
         DIP("mrc p15, 0, r%u, c14, c0, 0\n", rD);
         goto decode_success;
     }
}
```

Figure 18. *Unhandled instruction implementation.*

After the unhandled instructions were implemented the eNB software crashed at startup when ran under Valgrind's control. Valgrind informed that the crash was caused by an arithmetic error but the root cause behind that was not found. It is possible that the restrictions Valgrind sets on the software caused it. For example, the execution overhead and serialization of threaded applications might have caused this problem. Even the minimalistic tool Nulgrind, that does not do any instrumentation or analysis and slows down the client program only by a factor of five, caused the same crash [15].

There were some problems with the host environment as well. Because it was still under development, some time had to be used to improve it. The most time consuming task was caused by global variables in the eNB software. On the real target hardware each CPU core has its own copy of some global variables but in the host environment the same global variables were shared between threads which simulate different layer 2 nodes. The first idea to solve this problem was using a C++ wrapper class that would create a new instance of the variables for each thread. This required making many changes in the source code. Each definition of the global variables was different between the host environment and the real hardware target. This required using preprocessor macros to separate these two cases between different builds. The same process had to be done for each case where the variables were used for the first time. In the host build the C++ class had to be used to check if the variable was already allocated and to create the new instance if it was not. Another idea to solve the problem was creating multiple processes in the host environment similarly to how they are done on the actual hardware. This way each process would have its own global variables. However, this option was quickly dumped because implementing it would have required even more changes than the first option. The problem was finally fixed by adding C++11's *thread_local* storage duration specifier in front of most global variables in the host build. The specifier guarantees that each thread has its own copies of the variables [44]. This process was done by adding a new preprocessor macro in the source code. The macro was put in front of the variables where needed and it expands to the *thread_local* keyword only in the host build, otherwise it does nothing.

Another problem in the host environment was that some of the Robot Framework test cases were unstable. The tests would sometimes fail and sometimes pass even though no changes were made in any components between the test runs. This situation was made even worse by Cachegrind, which caused the amount of failure cases to increase or cause some tests to fail every time. Fortunately, this problem was easily overcome by running the tests on a different machine. The Robot Framework logs showed that the tests were failing because of timeouts. Apparently, the host environment has some timing problems that are machine dependent. The IP messaging of the system seems to require longer delays on some machines. The overall performance and network latency of the system that is running the tests seems to affect the delay requirements. The layer 2 software runs slower in the host environment than on the real eNB hardware and small changes in the performance of the system can cause the tests to fail.

7. EVALUATIONS

This chapter discusses about the accuracy of the cache usage results. The inaccuracies of Cachegrind and the effects of using the host environment are considered. In the second part of the chapter future developments are thought about.

7.1 Accuracy of the Results

Some cache memory properties are different between the ARM processor and Cachegrind’s simulation. Cachegrind uses LRU replacement policy for all the caches but the ARM processor uses random replacement policy for L2. According to a study that compared different replacement policies random usually has higher cache miss rates than LRU [45]. Random is better than LRU only in cases where a memory block slightly larger than the cache is swept through linearly and repeatedly. In this case LRU drops each entry right before it is needed while random can select any entry. Because such cases are rare in the eNB layer 2 software, this decreases the amount of LL cache misses reported by Cachegrind compared to the real hardware environment.

Another difference is the write hit policy. Cachegrind does not specify any difference between the write-back and write-through policies but this does not actually matter. Inclusive L2 cache means that the L1 uses write-through policy. For L2 a copy of a written cache line will stay in cache for both policies. Therefore the write hit policy has no effect on the event counts Cachegrind collects.

The write miss policies are also different or more precisely they can be different in some cases. The ARM processor uses the inner memory attributes from the memory management unit to determine which write miss policy it uses for the data caches [16]. Cachegrind always uses write-allocate policy. Since the processor can use no-allocate policy, this causes even more optimism in the results.

Several other things also cause differences between the cache usage simulation and the real cache usage. As mentioned earlier the serialization and scheduling of threads can change the execution order and thus the results. Whether this decreases or increases the cache miss counts depends on the situation. The client program has different memory layout under Cachegrind, which can cause the amount of conflict misses that occur to be different. Cachegrind also does not support software pre-fetching which is used in the eNB software. This means that the cache usage results for data that is pre-fetched are wrong and should be ignored.

When Callgrind was used to collect the event count numbers, the event collection was switched on in the middle of the running program. This means that the simulated caches were empty when the event collection began but in reality the caches should contain data from the earlier setup functions. This affects the number of compulsory cache misses detected by Callgrind.

On the real hardware some of the processes run on different CPU cores but in the host environment under Cachegrind only one L1 cache is simulated. This means that the L1 cache usage can be very different on the real hardware. Because each core has its own L1 cache the Cachegrind L1 results are too pessimistic. Simply put, the ARM processor actually has four times more L1 cache than was used in the simulations. The operating system processes running on the real hardware environment use a dedicated CPU core. For this reason they do not interfere with the L1 cache usage of the layer 2 software. However, the shared L2 cache contains data and instructions from all the processes on the system. This can cause small differences between the simulated and actual L2 cache usage results.

The host environment simulated a DSP platform instead of the ARM platform. This means that some code paths and data structures that were used in the simulations were different than on the real hardware. When the results were examined the DSP specific parts were ignored. In addition, many constants that are used to define array sizes are different on the two platforms. Loops that access the elements in these arrays were executed on the host environment the same amount of times as they are on the DSP platform. The result is that some loops have too low or too high miss event counts. However, if the array sizes are not significantly different between the two platforms this does not have any effect to the results in practice. If a loop has several miss events, a small change in the array size the loop iterates has only a small effect on the miss counts. Only if the differences are so big that in the other case the array would fit entirely in the cache this would cause an erroneous miss event report.

Overall, the results Cachegrind and Callgrind’s cache analysis give are not an accurate representation of the real cache usage. The analysis only gives a general picture of how the code and data structures of the client program affect the cache usage in a specific cache architecture. Despite all the inaccuracies, the results are still useful enough to give an indication of where the cache misses are happening in the client program source code.

7.2 Further Developments

Getting the analysis to work on the real hardware environment would be the ideal case because it would give the most accurate results. Because the eNB software has some strict real time requirements this might be difficult to implement in practice. The overhead Valgrind causes in the execution time of the client program would cause timeouts in the critical parts of the system and the cache usage results would be incomplete. It might be possible to circumvent this problem by relaxing the real time requirements of the system.

Callgrind offers some additional cache analysis features that were not used in this work. The first one of them is the instruction level event collection that was mentioned earlier in Chapter 5. The assembly code level annotations might be helpful in detecting some patterns that could be improved. However, since the host environment was running on a different CPU architecture than in the real use case the machine dependent assembly level code between the two environments is very different. For this reason the option was not used in the cache analysis.

Another additional feature is write-back behavior simulation. This can be enabled with the Callgrind option *--simulate-wb=yes*. This allows distinguishing between LL cache misses with and without write backs. It adds three new miss events that are ILdmr, DLdmr and DLdmw. The small d-letter in the event names refers to the dirty cache line that has to be updated to the main memory when a new memory block replaces an old block in a cache line. As these new events produce two memory transactions they account for a double time estimation compared to the normal misses. [15]

Callgrind can also simulate a simple hardware pre-fetcher which can be enabled with the option *--simulate-hwpref=yes*. It detects stream access in the last level cache and assumes always that the triggered pre-fetch succeeded before the real access is done. This means that it gives the best-case scenario by detecting all possible stream accesses. [15] Using this option helps reduce the amount of cache misses that are wrongly reported because of the missing hardware pre-fetcher.

The collection of cache line usage was also mentioned already in Chapter 5. It is enabled with the following parameter: *--cacheuse=yes*. With this option Callgrind collects usage information of every cache line. This includes the number of accesses and the actual number of bytes used. These events are related to the code that triggered the loading of the cache line. While the miss counters show the symptoms of where bad cache usage happens, the use counters try to pinpoint the reason. The option adds new event counters that help detecting bad temporal and spatial locality for both simulated caches. It also helps detecting bad data layout designs. [15] Using this option in the future might help especially optimizing the data layout designs in the eNB software.

Besides cache usage analysis, Valgrind offers several tools and other options that can be used to debug and optimize the performance of programs. For example, the branch predictor simulation of Cachegrind (and Callgrind) might be worth testing out. Also, some of the other programs introduced in Chapter 5 might be helpful. Pahole could be used to analyze the data structures and C++ classes in the eNB software.

It might be beneficial to design a single test case for the cache usage analysis. The test case should be designed so that it executes all the interesting code paths and uses all the data structures that should be optimized. It should also execute these parts of the code several times to reduce the effect of the fluctuations in the cache usage event count numbers. The tests that were used in this work were also quite simple. They simulated cases where only a few users were present and the data transmissions did not use much data. Better cache usage results might be possible if these parameters are increased. With the tests that were used in this work the results were dominated by the setup functions and other rarely executed parts of the system. Once the host environment is developed further more comprehensive tests can be used to increase the effect of the critical paths of the system in the cache usage results.

8. CONCLUSIONS

The objective of this work was studying if Cachegrind can be used to analyze the cache memory usage of the layer 2 eNB software. Because the software has strict real time requirements and Cachegrind causes the execution of the client program to slow down, Cachegrind cannot be used on the actual device at the moment. If changes that relax the real time requirements are made in the software then using Cachegrind on the device might become possible. The software crashed under Valgrind even with the minimalistic tool Nulgrind which might mean that some other changes are also required. The actual cause of this crash was not found.

Even though the cache usage analysis cannot be run on the eNB device, the host environment offers a viable alternative. The event miss counts reported by Cachegrind in the host environment might not be very accurate but they do still point out the places in the source code where the misses are happening. Any optimizations in the layer 2 software that are done based on these results will also decrease the miss counts on the actual device.

Because the startup process of the software includes uninteresting code for optimizations, using Callgrind makes examining the results easier. The setup functions that are executed in the beginning of the system startup cause the highest cache miss event counts but they can be ignored with the help of Callgrind. In a real use case the system is started only once and after that it runs continuously. The most interesting targets for optimizations are the code paths that are used in the data transmission related functions. Optimizations in these parts of the system code can improve the transmission speeds of the system.

Another reason for using Callgrind is that it helps examining the software behavior. It can show how many times functions are called and what functions called them. This information combined with the inclusive costs Callgrind collects helps identifying the bottlenecks in the program.

The statistical profiler Perf was already in use to improve the software but Cachegrind and Callgrind found code fragments where further improvements can be made. Because they simulate the cache usage at instruction level, they can more accurately pinpoint the places in the source code where the cache misses are happening. Using the results some changes were made in the source code. According to Cachegrind and Callgrind the changes decreased the cache miss counts by several thousands.

Because only a few improvements were made in the source code, testing their effect on the data transfer speeds was not possible. Any visible changes in the transfer speeds would require much higher amount of removed cache misses. However, the changes were tested on the DSP platform that can detect the amount of CPU stalls during the execution of a program. The amount of CPU stalls decreased after the optimizations. Because cache misses cause stalls this means that the optimizations increased the performance of the system.

At the moment Cachegrind and Callgrind seem to be one of the best dynamic instrumentation tools available for cache usage analysis. They are easy to use because a lot of documentation and examples can be found from the Internet. Because Valgrind is an open source program, making changes and additions in its source code is possible. Even though the results they give are not an exact presentation of the actual cache usage, they are still very helpful in detecting code that can be optimized.

REFERENCES

[1] Dahlman, E., Parkvall, S. and Sköld, J. *4G LTE/LTE-Advanced for Mobile Broadband.* Elsevier Ltd, 2011. p. 415.

[2] Ahmadi, S. *LTE-Advanced: A Practical Systems Approach to Understanding 3GPP LTE Releases 10 and 11 Radio Access Technologies.* Elsevier Inc., 2014. p. 1105.

[3] Kowarschik, M. and Weiß, C. *An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms.* p. 23.

[4] Drepper, U. *What Every Programmer Should Know About Memory.* 2007. p. 114.

[5] Central Processing Unit (CPU). [Online] [Cited: February 18, 2016.] Available: http://global.britannica.com/technology/central-processing-unit.

[6] Sharma, N. *Computer Architecture.* Laxmi Publications, 2009. p. 226.

[7] Murphy, D. *Liquid cooling vs. traditional cooling: What you need to know*. [Online] [Cited: April 25, 2016.] Available: http://www.pcworld.com/article/2028293/liquid-cooling-vs-traditional-cooling-what-you-need-to-know.html.

[8] Harris, D. and Harris, S. *Digital Design and Computer Architecture.* Morgan Kaufmann Publishers, 2007. p. 592.

[9] ARM Information Center. [Online] [Cited: April 25, 2016.] Available: http://infocenter.arm.com/help/topic/com.arm.doc.faqs/3738.html.

[10] Shibu. *Introduction to Embedded Systems.* Tata McGraw Hill, 2009. p. 709.

[11] Smith, B. *ARM and Intel Battle over the Mobile Chip's Future*. *Computer*. 2008, Vol. 41, Issue 5, pp. 15-18.

[12] Roberts-Hoffman, K. and Hegde, P. *ARM Cortex-A8 vs. Intel Atom: Architectural and Benchmark Comparisons.* Dallas, University of Texas, 2009. p. 7.

[13] Sims, G. *ARM vs x86 – Key differences explained*. [Online] [Cited: April 25, 2016.] Available: http://www.androidauthority.com/arm-vs-x86-key-differences-explained-568718/.

[14] Wolf, M. *Computers as Components: Principles of Embedded Computing System Design.* 3rd Edition. Morgan Kaufmann Publishers, 2012. p. 529. Books24x7 version.

[15] Valgrind Home Page. [Online] [Cited: May 6, 2016.] Available: http://valgrind.org.

[16] ARM Cortex-A15 MPCore Processor Technical Reference Manual. [Online] [Cited: April 25, 2016.] Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0488d/BABIHFFG.html.

[17] Darte, A. and Guillaume, H. *New Results on Array Contraction.* Lyon, France : Laboratoire de l'Informatique du Parallélisme. p. 22.

[18] How to Use Loop Blocking to Optimize Memory Use on 32-Bit Intel Architecture. [Online] [Cited: April 25, 2016.] Available: https://software.intel.com/en-us/articles/how-to-use-loop-blocking-to-optimize-memory-use-on-32-bit-intel-architecture.

[19] Naik, S. *Coding for Performance: Data alignment and structures*. [Online] [Cited: April 25, 2016.] Available: https://software.intel.com/en-us/articles/coding-for-performance-data-alignment-and-structures.

[20] Rentzsch, J. *Data alignment: Straighten up and fly right*. IBM Corporation, 2005.

[21] Gcc Manual. [Online] [Cited: April 25, 2016.] Available: https://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Variable-Attributes.html.

[22] Chilimbi, T., Davidson, B. and Larus, J. *Cache-Conscious Structure Definition.* 1999. p. 12.

[23] Bacon, D., Graham, S. and Sharp, O. *Compiler Transformations for High-Performance Computing.* Berkeley, University of California. p. 76.

[24] Wong, D. *Fundamentals of Wireless Communication Engineering Technologies*. John Wiley & Sons, 2012, Chapter 13.

[25] Holma, H. and Toskala, A. *LTE for UMTS: OFDMA and SC-FDMA Based Radio Access.* John Wiley & Sons, 2009. p. 450. Books24x7 version.

[26] Robot Framework Homepage. [Online] [Cited: March 15, 2016.] Available: robot-framework.org/.

[27] Alapuranen, S. *Performance Optimizations for LTE User-Plane L2 Software, Master's Thesis.* University of Oulu, 2015.

[28] Fowler, M. *Continuous Integration*. [Online] [Cited: March 15, 2016.] Available: http://martinfowler.com/articles/continuousIntegration.html.

[29] Jenkins Wiki. [Online] [Cited: April 25, 2016.] Available: https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins.

[30] Guide to Callgrind. [Online] [Cited: April 25, 2016.] Available: https://web.stanford.edu/class/cs107/guide_callgrind.html.

[31] Inline Specifier. [Online] [Cited: April 25, 2016.] Available: http://en.cppreference.com/w/cpp/language/inline.

[32] KCachegrind Homepage. [Online] [Cited: April 25, 2016.] Available: https://kcachegrind.github.io.

[33] Eclipse Homepage. [Online] [Cited: April 25, 2016.] Available: http://www.eclipse.org/linuxtools/projectPages/valgrind/.

[34] DynamoRIO Homepage. [Online] [Cited: April 25, 2016.] Available: http://dynamorio.org/.

[35] OProfile Manual. [Online] [Cited: April 25, 2016.] Available: http://oprofile.sourceforge.net/doc/index.html.

[36] Jackson, M. *Cache Miss Rates in Intel VTune Amplifier XE*. [Online] [Cited: April 25, 2016.] Available: https://software.intel.com/en-us/articles/cache-miss-rates-in-intel-vtune-amplifier-xe.

[37] Perf Wiki. [Online] [Cited: April 25, 2016.] Available: https://perf.wiki.kernel.org/index.php/Main_Page.

[38] The Prism Technology Platform. [Online] [Cited: April 25, 2016.] Available: https://criticalblue.com/prism-technology.html.

[39] Goldwyn, R. *Poke-a-hole and friends*. [Online] [Cited: April 25, 2016.] Available: https://lwn.net/Articles/335942/.

[40] Export Manual Page. [Online] [Cited: April 25, 2016.] Available: http://ss64.com/bash/export.html.

[41] Automake Manual. [Online] [Cited: April 25, 2016.] Available: http://www.gnu.org/software/automake/manual/automake.html.

[42] Scp Manual Page. [Online] [Cited: April 25, 2016.] Available: http://linux.die.net/man/1/scp.

[43] Trace Analyzer User’s Guide, Texas Instruments, Literature Number: SPRUHM7B, March 2014. p. 102. Available: http://www.ti.com/lit/ug/spruhm7b/spruhm7b.pdf

[44] Storage Class Specifiers. [Online] [Cited: April 25, 2016.] Available: http://en.cppreference.com/w/cpp/language/storage_duration.

[45] Al-Zoubi, H., Milenkovic, A. and Milenkovic, M. *Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite.* Huntsville, University of Alabama, 2004. p. 6.

APPENDIX A: CACHEGRIND’S DETAILED OUTPUT EXAMPLE

```
[orantala@ouling09 cg_tests]$ cg_annotate cachegrind.out.878 --auto=yes
--------------------------------------------------------------------------------
I1 cache:         32768 B, 64 B, 4-way associative
D1 cache:         32768 B, 64 B, 8-way associative
LL cache:         12582912 B, 64 B, 24-way associative
Command:          ./slow_loop
Data file:        cachegrind.out.878
Events recorded:  Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Events shown:     Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Event sort order: Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Thresholds:       0.1 100 100 100 100 100 100 100 100
Include dirs:
User annotated:
Auto-annotation:  on

--------------------------------------------------------------------------------
           Ir I1mr ILmr            Dr  D1mr  DLmr          Dw        D1mw        DLmw
--------------------------------------------------------------------------------
2,519,190,335  757  740 1,007,229,048 7,265 4,561 167,974,660 167,773,298 167,773,017  PROGRAM TOTALS

--------------------------------------------------------------------------------
           Ir I1mr ILmr            Dr  D1mr  DLmr          Dw        D1mw        DLmw  file:function
--------------------------------------------------------------------------------
2,517,033,082    1    1 1,006,755,874     0     0 167,813,133 167,772,160 167,772,160  slow_loop.cc:loop()

--------------------------------------------------------------------------------
-- Auto-annotated source: /home/orantala/cg_tests/slow_loop.cc
--------------------------------------------------------------------------------
           Ir I1mr ILmr          Dr D1mr DLmr          Dw        D1mw        DLmw

            .    .    .           .    .    .           .           .           .  const int SIZE = 4096;
            .    .    .           .    .    .           .           .           .
            .    .    .           .    .    .           .           .           .  void loop()
            3    0    0           0    0    0           2           0           0  {
            .    .    .           .    .    .           .           .           .    int h,i,j;
            .    .    .           .    .    .           .           .           .    static int a[SIZE][SIZE];
            .    .    .           .    .    .           .           .           .
           56    1    1          21    0    0           1           0           0    for(h = 0; h < 10; h++)
            .    .    .           .    .    .           .           .           .    {
            .    .    .           .    .    .           .           .           .
      204,860    0    0      81,930    0    0          10           0           0      for(i = 0; i < SIZE; i++)
            .    .    .           .    .    .           .           .           .      {
            .    .    .           .    .    .           .           .           .
  839,106,560    0    0 335,585,280    0    0      40,960           0           0        for(j = 0; j < SIZE; j++)
            .    .    .           .    .    .           .           .           .        {
1,677,721,600    0    0 671,088,640    0    0 167,772,160 167,772,160 167,772,160          a[j][i] = i + j;
            .    .    .           .    .    .           .           .           .        }
            .    .    .           .    .    .           .           .           .
            .    .    .           .    .    .           .           .           .      }
            .    .    .           .    .    .           .           .           .
            .    .    .           .    .    .           .           .           .    }
            .    .    .           .    .    .           .           .           .
            3    0    0           3    0    0           0           0           0  }
            .    .    .           .    .    .           .           .           .
            .    .    .           .    .    .           .           .           .  int main()
            2    0    0           0    0    0           1           0           0  {
            1    0    0           0    0    0           1           0           0    loop();
            1    0    0           0    0    0           0           0           0    return 0;
            2    0    0           2    1    1           0           0           0  }
            .    .    .           .    .    .           .           .           .

--------------------------------------------------------------------------------
 Ir I1mr ILmr  Dr D1mr DLmr  Dw D1mw DLmw
--------------------------------------------------------------------------------
100    0    0 100    0    0 100  100  100  percentage of events annotated

[orantala@ouling09 cg_tests]$
```