Masters thesis

Heuristic Online Profile Based Instruction Cache Optimisation in a Just-In-Time Compiler

by

Stefan Eng

LITH-IDA/DS-EX–04/001–SE

2004-01-23
Masters thesis

Heuristic Online Profile Based Instruction Cache Optimisation in a Just-In-Time Compiler

by Stefan Eng

LiTH-IDA/DS-EX–04/001–SE

Supervisor: Karl-Johan Karlsson
Compiler Technology, IL/EAB/UZ/DI
Ericsson AB

Examiner: Associate Professor Christoph Kessler
Department of Computer and Information Science at Linköpings universitet
Abstract

This master’s thesis examines the possibility to heuristically optimise instruction cache performance in a Just-In-Time (JIT) compiler.

Programs that do not fit inside the cache all at once may suffer from cache misses as a result of frequently executed code segments competing for the same cache lines. A new heuristic algorithm LHCPA was created to place frequently executed code segments to avoid cache conflicts between them, reducing the overall cache misses and reducing the performance bottlenecks. Set-associative caches are taken into consideration and not only direct mapped caches.

In Ahead-Of-Time compilers (AOT), the problem with frequent cache misses is often avoided by using call graphs derived from profiling and more or less complex algorithms to estimate the performance for different placements approaches. This often results in heavy computation during compilation which is not accepted in a JIT compiler.

A case study is presented on an Alpha processor and an at Ericsson developed JIT Compiler. The results of the case study shows that cache performance can be improved using this technique but also that a lot of other factors influence the result of the cache performance. Such examples are whether the cache is set-associative or not; and especially the size of the cache highly influence the cache performance.

Keywords: Alpha processor, Cache, Compiler, Heuristic, Hot, Instruction, Model, Online, Optimisation, Profile, Just-In-Time, Set-Associative
Acknowledgements

Thanks to everyone at Ericsson AB (UZ), especially Kaka, Patrik S. and Mikael H. for all the guidance on the way. Thanks to Jonas Frid for introducing me at Ericsson as well as being a great friend. My family deserves all my gratitude for their support in everything I have ever considered doing, or actually done. Dahlbäck, Grahn, Granstedth, Heinonen, Hesslevik, Jannok and Strandelin are some of my friends that have inspired me and helped me to get through this education. To all of you, thank you.

I would also like to thank my examiner Christoph Kessler for supporting me and for all great advice along the way. Frida Gunnarsson have given me many moments of inspiration and also aided me in my struggles with LaTex.

This report would not look like this, if it was not for all the proof reading done by my father Torsten and my brother Anders. Last but certainly not least, thanks to Camilla for all support and help you have given me.

Thanks,
Stefan
## Contents

1 Introduction .................................................. 1
   1.1 Background ............................................ 3
   1.2 Assignment ........................................... 4
   1.3 Purpose ............................................... 4
   1.4 Structure .............................................. 5
   1.5 Reading Guide ......................................... 5
   1.6 Definitions and Explanations .......................... 6
   1.7 Abbreviations ......................................... 9

2 Approach ....................................................... 10
   2.1 Process and Code Behaviour ............................ 10
   2.2 Memory System .......................................... 11
   2.3 Ahead-Of-Time Compilation ............................. 16
   2.4 Just-In-Time Compilation ............................... 18
   2.5 New Algorithm - LHCPA ................................ 19

3 Case Study ...................................................... 31
   3.1 Alpha Hardware and ASA Compiler ...................... 31
   3.2 Results .................................................. 35

4 Conclusions and future work ................................ 41
   4.1 Thoughts and Conclusions ............................... 41
   4.2 Future work and further research ....................... 44
List of Figures

1.1 Performance gap over time ........................................ 2
1.2 AXE system layer abstraction ..................................... 4

2.1 Address bit distribution ............................................. 13
2.2 Cache topology ....................................................... 14
2.3 Address to cache location mapping scheme ..................... 15
2.4 Hot function selection from profile data ......................... 21
2.5 Functions mapped to cache ......................................... 23
2.6 LHCPA: Pointer clarification in placeCompleteFunction .... 28
2.7 LHCPA: Pointer clarification in placePartFunction .......... 29

3.1 Alpha EV67 cache memory layout .................................. 32
3.2 APZ VM, ASA Compiler and JIT code relations ................. 33
List of Tables

1.1 Abbreviation table ........................................... 9
3.1 Test result table ............................................. 37
Chapter 1

Introduction

Early in the computer age, all programs were written directly in machine code. In 1957 the first programming language (Fortran) appeared and with it came the need for translators, so-called compilers. The compiler’s job is to translate programs into machine or assembly code. This improved portability of programs to different hardware architectures and larger programs became easier to develop and maintain.

As time has passed, hardware has become increasingly more complex with performance demands increasing at every aspect of the computer world, ranging from computational speed to graphical presentation requirements. Performance increases at different rates for different parts of hardware, as in the case of speed concerning processors and memory. During the past 20 years or so, the speed of processors has on average increased by a factor of 1.55 every year and the memory access time has on average decreased by a factor of 1.07 every year [1]. Since the processor needs input from the memory, the increasing gap in performance between the two types of hardware has become a problem. Figure 1 illustrates the increasing performance gap over time.

To bridge the gap, or at least to reduce the problem, smaller and faster memories called caches have been introduced between the main memory
and the processor. The idea is to keep needed information as close in time as possible to the processor and also to utilize the locality property of the program code [1, 2]. If the needed information has to be fetched from main memory, many CPU cycles are lost where no computation can occur.

The performance of the programs executed on a machine with cache memories is highly dependent on the program itself. If the program is larger than the cache, there is a risk of not using its full potential, since code has to be fetched from the main memory more often than if it could stay in cache at all time (i.e. for programs smaller than the cache size). Compilers of today do not often produce cache utilization considerate code, which leads to lowered execution performance. Much research has been done on how to utilize the caches as effectively as possible.

The most commonly used method for compilation is static compilation or Ahead-Of-Time (AOT) compilation. This means that no information is known about the status of the computer during runtime of the process. The compiler can use the information about the hardware and some optimisation techniques but no dynamic information. The concepts of virtual machines and feedback-directed optimisation have created new demands and also possibilities of optimisation of cache utilization [13]. In a Just-
In-Time (JIT) compiler accessible dynamic information about e.g. the memory or register contents makes it possible to build more problem adapted program code at run time. Other types of adaptions to the process behaviour can also be performed based on for instance profiling information.

1.1 Background

The Ericsson AXE platform [17] is used for exchanges in telephone networks and central traffic-systems in cellular networks. The platform has until recently been executed on an Ericsson internal processor architecture APZ [17]. Programs for this system are written in an Ericsson-internally developed language called Plex-C [18, 19], specialized for the AXE platform and hardware.

When Ericsson changed processors to Alpha [6], they developed a layer between the Alpha processor and AXE OS. They called the layer APZ VM (APZ Virtual Machine) since the layer simulates the old APZ hardware to the OS and the processes run on top of it. This layer contains a JIT compiler that translates the APZ assembly instruction language called ASA to Alpha assembly code that can be executed on the Alpha processor. This implies that old AXE programs can be run on new hardware without any alterations, and therefore no compiler from Plex-C to Alpha has to be developed. The JIT compiler is called ASA Compiler since it translates from ASA to Alpha code (Reference [14]), see Figure 1.2.

The greater demands from customers, which lead to the hardware switch in the AXE system, have stressed the problem with the gap in performance between processor and main memory even further. It is therefore necessary for Ericsson to examine the possibility of improving cache utilization of their systems.
1.2 Assignment

Much research has been done on how to optimise cache utilization. However, the research has mostly been done in AOT compilers and not in JIT compilers. The difference in restrictions on the compile time in AOT and online JIT systems, makes it impossible to use the same technique in the two different types of compilation. Most researchers have introduced heavy profiling and optimisation calculation algorithms that simply would not work in the JIT compiler environment.

The task is to create a fast heuristic algorithm that does not require extensive resources during compilation but still improves cache utilisation and hopefully the overall performance of the process during its execution. The task is also to test an implementation of the heuristic algorithm in the ASA Compiler.

1.3 Purpose

The purpose of this report is to describe the work and accomplishments during my master’s thesis work at Ericsson on the above described assignment.

Ericsson is interested in the solution to the problem of poor cache perfor-
ance, since it may improve the competitiveness of the upcoming versions of the AXE system.

1.4 Structure

This master’s thesis report consists of several chapters each described below.

- **Introduction** (Chapter 1) - Gives background information for the work.

- **Approach** (Chapter 2) - Provides the ideas and algorithm behind the new heuristic algorithm LHCPA as well as concepts used in AOT environments.

- **Case Study** (Chapter 3) - Describes the case study, the system restrictions, properties and the implementation along with its results.

- **Conclusions and future work** (Chapter 4) - Presents the conclusions and some thoughts on further research.

1.5 Reading Guide

Depending on the reader’s experience and knowledge within the area of compilers and computer architecture, some parts of the report may be skipped, or read extra carefully in order to understand the report.

Readers familiar with the area of compilers may skip Sections 1.6 - 1.7 and jump directly to Chapter 2, reading through Sections 2.1 - 2.2 more quickly and focusing more on JIT Compilation issues in Section 2.4. The case study in Chapter 3 is important for the understanding of the conclusions and future work in Chapter 4.

Readers who are not so familiar with the area of compilers may in addition to the above read through the introduction in Chapter 1 until Section 1.2. They should also read the definitions and abbreviations in Sections 1.6 -
1.6 Definitions and Explanations

This section provides the reader with definitions and explanations of the expressions and concepts within the area of compiler and computer technology that are being used in this report. Some expressions are explained where they are used inside the report.

A computer consists of many different hardware components, one of the most important is the processor (Reference [1, 2, 20]), in which all computations occur. All programs consist of instructions (Reference [1, 2, 20]) that are interpreted by the processor. Different functional units for integers and floating numbers and logical operators etc. inside the processor are commonly referred to as resources. The processor can not contain much data and a main memory (Reference [1, 2, 20]) therefore exists where the program and its data is located during execution.

Smaller and faster memories called caches (Reference [1, 2]) are used in the communication of data and instructions between main the memory and the processor to minimise the impact of the technical difficulties of building big and fast enough memories. A memory system of a modern computer consists of registers (Reference [1, 2, 20]) within the processor, one or multiple caches arranged in one or many levels and the main memory. A cache miss is a memory access when data needed by the processor is not present in cache and have to be fetched from main memory. The reverse is a cache hit i.e. data needed by the processor is present in cache. A cache line (Reference [1, 2]) is a chunk of consecutive storage space within the cache. Each location within the cache belongs to a cache line.

Whenever an algorithm or method does not guarantee improvement in all cases, or does not give the optimal solution in some cases it may be referred to as heuristic algorithm (Reference [21]). Optimisation is the process of
improving the performance of any process or finding a solution that is more accurate than previous results. An optimal solution is therefore the most exact or efficient solution that exists.

A compiler is a translator for code of one language into another. A C-compiler may for instance translate a program written in C code into an assembly language (machine instruction code). (Reference [1, 2, 20])

There are different types of compilers, Ahead-Of-Time (AOT) or Just-In-Time (JIT) (Reference [5, 11]). The difference between the types lies in when the translation is performed. In the AOT case the translation is performed only once in advance for every execution of a program. In the JIT case the translation is performed once prior and during every execution of the program. This difference also implies that different types of information can be used during translation in order to improve the process behaviour. In JIT compilation dynamic information such as register contents during runtime is known and used as guide to improve the process efficiency. In AOT compilation only static information is known and not anything about the processor state during execution of the program. AOT compilation is therefore also referred to as static compilation.

Conditional branches (Reference [1, 2, 3, 5, 20]) are instructions that have different functionality depending on the contents of a register. They either jump to another address (branch is taken) or continue with the next instruction. A fall-through path (Reference [3, 5]), is the execution path of the function when no conditional branches are taken.

When a program is executed it is called a process [22]. Simultaneously there may be several processes of the same program running i.e. multiple copies of a program are running. Information about a process can be obtained using a profiler [1]. A profiler can gather data such as repetitions of instructions or how many cache misses that have occurred etc. An important concept of programs is the function [1, 2, 3, 5, 20]. A function is a smaller part of a program describing a particular functionality such as for instance; give the lowest prime number higher than $x$ ($x$ is a parameter). Functions sometimes call other functions and therefore chains or structures of function
calls called \textit{call graphs} \cite{1, 5} are formed. Profilers may analyse a process in order to build its call graph. A call graph can tell a lot of the behaviour of the program and is often used as a guide tool when programmers and compilers optimise programs. \textit{Feedback-directed optimisation} \cite{13} is when dynamic information of the behaviour of a process such as for instance call graphs is used by the compiler in order to build a more efficient program.

There are many strategies on how to make processes more efficient. Some are used in this work since they affect the cache performance. \textit{Code placement} (Reference \cite{5}) is a tool that allows the compiler to choose the locations of the functions of a program inside the address space of the memory. \textit{Code replication} (Reference \cite{5}) is a common tool to reduce the number of function calls. If the code of a function is inserted instead of a function call, the time of the actual function call is reduced. This method is also often referred to as \textit{function inlining}

The \textit{operating system} (Reference \cite{1, 5}) is a software that provide services of the hardware to the programs and administer system resources such as CPU, memory and I/O (Input/Output). If there were no operating system all programs would have to contain information about how to use the needed hardware. A program is compiled to match both the specific hardware and the operating system. In case of a hardware switch, the operating system must be either exchanged or run on top of a \textit{virtual machine} (Reference \cite{5}). The virtual machine is a layer on top of the new hardware that simulates the old hardware to the operating system.
1.7 Abbreviations

Here is a list of commonly used abbreviations within this report, and their explanations:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASA</td>
<td>Assembly language of APZ</td>
</tr>
<tr>
<td>AOT</td>
<td>Ahead-Of-Time</td>
</tr>
<tr>
<td>APZ</td>
<td>Ericsson developed Processor</td>
</tr>
<tr>
<td>D-cache</td>
<td>Data Cache</td>
</tr>
<tr>
<td>FDO</td>
<td>Feedback Directed Optimisation</td>
</tr>
<tr>
<td>I-Cache</td>
<td>Instruction Cache</td>
</tr>
<tr>
<td>JIT</td>
<td>Just-In-Time</td>
</tr>
<tr>
<td>LRU</td>
<td>Least Recently Used</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
</tbody>
</table>

Table 1.1: Abbreviation table
Chapter 2

Approach

In order to be able to improve performance of a process, full understanding of the underlying systems or mechanisms has to be obtained. In this case the behaviour of the process and the memory system used needs to be studied. It is also necessary to examine how others have solved the studied problem, in this case what the solutions in AOT compilation look like. Having done this, the development of the new algorithm can start.

2.1 Process and Code Behaviour

Modern programming languages describe program functionality with a set of functions. Each function has its own functionality and may call other functions. Structures, loops or chains, are formed when looking at function calls from many functions. These structures describe greater functionality within the program.

Compilers translate each function description into machine code - a list of machine instructions that expresses the same functionality as the description does. These lists of instructions are in an AOT compiler placed in an executable file. In a JIT compiler they are instead placed directly in main memory.
When a program is executed on a machine it is called a process. The program is read into main memory and executed or, as in the JIT environment, compiled directly into main memory and executed. Each instruction has its own unique address in main memory and a program counter keeps track of where in the address space the process executes at a certain time.

When optimising a program it is important to understand its process. In order to do so, so-called profilers are used. A profiler may count how many times a function within a process has executed or how much iteration a certain loop of instructions has executed etc. The information from a profiler may indicate where it is suitable to focus the optimisation effort.

Profiling requires resources and slows the program process down. There are different kinds and degrees of profiling that requires different amounts of process resources.

Some functions, or code segments of the process, may be executed more frequently than others. The performance of these functions affects the performance of the whole process more than the less frequently executed functions. For example: In an image filtering program there may only be three different functions - one to load the image from a file, one to filter each pixel in the image and one to store the image back to file. Depending on the image size these functions are executed with different frequency. The load and store are only executed once for every image but the filtering function is executed \(x \times y\) times in a \(x\) pixels wide and \(y\) pixels high image. Improving the load and store functionality may of course be needed, but disregarding the effect of optimising the actual filtering function may be devastating to the performance of the process when filtering larger images. Some functions may therefore be considered hot, i.e., considered extra important to optimise within a program.

### 2.2 Memory System

Ever since the cache was invented, the need to understand the memory system has been great, since understanding it is required in order to build
programs with good cache performance. The description below explains
the relations between the address space and the storage location within the
cache. This relation is also called mapping.

The different architectures of processors and their memory hierarchies can
often be described with some parameters (Reference [1, 2]). The most
relevant parameters concerning cache layout and its performance in any
system are:

- Instruction word size
- Instruction alignment policy (by byte or word)
- Cache levels
- Cache data/instruction separation
- Cache size ($= 2^γ$ bytes)
- Cache line size ($= 2^w$ bytes)
- $k$-way set-associative or direct-mapped cache.
- Cache load line delay
- Cache replacement policy
- Cache update policy

The last seven parameters may be different for each cache level and they all
affect the performance of the memory architecture. Using these parameters
it is possible to build a model of the cache, usable to calculate where each
address is mapped to in the cache. The different parameters affect the
complexity and structure of the model.

If the instruction size is not constant or the instruction alignment policy
is by byte instead of words, calculations of where each instruction can be
placed become more complex. In case of more than one level of caches the
model becomes more complex, since this implies an increasing amount of relations between address and placements in the different caches. Whether the cache is set-associative or direct mapped matters, since set-associative caches introduce more flexibility in the mapping of addresses, i.e. how many places a specific address may be mapped into.

The cache load line delay is highly relevant since it describes the penalty of every cache miss. If there was no delay there would be no need for cache optimisations. The total and line size of the cache have great influence on the memory system since they affect miss rates. Greater sized caches often have less cache misses but are slower at hits. Greater sized lines may increase the delay at cache misses but may also reduce the miss ratio. Separate caches for data and instructions affect the model in many ways. Data and instruction placements affect each other less if they are separated. If there are unified cache at higher levels their placements still affect each other. This since a replacement caused by for instance a data miss may replace a line in a higher cache level containing instructions needed later in the process.

The update policy can either be write-through or write-back and decides when edited data in cache is stored back to main memory. This policy is only important to caches containing data, since only data is edited in cache.

<table>
<thead>
<tr>
<th>$p$ bits</th>
<th>$b$ bits</th>
<th>$w$ bits</th>
<th>$\alpha$ bits</th>
</tr>
</thead>
</table>

Figure 2.1: Address bit distribution

Main memory can be seen as a lot of different things: a memory, many pages, many more blocks or even more words. A block is a consecutive sequence of $2^w$ memory locations that can be held in a cache line. The address, represented as a binary number, can be split up into sections each describing the offset within these concepts. The physical address consists of $\alpha$ bits, where
2.2. Memory System

\[ \alpha = p + b + w \]

\( p \) is the number of bits required to index the page, \( b \) is the number of bits required to index each block within a page and \( w \) is the number of bits required to index each word within a block, see Figure 2.1. There are two types of addresses: physical and virtual. A physical address points to a specific location in main memory, a virtual address also points to a specific location but it may change over time with the change of virtual to physical page mapping.

A cache has a similar topology, starting from the other end, such as words, blocks or lines and sets as depicted in Figure 2.2. A \( k \)-way set-associative cache has \( n \) sets with \( k \) lines within each set. A direct mapped cache only has one line in each set, thus making the use of set notation unnecessary. The cache lines are commonly as big as a block in main memory (i.e. \( w \) bits to describe its size). Given a specific physical or virtual address, it is always possible to point out the set it is mapped to within the cache hierarchy. Within the set, a line is placed according to the replacement strategy of the cache.

![Figure 2.2: Cache topology](image-url)
Figure 2.3: Address to cache location mapping scheme
2.3. Ahead-Of-Time Compilation

The least significant \( w \) bits of the physical or virtual address describe the offset within a line inside the cache. The \( s = \log(2^{\gamma - w}/k) \) next bits describe to what set the address is mapped to inside the cache. The rest of the bits in the address is used as a tag with length \( t = p + b - s \); making it possible to uniquely identify what block is actually stored in the actual cache line, see Figure 2.3.

A given address \( A \) is therefore mapped to set \((A \mod ((2^{\gamma})/k))/2^w\) and offset within the line \((A \mod 2^w)\).

2.3 Ahead-Of-Time Compilation

Ahead-Of-Time compilation have been and still is the most commonly used compilation method. The academic research world has produced numerous approaches on how to optimise performance of programs compiled with this method. In many such approaches, the basic idea is to take advantage of the nature of the program that is to be optimised. The program behaviour can be identified either by examining the code directly or by looking at profile information. Since compilation is only performed once in advance for all executions of the program in a AOT system, a lot of time may be spent on calculating optimal placement and optimal code alterations.

Code placement and code replication are tools that a programmer/compiler can use in order to affect cache performance considering an already existing memory system. Bad placement causes cache misses that lower overall performance. Good placement requires knowledge of both the memory system and the process. Code replication may, if implemented correctly, for instance reduce function calls to gain performance by exploiting the code locality property better, but also increase code size, which may in the worst case imply more cache misses, reducing the process performance.

In 1997 graph colouring was introduced as a tool to optimise code placement by Hashemi, Kaeli and Calder [4]. Their method uses a profile derived call graph (Reference [3]), function size and cache layout as a basis for all code positioning decisions. To decide where each method or function should be
placed, Hashemi et al. [4] use a graph traversal algorithm with node colouring to avoid cache conflicts of parent and sibling functions (call functions and called functions). Each colour represents a set or line in the cache, and each function (node) must not contain any colours of either parent or children if the positioning shall have desired effect. This algorithm also divides the functions into popular and unpopular groups, depending on the execution count of each function found in the call graph. The unpopular functions are used as padding in the empty spaces between the repositioned popular functions. This algorithm is very efficient in reducing cache collisions and has a clearly noticeable effect on the process performance. The algorithm was created for direct mapped caches but may easily be extended to set-associative caches as well.

The graph colouring algorithm (Reference [4]) is a profile-based algorithm. Another algorithm not using profile data is for instance the “Compile time Instruction Cache Optimisation” heuristic algorithm presented in 1994 by Mendelson, Pinter and Shtokhamer [12]. The heuristic algorithm, unlike Hashemi’s graph colouring algorithm, uses static information only and, by using code replication and code placement strategies, avoids cache collisions of code segments called from within the same loop. Greater sized patterns of function calls are harder to detect using this approach. The optimisation does not separate functions depending of their optimisation interest, since no such data is available.

There exist several hardware-dependent methods to avoid problems with cache performance as well. One such solution is for instance the dynamic exclusion hardware presented in 1992 by McFarling [15]. These types of solutions are not very interesting since they require extra hardware, making each optimisation really expensive. Process performance still depends on the compilation, as collisions are avoided by the hardware by reading directly from main memory if the cache contains vital information at that particular position. Frequently executed functions that map each other out of the cache will however still reduce performance. This “solution” is both expensive and does not handle the actual problem.

McFarling also presented a far more interesting work in 1989 with his article
“Program Optimisations for Instruction Caches” [7]. Here McFarling uses profiling information to reposition code, as was done in the graph colouring algorithm by Hashemi [4].

In the same year, 1989, Whu and Chang [8] presented another algorithm based on the same principle. The cache mapping conflict was minimized by placing functions with overlapping lifetimes into memory locations that do not contend with each other in the cache.

Separating functions and code sections into different optimisation interest categories have shown great results in many reports. This was done for instance in Cohn’s article from 1996 about optimisations of large Windows/NT applications [16], and in the graph colouring algorithm mentioned above. The simplest of these approaches is to divide the program functions into two such categories: one that is aimed for optimisation and one that is not, i.e. hot and cold code sections or functions.

2.4 Just-In-Time Compilation

In some circumstances a JIT compiler environment contains many advantages compared to the AOT compiler. For instance, runtime information such as register contents may be used to evaluate conditional branches and create fall-through execution sequences for special machine states. Non-executed code will never be compiled or occupy memory space. Code can be deleted to free space and later be recompiled to adapt to new environmental conditions. All this makes the code adaptable to changes in the process execution profile, and the code may be tuned to run efficiently on different data. Researchers have shown the benefits of optimising the code layout to improve utilisation of the cache. The same principles used in AOT compilers may be used in a JIT compilation environment, with some changes.

The idea of replacing code has been successfully used in static compilation. It was for instance used in Hashemi’s graph colouring algorithm [4]. Using this idea it may also be possible to improve cache miss ratio in the JIT
environment. In order for this to work, a reduction in the calculation effort of the algorithm must be achieved. This can be done by considering the hot code only (i.e. the code most appropriate to optimise).

Since 1997 there exist algorithms for improving instruction code locality in JIT environments [11], when Chen and Leupen introduced the JITCL algorithm. It dynamically positions the code while running. Only executed code will be compiled, and consecutively executed functions will be placed next to each other according to activation order (first time executed order). This does not improve cache miss rates caused by collisions of hot functions in cache, but it shows that algorithms for cache utilisation optimisation also exist in the JIT compiler environment.

In an article on FDO, published in 2000 by Michael Smith [13], the benefits of focusing the optimisations on frequently executed portions of programs are stressed and explained, as well as the use of profiler instrument in order to do so.

2.5 New Algorithm - LHCPA

By using the ideas from AOT and JIT compilation methods for improving cache performance, a new algorithm was created. The new algorithm suggested below is named “Linear Hot Code Positioning Algorithm” or just short LHCPA.

The heavy profiling used in many of the AOT algorithms to build call graphs upon which the replacement is calculated, is not an option in this type of environment. Smaller profiling tasks, such as logging the number of calls to each function, can be acceptable from a performance perspective. They are however still considered to be expensive. This profiling may be turned on or off depending on the load of the processor, in order to minimise the impact of the profiling. The result of this limited profiling provides a base for determining what code is the most appropriate to optimise. Profiling is an easy alternative to get highly interesting information about the actual behaviour of the process. Adapting the code to new environmental
properties, such as changes in program behaviour, is a fundamental prop-
erty of the JIT compiler environment. The compiler should make an effort
not to change that property.

Let us consider the most frequently executed functions for optimisation,
see Figure 2.4, found by a minimal function profiling. These functions are
the ones most likely to cause serious cache performance degradation, as a
result of mapping each other out of cache. To prevent this, the functions
have to be placed in memory so that they do not contend in cache. It is
always known in a JIT compiler where generated code is to be placed in the
memory. Therefore it is possible to calculate to what sets and set offsets the
generated code is mapped in the cache, see Section 2.2. The replacement
policy will decide which line to use within the set. It is controlled by
hardware only. By considering previous placements of other hot functions,
and placing hot functions at addresses mapped to non-occupied locations
in cache, conflicts in the cache between hot functions can be reduced.

In a direct mapped cache, the conflicts between hot functions can be not
only reduced but also completely removed since we do not have to rely on
any replacement hardware making the right decisions. In set-associative
caches it is desired to make the best use of cache space in order to maintain
performance. To achieve this, all lines in each set must be used to store
hot functions. This makes a demand upon the replacement algorithm, i.e.
to replace the right line at the right time.

This idea will only reduce the I-cache conflicts between the considered
hot functions. Cold functions (i.e. less frequently executed functions not
optimised by the LHCPA) may map out each other and hot functions, thus
causing I-cache misses. The idea however is that these cold functions are
not executed as often, and do not affect process performance as much as
hot functions do. The sizes of the cache and the functions influence the
number of functions that can be considered hot. The more functions that
can be considered hot, the more percentage of the complete process code
may be optimised and less conflicts will arise.

Positioning of compiled hot code is based on previous hot code placement.
Figure 2.4: Hot function selection from profile data
This becomes possible by implementing a virtual cache, which corresponds to the actual level-1 cache hardware in aspect of size, degree of set-associative and line size. By storing placements of previously compiled hot code at their mapped locations in the virtual cache, it is possible to decide where a function can be placed or is best placed. Some requirements are put on the compiler. [1. Known function size] Compiled function code size must be known before placement occurs, to ensure that no other conflicts with other hot functions can occur. [2. Free positioning] The memory allocation system must allow the functions to be placed within the whole process memory area, enabling code positioning at all positions within the cache.

The new algorithm does not consider data placement, because most modern processors have split data and instruction caches at level 1. It is however possible to make smaller adjustments in the algorithm to consider placement of important data. This can be done in the same way as is done with hot functions in case of a unified cache at level 1. Nor will this algorithm consider the memory system design at higher levels, since it would then have to consider both data placement and possibly multiple sizes of caches. This would generate a too complex algorithm that does not fit inside time constraints of JIT compiler systems.

Since the process execution profile may differ from execution to execution, it is important to use fresh profile information. This can be achieved by starting to execute the program without the algorithm turned on from the beginning. When an appropriate amount of profile information has been gathered, the algorithm is turned on while the profiling may continue or stop. The $x$ most frequently executed functions are selected as hot and treated specially by the algorithm. The $x$ is selected in such a way that the code of the $x$ functions fit inside the cache all at once, see Figure 2.4. The black boxes represent the hot functions and are to be positioned by the heuristic optimisation algorithm.

Looking at a small example in Figure 2.5, the A example is the function code mapping of hot and cold functions to the cache in a JIT compiler without any optimisation. The B example is the result of the heuristic
algorithm LHCPA executed on a machine with a 2-way set-associative cache. The same cache is used in both example A and example B. The black boxes represent hot functions and the grey ones represent cold functions. The compile time for boxes to the right or below is later than boxes to the left and up. Looking at the first two columns, nothing is changed between example A and B, since a 2-way set-associative cache has room for code from two hot functions in each set, without them competing for the same cache line. The hot function in column 3 is repositioned, since it would otherwise map out parts of the already positioned code from hot functions in column 1 and 2 in set 7 and 8. The following hot functions are all repositioned in order to avoid collisions in cache in the same fashion.

A search through the address space is conducted in the virtual cache starting from the last address code was generated to, in search of a suitable location. If the size of the $x$ functions selected as hot is smaller than the
cache, it is always possible to place a function if it can be divided into smaller parts using jumps between them. The extra space required for the jump instruction must be considered here. The pseudo code of the algorithm functions is presented in Algorithm 2.5.1, 2.5.2 and 2.5.3. The algorithm would be described as an greedy algorithm by Russel and Norvig [21].
Algorithm 2.5.1 Pseudo algorithm - LHCPA

LHCPA(startAddress, function)

1 /* startAddress, returnAddress, jumpAddress = pointers into memory */
2 /* function = instructions[1...sizeof(function)] */
3 /* global variable: abstractCache */
4 /* startSet, currentSet, functionSize = type integer */
5 /* Sets = *constant* 2^γ/2^w/k */
6 /* Function findFreeSet finds first free set in abstractCache and its address based on the address parameter. */
7 startAddress ← findFreeSet(startAddress)
8 returnAddress ← placeCompleteFunction(startAddress, function)
9 if returnAddress ≠ NULL
10      then return returnAddress
11 else placePartFunction(startAddress, function)
12      return startAddress

In the functions in Algorithm 2.5.1, 2.5.2 and 2.5.3 there exist several functions and variables used that needs to be explained. The most important variable is the abstractCache which contain all information about the placements of the hot functions. It consists of a structure of sets, each set contains lines and each line contains information about the locations of the instructions within, as described in Section 2.2. The function variable is the compiled part of the function that needs to be positioned in memory. It consists of an array of instructions. Both these variables are limited in their size by the cache size $O(2^γ)$. The other variables are explained in the pseudo code and are either pointers or integers. In Figure 2.6 and 2.7 the pointers usage is explained. The A case in Figure 2.6 is the start situation when LHCPA is called upon. The B case is before returning from the placeCompleteFunction algorithm. Figure 2.7 is the case when
Algorithm 2.5.2 Pseudo algorithm - LHCPA: placeCompleteFunction

PLACECOMPLETEFUNCTION\newline(startAddress, function) \newline
1 /* See LHCPA pseudo code variable descriptions. */ \newline
2 startSet ← MAPADDRESSTOSETNO(startAddress) \newline
3 currentSet ← startSet \newline
4 currentAddress ← startAddress; \newline
5 functionSize ← sizeof(function); \newline
6 while functionSize > 0 \newline
7 do if MOD(currentSet + 1, Sets) = startSet \newline
8 then return NULL \newline
9 if isSetFull(currentSet) \newline
10 then startAddress ← FIRSTADDRESSINNEXTSET( \newline
11 currentAddress) \newline
12 currentSet ← MAPADDRESSTOSETNO( \newline
13 startAddress) \newline
14 currentAddress ← startAddress \newline
15 functionSize ← sizeof(function); \newline
16 else functionSize ← functionSize − 2^w + \newline
17 MOD(startAddress, 2^w) \newline
18 currentAddress ← FIRSTADDRESSINNEXTSET( \newline
19 currentAddress) \newline
20 currentSet ← MAPADDRESSTOSETNO( \newline
21 currentAddress) \newline
22 if functionSize = 0 \newline
23 then COPY(function, sizeof(function), \newline
24 startAddress) \newline
25 /* Function addFunction adds entire function to abstractCache */ \newline
26 ADDFUNCTION(sizeof(function), \newline
27 startAddress) \newline
28 return startAddress
Algorithm 2.5.3 Pseudo algorithm - LHCPA:placePartFunction

PLACEPARTFUNCTION(startAddress, function)
1 /* See LHCPA pseudo code variable descriptions. */
2 startSet ← MAPADDRESSTOSETNO(startAddress)
3 currentSet ← startSet
4 currentAddress ← startAddress;
5 functionSize ← SIZEOF(function);
6 while functionSize > 0
7    do if ISSETFULL(currentSet)
8       then writeSize ← (currentAddress − 1 − StartAddress −
9                              SIZEOF(jumpInstruction))
10          COPY(function, writeSize, startAddress)
11          ADDFUNCTION(writeSize, startAddress)
12          UPDATEFUNCTION(function, −WriteSize)
13          /* Save jumpAddress for backpatching. */
14          jumpAddress ← currentAddress − 1−
15          SIZEOF(jumpInstruction)
16          currentAddress ← FINDFREESET(currentAddress)
17          currentSet ← MAPADDRESSTOSETNO(
18              startAddress)
19          BACKPATCH(jumpAddress, startAddress)
20    else functionSize ← MAX(((functionSize − 2^w) +
21                             MOD(startAddress, 2^w)), 0)
22          currentAddress ← FIRSTADDRESSINNEXTSET(
23              currentAddress)
24          currentSet ← MAPADDRESSTOSETNO(
25              currentAddress)
26    if functionSize = 0
27       then writeSize ← SIZEOF(function)
28          COPY(function, writeSize, startAddress)
29          ADDFUNCTION(writeSize, startAddress)
30          UPDATEFUNCTION(function, −writeSize)
31       return
placePartFunction is returning and the code of a function is divided into two parts. In both the figures the black boxes represents the newly added function and the grey already compiled functions.

![Diagram showing LHCPA: Pointer clarification in placeCompleteFunction](image)

Figure 2.6: LHCPA: Pointer clarification in placeCompleteFunction

In Algorithm 2.5.1 a function called FindFreeSet is called at line 7. This function simply performs a linear search in the sets of the abstractCache in search of a free line. This search is started at the parameter address sent to the function. Since LHCPA requires that all code considered hot must fit inside the cache, there will always be free lines where code can be positioned. The worst case search is therefore limited by the number of lines in the abstractCache and can be expressed as $O(2^{\gamma-w})$.

In Algorithms 2.5.2 and 2.5.3 several functions needs explaining. MapAddressToSetNo translates the parameter address into the set it is mapped to in the abstractCache by using the formulas in Section 2.2. It is a simple mathematical function and its execution time does not depend
on input length and therefore is limited to $\Theta(1)$. The function $\text{IsSetFull}$ checks the abstract cache if the set pointed out by the parameter contains any free lines. This is only a lookup operation that also belongs to the $\Theta(1)$ execution time complexity class. $\text{FirstAddressInNextSet}$ is a function that calculates the first address in the set after that to which the parameter address is mapped to. This function is also a simple mathematical function in the $\Theta(1)$ time complexity class. The function $\text{addFunction}$ updates all sets and lines in the $\text{abstractCache}$ that the function is occupying. The execution time is thus limited by the function size and the updates to the sets it is positioned to; $O(\psi + 2^s)$ where $\psi = \text{sizeof(function)}$. The function $\text{updateFunction}$ in Algorithm 2.5.3 removes the first part of the instruction array from the $\text{function}$ according to the size of the second parameter. This function is limited in its execution time by the same conditions as the $\text{addFunction}$. The function $\text{backPatch}$ updates the jump address to

<table>
<thead>
<tr>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
</tbody>
</table>

Figure 2.7: LHCPA: Pointer clarification in placePartFunction
2.5. New Algorithm - LHCPA

the new code part positioned from the old, as the location of the new code part was unknown when the old was written to memory. This is a single write instruction and it belongs to the $\Theta(1)$ time complexity class. When breaking apart functions it is important to update the program-counter-relative address used within the function that are to be positioned so they do not point incorrectly. This is not included as a function in the algorithm description of Algorithm 2.5.3 but it would also belong to the $O(2^\gamma)$ complexity class.

In both Algorithm 2.5.2 and Algorithm 2.5.3 there is a function called $COPY$, this function only copies the function instructions from the function variable to the memory with size and location according to the parameters. The complexity of this function is irrelevant since the copy into memory has to be done in all compilation. It does not matter if the LHCPA algorithm is used or not.

The complete Algorithms 2.5.2 and 2.5.3 both belong to the same time complexity class, $O(2^\gamma)$, since the while-statement conditions limits the actual complexity of the functions $addFunction$ and $updateFunction$. Since Algorithm 2.5.1 executes at most both the algorithms one time, the time complexity of the complete LHCPA algorithm is $O(2^\gamma + 2^\gamma/w) \subseteq O(2^\gamma)$. Note that the notation is the same as in Section 2.2.
Chapter 3

Case Study

In order to evaluate the LHCPA algorithm, a case study was made. An incomplete implementation of the heuristic algorithm LHCPA has been tested on a Alpha EV67 processor in a Unix environment system at Ericsson in Mjärdevi, Linköping.

3.1 Alpha Hardware and ASA Compiler

The memory system of the Alpha processor can be described with the parameters in Figure 3.1. The parameters can be used inside the memory model described earlier.

This memory architecture allows us to implement an algorithm that does not consider data placement, since data and instructions are separated at cache level 1. Instructions are however still affected by the data placement as the cache level 2 is unified. This was not considered during this case study, as a result of compilation time restrictions within the JIT Compiler.

The ASA compiler, see Section 1.1, has a few special properties that does not allow a full implementation of the LHCPA algorithm described in the previous chapter. Neither of the two requirements ([1. Known function size],
3.1. Alpha Hardware and ASA Compiler

### Cache Layout

<table>
<thead>
<tr>
<th></th>
<th>Level 1</th>
<th>Level 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction word size</td>
<td>32 bits</td>
<td></td>
</tr>
<tr>
<td>instruction alignment policy</td>
<td>by word</td>
<td></td>
</tr>
<tr>
<td>cache levels</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>data/instruction separation</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>layout</td>
<td>2-way set-associative</td>
<td>direct mapped</td>
</tr>
<tr>
<td>size</td>
<td>32 Kb x2</td>
<td>8 Mb</td>
</tr>
<tr>
<td>line size</td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td>load delay [CPU cycles]</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Cache replacement policy</td>
<td>LRU</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 3.1: Alpha EV67 cache memory layout

[2. Free positioning]) needed for the algorithm to work properly are fulfilled. Still the algorithm may reduce the cache miss rate, as other possibilities exist to work around some of the problems concerning these non-fulfilled requirements.

To see how the requirement was not fulfilled but worked around in some sense, a better understanding of the system must be obtained. Before the switch of hardware in Ericsson AXE exchanges, the AXE OS ran on top of the APZ hardware. After the swap the AXE OS runs on top of the APZ VM containing the ASA Compiler, both running in a True 64 Unix system on top of the Alpha hardware. The relations between the APZ VM, ASA Compiler and JIT compiled code is explained in the internal document “The APZ 212 40 ASA Compiler [14]”. Functions that are to be executed are called by the AXE OS to the APZ VM; depending on the status of this function (compiled or not) APZ VM either calls the ASA Compiler to compile the function or calls the code directly, see Figure 3.2.

If the APZ VM calls the ASA Compiler, the compiler compiles the function using current dynamic information to decide the fall-through path in the function. All paths not taken are compiled into exit points called stubs...
into the ASA Compiler. Whenever a condition may not be evaluated, compilation stops and another exit point is generated. The code executes until an encountered stub and the compiler gets invoked again. The ASA Compiler uses the new dynamic information from the execution and continues to compile the next taken part of the function. The size of the code for a signal (function) may differ, as it uses dynamic information to decide fall-through path when compiling. This combined with the fact that code not executed will never be compiled, results in the unpredictability of the total signal (function) size. This compile method is actually an advanced form of the algorithm presented by Michael Smith [13], described earlier in the JIT Chapter 2.4.

Since a JIT Compiler writes directly into main memory, it must keep track of all code and data. In the ASA Compiler case, this is done with two classes called "data allocator" and "code allocator". Each program class (called "Block" in ASA) own a unique instance of both the "data allocator"
and "code allocator", enabling easy de-allocation of all data or code for every compiled class. This however makes it impossible to place code outside the allocated area of each "code allocator", making the [2. Free positioning] requirement of the heuristic algorithm unfulfilled. This makes it hard to place functions at desired locations in cache according to the new heuristic algorithm.

The unpredictability of the function size does not necessarily make it impossible to achieve the goal of the algorithm since it only takes into consideration the already placed hot code of the previously compiled hot functions. Every time a sequence of instructions is written to main memory the algorithm can check if the code will compete for the same cache lines as other hot code sequences. If it does, it may be relocated. This functionality is however not implemented. Only the beginning of the functions is really repositioned in the implemented algorithm, in order to not compete with any other already positioned hot code segments. This would imply that only Algorithm 2.5.1 and 2.5.2 was implemented and not Algorithm 2.5.3.

The choice of not implementing the replacement in all cases depends on two things. First, it would generate a lot of jumps in the code, reducing the performance of the prediction hardware within the hot functions. Secondly, it may be impossible to place the code wherever the algorithm suggested due to the unfulfilled [2. Free positioning] requirement, and thus not guaranteeing a proper placement anyway.

Other optimisations have already been implemented, such as for instance another type of code protection concerning hot code inside the APZ VM. Some code inside APZ VM is very frequently executed and if replaced by any JIT compiled code, performance would suffer gravely. This is a well working optimisation that is based on the same basic idea that this heuristic algorithm is based on. The difference is the consideration between two different processes instead of between functions within a process. The two different approaches differ as follows. The code protected in the APZ VM case does never change and is always considered equally important to protect from other code. The functions considered hot within the JIT Compiler may differ during time, both concerning code origin (hot function
3.2 Results

Several different types of tests were performed. Two different applications were tested in order to evaluate the algorithm. The two different applications are the cms88 and cme20, both telecom switching applications (Plex-C compiled into ASA assembly language) but with different functionality. The important difference between them is the size of the active code footprint. The cme20 application code footprint is almost three times as large as the code footprint of the cms88 application.

To be able to evaluate the test results of the LHCPA, some tests with and without the algorithm had to be made for comparison. Another approach of optimisation based on the LHCPA, from now on referred to as LHCPA-E (Extended), was also tested (the algorithm is described later in this Chapter). Three tests were conducted for all selections of application and algorithm, with one exception. The tests were very time consuming and LHCPA-E was reducing cache performance instead of increasing it, and thus excluded from more testing. The reason for making three tests in each case was that no test is equal to any other concerning the input (incoming calls and signals) and thus neither the output.

Input data affects the performance in many ways. The input data is, in this case, the application traffic. The traffic decides which functions are compiled and what parts of these functions are compiled. The process code size is thus dependent on the traffic. The order of compilation, dependent on incoming calls, decides in what order functions are placed, giving different placements in each test run. Therefore no test is representative to all test cases but a number of tests may show a tendency of the cache performance.

Each test gave an extensive amount of data: total cycles lost to cache misses, total code size, function profiling data and mapping of hot functions to the cache. The results are shown in Table 3.1 and they are the basis for the analysis below. Profiling were performed on all tests in order to collect
3.2. Results

data about the test runs for better analysis. Placement of hot functions were also logged for the same reason. In test case 1-3 and 10-12 (original placement) there were no alterations of the code positioning.

Table 3.1 is quite large and needs some explanation. The three first columns enumerate each test and describe what application was executed and with which placement algorithm the test was performed. The fourth column describes the generated code size in Kb. The fifth column describes the number of different functions executed during the profiling and the sixth the selected number of hot functions based on the same profiling. Please note that in the sixth and seventh column some numbers are within parenthesis to indicate that these are not actually treated as hot. The seventh column contains the execution count of hot functions in % of all executed functions. Column number eight contains the total amount of cycles executed during measurement with the performance counter tool (DCPI). The last two columns describe the total amount of lost cycles due to cache misses specific to each test and the average within each group (same algorithm and application).

To interpret the results correctly it is important to understand that the two last columns account for all types of instruction cache misses. There are three types of cache misses: cold start, capacity and collision. The cold start describes the case where the code has never been executed previously and is loaded for the first time. Cache misses due to capacity occurs when the code footprint is larger than the cache itself i.e. all code cannot reside in the cache all at once and code has to be read in again and again. Collisions describe the situation where two or more functions are situated in memory space so that they compete for the same cache lines and map each other out. The collision case is often combined with the capacity case since the code can be positioned not to collide if there is enough space. The performance counters were in each test started after all the functions were compiled and thereby executed since this is a JIT compiler. Therefore the only cold start effect possible seen in the result table are those resulting from process switching.

Looking at the results between tests 1 to 3 and tests 10 to 12 some large
## LHCPA - Test results

<table>
<thead>
<tr>
<th>Test nr</th>
<th>App.</th>
<th>Algorithm</th>
<th>Code size [KB]</th>
<th>No of funct.</th>
<th>No of hot funct.</th>
<th>Exec. count of hot funct. [%]</th>
<th>Total cycles</th>
<th>Lost cycles [%]</th>
<th>Lost c. [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cms88</td>
<td>Original</td>
<td>10723</td>
<td>900</td>
<td>(6)</td>
<td>(15.36)</td>
<td>708573</td>
<td>10.26</td>
<td>9.83</td>
</tr>
<tr>
<td>2</td>
<td>cms88</td>
<td>Original</td>
<td>10633</td>
<td>917</td>
<td>(6)</td>
<td>(15.22)</td>
<td>611636</td>
<td>8.70</td>
<td>9.29</td>
</tr>
<tr>
<td>3</td>
<td>cms88</td>
<td>Original</td>
<td>10574</td>
<td>896</td>
<td>(6)</td>
<td>(15.26)</td>
<td>693474</td>
<td>10.53</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>cms88</td>
<td>LHCPA</td>
<td>10692</td>
<td>953</td>
<td>6</td>
<td>15.49</td>
<td>699989</td>
<td>9.17</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>cms88</td>
<td>LHCPA</td>
<td>10633</td>
<td>904</td>
<td>6</td>
<td>15.52</td>
<td>695762</td>
<td>9.28</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>cms88</td>
<td>LHCPA</td>
<td>10604</td>
<td>912</td>
<td>6</td>
<td>15.52</td>
<td>680506</td>
<td>9.41</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>cms88</td>
<td>LHCPA-E</td>
<td>10780</td>
<td>906</td>
<td>18</td>
<td>23.97</td>
<td>1088771</td>
<td>12.04</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>cms88</td>
<td>LHCPA-E</td>
<td>10745</td>
<td>904</td>
<td>18</td>
<td>24.16</td>
<td>695001</td>
<td>10.42</td>
<td>11.18</td>
</tr>
<tr>
<td>9</td>
<td>cms88</td>
<td>LHCPA-E</td>
<td>10456</td>
<td>919</td>
<td>18</td>
<td>23.87</td>
<td>854704</td>
<td>11.07</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>cme20</td>
<td>Original</td>
<td>29051</td>
<td>2206</td>
<td>(6)</td>
<td>(9.59)</td>
<td>842156</td>
<td>26.20</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>cme20</td>
<td>Original</td>
<td>29256</td>
<td>2202</td>
<td>(6)</td>
<td>(9.59)</td>
<td>885962</td>
<td>24.40</td>
<td>25.61</td>
</tr>
<tr>
<td>12</td>
<td>cme20</td>
<td>Original</td>
<td>30457</td>
<td>2197</td>
<td>(6)</td>
<td>(9.59)</td>
<td>939853</td>
<td>26.23</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>cme20</td>
<td>LHCPA</td>
<td>29698</td>
<td>2196</td>
<td>6</td>
<td>9.59</td>
<td>939853</td>
<td>22.60</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>cme20</td>
<td>LHCPA</td>
<td>29260</td>
<td>2211</td>
<td>6</td>
<td>9.59</td>
<td>1013425</td>
<td>23.39</td>
<td>23.47</td>
</tr>
<tr>
<td>15</td>
<td>cme20</td>
<td>LHCPA</td>
<td>29462</td>
<td>2204</td>
<td>6</td>
<td>9.58</td>
<td>1069812</td>
<td>24.41</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Test result table
3.2. Results

Differences are noticed. The number of different functions are more than twice as many in test 10 to 12 than tests 1-3 due to the application switch. The code size is almost three times as large and the average lost cycles percentage is about 2.5 times as large. Since more code competes for the same area in the cache when the code footprint is enlarged, more conflicts and more lost cycles due to cache misses are to be expected. This is clearly shown in the result table.

Tests 1 to 3, when compared to tests 4 to 6, also differ in other ways. The switch of the positioning algorithm does not affect code size or number of functions, but should, according to our idea in this report, affect the cache performance. Considering the same number of functions as hot and verifying that approximately the same percentage of all executed functions are hot in all tests 1 to 6, the positioning is the main factor that influences the cache performance in these tests. In average the cache misses are reduced by 5.5% when using the new algorithm LHCPA. Looking at each test result for all tests, some things stand out. Test 2 for instance has a lower cache miss ratio than all the other tests 1 to 6. The code positioning done in this case is therefore supposedly better than in the other tests. This can occur even though there is no clear strategy for placement (Original positioning). The number of lost cycles in tests 4-6 are about the same, possibly indicating that LHCPA handles the code similarly each time. LHCPA positioning increase average cache performance even though there exist clearly better placement strategies and room for improvement.

When comparing tests 10 to 12 with tests 13 to 15 there are differences in code size and number of functions affecting the effectiveness of the optimisation. Here we can see that the average improvement is approximately 8.4%. LHCPA increased its effectiveness in tests 10 to 15 more than in tests 1 to 6 even though the code size increased and the part of code considered hot and treated specially by LHCPA decreased in percentage. This was not expected and may indicate that other factors not considered here might influence even more than the hot functions property exploited by LHCPA.

An early discovered problem was that only a few functions were able to
be considered as hot due to their considerate size and the limited cache size. Even though a function is considered hot it may contain infrequently executed code parts handling input data exceptions etc. In LHCPA these code parts are considered hot as there exists no possible way to locate these parts and exclude them from the hot code with help from the limited profiling done on the function level. One idea is therefore to make a “guess” of which part is most likely hot and only consider that part as hot. This was done in the extended version of LHCPA, named LHCPA-E. The “guess” was guided by the JIT compiler nature, by considering only the first executed fall through path in each function, see Chapter 2.4. In tests 7 to 9 the LHCPA-E was tested and it should be compared to tests 1 to 6 as the same application was used. The results were not as expected and it actually decreased cache performance. There are several possibilities of why this did not work as intended. One explanation might be the fact that the first taken path may not be the most frequently executed path and thus should not always be considered hot. Potentially the really hot code in the functions was not considered hot at all. Another explanation might be the increased density of unconsidered hot code within the same cache area, causing unconsidered hot code to collide with more hot code from other hot functions. This would directly show up as an increase of the cache miss ratio percentage.

All the above analysis was performed with the assumption that the tests performed in some sense described the average case of positioning and the average resulting cache performance. This is clearly a naive way of looking at it since three tests in each group is far to little data to form a reliable average behaviour in this case. Only the spread of lost cycles in the first group is in its own indicating that much more testing is needed in order to show with certainty that the LHCPA is actually improving performance. Often there are special cases in which prefered algorithms fail to improve performance. All cases of input can of course not be tested, as the input data space is very large. Increasing the number of tests to at least five or ten times more than used in this study would make a better foundation to build reliable average performance assumptions. This could not be done since the tests in this system was very time consuming. Every test needed a long initialisation time and also a lot of interaction with many different
servers and programs was needed. During the tests the performance of the processes had to be closely monitored to ensure that nothing went wrong in the unstable test environment.

As there was no performance counter in the Alpha processor covering only cache misses as result of collisions, there exist cache misses of the other two types of unknown amount in the result data. The effectiveness of the LHCPA heuristic algorithm only depends on the amount of cache misses resulting from collisions. Since the real amount of this type of cache misses is unknown, the real improvement in cache performance by LHCPA is hard to estimate without further testing and with other types of performance counters.
Chapter 4

Conclusions and future work

The case study was in many ways a success. It confirmed many expectations and raised some questions not considered in advance. The topic of cache optimisation is still developing even in AOT compiler environments. A recent article “FICO: A Fast Instruction Cache Optimizer” by Marco Garatti [23] shows that new approaches are still developed. In the JIT compiler environment, the development of these types of optimizations has only just begun.

4.1 Thoughts and Conclusions

A smaller than anticipated but noticeable positive effect was shown by the tests performed in the case study. This despite the fact that there was no guarantee of proper placement of hot code by the incomplete implementation. The unfulfilled requirements ([1. known function size], [2. Free positioning], see Section 2.5) made a full implementation impossible, see Section 2.5, but the test shows that the approach is promising, producing a more appropriate placement of hot functions on the average than with the original placement algorithm.
4.1. Thoughts and Conclusions

A full implementation would possibly improve the cache performance even more than the achieved 5 – 8% reduced I-cache miss ratio as it would guarantee that hot functions do not contend in cache. There are though still several problems with this approach. For instance there is large uncertainty about the average assumption of the tests results. Much more testing is required in order to pinpoint the actual effect of the LHCPA on cache performance.

One of the biggest problems of the LHCPA is that the algorithm tries to protect too large segments of code (complete functions) from each other. A function may be executed a lot, but big parts of it may not be executed at all due to special input cases etc. If profile data could be more detailed, a better placement strategy may be constructed. The problem is then transformed into getting the detailed profile data. By protecting complete functions from each other, too few functions may be considered for this method to be really effective.

Another aspect is the use of the algorithm on programs without specific hot functions, i.e. the execution frequency is similar in all, or too many functions. Without distinctive hot functions, we would not gain cache performance in treating some functions in a different way. If all the functions in a large code footprint process are equally important to optimise, proper placement is still important. This will though be much harder to do properly without heavy profiling and use of static compilation techniques.

Even though two functions are considered hot, they may be placed in the same cache sets if they are not executed in the same phase of the process. A Phase in this case could be a time interval during the execution. Separating these in cache would only be a “waste” of cache space. There is however no chance of knowing this simply by looking at one sample of the functions’ execution profile data. A set of profile data may reveal such code properties without increasing the execution cost of profiling.

The placement algorithm (LHCPA) is likely to place code in a wider address space since holes are introduced when hot functions are relocated to higher addresses. The code density is then affected, but only between functions
and not inside them. This is a small density change and will not change the cache performance noticeably [9].

In 1989, Scott McFarling [7] stated that caches were too small to use this approach since the functions as units are just too large. Function units have not decreased in size but caches have increased allowing this approach to give better result today than 1989. One problem with that approach was that there already existed better solutions in AOT compilers for the same problem, while for a JIT compiler there does not exist a variety of cheap working online algorithms with these results.

Larger caches have less to gain in cache performance from code positioning algorithms since a larger amount of code can reside in cache at all time. Thus it is more unlikely that hot functions compete for the same cache lines [10]. Larger caches are both more expensive and slower, causing medium size caches to be more frequently used in computers.

When only considering one system, tuning can affect the cache performance a lot. Traditional compilers are built to handle many different programs and systems on similar hardware. JIT Compilers are often tuned particularly to specific environments and therefore have a lot of optimisations only for that hardware or software. Tuning is essential in any such system to get the most out of the optimisation. In the ASA Compiler case, a lot of further tuning can be made, which is important to keep in mind when interpreting the results in the case study, see Chapter 3.2. Additional tuning can help resolving problems that interfere other optimisation processes, further tuning may therefore influence the results positively.

The Alpha processor has a number of hardware optimisations for cache utilisation. These hardware optimisations prefetch instructions needed ahead of execution so that less cache misses occur. The true impact on cache performance of the LHCPA algorithm on a processor without these hardware optimisations is hard to predict. Since these hardware optimisations mostly reduce cache misses inside functions and have problems making predictions over branches and function calls, it is likely that the results would look like the tests performed even without the hardware optimisa-
4.2 Future work and further research

Since all protective functionality around the hot code was not implemented, see Section 2.5, the first step would be to examine the true impact of the complete algorithm. This is not possible in the ASA Compiler due to its current limitations of its architecture. Therefore the algorithm has to be implemented in some other JIT compiler such as for instance a Java compiler on Alpha or other similar hardware. This would perhaps also help in the evaluation of the algorithm since it may increase testability and reduce the number of required surrounding systems. One advantage with the Java compiler environments is that complete functions are compiled at once.

The impact of the algorithm on different hardware with different memory systems would also be of interest since it would tell if this algorithm is worth refining. Such refinements could be improved profiling details on selected functions allowing even better placement decisions. The introduction of function classification levels between hot and cold, and the evaluation of the impact of different placement strategies at different levels would aid in the problem understanding. If it would be possible to compare and draw intelligent conclusions by comparing different profile data, it may be possible to find relationships between functions that are currently not yet exploited, in contrast to many static compilation methods.

AOT compilers have many refined methods and algorithms that are interesting to study when developing optimisations in the JIT environment.

JIT compilers will increase in popularity since they allow very dynamic and problem specific code optimisations that in many cases are impossible to achieve using AOT compilers.
Often the techniques used in AOT compilers may not be used but the same principles and shortcuts can often be exploited in other ways. The research around JIT compiler environment optimisations is intensifying as the JIT technique is gaining more attention in the academic world and in the industry.
Bibliography

[1] J. Hennesy, D. Patterson:
Computer Architecture A Quantitative Approach,
Third Edition (2003), Morgan Kaufmann Publishers,
San Francisco

Computer Organization & Architecture
Designing For Performance,
Sixth (International) Edition (2003), Prentice Hall,
New Jersey

Compilers Principles, Techniques and Tools,
(1986/88), Reproduced (2002), Addison Wesley,
Massachusetts

Efficient Procedure Mapping Using Cache Line Coloring
Proceeding of ACM (1997), Conference PLDI’97 Las Vegas,
pages 171-182

[5] Steven S. Muchnick:
Advanced Compiler Design Implementation,
(1997), Morgan Kaufmann Publishers,
San Francisco
[6] Alpha Architecture Committee:  
*Alpha Architecture Reference Manual*,  

[7] Scott McFarling:  
*Program Optimization for Instruction Caches*,  
ACM (1989), ASPLOS-III, pages 183-191

*Achieving High Instruction Cache Performance with an Optimizing Compiler*,  

[9] Peter Steenkiste:  
*The Impact of Code Density on Instruction Cache Performance*,  

[10] Chen W.Y. Chang, P.P., Conte, T.M., Hwu, W.W.:  
*The Effect of Code Expanding Optimizations on Instruction Cache Design*,  
IEEE Article (1993), IEE Trans. Comput (USA), pages 1045-57

*Improving Instruction Locality with Just-In-Time Code Layout*,  
Workshop (1997), USENIX Windows NT Workshop, pages 25-32

[12] A. Mendlson, S. P. Pinter, R. Shtokhamer:  
*Compile Time Instruction Cache Optimizations*,  
Proceedings (1994), Compiler Construction 5th International Conference, pages 404-418

[13] M. D. Smith:  
*Overcoming the Challenges to Feedback-Directed Optimization*,  
ACM (2000), Workshop on Dynamic and Adaptive Compilation and Optimization, pages 1-11
[14] Ericsson Staff:  
*The APZ 212 40 ASA Compiler*,  
Internal Documentation (2003), Revision B, Ericsson, Mjärdevi, Linköping

[15] S. McFarling:  
*Cache Replacement with Dynamic Exclusion*,  

[16] Robert Cohn:  
*Hot Cold Optimization of Large Windows/NT Applications*,  

[17] Ericsson Staff:  
*AXE 10 Survey*, EN/LZT 101 1513 R2A,  
Internal Documentation (1995), Ericsson, Stockholm

[18] Ericsson Staff:  
*Plex-C 1*, EN/LZT 101 1279 R6A,  
Internal Documentation (1998), Ericsson, Stockholm

[19] Ericsson Staff:  
*Plex-C 2*, EN/LZT 101 1280 R2A,  
Internal Documentation (1995), Ericsson, Stockholm

[20] Olof Roos:  
*Grundläggande datorteknik*,  

[21] Stuart Russel, Peter Norvig:  
*Artificial Intelligence*, A Modern Approach  

[22] Harald Abelson, Gerald Jay, Jullie Sussman:  
*Structure and Interpretation of Computer Programs*,  
[23] Marco Garatti:
   *FICO: A Fast Instruction Cache Optimizer*,
Proceedings (2003), SCOPES, page 388-402
This master’s thesis examines the possibility to heuristically optimise instruction cache performance in a Just-In-Time (JIT) compiler.

Programs that do not fit inside the cache all at once may suffer from cache misses as a result of frequently executed code segments competing for the same cache lines. A new heuristic algorithm LHCPA was created to place frequently executed code segments to avoid cache conflicts between them, reducing the overall cache misses and reducing the performance bottlenecks. Set-associative caches are taken into consideration and not only direct mapped caches.

In Ahead-Of-Time compilers (AOT), the problem with frequent cache misses is often avoided by using call graphs derived from profiling and more or less complex algorithms to estimate the performance for different placements approaches. This often results in heavy computation during compilation which is not accepted in a JIT compiler.

A case study is presented on an Alpha processor and an at Ericsson developed JIT Compiler. The results of the case study shows that cache performance can be improved using this technique but also that a lot of other factors influence the result of the cache performance. Such examples are whether the cache is set-associative or not; and especially the size of the cache highly influence the cache performance.
Copyright

Svenska

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under en länge tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera åktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Uphovsmannens ideella rätt innefattar rätt att bli nämnt som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart. För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Stefan Eng
Linköping, 20th February 2004