FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Optimierung von OpenJDK8 für ARMv8 64 bit

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software and Information Engineering

eingereicht von

Benedikt Wedenik
Matrikelnummer 1227151

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Wien, 2. Juli 2015

Benedikt Wedenik	Andreas Krall

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Optimization of OpenJDK8 for ARMv8 64 bit

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software and Information Engineering

by

Benedikt Wedenik
Registration Number 1227151

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Vienna, 2nd July, 2015

Benedikt Wedenik	Andreas Krall

Erklärung zur Verfassung der Arbeit

Benedikt Wedenik
Raffaelgasse 1A / 18, 1200 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 2. Juli 2015

Benedikt Wedenik

Kurzfassung

OpenJDK ist die offizielle Implementierung der Java Plattform und daher sehr populär, sowie weltweit in Verwendung. ARMv8 64 bit execution mode, genannt *aarch64*, ist eine im Jahr 2013 eingeführte Architektur, die momentan immer populärer wird. Die ARM Architektur ist besonders im Bereich der eingebetteten Systeme sehr stark verbreitet und es gibt einen Trend, diese Systeme auch im Serverbereich einzusetzen. In dieser Arbeit wird ein Überblick über OpenJDK8 für ARMv8 gegeben und Methoden zur Messung und Analyse der Performance vorgestellt. Des weiteren werden Ansätze gezeigt, mit denen der generierte Code optimiert werden kann, was an einem konkreten Beispiel illustriert wird, das die Performance bei gewissen Benchmarks um 15.2% steigert. Die Evaluierung der jeweiligen Performance wird duch den Vergleich der Ergebnisse des SPECjbb2005-Benchmarks durchgefürt. Zum Schluss werden die Vorteile dieser Optimierung aufgezeigt und zukünftige Forschung diskutiert.

Abstract

OpenJDK is the official implementation of the Java Platform and therefore very popular and frequently used. ARMv8 is an architecture introduced in 2013, which is constantly gaining popularity, notably the 64 bit execution mode is called *aarch64*. The ARM architecture is very popular in the field of embedded systems but there are also implementations targeted to server applications. In this work, an overview of OpenJDK8 for ARMv8 is given and methods to measure and analyze the performance are explained. Furthermore, ways of optimizing the generated code are shown and are illustrated on a concrete example, which increases the performance up to 15.2% on certain benchmarks. The evaluation of the respective performance is done by comparing the results of the SPECjbb2005 benchmark. Finally the advantages of this optimization and future work are discussed.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Motivation

The Advanced RISC Machine (ARM) v8 in the 64 bit execution mode, called "aarch64", is an architecture introduced in 2013. Aarch32 - the 32 bit execution mode of ARMv8 - as well as older ARM versions like ARMv7 are widely spread but aarch64 is constantly gaining popularity and replacing older hardware. It is the most frequently used architecture in the field of embedded systems and also the vast majority of todays smarthpones and tablets are using the ARM technology. Java is one of the most popular programming languages and it runs on many platforms. According to the "RedMonk-Index 2015", which is a global index for comparing the usage of programming languages, Java is ranked as language number 1 [O'G15]. The "TIOBE-Index" is another worldwide ranking index for the popularity of programming languages. Java was ranked number one in august 2013 and is ranked number two in march 2015, behind C [McM13], [Com15].

OpenJDK is the official free implementation of the Java platform and also the most commonly used Open Source Java implementation, available for almost any Java supporting platform (eg. Linux, Solaris, Windows Client, Windows Server, Mac OS X, Firefox, Chrome, Safari, etc.). It is written in C++ and Java and is distributed under the GNU General Public License (GPL) by Oracle, which is the main contributor. Oracle announced in 2008 that OpenJDK will be the official replacement for the non-free Java Development Kit (JDK).

When speaking about Java technology there are basically three components:
The *programming language* Java to write programs, the *JDK* to build programs and the *Java Virtual Machine (JVM)*, which is a standardized software platform, is needed to execute Java programs.

The need of an implementation, which reduces *execution times*, as well as *memory and energy consumption* and therefore *costs*, is obvious. These two factors are important performance metrics which need to be considered when speaking about optimizations. A version for aarch64 exists but it is not yet able to compete with the one for common x86 architectures when comparing systems with the same number of cores and the same clockspeed with Java benchmarks like SPECjbb2005 [Cor05] or DaCapo [Pro09]. Therefore, this work aims to optimize the OpenJDK for aarch64 to gain execution speed and reduce memory consumption, which causes a reduction of energy consumption as well.

1.2 Problem statement

Program code, written in a programming language needs a compiler or interpreter (or a combination of both) to be executed on a computer. A compiler translates one language (program code) into another one (machine readable instructions or "binary code"). Optimized program code still depends on a well optimized compiler, because the final application will only perform as well, as the compiler generated binary code can be executed. As a logical consequence a well optimized compiler speeds up almost any application and / or reduces the application's memory consumption.

The direct impact of compilers on the code performance, necessitates the science of "compiler construction". Compilers are platform dependent and so they need to deal with strengths and weaknesses of the underlaying system. Most modern compilers use different stages of optimization which can be separated into "machine dependent" and "machine independent". The independent part uses a so called Intermediate Representation (IR) - which allows to perform general and generic optimizations. The platform dependent part needs to know the exact machine model to be able to select the best register allocation, to know which instructions are cheap and which are expensive and a lot more. As an example *loop unrolling* and *dead code elimination* are machine independent optimizations, while *vectorization* depends on the available registers of the respective architecture. This work will only focus on the "machine dependent" part of OpenJDK optimizations for ARMv8 64 bit architecture.

1.3 Aim of the work

To determine how well a compiler performs on a specific platform, benchmarks can be used for a comparison. Intentionally these benchmarks are built to compare two systems, or two processors with each other. Depending on the benchmark, the results only express how fast the benchmark was executed, which can also include network tests, memory tests, harddrive speed, etc., not how well the used compiler or Java Virtual Machine is working. These benchmarks try to simulate some real-world-problems, as well as some special cases, which are intended to find weaknesses and corner cases in the system.

By comparing results of benchmark runs of the same system, using different compilers, one can see which one performs the best. For example one run uses OpenJDK7 and the other one uses OpenJDK8. Now the results show the difference between both implementations. The primary benchmark used in this work is the "SPECjbb2005" [Cor05]. SPEC stands for the Standard Performance Evaluation Corporation, which was founded in 1988 and is now a non-profit organization. The aim of this work is to show ways to get better scores at the very common benchmark SPECjbb2005 using aarch64 as platform together with OpenJDK8.

1.4 Structure of the work

This thesis is divided into four chapters. The first chapter gives an overview of the whole work. Chapter 2 presents related work, while in chapter 3 the applied methods and a more detailed overview of the analysis is given and in the end the final results of the optimizations are shown. Conclusion, discussion and also future work are presented in chapter 4.

1.5 Methodological approach

This work will use a methodology which consist of these steps:

- **Benchmarking**
  At first some data for a comparison is required to be able to determine the difference between the OpenJDK with and without the new optimizations. Therefore SPECjbb2005 [Cor05] and the DaCapo [Pro09] benchmark will be used to evaluate the performance.

- **Assembly code review**
  In this step the quality of the generated code is judged. To be able to decide which optimization is possible for the current implementation the generated assembly code itself needs to be reviewed manually. Methods which performed poorly in the benchmarks are the primary subject of this analysis.

- **Machine model and compiler rules improvements**
  The machine model is used to describe the underlaying hardware, while compiler rules define the behaviour of the compiler and therefore also influence when to emit which code. In this step the machine model is improved by adapting the pipeline model more accurately to the real hardware. In addition new compiler rules are introduced to make use of yet unsupported instructions, as well as existing rules are improved. These so called *matching-rules*, which are explained in detail in section 3.6.1, define which assembly code is emitted by matching on specific patterns of the parsed Java code. For example a more optimized rule emits only two instead of three instructions for the same task.

- **Optimization evaluation and documentation of results**
  Finally the work of the previous steps needs to be evaluated. This is done by comparing the specific benchmarks against the original results. In addition the results are displayed visually by plotting a chart to easily see the changes in performance.

CHAPTER 2

Related work

In this chapter, related work about parameter finding and optimization, instruction selection, tree matching and Java synchronization is presented.
Finding the optimal combination of parameters is a NP-hard problem, which is relevant for this thesis in the aspect of selecting the optimal command line options for the execution of SPECjbb2005 with OpenJDK8.

In their paper about rapidly selecting good compiler optimizations Cavazos et al. show that by using performance counters in combination with machine learning strategies the finding of optimal compiler options can be performed automatically [ABT]. Traditional approaches can be very expensive, because the used search space consists of all possible combinations of compiler options. The used dataset contains of 500 transformation sequences, which are randomly processed and afterwards the performance is evaluated. An important advantage of the introduced algorithm is that it can be generalized and then applied to new programs, which were not yet available at the time of the evaluation. By comparing the results of SPEC benchmark runs with `-Ofast` and the compiler options generated by their method, Cavazos et al. measured a speedup of 10%.
Another interesting approach is realized by Haneda et al., who are using a statistical model to reduce the search space significantly [HKW05]. The presented algorithm is capable of finding a combination of compiler flags not only for a single program but for a collection of applications, like the SPECint95, which is used for the evaluation. The results show that all `-O` flags of gcc perform worse, than the statistically generated compiler options.

In the paper of Triantafyllis et al. another method is introduced, which reduces the search space [TVV+03]. This is done by limiting the use of heuristics, while exploring compiler options, combined with a compiler tuning phase. The innovative approach in their work is, that the predictive heuristic calculates the quality of the generated options a posteriori and not, like traditionally implemented, a priori. In addition the iterative

compilation method enables Triantafyllis et al. to gain accuracy towards non-iterative algorithms which cannot handle unpredictable optimizations and complex architectures.

Agakov et al. present another algorithm which uses an iterative approach, trying to reduce costs by minimizing the number of iterations [ABC+06]. The expensive part of iterative compiler optimization algorithms is the evaluation of the program, which needs to be performed after each iteration. A combination of machine learning and predictive modeling can be used to detect areas of the search space, which have the highest potential for optimizations. The algorithm of Agakov et al. learns with an off-line training set and stores the gathered information into independent Markov models. Experiments show that there is a measurable speedup of selecting compiler options, due to the reduced number of iterations.
This algorithm is relevant for this thesis, because the costs of one benchmark run are relatively high (about 3 minutes) and the number of relevant compiler parameters are large (20 - 30).

Kohavi et al. present a learning algorithm to select parameters for the optimal performance by minimizing the estimated error [KJ95]. Their algorithm uses a dataset as training data and is implemented using a wrapper method, assuming that the task can be reduced to a discrete function optimization problem. Using the training set as input, the search is performed by best-first search in combination with cross validation. In the next step the parameters are evaluated and then the information about the estimated error of each parameter combination is calculated. The final selection of parameters is then tuned with the specific induction algorithm and the according dataset.
When applying the algorithm to 33 standardized datasets from the UCI repository of machine learning databases, the result is that at a confidence level of 95%, the algorithm improves six domains of C4.5 while having no negative influence on the remainder.

One topic of this thesis is *instruction selection*, which is the stage of the compilers backend where, based on the tree-based IR, the respective instructions are selected, which is often combined with *register allocation*.
In the paper of Fraser, Henry and Proebsting the *BURG* program is introduced, which is a quick tree parser using a bottom-up rewrite system [FHP92]. With BURG it is possible to use a tree grammar, which is extended by a cost value per node, to generate C-code that finds an optimal parse in the given language in $\mathcal{O}(n)$. The result is an optimal instruction selection, which can be used in code generation.
In OpenJDK8's Hotspot, BURG is not used directly, but a program which uses an extended version of a similar algorithm.

Aho et al. introduced the tree-manipulation language *twig*, which uses a top-down matching algorithm to generate code [AGT89]. Twig is able to discover the minimum costs of a given tree automatically and therefore generating an optimal instruction selection by using dynamic programming. Because of this ability there are no constraints on the order

of the occurring patterns in the matching tree. Common subexpression elimination, as well as algebraic simplification are not performed during the parse, but they are subjects of future work.

One optimization described in this thesis deals with Java's synchronization, where the performance is increased by removing unnecessary lock and release instructions. Doing some research, one can find out, that the synchronization implementation of Java has still potential of further optimization.
According to Bogda et al., some of the synchronization operations are unnecessary, like the case when a synchronized object is only accessible by a single thread [BH99]. In this scenario a concurrent access is never taking place and therefore all locking and releasing operations on the given object can be omitted. Using two benchmarks as metric, they measured performance improvements of 36% and 20%.
In a similar way, Ruf and Erik perform an analysis of statically compiled code and are able to remove 100% of the synchronization operations performed in single-threaded applications [Ruf00]. In addition they show, that with a small overhead in compile time, they can remove redundant locking of objects in concurrent programs. For this method, *alias analysis* and *thread closure* strategies are used in a context-sensitive way.

CHAPTER 3

Method

3.1 OpenJDK internals

This sections gives an overview about the structure of OpenJDK, as well as a deeper look into the used syntax to describe the machine model and matching rules.

3.1.1 Structure

The internal directory structure is as follows:

- `root` - contains infrastructure files as well as various scripts
- `corba` - Common Object Request Broker Architecture (CORBA) is used to exchange object messages
- `hotspot` - JVM
- `jaxp` - Java API for XML Processing (JAXP) is an API to parse, validate, generate and transform XML
- `jaxws` - Java API for XML Web Services (JAX-WS)
- `jdk` - contains the required class libraries
- `langtools` - javac compiler
- `nashorn` - JavaScript Engine

All optimizations presented in this work concern files from the hotspot directory. Hotspot is the JVM, which supports just-in-time compilation (JIT) and adaptive optimization.

The structure of Hotspot is the following:

- `agent` - serviceability agent
- `test` - contains the testing suite
- `src`
  - `share` - contains shared code like vm and tools
  - `cpu` - CPU specific code
    * `aarch64` - the code relevant for this thesis
    * `ppc`
    * `sparc`
    * `x86`
    * `zero`
  - `os` - OS specific code
    * `aix`
    * `bsd`
    * `linux`
    * `posix`
    * `solaris`
    * `windows`
  - `os\_cpu` - contains OS and CPU specific code like `linux\_x86`, `windows\_x86`, etc.

HotSpot consists of two compilers: *c1* and *c2*.
While c1 is intended to compile code quickly without sophisticated optimizations, c2 generates optimized code and therefore is slower.
The internal profiler decides when and which methods needs to be compiled. If one method is called frequently, it is very likely that it gets compiled.
Usually HotSpot will first use c1 to generate binary code, which is then observed by the profiler if it would be worth compiling it again with the more expensive c2 compiler.

3.1.2 Architecture Description (AD) file - machine model

To describe the underlaying hardware, HotSpot uses a so called "Architecture description file" for each architecture. In this file the available registers, encoding classes, pipeline model, cost model and matching rules are defined.

According to the comments in `hotspot/src/cpu/aarch64/vm/aarch64.ad` "**encoding classes** are parameterized macros used by Machine Instruction Nodes in order to generate the bit encoding of the instruction". This means that these classes are collections of C-macros which generate the respective instruction in binary form.

The **pipeline model** describes the machine's pipeline-stages and associated resources. This information is used to select and schedule instructions such that an (almost) optimal chain of instructions can be generated, by inspecting dependencies and latencies of sequential instructions. The aim is to avoid waiting for results and executing independent tasks in the meantime. The model consists of *resources*, *stages* and *pipeline classes*.
Resources represent the functional units which are available, while a stage represents one clock cycle.
Pipeline classes are the most comprehensive part of the pipeline model, because each instruction belongs to one pipeline class. An example of a pipeline class can be found in table 3.1.
In line 1 the name `pipe_IX` is defined. Line 3 indicates that this class corresponds to a single instruction, while line 4 defines the latency of the used instruction. In line 5 and 6 the uses resources in the different stages are defined - the syntax is `resource : stage`. In this example line 5 indicates that `DEC`, which represents the resource for the decoder unit, is needed in `C0`, which stands for the first cycle.

	Pipeline class
1	`pipe_class pipe_IX()`
2	`%{`
3	`  single_instruction;`
4	`  fixed_latency(2);`
5	`  DEC : C0;`
6	`  IX  : C1;`
7	`%}`

Table 3.1: Example of a pipeline class.

The **cost model** helps the compiler to decide which instruction to select if they are semantically identical.
In HotSpot the cost model is implemented as a simple cost function which has to be defined for each instruction. The value needs to be defined inside the respective matching rule, and looks like this: `ins_cost(n);`, where `n` is an integer.

About 90% of the whole AD-file consists of **matching rules**.
An example of such a rule can be found in table 3.2. These rules consist of a *match* element and an obligatory *encoding*. In addition a rule can be associated with costs (cost model) and a pipeline class. Further a format string can be inserted, which is displayed only when debugging. The match-function takes a binary tree as argument, which describes a part of the compiler generated abstract syntax-tree.
If the compiler matches a rule while analyzing the actual tree, it compares if there are one or more matches. In the case of multiple matches, the rule with the lowest costs will be selected.

In the example the rule is called `addP_reg_reg`, which stands for the *add*-instruction with two registers as parameters. The match-function is called in line 2 and matches any (part of a) tree which consists of `Set` node which has a `iRegPNoSp` left child and a `AddP` node as right child. The `AddP` node has to have again a left child of type `iRegP`, which stands for "integer-register-pointer", and a right child of type `iRegL`, "integer-register-long".

`Set` is a generic data type used in HotSpot to indicate a set of nodes, while `AddP` indicates an addition of pointers. The types of the variables used in the rule can be found in line number 1, where the variables are declared.
Types and nodes are defined in `hotspot/src/share` in C++.

Line 4 calls the cost function and assigns the rule a value of two. In line 5 a format string is added, which is only used for debugging purpose. This string is displayed when using a debugging version of OpenJDK when disassembling.
The obligatory `ins_encode` is used in line number 7, where a simple `add` instruction is emitted with three operands. The first operand is the destination register followed by source register one and two.
This instruction adds `src2` to `src1` and stores the result in `dst`. In line 13 the corresponding pipeline class is assigned, which is in this case the same class like listed in table 3.1 and described earlier.

	Matching rule
1	`instruct addP_reg_reg(iRegPNoSp dst, iRegP src1, iRegL src2) %{`
2	`  match(Set dst (AddP src1 src2));`
3	
4	`  ins_cost(2);`
5	`  format %{ "add $dst, $src1, $src2\t# ptr" %}`
6	
7	`  ins_encode %{`
8	`    __ add(as_Register($dst$$reg),`
9	`           as_Register($src1$$reg),`
10	`           as_Register($src2$$reg));`
11	`  %}`
12	
13	`  ins_pipe(pipe_IX);`
14	`%}`

Table 3.2: Example of a matching rule of a simple pointer addition.

3.2 Overview

1. **Setup of the working environment**

   At first the whole working environment needs to be set up. Therefore the latest version of OpenJDK8 for aarch64 is required and has to be compiled. This can be done either *natively*, which means that the code is directly compiled on the target platform, or *cross*, where the code is compiled on another platform with a compiler which generates code for the target system.
   Usually it is easier to build natively if the compiler and the needed libraries are available for the used operating system. In this step all required dependencies, like header-files and libraries are collected and installed. Further the benchmarks for the evaluation are deployed on the target system.
   As a last step the OpenJDK8 is compiled in a release and in a debug version. The release version is the standard version of the OpenJDK and is intended to be used in production, while the debug version allows to gather more detailed information about the codegeneration, runtime behavior, instruction selection, etc. This version offers more possibilities to analyze but is therefore slightly slower and so it's not recommended to use it on productive servers, but only for development.

2. **Running the benchmarks**

   The benchmarks used for this work are primary SPECjbb2005 [Cor05] and as a second DaCapo [Pro09]. Both benchmarks are executed on the aarch64 platform under such conditions, that the result can be reproduced easily. To compensate fluctuations which may occur during the runs both benchmarks need to run multiple times. Further the x86 architecture will be used for benchmarking. This is done to be able to match differences with the results of aarch64 and to find weaknesses in the current OpenJDK implementation.
   The assumption is, that OpenJDK for x86 is already better optimized, because it is used a lot more than the one for ARMv8.

   Another important aspect is to record profiling information during the runs to be able to conclude which parts of the benchmarks perform better and which worse. This is important, because the results of a benchmark usually, and in this case too, do not tell anything detailed about performance bottlenecks. Benchmarks are intended to be a metric to compare hardware / systems / software and most benchmarks just generate one single score which represents the overall performance. Eg. SPECjbb2005 just provides one value per "level", where level n is a run with n threads in use. DaCapo supplies more values because it consists of many smaller benchmarks, but still the amount of scores is relatively small (<10). Due to the fact that the result of a benchmark are only a few figures, further profiling information is needed to be able to compare aarch64 and x86 in detail. During the runs the profiler collects data about which method is executed how often, how much CPU time is used, memory consumption, heap allocation and can even dump the whole heap. In this work the two built-in profilers Xprof and Hprof of Java are used.

These two profilers can be activated with the command line options `-Xprof` and `-agentlib:hprof`.

3. **Evaluation of the benchmarks**

   As a first step the results of the runs of the original, unmodified OpenJDK on aarch64 needs to be stored. The next step is to calculate the average of the results to have one diagnostically conclusive value. The most complex part of this task is to evaluate the information gathered by the profilers. The extraction of the information is done in the following steps:

   Per benchmark:

   a) Run the benchmark n times.

   b) Extract the execution time $t_{i,method}$ of each method.

   c) Accumulate the values of each method, $sum_{method} = \sum_{i=1}^{n} t_{i,method}$.

   d) Divide the sum of respective value by the total amount of execution time to get a percentage of the execution time for each method , $avg_{method} = \frac{sum_{method}}{n}$.

   Per method:

   a) Calculate the average profile $avg_{method,x86}$ for each method on x86.

   b) Calculate the average profile $avg_{method,aarch64}$ for each method on aarch64.

   c) Normalize the data to be able to compare the profiles.
   This is done by calculating the relative execution time to be able to exclude differences in clock speed and other architectural influence factors.
   The absolute execution time is the consumed CPU time of all methods during one profiling session, which is usually one benchmark run.
   Assume there are m methods, then $t_{abs} = \sum_{i=1}^{m} t_{i,method}$.

   d) Plot a bar chart per relevant method (using a proper threshold for relevance). For example methods which occur only once in the benchmark and do only consume very little CPU time will not be displayed. This threshold needs to be adapted for every benchmark separately.
   For example experiments showed that a threshold of 1% CPU time is a good value for SPECjbb2005 running on the system used in this thesis, which is described in more detail in section 3.3.

   With this information a conclusion about bottlenecks and weaknesses is possible, because a direct comparison of two implementations of OpenJDK is given.
   It is also necessary to calculate this information, because after the optimization a more detailed comparison of the results can be performed.

4. **Analysis of suspicious code**

If a method is, for example, executed only 10% on x86 and on aarch64 it took the CPU 35% of the total execution time, this is suspicious and possibly a good entry point for further investigation. In the next step the benchmark is executed again, but with the option to print out the generated assembly code. This code is the subject of further analysis.

By manually checking the quality of the assembly, situations in which the compiler did not generate the optimal code, can be found. If the found pattern occurs in a *hot loop*, which is a code fragment which is executed very often in a loop, or in another frequently called method, this means that the optimization of it would significantly influence the results. Another tool to inspect the code generation is JITWatch, which is a "Log analyser and visualiser for the HotSpot JIT compiler" [Gro15]. It allows to analyze logfiles in a convenient way, which may be really large (>150 MiB), print call graphs, split-view of Java vs. assembly and so on.

5. **Optimization in the OpenJDK**

   The optimizations will be done in the following places:

   - Better mapping of the machine-description model to the architecture.
   - Rewriting of existing matching-rules for the compiler.
   - Introduction of new rules for the compiler, e.g. because of newly available instructions.

6. **Evaluation of the implemented optimization**

   In this step the performance is measured again to evaluate whether the newly introduced optimizations really improve the generated code, using the same metrics as in step 3.

3.3 Setup of working environment

This sections introduces the required tools and dependencies to be able to build the OpenJDK, run the benchmarks and to analyse the generated assembly.

3.3.1 Build OpenJDK

To be able to build the OpenJDK natively on ARMv8 64 bit the following requirements needs to be fulfilled:

- ARMv8 64 bit execution mode hardware.
  The hardware used for this thesis is an *APM(R) X-Gene1 ARMv8 CPU @ 2.40GHz* with *8 cores* and *16 GB RAM*.

- Supported operating system version.
  The used version of this thesis is Ubuntu 14.10.

- The latest version of the OpenJDK8 for aarch64.
  This version is available in the official mercurial repository which can be found at hg.openjdk.java.net. The version used in this thesis is based on the mercurial tag `jdk8u45-b13`.

- A C-compiler, like gcc.
  In this work a non-standard gcc, version 4.8, is used which is optimized for the aarch64 architecture.

- Dependencies for OpenJDK, like headers and libraries including *alsa*, *freetype*, *cups*, and *xrender*.

The compilation of the OpenJDK8 needs to be done in two ways: in the *server-release* and the *server-slowdebug* version. The debug version is needed to be able to better understand why which code is generated and for further assembly code analysis. The release version is used for benchmarking.

3.3.2 Benchmarks

The next step is to setup the benchmarks:

- Get the benchmarks
  For this thesis the SPECjbb2005 [Cor05] and DaCapo [Pro09] benchmark will be used. The respective files can be found at the website of the vendor (see references).

- Installation
  In the respective files one can find a install script which does all the needed work. No manual work is required in this step.

- Parametrization
  Command line options have a huge influence on the results and therefore they need to be examined and adopted carefully. In addition the self-built Java binary needs to be used to run the benchmarks, which can be done by setting the Java classpath. To find out the best combination of options a lot of runs (> 100) are needed to compare the results.
  This is still not the optimal configuration because there exists a huge amount of combinations, but it is a solution which is at least better than the standard settings. "Huge amount" means that there are about 900 different flags, where about 170 flags need a signed integer and about 130 an unsigned integer as parameter, while the rest of the flags can either be enabled or disabled. A signed integer has values from $-2.147.483.648$ to $2.147.483.647$, while an unsigned integer has a range from 0 to $4.294.967.295$.
  This would mean there are about $(2^{600} - 1) * (4.294.967.296^{300} - 1) \sim 3.2 * 10^{3070}$ different combinations of flags, for which the optimal combination cannot be found in appropriate time by trying out all combinations, considering that one run takes about 3 minutes to evaluate.

3.3.3 Analysis tools

Logfiles may be huge (> 150 MiB) and so it can be really frustrating to examine these files per hand. To improve the readability and to be able to find the needed information faster analysis tools are the key to success. In this thesis the following tools are used:

- JITWatch [Gro15]
  With the help of this tool the behavior of the Just-In-Time (JIT) compiler of Java can be examined. JITWatch therefore parses the generated logfiles of the hotspot compiler. Some of the features listed in the JITWatch wiki:
  - Browse class trees and view which methods were JIT compiled, when JIT compilation occurred, and information about the compilation.
  - Mount your source, jars, and class trees to jump to the source, bytecode, and assembly for a method.
  - Plot JIT compilations over time and visualise when a method was JIT compiled.
  - View toplists of the largest native methods, methods with most bytecodes, longest compilation times, etc.
  - Open sourced under the Simplified BSD license.
- IdealGraphVisualizer (IGV) [OoL07]
  During the compiling phases the OpenJDK generates an XML-file which contains information about the internal IR. IGV parses this file and graphically displays the abstract intermediate-tree in the respective state of the code generation. More information about this tool can be found in the master thesis *Visualization of program dependence graphs* of T. Würthinger who demonstrated the program in short at the Intl. Conf. on Compiler Construction [WWM08].
- `vim` & `grep` and other command line tools
  Still a lot of work - probably even the major part - needs to be done manually. Most of the time tools like `vim`, `grep` and `find` are used to search occurring patterns and to modify them.

3.4 Running the benchmarks

By executing the benchmarks, information about the performance is gathered. By running the same benchmark with different compiler versions on the same machine on the same platform, the results can be used as metric to compare the versions. This information is crucial because it is a good indicator to see if the newly introduced optimizations are really improving the performance. Of course one benchmark cannot prove a performance increase for all possible situations, but at least it shows that in some cases the program executes faster.

In this thesis the main focus is on SPECjbb2005 because of its wide international distribution and popularity. The parameters of the benchmark, or to be more precise, of the Java call of the benchmark, needs to be adopted. By reading the official Java manuals some general optimization flags like `-server -XX:+AggressiveOpts` can be found. Increasing the heap and the stack available for the Java Virtual Machine (**JVM**) can be done by using these flags: `-Xms10g -Xmx10g -Xmn4g -Xss64`.
Depending on the chosen settings the benchmark needs between 2 and 30 minutes to execute on current hardware, which is in this thesis an ARMv8 with 8 cores and 16 GiB RAM.

By doing some testing these options leaded to the best results:

```
-server -XX:+AggressiveOpts -XX:+UseFastAccessorMethods \
-XX:+OptimizeStringConcat -XX:+UseBiasedLocking \
-XX:+UseParallelGC -XX:ParallelGCThreads=10 -XX:+UseParallelOldGC \
-XX:SurvivorRatio=8 -XX:TargetSurvivorRatio=90 \
-XX:MaxTenuringThreshold=15  -Xms10g -Xmx10g -Xmn4g -Xss64
```

3.5 Evaluation of the benchmarks

This step is required to prove an increase of performance. In this case the end result of the benchmark is an indicator for the measured performance which can be found in the log-file generated during the execution.

To find out which parts of the benchmark or parts of the OpenJDK did perform better and which worse a profiler can be used. As mentioned earlier, the assumption is, that OpenJDK for x86 is already better optimized, because it is used a lot more than the one for the aarch64 architecture. This can be used to compare the results of the profilers and find out the bottlenecks in the respective implementation, especially weaknesses in aarch64.

By executing the benchmark twice, the results vary a little ($< 5\%$) because of the non-constant load of the machine, like cron jobs, ssh connections, network traffic and so on. These fluctuations need to be compensated, which can be realized by running the benchmark multiple times and afterwards calculating the average. Experiments showed that under the given conditions 40 to 50 runs are needed to be able to calculate a solid, stable and convincing mean. With this figure a robust and stable and informative value is determined.

3.5.1 Comparison of benchmark results between architectures

To determine the total result, SPECjbb2005 simulates warehouses which are implemented as threads. Each run consists of internally so called *levels*, which correlate with

the amount of threads in so far as in level n there are n threads used. For example in level 4, there are 4 threads used to simulate the orders and the traffic between warehouses.

Theoretically there is an unlimited number of levels, but usually twice the amount of CPU cores is the user-selected upper limit, as recommended by SPECjbb2005. SPECjbb2005 requires the user to run at least 8 levels to generate a valid and comparable test result. Usually the peak of performance is at the level which equals the amount of cores of the used machine.
Each levels performance is measured and a total result, as well as a result per level is generated.

The following steps describe the procedure to calculate representative values:

1. Assume $i \in \{1, ..., max(8, |cores|)\}$, $40 \leq n \leq 50$.
2. Run the benchmark n times on x86.
3. Run the benchmark n times on aarch64.
4. Extract the result of $level_i$ from the log-file, eg. with a regular expression `"throughput =\s*(\d+).*?bops"` (real example of SPECjbb2005 extraction).
5. Calculate the averages for x86.
6. Calculate the averages for aarch64.
7. Compare the average of level $\{1, ..., i\}$, eg. on a plot.

An example plot of a comparison between 40 runs on x86 and 40 runs on aarch64 can be seen in figure 3.1. In addition there is an example of a non-normalized plot, which can be found in figure 3.2.
The plot in figure 3.2 was generated without performing a normalization (calculating the average), to show the fluctuation between single runs. For this example only 10 runs instead of the previously mentioned 40-50 runs are displayed, because else the plot would get overfilled of lines.
The used x86 system is an *Intel(R) Xeon(R) CPU E31220 @ 3.10GHz* with *4 cores* and *20 GB RAM*.
The aarch64 machine has an *APM(R) X-Gene1 ARMv8 CPU @ 2.40GHz* with *8 cores* and *16 GB RAM*.

The peak is at about 210000, reached by the x86 machine which can be explained by the higher clockspeed, the more complex instruction set of x86 and the better optimization of OpenJDK for this architecture. As expected the peak is at level 4, which correlates with the 4 cores of the machine.

This plot shows that both - the ARMv8 and the x86 - scale well, which means, that until the level which equals the number of cores, the score increases linearly. Due to the 8 cores of the aarch64 the local peak is in the 8th level.

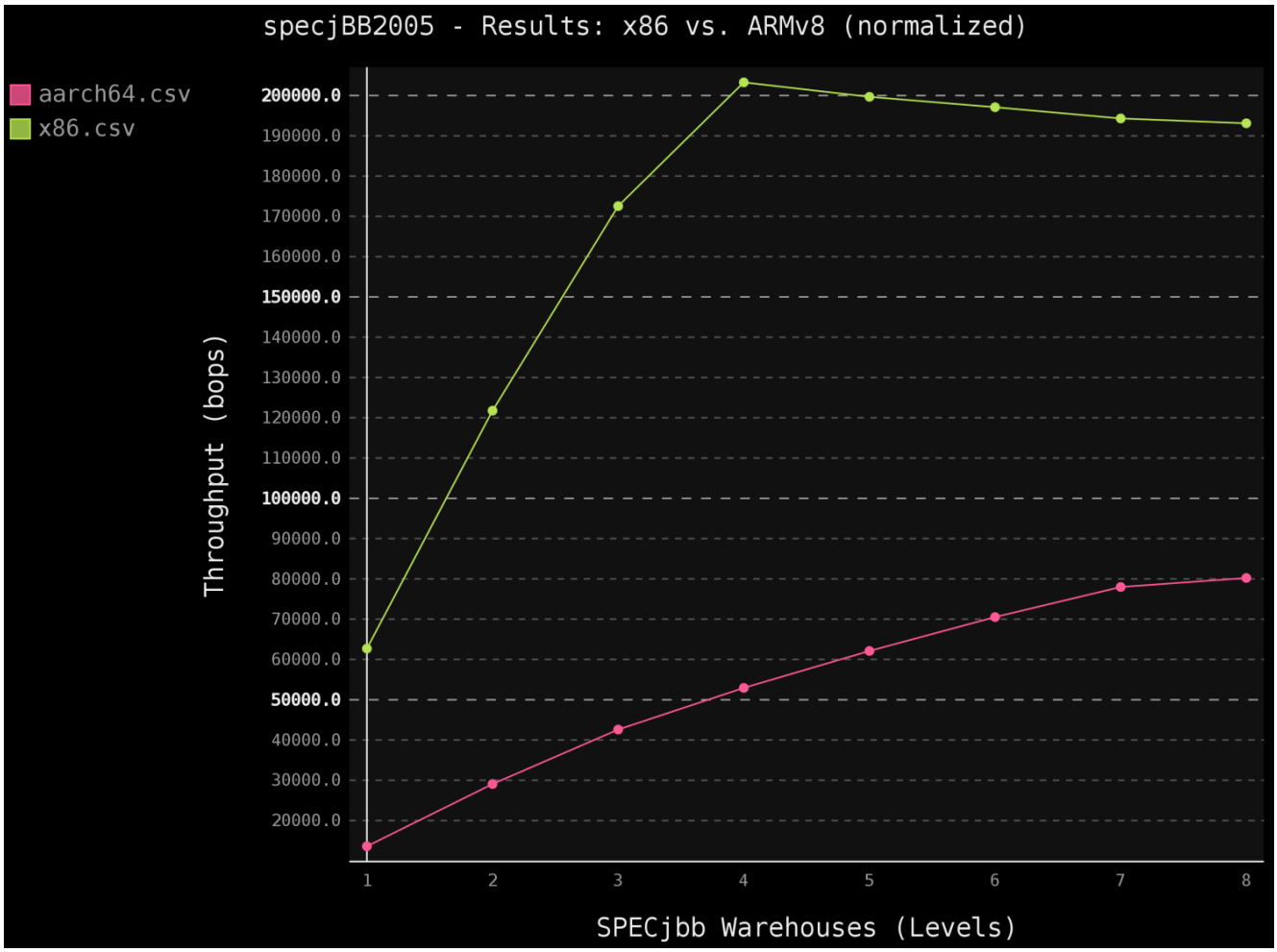

Figure 3.1: SPECjbb2005 - x86 vs. ARMv8 (normalized)

Another interesting fact is the fluctuation occurring on both machines, but significantly stronger on x86. The strong variations of the score at aarch64 in level 8 can be explained because in all other levels there was still at least one core available for tasks of the operating system. In level 8 the benchmark needed the whole CPU time and so, depending on the load of the OS, context switches lowered the score.
The higher fluctuations on x86 can be explained by mentioning that this machine is a multi user server which is not dedicated for benchmarking, and so the tasks of other users influenced the result more. Even the outlier (*x86_10.csv*) can be interpreted as an unfavorable time for benchmarking, because the system was used more intense by other

users.
Still the result is clear and both peaks can be seen clearly.

The much inferior results of the ARMv8 machine in this benchmark can be explained by two facts:

1. The lower clock speed of 2.4 GHz versus the x86 with 3.1 GHz.
2. The OpenJDK is better optimized for x86 machines.

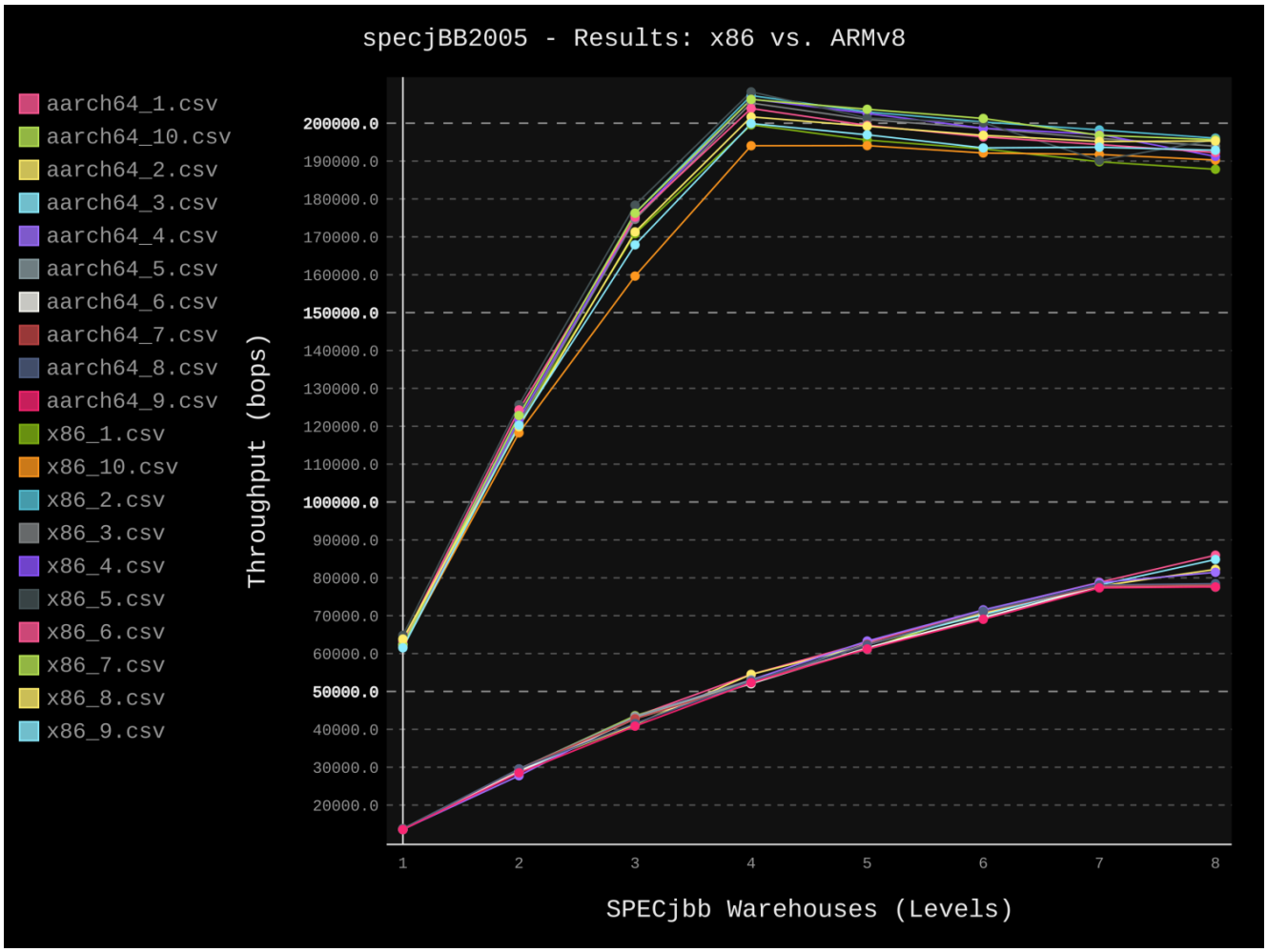

Figure 3.2: SPECjbb2005 - x86 vs. ARMv8 (without normalization)

3.5.2 Profiling

Profiling data contains information about used CPU time, memory consumption and the ratio between interpreted and compiled code. This information is crucial because

the OpenJDK is too large to inspect all aspects at once and so the parts which can be optimized need to be found and isolated. As mentioned in section 3.2, in this work the two built-in profilers Xprof and Hprof of Java are used.

The call for Hprof looks like this:
```
$JAVA -Xrunhprof:depth=10,verbose=n,heap=sites,cpu=samples \
-Xms256m -Xmx256m spec.jbb.JBBmain -propfile SPECjbb.props
```
where `$JAVA` is the path of the self-built binary.

- The parameter `depth=10` means, that the stack trace depth is 10 which is by default 4.
  As mentioned by A. Schroeter et al, "Software debugging is difficult and often involves searching through millions of lines of code to identify the cause of a defect - akin to finding a needle in a haystack. But stack traces can potentially narrow down the list of candidate files that are likely to contain the defect to speed up debugging." [SBP10].
  Therefore a deep stack trace is a valuable tool for identifying methods which have a high potential for optimization.

- With `verbose=y|n` it can be decided whether the profiler does generate verbose output or not.

- The flag `heap=` can be called with either `all` (default) , `sites` or `dump`.
  With `dump` the profiles creates huge logfiles because all available information about every created object is stored.
  By using `sites` not the entire information is printed but only generate as much output to be able to print the stack trace and a more general statistic of the memory consumption. More information can be found in the official documentation - A Heap/CPU Profiling Tool.

- `cpu=samples` means that the profiler uses sampling threads to periodically check all running threads and to print the most frequently stack traces.
  There is also the option to use `cpu=times` which would use Byte Code Injection (BCI) to determine the calls and stack traces of every called method. This would be more accurate but would also generate much more overhead than `samples`.

- The other parameters are used for the stack and heap size, while `spec.jbb.JBBmain` indicates which program to start.

Xprof does not support any flags and the call for it looks like this:
```
$JAVA -Xprof Xms256m -Xmx256m spec.jbb.JBBmain -propfile SPECjbb.props
```

A sample of the generated logfiles of **Hprof** looks like the outputs which can be found in table A.1, A.2 and A.3.

Table A.1 gives information about the heap usage but only the top 20 methods are shown in this example - Hprof analyzed more than 300 methods in this benchmark. The results are ranked by the usage percentage and in column 3 the accumulated percentages are displayed. Further the used bytes and objects can be seen and one very important value is the name of the used method which can be seen in the last column.

To find out which methods were called, the trace number is given. This number represents the unique id of the stack trace which is generated by Hprof. With that number the corresponding stack trace can be identified.
For example 310683, which can be found in the column *trace* of the first row in table A.1.
As previously mentioned the depth of the trace can be defined with the parameter `depth=n`, which was in this example set to $n = 10$. As a result of this flag, the maximum amount of displayed methods is 10, which is the case in this example.
A detailed explanation of how to interpret the stack trace id is given in the next paragraph.
Table A.2 gives an overview about the CPU usage. The values are sorted and listed with accumulated percentages like in table A.1. This samples show that `java.lang.Object.<init>` takes the vast majority of the computation time. In addition `java.util.Arrays.copyOf` needs also a lot of CPU time, which is explicable because the SPECjbb2005 benchmarks performs a lot of memory operations which are using arrays for storing the values.
To get a deeper understanding of who uses this method so often, the stack trace gives a lot of information. In this example the corresponding stack trace id of `java.lang.Object.<init>` can be found in the first row of table A.2, which is in this case 312498.
With this information one can look up the id in the log file and gather information about the caller(s).
In table A.3, the corresponding trace can be found. Here the stack indicates that the method `spec.jbb.TransactionManager.goManual`
called `spec.jbb.TransactionManager.runTxn`, and so on, which ended up in a call of `java.lang.Object.<init>`.
This is crucial information because it indicates which methods do use a lot of CPU time and which not. In this example one can conclude that the optimization of the initialization procedure of an object would yield the most performance.

A sample output of **Xprof** can be found in table A.4 and A.5.
Because Xprof samples the runtime stack in periodic intervals, it causes an impact on the performance. This impact is reduced by waiting considerable time between the samples, making this profiler suitable for measuring without the loss of much accuracy.
Usually times between 5ms and 20ms are used [Kli14].

Javas HotSpot first interprets the code and after finding very frequently used fragments, so called "hot spots", it compiles them.
In the so called *CompilePolicies* of HotSpot, the behavior of when to compile a method is described. The c2-compiler of HotSpot gets the needed information out of the C++ class `StackWalkCompPolicy`, which can be found in `src/share/vm/runtime`.

The name of this class is based on the algorithm, which walks up the stack as long as the first method is detected, which does not lead to inlining.
The JVM internal variable `CompileThreshold`, which is defined in `cpu/aarch64/vm/c2_globals_aarch64.hpp` for aarch64, is set to the following defaults: 10000 for c2 and 1500 for c1. This means that a method needs to be executed more than 1500 times to get compiled by c1 and after 10000 calls to that method it gets compiled by c2.
With flags like `-Xcomp` and `-XX:+CompileTheWorld` the compilation can be enforced.
Another influence factor, of whether a method gets compiled or not, is the amount of loop cycles inside a method. If a loop has many iterations, it gets marked to get compiled, while the amount of iteration is stored in so called *backedge-counters* separately for each method.
At this point the *two-tier-compilation* should be mentioned, which uses c1 in the first place to compile a method quickly, but with just very basic optimization, and afterwards - if needed - compile it again with c2 to generate highly optimized code.

In table A.4 the statistics of an sample benchmark run for the interpreted methods are shown.
The percentage indicates the time spent inside the method relative to the total execution time. With 2.0% the method `spec.jbb.Company.loadItemTable`, which is the most frequently used interpreted method in this example, consumed only as much CPU time as the 5th frequently used compiled method.
This is not surprising, because if the method would be used more, HotSpot would have decided to compile it. In conclusion it can be said, that in general interpreted methods will use less CPU time, than compiled functions. This can also be seen in the total amount of interpreted methods which is in this case 33.8%.

Table A.5 has an identical structure as table A.4, with the only difference that only compiled methods are displayed. Obviously `spec.jbb.Stock.<init>`, which is an benchmark internal method, consumes the most CPU time. Unfortunately Xprof does not support stack traces yet, so the information about the caller cannot be concluded.
As explained above, the total number of compiled methods is usually greater than the number of interpreted ones, which is true in this case. By analyzing the given figures one can see that about 60% of the total CPU time is used by compiled methods.

3.5.3 Comparison of profiling data

The data gained from profilers like Xprof and Hprof contains information about which methods are "important". Important means, that they are called very frequently and consume the major part of the CPU time.
When talking about optimization, these methods are of high interest, because there is the most potential to gain performance by optimizing them. By taking a closer look at

the generated assembly, weaknesses and suboptimal code generation can be discovered.

As mentioned above, this thesis compares an x86 and an aarch64 implementation of the Java OpenJDK to gather information about "where to optimize". Two examples of a comparison between profiling data are given in figure 3.3 and 3.4.

The algorithm to calculate differences in profiling data (eg. of Hprof) of a given benchmark is the following:

1. Assume $40 \leq n \leq 50$,
   $m = |methods|$,
   $A = \{x86, aarch64\}$.
2. Run the benchmark with profiling n times on architecture $a \in A$.
3. Extract the result of each occurring $method_{i,j,a}$ and the total CPU time $total\_CPU_{i,a}$ $i \in \{1, ..., n\}, j \in \{1, ..., m\}, a \in A$, eg. with a regular expression `"\s*\d+\s+?(\d+\.\d+)%\s+\d+\.\d+%\s+\d+\s+\d+\s+(.*?)$"` (real example for an Hprof extraction).
4. Calculate the relative CPU usage of each method.
   $rel_{i,j,a} = \frac{method_{i,j,a}}{total\_CPU_{i,a}}$ $, i \in \{1, ..., n\}, j \in \{1, ..., m\}, a \in A$
5. Select all methods which occurred in all runs on aarch64 and x86.
   $M = \{x | x \in \{method_{i,j,x86} \cap method_{i,j,aarch64}\}\}$,
6. Calculate the averages $avg_{j,a}$ for all methods $method \in M$ on architecture $a \in A$.
   $j \in \{1, ..., |M|\}$
7. Calculate the relative ratio between the average values of both platforms.
   $ratio_j = | \frac{avg_{j,x86}}{avg_{j,aarch64}} |$
8. Sort the results by their relative ratio $ratio_j$.
9. Plot the data on a bar chart.

In figure 3.3 an example result of this algorithm for a profiling with Xprof is shown. The methods are sorted by their relative ratio, which is the value in brackets after the method name.
Obviously `java.math.BigDecimal.layoutChars` has with 33819.27% the biggest relative ratio in execution time. This can be explained because this specific method is almost never called during the benchmark on aarch64.
In contrast to this example,
`spec.jbb.CustomerReportTransaction.processTransactionLog` is used and executed with almost identical relative CPU time on both architectures, where the relative ratio is only 0.03%.

Methods with a high relative ratio and additionally a high amount of total execution time are the most interesting. In this figure, this is `spec.jbb.DeliveryTransaction.preprocess`, which consumes on both platform the major part of CPU time and the relative ratio is with 49.32% significant. An important aspect is, that the method is spending more CPU time on aarch64 than on x64, which is suspicious, and it can be assumed, that in the generated code, there was a high optimization potential. Another very interesting method is `spec.jbb.Order.processLines`, which has a relative ratio of 195.91% and especially on x86 it has a very high CPU time consumption. In this case one can analyze both generated assembly codes and find out where x86 is losing the performance and what are the strenghts of aarch64 in this situation. With this information similar patterns, which do not perform as well, can be found and replaced. In addition the knowledge of existing strenghts helps, when implementing new features.

Figure 3.4 shows the result of a comparison using Hprof. This figure is structured in the same way as figure 3.3.
In this plot `spec.jbb.JBButil.create_random_a_string` is the eye catcher, because it consumes the most CPU time by far, but only on aarch64. The relative ratio is with 1411.24% significant and the generated assembly code is a good place to start with the manual review.
Further `spec.jbb.JBButil.create_a_string_with_original` has the highest ratio with 1909.64%. This leads to the suspicion, that something regarding the creation of strings is not implemented optimally.
The assumption is, that by disassembling the discussed methods, the chances are higher to find code fragments with optimization potential, than when reviewing dissassembly randomly.

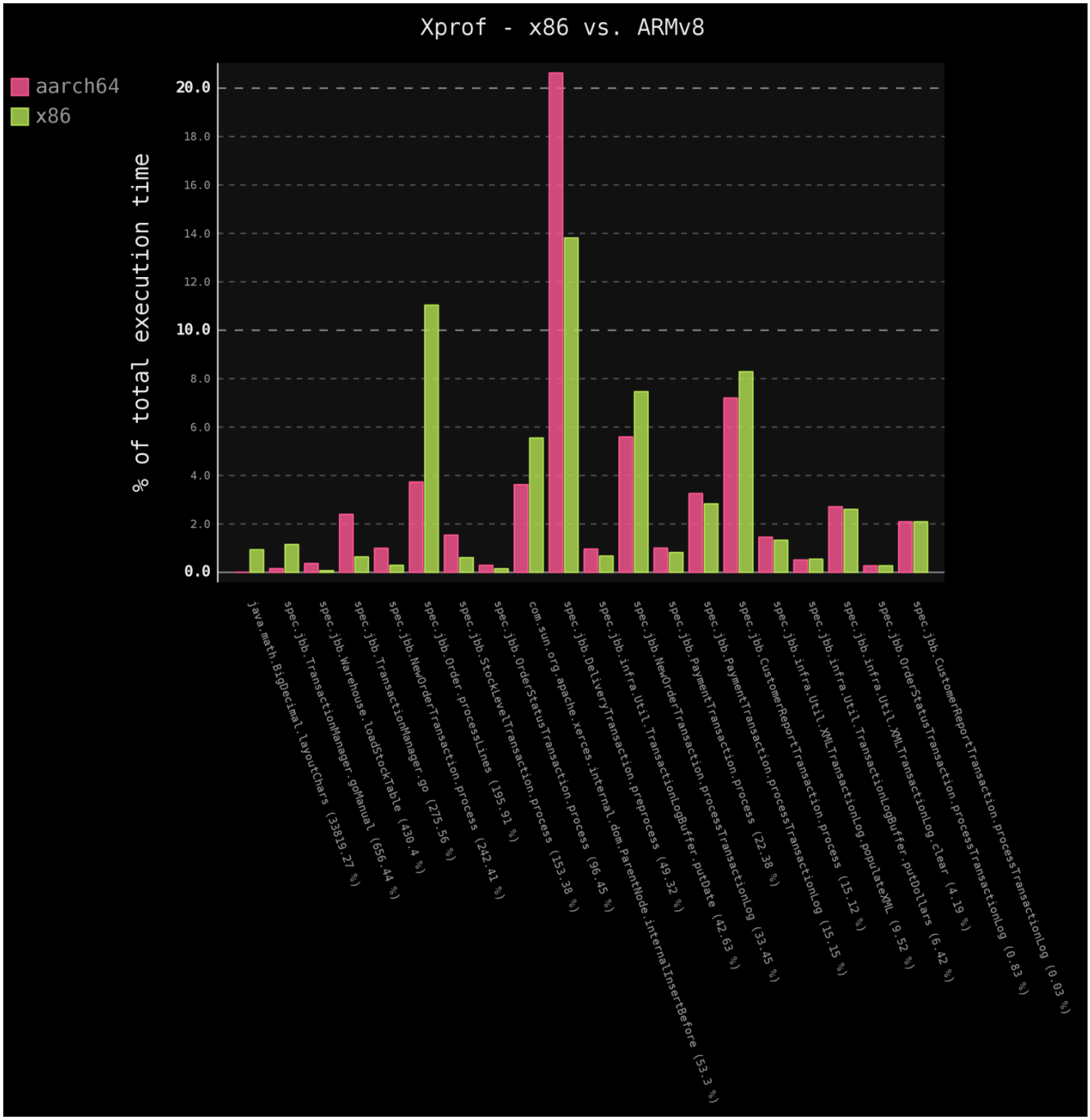

Figure 3.3: Xprof - x86 vs. ARMv8

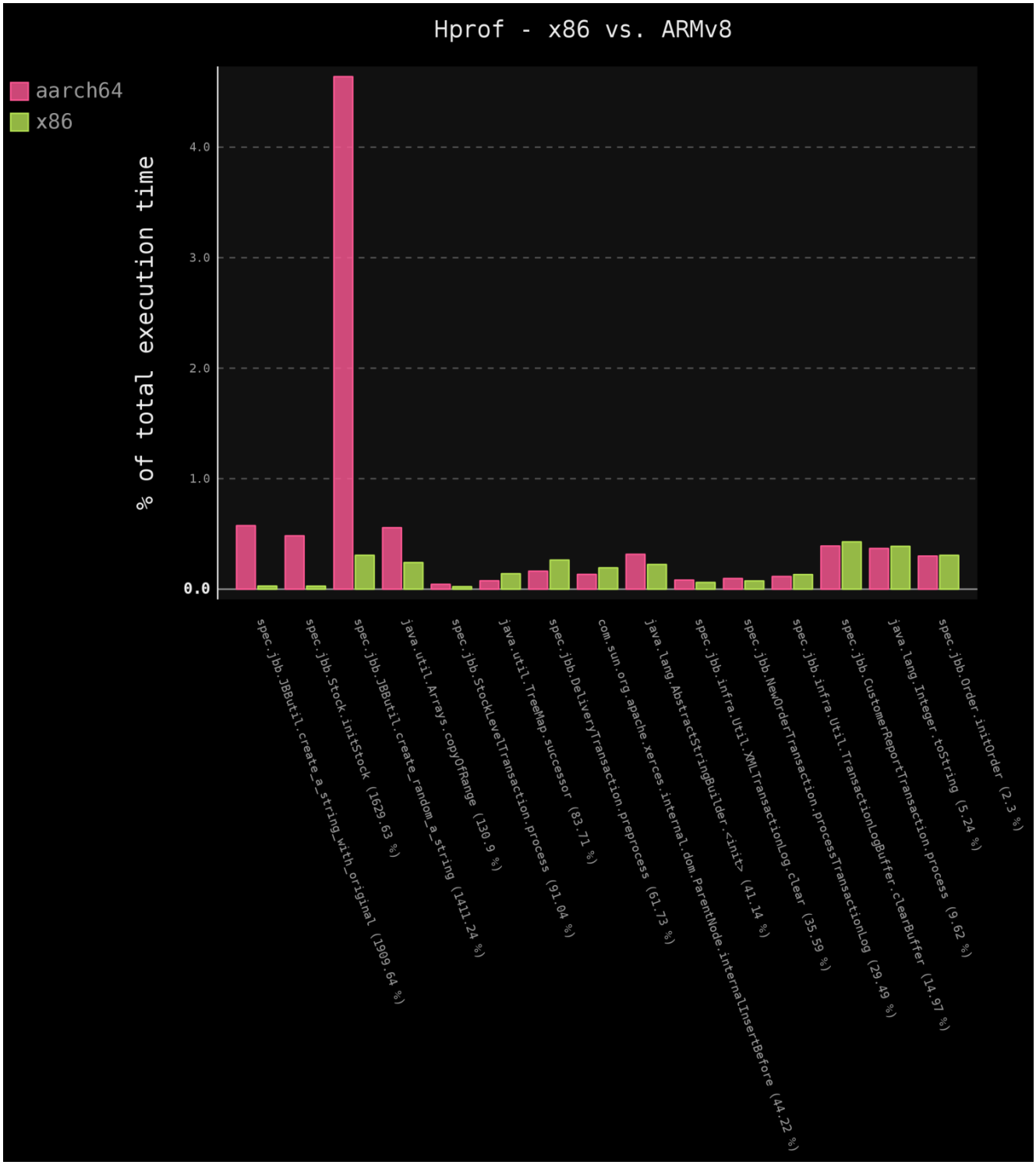

Figure 3.4: Hprof - x86 vs. ARMv8

3.6 Implementation of optimizations

This sections gives an overview of how to implement an optimization in OpenJDK. There are two real-world examples - firstly the introduction of a new matching rule is illustrated and then the analysis and improvement of the existing synchronization mechanism is shown.

3.6.1 Matching rule

While reviewing the generated assembly code manually, the pattern displayed in table 3.3 can be observed.
The structure is the following: `memory address: instruction  operands`.
In line number 1 an `and` instruction is used with three operands. In ARM assembly, the first operand is always the *destination operand*, which is in this case `w2`. This means that the result of the logical and instruction of `w2` and `#0x7` is stored in `w2`. Line number 2 shows the `cmp` instruction, which compares two values, setting the *condition flags* accordingly.
There are four flags: C (carry), Z (zero), N (negative) and V (overflow). In this example the value of `w2` is compared to zero, while in line 3 a conditional branch is executed. The `b.eq` (branch if equal) instruction reads the condition flags and if Z is set, it jumps to the given memory address, else the program continues with the next instruction.
There are many options available for the `b` instruction - a complete list can be found in table A.7.

This can be summed up like this: "Add 7 to `w2`, compare it to zero and if it is zero jump to `0x0000007f8c2968f4`."
In the ARM instruction set, there is an `ands` instruction, which is a bitwise "and" which additionally sets the condition flags. Obviously this instruction can be used in this case, like shown in table 3.4.
In line number 1 the bitwise "and" is computed and after that, the N, Z, and C flag is updated, according to the result. Line 2 executes the conditional branch.
Semantically this is identical and therefore it is ensured that the program is still valid. The advantage is, that the CPU needs only two instead of three instructions for the same work to be done, which inceases the performance.

	Observed assembly code pattern
1	`0x0000007f8c2961b4: and  w2, w2, #0x7`
2	`0x0000007f8c2961b8: cmp  w2, #0x0`
3	`0x0000007f8c2961bc: b.eq  0x0000007f8c2968f4`

Table 3.3: Non-ideal assembly code pattern

To transform the old pattern into the new one, OpenJDK uses matching rules, which are explained in detail in section 3.1.2.

	Optimized assembly code pattern
1	`0x0000007f8c2961b4: ands  w2, w2, #0x7`
2	`0x0000007f8c2961b8: b.eq  0x0000007f8c2968f4`

Table 3.4: Ideal assembly code pattern

First of all the `match` function needs to be parametrized correctly. This means that the corresponding abstract syntax tree needs to be defined, which contains the desired matching pattern, which is in our case "and,cmp,b".
Figure 3.5 illustrates the desired syntax tree and can be can be interpreted in this way: The root element `If` takes two operands - `cmp` and the result of the `CmpI` node, which takes the result of `AndI` and `zero` as input, while `AndI` computes the bitwise "and" of `src1` and `src2`.

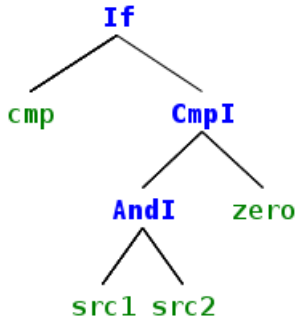

Figure 3.5: Syntax tree of observed pattern

In the next step the types of the operands need to be defined. `cmp` is a compare operation, while `zero` and `src2` are integer immediates. `src1` is an integer register and for the branch instruction a label is needed, which is of type `label`.

The final rule can be found in table 3.5.
Line 1 and 2 describe the operands, while the match function in line 3 takes the previously described binary syntax tree as input.
Line 4 tells OpenJDK to use `lbl` as label, which is then used in line 7, to indicate where to jump at from the conditional branch. In line 5 the costs of two are assigned and line 14 sets the pipeline class to `pipe_IX_BU`, which means "an arithmetic operation followed by a branch".
From line 6 to 13 the code to emit is defined, where line 7 the label is set to the corresponding condition. Line 9 to 11 emit the `ands` instruction and in line 12 the branch is emitted.

Experiments showed, that there is no measurable performance increase when running SPECjbb2005 with this rule, but a review of the generated assembly shows, that the rule

emits the code. Apparently this pattern does not occur in a hot loop or in a frequently called method, so that there is no significant reduction of CPU time. This does not mean, that the rule does not improve the code quality, because the evaluation is done with only one benchmark.
In programs, which use this pattern very often, the performance increases, because the CPU needs to execute only two instead of three instructions.

New matching rule for "and,cmp,b" pattern

```
1  instruct ands_branch(cmpOp cmp, immI0 zero, iRegIorL2I src1,
2                       immILog src2, label lbl) %{
3   match(If cmp (CmpI (AndI src1 src2) zero) );
4   effect(USE lbl);
5   ins_cost(2);
6   ins_encode %{
7     Label* L = $lbl$$label;
8     Assembler::Condition cond = (Assembler::Condition)$cmp$$cmpcode;
9     __ andsw(as_Register($src1$$reg),
10       as_Register($src1$$reg),
11       (unsigned long)($src2$$constant));
12    __ br ((Assembler::Condition)$cmp$$cmpcode, *L);
13  %}
14  ins_pipe(pipe_IX_BU);
15 %}
```

Table 3.5: Newly introduced matching rule

3.6.2 Data Memory Barrier (DMB)

By analyzing and profiling the performance of OpenJDK like discussed above, the assembly code of synchronization in Java, using the keyword `synchronized`, looks suspicious because a lot of `dmb` instructions are used, which are very costly. This instruction is expensive because all load and store operations on all CPU cores are forced to be finished before the program continues with the next instruction after the `dmb`.
"A `dmb` ensures that all explicit memory accesses before the `dmb` instruction complete before any explicit memory accesses after the DMB instruction start." [Ltd14]
This means that with the help of this instruction the order of the load and store operations can be forced, and the CPU must not reorder any of those.
These barriers are required in the instruction set of todays CPUs because the so called *out-of-order-execution* is a common hardware optimization. When dealing with single threaded applications the reordering does not have any impact on the correctness, but in concurrent programs, unpredictable behavior can be caused, when neglecting synchronization.

An illustrative example for an out-of-order-execution problem can be found in table 3.6. One would expect the program to print "42" in every case, because if processor #2 sets x to 42 and afterwards setting f to 1, processor #1 will exit the loop and print the current value of `x`, which is then 42. But in the case, the CPU reorders these operations it could happen, that this program would output "0".
This would happen if `f` gets updated before `x` and so the loop would be exited and the value of `x`, which is initially 0, printed, before processor #2 sets the value to 42.
This shows the need of the `dmb`-instruction, which enforces the ordering constraints and therefore eliminates the unpredictable behavior in this example.

Illustrative example for an out-of-order-execution problem - from wikipedia.org

```
1  x = f = 0;
2
3  Processor #1:
4     while (f == 0);
5     // Memory fence required here
6     print x;
7
8  Processor #2:
9     x = 42;
10    // Memory fence required here
11    f = 1;
```

Table 3.6: Out-of-order-execution problem

In Java, the built-in synchronization can be used by declaring something as `synchronized`. There are four different kinds of entities which can be defined as synchronized:

1. Instance methods
2. Static methods
3. Code blocks inside instance methods
4. Code blocks inside static methods

This ensures, that all operations done inside this entities can only be executed by exactly one thread concurrently, which is especially important when reading or writing to shared resources.

For exclusive access, including the acquiring of a lock for mutual exclusion, there are dedicated instructions available for ARMv8 systems.
There are *Load-Acquire* instructions like the `LDAXR`-family and *Store-Release* instructions like the `STLXR`-family.

A complete list of exclusive Load-Acquire and exclusive Store-Release instructions can be found in table A.6.
Table 3.7 shows the pattern, which is typically generated by OpenJDK8 when accessing to shared resources. To read the shared memory, `ldaxr` is used, after that some work with the loaded values is performend and afterwards a `stlxr` stores the value back to the memory. The whole procedure is surrounded by two `dmb` instructions, to enforce the ordering constraints.

Typical DMB-pattern in OpenJDK's generated assembly
`; load-aqcuire, do work, store-release`
`dmb`
`ldaxr`
`.....`
`stlxr`
`dmb`

Table 3.7: Typical DMB pattern in OpenJDK8

In the official *ARMv8 Architecture Reference Manual* a suggestion for how to implement the acquiring of a lock for mutual exclusion is described, which can be found in table 3.8. Here the Lock-Acquire is implemented as spinlock, which means the CPU performs busy waiting until the lock was acquired successfully.
In this example no `dmb` instructions are used, which leads to the question if they are redundant when implementing exclusive loads and stores. According to the manual, chapter *B2.7 Memory ordering, Load-Acquire Store-Release*, the dmb is not needed in this case:

page 88:
> "There are no additional ordering requirements on loads or stores that appear before the Load-Acquire."
> "There are no additional ordering requirements on loads or stores that appear in program order after the Store-Release."

page 89:
> "The Load-Acquire/Store-Release instructions can remove the requirement to use the explicit DMB memory barrier instruction."

In HotSpot there are two matching rules, which match on acquiring a lock and on releasing a lock. Both rules are shown in table 3.9 in their original state.
As one can see in line 8 and 23 the rules emit a `dmb` instruction, which is in this case never needed, because the mechanism of acquiring a lock already causes ordered access. This redundant instruction results in a significant performance decrease, because the

	Acquiring a lock
1	`PRFM PSTL1KEEP, [X1] ; preload into cache in unique state`
2	`Loop`
3	`LDAXR W5, [X1] ; read lock with acquire`
4	`CBNZ W5, Loop ; check if 0`
5	`STXR W5, W0, [X1] ; attempt to store new value`
6	`CBNZ W5, Loop ; test if store succeeded and retry if not`
7	`; loads and stores in the critical region`
8	`; can now be performed`

Table 3.8: How to acquire a lock in aarch64 assembly.

CPU is forced to complete all store and load operations before and after the `dmb`, even if they are not dependent.
Obviously the next step is to stop emitting these two `dmb` instructions when acquiring and releasing a lock. It is important not to delete the whole matching rule, because this would result in selecting the default rule for `MemBarAcquireLock`- and `MemBarReleaseLock`-nodes.
By changing line 7-9 and 22-24 to `ins_encode();`, setting the costs in line 3 and 18 to 0, as well as setting the pipeline class to `pipe_class_empty`, removes the `dmb` from the generated code but keeps the matching. Comparing the results of SPECjbb2005 with and without this optimization, an performance gain of up to 15.2% can be measured, which can be seen in figure 3.6.
For this plot the benchmark was executed 40 times in the original state and 40 times with the optimization, while calculating the respective values like shown in section 3.5.1 but with both times aarch64 as platform.

original DMB rules in aarch64 AD-file

```
1  instruct membar_acquire_lock() %{
2    match(MemBarAcquireLock);
3    ins_cost(VOLATILE_REF_COST);
4
5    format %{ "membar_acquire_lock" %}
6
7    ins_encode %{
8      __ membar(Assembler::LoadLoad|Assembler::LoadStore);
9    %}
10
11   ins_pipe(pipe_serial);
12 %}
13
14 ###############################################
15
16 instruct membar_release_lock() %{
17   match(MemBarReleaseLock);
18   ins_cost(VOLATILE_REF_COST);
19
20   format %{ "membar_release_lock" %}
21
22   ins_encode %{
23     __ membar(Assembler::LoadStore|Assembler::StoreStore);
24   %}
25
26   ins_pipe(pipe_serial);
27 %}
```

Table 3.9: original DMB rules in aarch64 AD-file

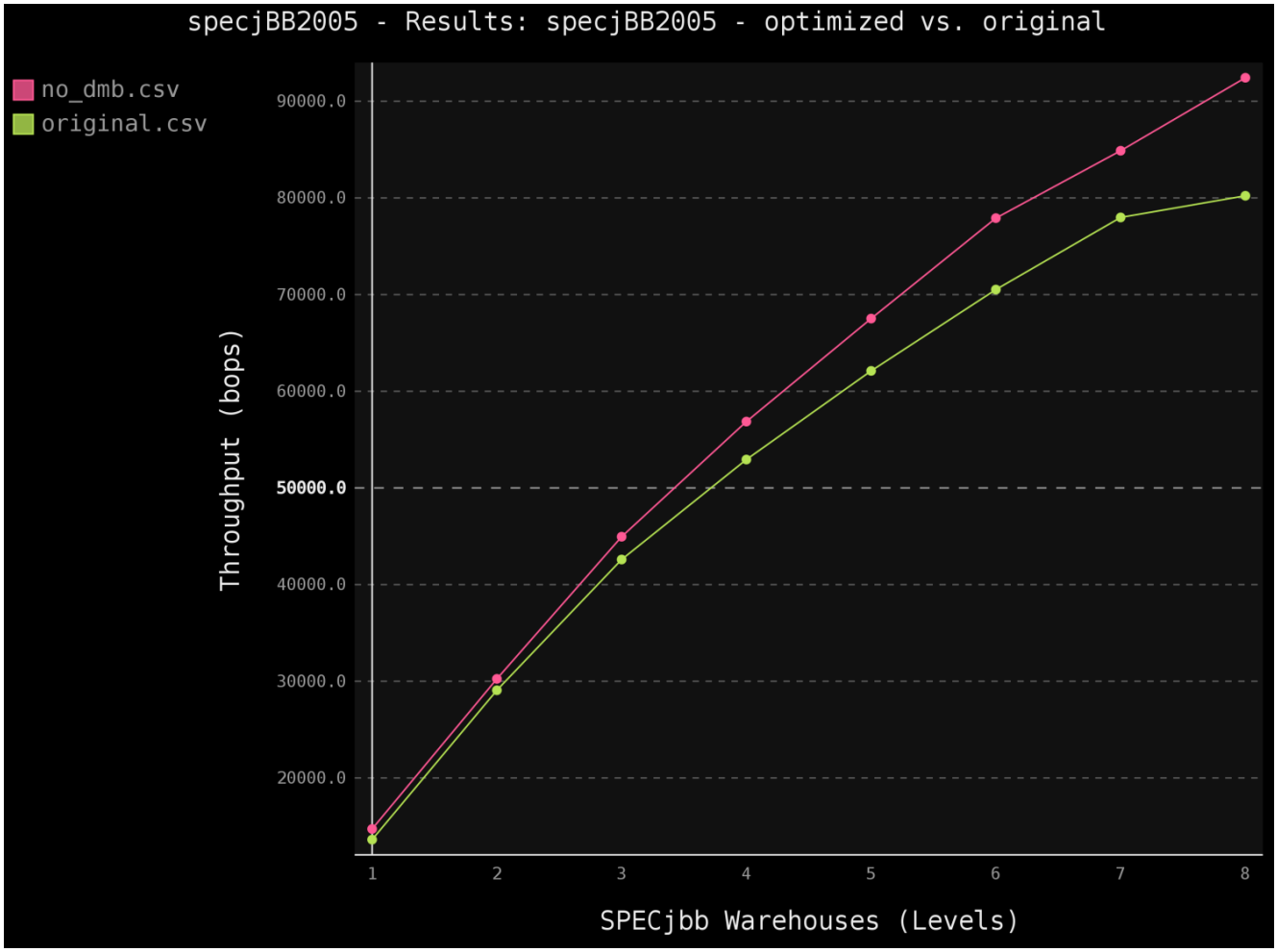

Figure 3.6: specjBB2005 - optimized vs. original

CHAPTER 4

Discussion and conclusion

In this work a methodological approach for analyzing, evaluating, profiling and improving the performance of OpenJDK8 for ARMv8 systems was presented. The aim was to improve the performance using the SPECjbb2005 benchmark as metric. In the first step the internal structure of OpenJDK was presented, which is important to understand the implementation of the optimizations in detail. Further the setup of the working environment was explained, followed by how to run the benchmarks.
To be able to compare the results a detailed step-by-step instruction was given, which was illustrated with examples. Finding weaknesses and bottlenecks was solved by using the Java built-in profilers Xprof and Hprof. In addition an algorithm to evaluate the collected information from the profilers was described and applied to real-world data and shown in various figures.
Concluding it can be said, that analyzing and profiling of OpenJDK8, as well as manual assembly code review are main aspects of this work, which are obligatory requirements to be able to identify the parts of this comprehensive software, which have a high potential for optimizations.
Finally two practical and real-world improvements of OpenJDK8 were presented and their implementation was illustrated in detail. The optimization of Java's synchronization mechanism, which was realized by removing redundant data memory barriers, improved the performance by up to 15.2% in SPECjbb2005.
While the presented modifications of OpenJDK8 resulted in a gain of performance, there is still a lot of potential for further improvements - especially in the implementation of synchronization routines. In future work, a deeper analysis of emitted Lock-Acquire and Store-Release instructions will be done, because of the assumption, that still non-ideal synchronization-patterns get emitted. Additionally one of the presented algorithms in chapter 2 will be implemented and applied to OpenJDK8 and SPECjbb2005 to find a better combination of command line parameters.

APPENDIX A

Profiling and instruction tables

SITES BEGIN (ordered by live bytes) Thu Mar 19 17:31:51 2015

rank	self	accum	bytes	objs	bytes	objs	trace	name
1	46.37%	46.37%	86400000	1200000	86400000	1200000	310683	char[]
2	15.46%	61.82%	28800000	1200000	28800000	1200000	310682	java.lang.String
3	6.12%	67.94%	11403112	120000	11403112	120000	310679	char[]
4	3.61%	71.55%	6720000	120000	6720000	120000	310680	java.lang.String[]
5	3.09%	74.64%	5760000	120000	5760000	120000	310675	spec.jbb.Stock
6	2.23%	76.87%	4159872	57776	15833376	219908	311328	spec.jbb.Orderline
7	2.06%	78.93%	3840000	120000	4480000	140000	307215	java.util.HashMap$Node
8	1.55%	80.48%	2880000	120000	2880000	120000	310678	java.lang.String
9	1.46%	81.94%	2717736	3309	2957728	3600	310840	char[]
10	1.24%	83.18%	2311040	57776	8796320	219908	311332	java.math.BigDecimal
11	1.02%	84.20%	1907808	119238	1907808	119238	310688	java.lang.Integer
12	1.02%	85.23%	1904296	20000	1904296	20000	310618	char[]
13	0.77%	85.99%	1427712	8112	93012480	528480	311384	char[]
14	0.74%	86.74%	1386624	57776	5277792	219908	311333	java.lang.String
15	0.61%	87.35%	1136056	20000	1136056	20000	310609	char[]
16	0.55%	87.90%	1030656	5856	64111872	364272	312067	char[]
17	0.42%	88.32%	786528	6	1573152	66	310685	java.util.HashMap$Node
18	0.34%	88.66%	640000	20000	640000	20000	310606	spec.jbb.Item
19	0.34%	89.01%	640000	20000	640000	20000	310620	java.util.HashMap$Node
20	0.31%	89.32%	582912	3312	35177472	199872	311318	char[]
...								

Table A.1: Profiling - Hprof sites (heap)

CPU SAMPLES BEGIN (total = 29863) Thu Mar 19 17:31:51 2015

rank	self	accum	count	trace	method
1	15.82%	15.82%	4725	312498	java.lang.Object.<init>
2	5.84%	21.67%	1745	311384	java.util.Arrays.copyOf
3	5.59%	27.25%	1669	312317	java.lang.Object.<init>
4	4.16%	31.41%	1242	310686	java.lang.Object.<init>
5	4.03%	35.44%	1202	312403	java.lang.Object.<init>
6	3.97%	39.41%	1185	312067	java.util.Arrays.copyOf
7	3.87%	43.27%	1155	310683	java.util.Arrays.copyOf
8	3.86%	47.14%	1153	310681	spec.jbb.JBButil.create_random_a_string
9	2.81%	49.95%	840	312234	java.lang.Object.<init>
10	2.52%	52.47%	754	312450	java.lang.Object.<init>
11	2.52%	54.99%	752	312463	java.lang.Object.<init>
12	2.46%	57.46%	736	312399	java.lang.Object.<init>
13	2.43%	59.89%	727	312315	java.lang.Object.<init>
14	2.38%	62.27%	712	312455	java.lang.Object.<init>
15	2.26%	64.53%	675	312449	java.lang.Object.<init>
16	2.20%	66.74%	658	311318	java.util.Arrays.copyOf
17	1.89%	68.62%	563	312474	java.lang.Object.<init>
18	0.68%	69.30%	203	307221	java.lang.Object.<init>
19	0.64%	69.95%	192	307210	spec.jbb.JBButil.create_random_a_string
20	0.64%	70.59%	192	307212	java.util.Arrays.copyOf
...					

Table A.2: Profiling - Hprof CPU samples

TRACE 312498:
java.lang.Object.<init>(Object.java:37)
java.lang.Number.<init>(Number.java:55)
java.lang.Integer.<init>(Integer.java:849)
java.lang.Integer.valueOf(Integer.java:832)
spec.jbb.Warehouse.retrieveStock(<Unknown Source>:Unknown line)
spec.jbb.DeliveryTransaction.preprocess(<Unknown Source>:Unknown line)
spec.jbb.DeliveryHandler.handleDelivery(<Unknown Source>:Unknown line)
spec.jbb.DeliveryTransaction.process(<Unknown Source>:Unknown line)
spec.jbb.TransactionManager.runTxn(<Unknown Source>:Unknown line)
spec.jbb.TransactionManager.goManual(<Unknown Source>:Unknown line)

Table A.3: Profiling - Hprof stack trace

Interpreted		native	Method
2.0%	3	0	spec.jbb.Company.loadItemTable
1.3%	2	0	spec.jbb.MapDataStorage.put
1.3%	2	0	spec.jbb.Warehouse.loadStockTable
0.7%	0	1	java.lang.Object.clone
0.7%	1	0	java.util.HashMap.afterNodeAccess
0.7%	0	1	java.io.UnixFileSystem.getBooleanAttributes0
0.7%	0	1	java.io.FileInputStream.open0
0.7%	0	1	java.lang.ClassLoader.findBootstrapClass
0.7%	0	1	java.lang.Class.forName0
0.7%	1	0	java.lang.ClassLoader.defineClass1
0.7%	1	0	java.lang.ref.ReferenceQueue$Lock.<init>
0.7%	1	0	java.nio.charset.CharsetEncoder.maxBytesPerChar
0.7%	1	0	java.lang.Integer.intValue
0.7%	1	0	spec.jbb.TreeMapDataStorage.<init>
0.7%	1	0	java.util.logging.LogRecord.setSourceClassName
0.7%	1	0	java.util.ResourceBundle$BundleReference.<init>
0.7%	1	0	java.util.regex.Matcher.getSubSequence
0.7%	1	0	java.util.TreeMap.get
0.7%	1	0	java.nio.ByteBuffer.array
0.7%	1	0	java.math.MutableBigInteger.isOdd
0.7%	1	0	java.lang.Integer.valueOf
0.7%	1	0	spec.jbb.JBButil.create_random_a_string
0.7%	1	0	spec.jbb.Orderline.validateAndProcess
0.7%	1	0	java.lang.CharacterDataLatin1.toUpperCaseEx
0.7%	1	0	java.lang.String.compareTo
33.8%	45	6	Total interpreted (including elided)

Table A.4: Profiling - Xprof interpreted methods

Compiled		native	Method
18.5%	28	0	spec.jbb.Stock.<init>
11.3%	17	0	spec.jbb.Warehouse.loadStockTable
4.0%	6	0	spec.jbb.Item.setUsingRandom
2.6%	2	2	spec.jbb.Stock.initStock
2.0%	3	0	spec.jbb.Orderline.validateAndProcess
1.3%	1	1	java.math.BigDecimal.<init>
1.3%	0	2	sun.security.provider.SHA.implCompress
0.7%	0	1	java.lang.StringBuilder.<init>
0.7%	1	0	spec.jbb.MapDataStorage.put
0.7%	1	0	java.util.HashMap.put
0.7%	0	1	java.util.HashMap.get
0.7%	0	1	spec.jbb.JBButil.random
0.7%	0	1	java.util.HashMap.containsKey
0.7%	0	1	java.math.MutableBigInteger.clear
0.7%	0	1	java.math.BigInteger.trustedStripLeadingZeroInts
0.7%	0	1	spec.jbb.JBButil.create_random_n_string
0.7%	1	0	sun.nio.cs.US_ASCII$Decoder.decode
0.7%	1	0	spec.jbb.JBButil.create_random_a_string
0.7%	0	1	java.lang.String.compareTo
0.7%	0	1	java.math.BigInteger.addOne
0.7%	0	1	java.math.MutableBigInteger.rightShift
0.7%	0	1	spec.jbb.JBButil.create_a_string_with_original
0.7%	1	0	java.math.MutableBigInteger.divide
0.7%	1	0	spec.jbb.Address.setUsingRandom
0.7%	0	1	java.util.HashMap.getNode
58.3%	66	22	Total compiled (including elided)

Table A.5: Profiling - Xprof compiled methods

	Exclusive Load-Acquire and Store-Release instructions	
1	`LDAXR`	Load-Acquire Exclusive register
2	`LDAXRB`	Load-Acquire Exclusive byte
3	`LDAXRH`	Load-Acquire Exclusive halfword
4	`LDAXP`	Load-Acquire Exclusive pair
5	`STLXR`	Store-Release Exclusive register
6	`STLXRB`	Store-Release Exclusive byte
7	`STLXRH`	Store-Release Exclusive halfword
8	`STLXP`	Store-Release Exclusive pair

Table A.6: Exclusive Load-Acquire and Store-Release instructions

Suffix	Flags	Meaning
EQ	Z set	Equal
NE	Z clear	Not equal
CS or HS	C set	Higher or same (unsigned >=)
CC or LO	C clear	Lower (unsigned <)
MI	N set	Negative
PL	N clear	Positive or zero
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher (unsigned >)
LS	C clear or Z set	Lower or same (unsigned <=)
GE	N and V the same	Signed >=
LT	N and V differ	Signed <
GT	Z clear, N and V the same	Signed >
LE	Z set, N and V differ	Signed <=
AL	Any	Always. This suffix is normally omitted.

Table A.7: Condition code suffixes - from arm.com

List of Abbreviations

AD Architecture Description

API Application Programming Interface

ARM Advanced RISC Machine

BCI Byte Code Injection

CORBA Common Object Request Broker Architecture

CPU Central Processing Unit

DMB Data Memory Barrier

IGV Ideal Graph Visualizer

IR Intermediate Representation

JAXP Java API for XML Processing

JAX-WS Java Api for XML Web Services

JDK Java Development Kit

JIT Just In Time

JRE Java Runtime Environment

JVM Java Virtual Machine

Bibliography

[ABC+06] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael FP O'Boyle, John Thomson, Marc Toussaint, and Christopher KI Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305. IEEE Computer Society, 2006.

[ABT] John Cavazos1 Grigori Fursin2 Felix Agakov, Edwin Bonilla, and Michael FP OâĂŹBoyle1 Olivier Temam. Rapidly selecting good compiler optimizations using performance counters.

[AGT89] Alfred V Aho, Mahadevan Ganapathi, and Steven WK Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):491–516, 1989.

[BH99] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in java. In *ACM SIGPLAN Notices*, volume 34, pages 35–46. ACM, 1999.

[Com15] TIOBE Programming Community. Toibe-index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2015.

[Cor05] Standard Performance Evaluation Corporation. Spec jbb 2005. https://www.spec.org/jbb2005/, 2005.

[FHP92] Christopher W Fraser, Robert R Henry, and Todd A Proebsting. Burg: fast optimal instruction selection and tree parsing. *ACM Sigplan Notices*, 27(4):68–76, 1992.

[Gro15] OpenJDK Adoption Group. Jitwatch. https://github.com/AdoptOpenJDK/jitwatch/, 2015.

[HKW05] Masayo Haneda, Peter MW Knijnenburg, and Harry AG Wijshoff. Optimizing general purpose compiler optimization. In *Proceedings of the 2nd conference on Computing frontiers*, pages 180–188. ACM, 2005.

[KJ95] Ron Kohavi and George H John. Automatic parameter selection by minimizing estimated error. In *ICML*, pages 304–312. Citeseer, 1995.

[Kli14] P Klijn. How accurately do java profilers predict runtime performance bottlenecks? 2014.

[Ltd14] ARM Ltd. Arm dmb. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka14041.html, 2014.

[McM13] Robert McMillan. Toibe-index. http://www.wired.com/2013/01/java-no-longer-a-favorite/, 2013.

[O'G15] Stephen O'Grady. Toibe-index. http://redmonk.com/sogrady/2015/01/14/language-rankings-1-15/, 2015.

[OoL07] Oracle and University of Linz. Idealgraphvisualizer. https://wikis.oracle.com/display/HotSpotInternals/IdealGraphVisualizer, 2007.

[Pro09] DaCapo Project. Dacapo-benchmark. http://www.dacapobench.org/, 2009.

[Ruf00] Erik Ruf. Effective synchronization removal for java. In *ACM SIGPLAN Notices*, volume 35, pages 208–218. ACM, 2000.

[SBP10] Adrian Schroter, Nicolas Bettenburg, and Rahul Premraj. Do stack traces help developers fix bugs? In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 118–121. IEEE, 2010.

[TVV+03] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, David August, et al. Compiler optimization-space exploration. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 204–215. IEEE, 2003.

[WWM08] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Visualization of program dependence graphs. In *Compiler Construction*, pages 193–196. Springer, 2008.