Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Automated program analysis for predicting memory access collisions

Semester Thesis

Joel René Büsser

jbuesser@ethz.ch

Computer Engineering and Networks Laboratory
Department of Information Technology and Electrical Engineering
ETH Zürich

Supervisors:
Andreas Tretter
Stefan Draskovic

Prof. Dr. Lothar Thiele

January 15, 2018

Acknowledgements

My thanks go to my supervisors, Andreas and Stefan, who guided me through this thesis. It has been a diverse learning experience for me. They earn my gratitude for their patience, their clear communication, and for sharing their knowledge and experience at any time.

I also thank Professor Thiele and his fellow researchers for the positive atmosphere at the institute. Thanks to him I had the opportunity to work in an up-to-date research area.

Abstract

Parallelization is one of the buzzwords in computer architectures. Along with more cores, or more processors, come new challenges for the memory. Often, multi-core systems share memory between several cores. The partition of the shared memory into memory banks to allow parallel accesses boosts the performance. Still, if two cores access data located in the same memory bank, the requests cannot be served at the same time. An access conflict occurs. To resolve this, the memory bank serves the requests one after the other. Thus, one core does not immediately get the data it needs for the execution of the thread. Consequently, the thread's execution time increases because it cannot proceed the execution until the memory bank has delivered the necessary data.

In this thesis, we analyze how memory access conflicts affect the execution time of a thread. We achieve this by measuring the execution time of the thread without any access conflict. Also, we analyze the number of memory accesses the thread performs. These figures serve as inputs to a probabilistic model which returns the expected increase of the execution time due to access conflicts. To answer the initial question, we verify the model.

We confirm that memory accesses and the increase of the execution time are related, although our model proves to be too inaccurate. We see the main reason for this in the small set of verification data. In general, we are on the right track, but further research for the model and the memory access analysis – like how the thread accesses memory over time – is necessary.

Contents

Chapter 1
Introduction

Over the course of hardware development history, there have been significant advances in parallelization. Mainly processors have gone through major development steps. Multi-core or even multi-processor systems are common architectures found in today's systems. Other hardware components, like the memory, adapt to these more complex concepts, too: Memory components are often shared between cores.

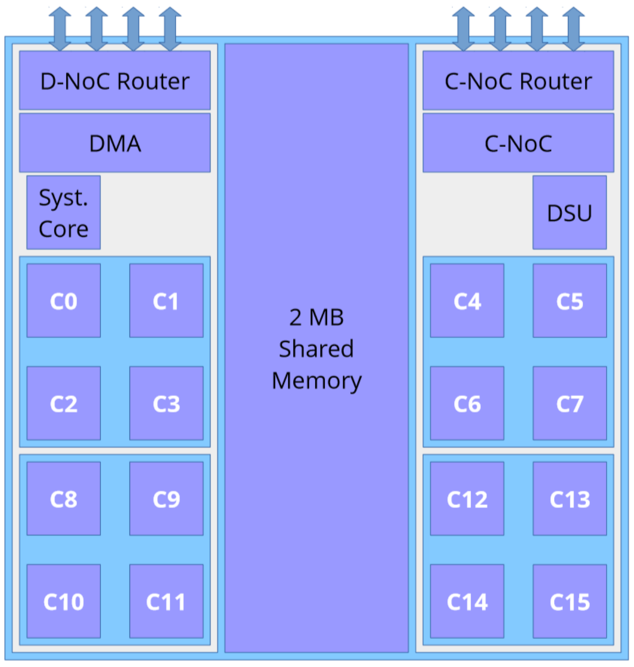

Figure 1.1 Single cluster of the Kalray MPPA-256 many-core CPU

In this thesis, we work with a multi-core system where several cores share memory. Figure 1.1 relates the theoretic concept to a concrete architecture. A phenomenon observed in this kind of architecture are access conflicts.

1.1 Access conflicts

The shared memory is partitioned into memory banks. Each bank can serve one request at a time. Thus, an access conflicts occurs whenever two (or more) cores access data stored on the same memory bank at the same time. To resolve the access conflict, the memory bank serves the requests one after the other. Consequently, lacking the requested data, the execution on the core that is served later is temporarily halted, hence increasing the execution time of the thread running on that core.
We want to model the increase of the execution time by relating the execution time without access conflicts to the number of memory accesses of a thread.

1.2 Goal of this thesis

To find out about the relation between the execution time without access conflicts and the number of memory accesses, we require a model which takes the execution time of a function and its memory accesses as inputs.

As there are no tools known to us which allow to analyze the memory accesses of a function, three steps are necessary:

1. Develop a tool that analyzes the memory accesses of a function.
2. Develop a model that predicts the average increase of the execution time.
3. Verify the model developed in step 2 by running tests on the Kalray MPPA-256 (MPPA).

Provided the number of memory accesses of a function, the model developed in step two, according to our statement, should provide accurate predictions. We verify the accuracy of the model by running tests on the MPPA. The MPPA is a suitable platform because of its determinism: the execution time of a program is the same whenever it is invoked with the same parameters.

1.3 Thesis outline

In Chapter 2, we introduce the LLVM compiler architecture which will help us developing the tool to determine the memory accesses of a function. In the same chapter, we discuss a related work which is relevant for the development of our model. In Chapter 3, we explain the problems we have to solve in greater detail. Chapter 4 describes how we solve steps one and two of the goals listed in section 1.2. It guides through the development of the access analysis tool and explains the different models we develop. In Chapter 5, we verify how well the prediction models developed in Chapter 4 match. Finally, Chapter 6 concludes the findings of the thesis and summarizes remaining and new questions.

Chapter 2
Background

This chapter discusses methods and techniques contributed by others which this work relies on. This includes an introduction to the LLVM compiler infrastructure in section 2.1. Why and how we use it follows in Chapter 3 and Chapter 4.
As for the theory, the results of the paper "On the Meaning of pWCET Distributions and their use in Schedulability Analysis" by Robert I. Davis, Alan Burns and David Griffin are relevant to us. The details are elaborated in section 2.2.

2.1 An introduction to LLVM

The term "LLVM" refers to a wide range of subprojects and has little to do with its original name, Low-Level Virtual Machine [1]. In our thesis, we refer to LLVM as the compiler we use and extend.

Classic compilers have three major stages, called frontend, middle end, and backend. The frontend as the first stage is language-specific, i.e. for different source code languages, a different frontend is to be used. [2] The task of the middle end is to optimize the code on a higher level than the backend optimizer which focuses on producing efficient code for the respective target platform. As the middle end optimizer is the only optimizer we consider, we henceforth use the term "optimizer" instead. The frontend translates the source code to the optimizer's language representation. To improve the code's runtime, the optimizer then performs a variety of transformations. Finally, the backend maps the code produced by the optimizer to the target instruction set.

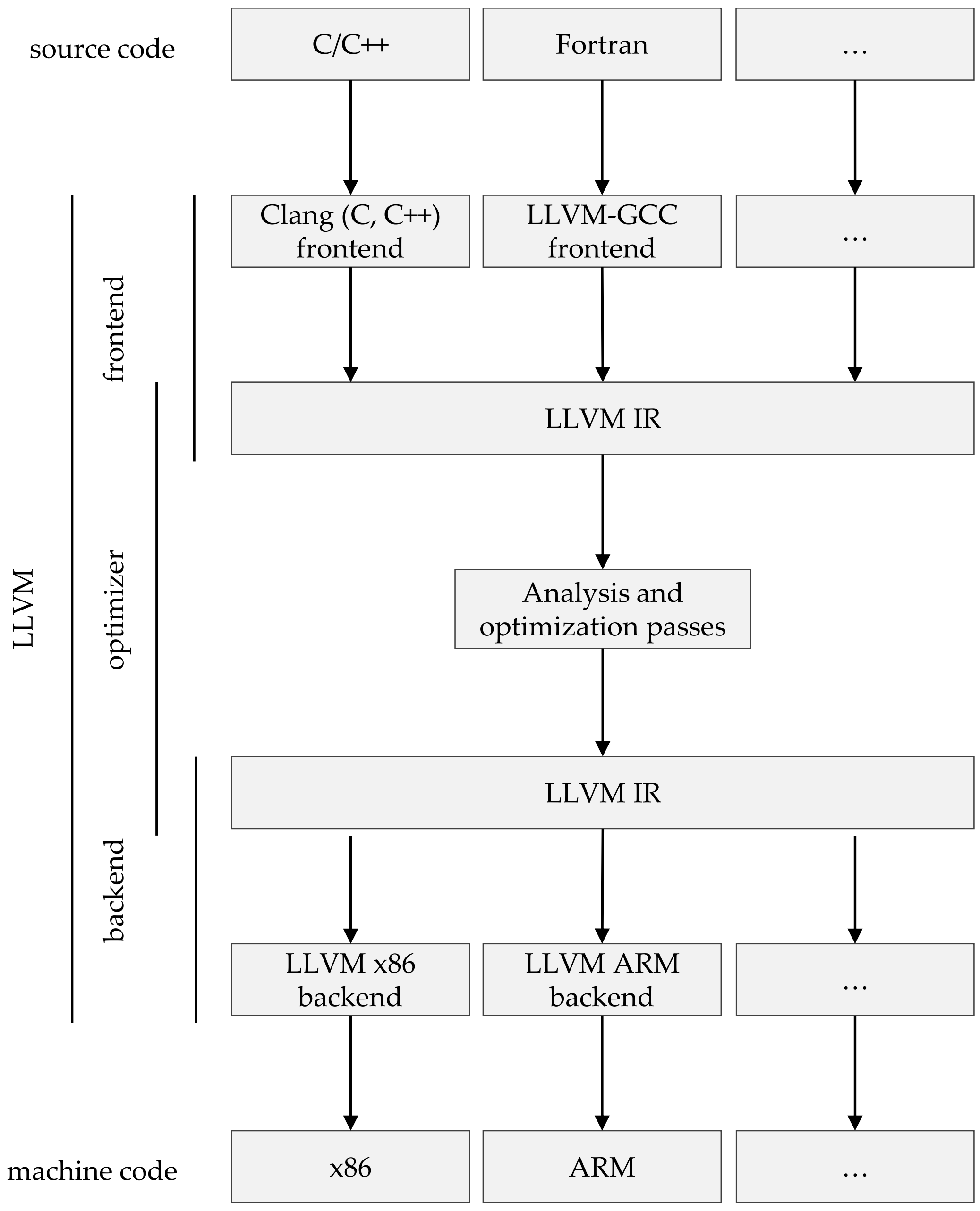

Figure 2.1 LLVM's modular architecture

The benefits of this three-stage design become apparent when a compiler – like LLVM – supports different source code languages and target platforms, see Figure 2.1. For such compilers, the common stage is a single optimizer which operates on a single type of code representation. Hence, the optimizer supports any frontend that can compile to the code the optimizer uses, and so it does any backend that can compile from the optimizer's code. The code the optimizer uses and produces is called the intermediate representation (IR). [2]

The Clang frontend, which was developed as part of the LLVM project, can be used to produce LLVM IR code from C and C++ source code.

2.1.1 IR code structure

The IR of the LLVM optimizer is mostly a three-address code which fulfils the Static Single Assignment (SSA) form property [3, p. 15f.]. A three-address operation takes one or two operands and produces a single result, therefore resembling an assembly language. The SSA property means that every operation that computes a new value stores this value to a new virtual register.

If we take a single operation – which, in LLVM, is called an instruction – as the smallest element in the IR code, the basic block is the next bigger, see Figure 2.2.

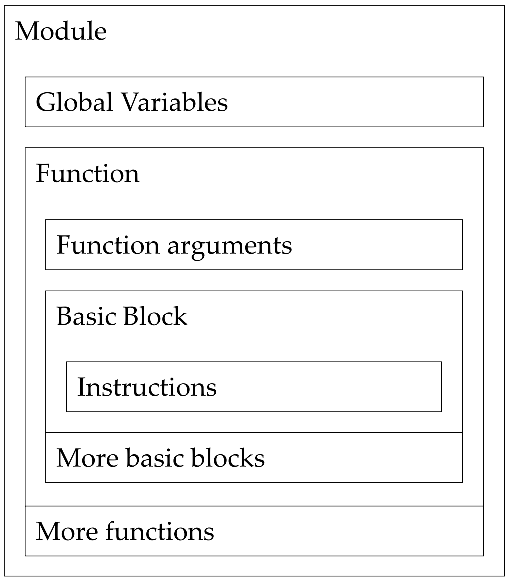

Figure 2.2 Internal structure of the intermediate code

A basic block, by definition, has a single entry and a single exit point [4, p. 231]. This implies that all instructions between the entry and exit point are executed once, if the basic block is entered.

A function, then, consists of one or more basic blocks and, possibly, of function arguments. Note that, although we are describing the LLVM IR, this matches with the concept of a function, or method, in a programming language like C++ or Java. For the lack of dedicated terms and the fact that a function in the source file is a function in the IR, we do not distinguish the two contexts.

Finally, all functions together with global variables (as known from the source code) are packed into a module. The IR code we will analyze contains a single module. If we view this from the source code perspective, the module includes all the functions and variables. Thus, we permit ourselves to use the less abstract term "program" instead of module.

We know by now that the IR is what the optimizer works with. Next, we discuss how it does so.

2.1.2 The LLVM optimizer

Optimizations are implemented as passes. A pass is source code that traverses a certain portion of a program's code. Analysis passes compute information which either serves visualization purposes (e.g. print a control flow graph) or are used by transform passes. Transform passes mutate the program to improve its execution time (ET), memory usage, or other measures. Finally, there are utility passes which, for example, write a module to bitcode. They neither fit the categorization of analysis nor transform passes.

For the different levels of the code structure in Figure 2.2, there are respective passes to match the abstraction level on which analysis or transformations are performed (module, function, or basic block pass). For specific constructs - functions may include loops, for instance – LLVM provides also corresponding passes.

2.2 The access conflict prediction model

In section 1.2, we explained that we use the MPPA as our test platform because of its deterministic behavior. However, a deterministic model would involve knowing all states of all cores and memory banks. In short, today's processor architectures are complex, and it is infeasible to understand every detail about it. This has consequences since precise ETs of a program cannot be calculated [5, p. 3].

Because we are not going to analyze the hardware on which we run the test programs, we face epistemic uncertainty. Epistemic uncertainty results from the lack of knowledge about the system. If measurements are based only on a subset of possible inputs and states, we cannot be absolutely confident that the resulting estimate we derive is correct [5, p. 3].

The paper by Davis, Burn, and Griffin [5] discusses this and concludes that a probabilistic approach can be taken. Applied to our problem, this means that the prediction model must be probabilistic.

Chapter 3

Problem to solve

A function, referring to code operating on inputs and output arguments, is to be analyzed. For some given source code of a function or a program comprising several functions, we are given the ET without any interference. We call this the "individual" or "sequential" ET. We are also given the number of memory accesses of every function and, hence, also of the program. Based on these two figures, we develop a model which predicts the average increase of the ET due to memory access conflicts. The increase refers to the difference between the "combined" or "parallel[1]" ET, which is the ET with access conflicts, and the individual ET.

3.1 Model

As we are interested in the memory access conflicts, we base our analysis on the number of memory accesses of the function of interest, subscript k, and on other functions running in parallel, subscript i (and others):

$$\Delta t_k = t_{k,comb} - t_{k,ind} = f\left(t_{k,ind}, N_k, t_{i,ind}, N_i, \dots\right) \tag{3.1}$$

Here, $t_{k,ind}$ indicates the sequential ET of function k. N_k refers to the number of memory accesses of function k. The parallel ET is denoted as $t_{k,comb}$. The figures with the subscript i refer to another function being executed on another core at the same time.

[1] Defined in section 1.1, an access conflict only occurs when two or more threads run in parallel on different cores.

Hence, according to equation (3.1), we need the individual ETs and the number of memory accesses of functions k, i, and all others being executed at the same time to predict the parallel ET of a function k. These numbers serve as inputs to the model, currently referred to as a blackbox function f.

While determining the ET $t_{k,ind}$ is possible with existing tools, a tool to detect memory accesses is to be developed.

3.2 Tool

The tool reports how often every variable in a program is read from and written to. Read and write operations are summarized as (memory) access. We group variables roughly into global variables, function arguments, and local variables. Though we are interested in all of them, we focus on global variables and function arguments. For now, this is sufficient[2].
For the first analysis and verification process, we limit the tool to report the total number of memory accesses (the sum of all reading and writing accesses to every variable considered) of each function of a program, N_k in equation (3.1). The model and the tool are targeted towards the analysis of individual functions; knowing the memory accesses and the prediction of every function, we can also draw conclusions for the entire program.

[2] We make sure that the programs we test our pass on mainly operates on function arguments and global variables.

Chapter 4
Development

We split the entire problem into two parts in Chapter 3. Part one involves the development of a tool which analyzes the memory accesses of a function. Part two is the model which, according to our findings in section 2.2, is a probabilistic model.

The question remains how to implement part one. We certainly need to analyze the source code of the function of interest. Ideally, we inspect the bitcode because it exactly describes what the hardware does. Especially, it contains load and store instructions, both of which are a memory access. The LLVM optimizer (section 2.1) is made to analyze and even modify bitcode. Hence, a possible and suitable way of implementing the tool, is to write an analysis pass as part of the LLVM optimizer.

4.1 Tool implemented as a compiler pass

We explained in section 2.1 what a compiler pass in LLVM is and what it can do. For the subsequent implementation of the tool – which we synonymously refer to as "pass" from now on – we follow a bottom-up approach. This means that we start by analyzing the instructions themselves. In a second step, we want to know how often the instruction is executed. Finally, we connect this information to obtain the number of memory accesses, n_i in formula (3.1).
This procedure corresponds to a start at the most detailed level as depicted in Figure 2.2.

4.1.1 Basic instruction analysis

In a first step, we detect memory accesses in the IR of the program we analyze. In LLVM, there are two instructions which access memory: load and store. By iterating through all instructions of a function (and through all functions in a program), we identify these. The important information is which variable is read or written. For variables holding a single value, this is easy to determine. Problematic are arrays and other indexed data structures. By indexing arrays, LLVM obtains a new address derived from the array base address. Since, however, this is accomplished with the "Get Element Pointer" (GEP) instruction, we can track the indexing back to the base address.

This first approach analyzes every instruction exactly once. That is, each load or store instruction increases the memory access count by exactly one. It could happen, though, that a function is never called or that there are loops. In any case, to get a true memory access count, it is crucial to obtain information about how often a basic block – and therefore the instructions therein – is executed.

4.1.2 Determine block execution count

We first think about what influences the execution count of a basic block. A coarse-grain number is how often the function to which the basic block belongs is called. In detail, we must know what the specific basic block is: Is it part of a loop or is it one of several basic blocks out of which only a few are executed, depending on the specific conditions (i.e. branches)?

Let us first consider loops. The loop count states how often a loop is executed. Clearly, the loop count, therefore, determines how often basic blocks which are part of a loop are executed. In LLVM, there are analysis passes which aid determining the loop count. However, the condition is that the code has been optimized before already. Specifically, scalar evolution analysis, induction variable simplifications and loop rotation are necessary. According to the LLVM documentation, "[the] ScalarEvolution analysis can be used to analyze and categorize scalar expressions in loops. […] Given this analysis, trip counts of loops and other important properties can be obtained." [6] This is precisely what we need.

Branch analysis is a complex topic by itself. Often, it is impossible to safely detect the branch that is executed. Like branch prediction algorithms in hardware, we,

too, needed to make assumptions. Branch prediction in compiler design, being an ongoing research topic [7], is beyond the scope of this thesis. Therefore, for simplicity reasons, we think of a rule we apply to all branches to avoid branch prediction. Such a rule could be to calculate the average number of memory accesses of all branches or to always pick the branch with the most memory accesses. In any case, this simplification affects the accuracy of the prediction model because we no longer obtain an exact memory access count. We choose to consider the worst-case branch. If the functions the pass will analyze do not have heavily unbalanced conditional branches – that is, one branch has many memory accesses while the other has none – the simplification still provides reasonable results.
The branch analysis is not implemented in our pass; for now, we are satisfied with an accurate loop analysis.

A last number the pass analyzes – again, this is not implemented in our pass – is how often each function is called. This requires analyzing of the call instruction. A construct to consider separately is recursion, i.e. when a function calls itself. The LLVM transform pass "Tail Call Elimination" [6] transforms recursive calls to loops. These loops can then be analyzed as described before.

Lastly, we must put together the instruction analysis of section 4.1.1 and the block execution count analysis of section 4.1.2 to obtain the memory access count.

4.1.3 Tree construction

There are multiple ways to obtain a result of the memory access analysis for every function. One is to use metadata in LLVM [8] [9]. This directly includes the information into the IR. Because it requires modifying and then reading the IR, it is conceptually a larger effort than the second solution we propose.
A more generic approach is to build a tree structure for each function. The tree consists of three different node types: Leave nodes represent a basic block and contain the number of memory accesses of that basic block, obtained by the steps described in sections 4.1.1 and 4.1.2. For loops in the function, "loop nodes" are created and conditional branches are children, of the "if-else node".

Figure 4.1 shows the control flow graph (CFG) of a sample function. Based on the CFG, we construct a tree. For the evaluation, every leaf node of that tree determines the memory accesses of the basic block this leaf node represents.

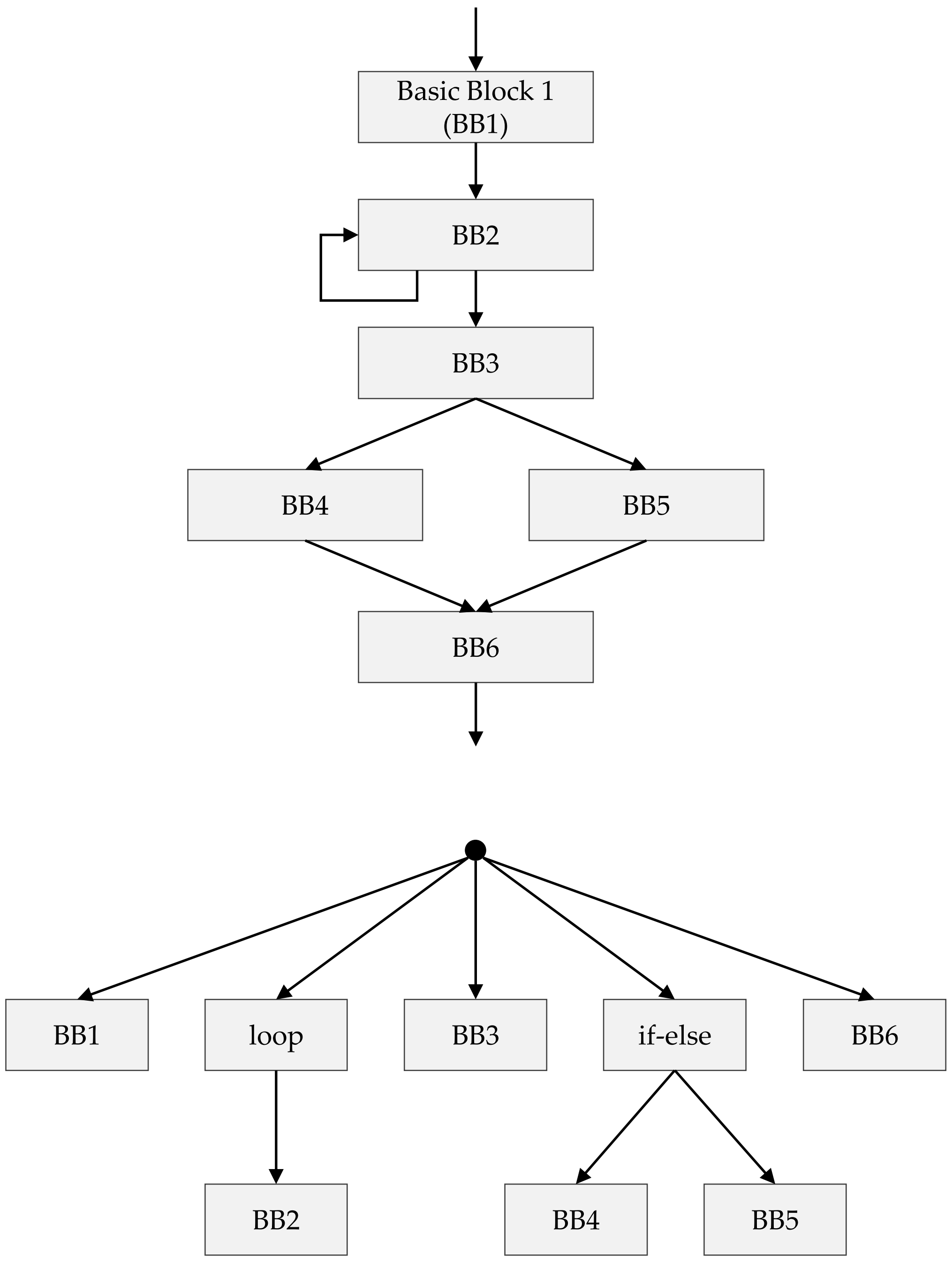

Figure 4.1 Control flow graph (top) and resulting tree ready for memory access evaluation (below) of a sample function

If one or several leaves are child nodes of a loop node, the loop node multiplies the values by the loop count of the corresponding loop. According to the simplification made in section 4.1.2, the if-else node, if it were implemented in our pass, takes the larger memory access count of the two branches.

Now that we have a tool which analyzes the memory accesses, we proceed to the development of the probabilistic model.

4.2 Model

Throughout the development of the model, we assume that there is a dependency between the number of memory accesses and the increase of the ET Δt due to access conflicts.

We start with a model that is independent of the function's memory accesses, see equation (4.1). The reasoning for this model is that, based on a set of measurements, the average increase c_1 is valid for every program in any setup. (4.1) expresses the expected increase of thread k. c_1 is the constant we train before using it on other data. T represents the number of threads[3] running in parallel, i.e. $(T-1)$ is the number of threads which thread k could collide with.

$$\Delta t_k \triangleq t_{k,par} - t_{k,seq} = c_1 \cdot (T-1) \cdot t_{k,seq} \tag{4.1}$$

As we state that the memory access count is relevant to compute the increase in the ET, we next derive a model which directly relates these two quantities, see equation (4.2). c_2 is, again, the constant we obtain by training.

$$\Delta t_k = c_2 \cdot (T-1) \cdot N_k \tag{4.2}$$

However, (4.2) does not consider the time span during which the memory accesses occur, i.e. the prediction is independent of $t_{k,seq}$.

We consider the sequential ET again for the last model. We assume a linear distribution of the memory accesses in time. Thus, we model the probability that thread k accesses memory at any cycle during its execution by

[3] Refers to the function we analyze being executed on a core.

$$\frac{N_k}{t_{k,seq}}.$$

For the other functions, the distribution is analogue. We use this distribution to calculate the probability of an access conflict, i.e. the probability that thread k accesses memory at the same time as any of the other threads running. Because we assume the threads to access memory independently of each other, the probability of an access conflict is p_c, the sum of the joint probabilities. The formula for this model is given in equation (4.3).

$$\begin{aligned} \Delta t_k = c_3 \cdot t_{k,seq} \cdot p_c = c_3 \cdot t_{k,seq} \cdot \left(\frac{N_k \cdot N_1}{t_{k,seq} \cdot t_{1,seq}} + \cdots + \frac{N_k \cdot N_n}{t_{k,seq} \cdot t_{n,seq}} \right) \\ = c_3 \cdot N_k \cdot \sum_{\substack{i=1 \\ i \neq k}}^{n} \frac{N_i}{t_{i,seq}} \end{aligned} \tag{4.3}$$

Now that we proposed three prediction models, we conclude section 4.2 by informally stating how well we expect them to correlate with the actual measurements. The actual results are discussed in section 5.2.

4.2.1 Expected results of the models

The simplest model (equation (4.1)) is expected to behave well for data that is similar to the training data. However, we expect it to be the least accurate of the three models with new data.

Because model two (equation (4.2)) takes additional parameters, namely the memory accesses, into account, we expect it performs better than the first model on new data. However, since the memory access count is unrelated to the individual ET, we still suppose that it fails in cases that differ from the training data.

Lastly, we assume the third model (equation (4.3)) to be the most accurate. The reasoning is that it includes a probability measure for the access conflicts depending on the other tasks running in parallel.

We verify the models and discuss the results obtained in Chapter 5.

Chapter 5

Verification and results

The general procedure to verify our prediction models developed in section 4.2 is as follows: First, we gather sequential and parallel ETs as well as the memory access count (using the pass developed in section 4.1) for different configurations of a program. We execute the program for each configuration with two, three, and four threads. The set of measurements of one thread setting serves as training data for the models. The other thread settings are then used for the verification of the model.

5.1 Obtain execution time

More precisely, we run a matrix multiplication on the MPPA. The crucial part is the use of the memory banks. For now, we limit the program to use only one bank, such as to force access conflicts. Because we want to find out about the relation between access conflicts and the ET, this is not a limitation per se. However, further tests with different memory configurations should be considered at a later stage.
The test program mainly runs two functions: The first generates two matrices (matrix generation function), the second multiplies the two matrices (matrix multiplication function). Increasing the number of threads running in parallel spawns more threads to be generated and multiplied (rather than splitting the functions themselves). We map the different threads – and thus the two functions – to the same core to obtain the sequential ETs and to different cores to obtain the parallel ETs.

Before we present and discuss the results of these measurements, we define the following figures in addition to the ones introduced previously:

- The relative increase of the ET: $\Delta t_k / t_{k,seq}$.
- The absolute increase of the ET predicted by model j for function k: $\Delta t_k^j,\ j \in \{1,2,3\}$.

5.2 Discussion of the results

We first take a general look at the measurements before applying our models. Figure 5.1 includes all measurement points of the matrix generation function. Due to the smaller sized matrices for three and four threads – recall that every thread creates two matrices – there is a clear reduction of the sequential ET. We also recognize that, for two threads running in parallel (red points), there is hardly any difference between the parallel and sequential ET. Despite this, we clearly observe different correlations for two, three, and four threads. That is, the points are aligned along three distinctly sloped lines. The exception are a few measurements obtained with four threads (blue dots), which, again, follow another (steeper) slope.

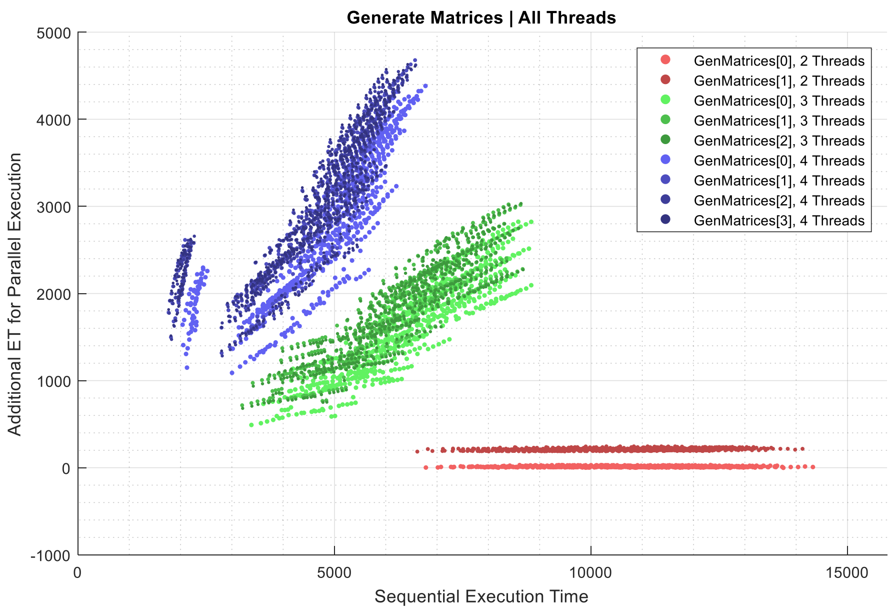

Figure 5.1 ET measurements of the matrix generation function

For the second function of the test program, the matrix multiplication, the measurements exhibit some interesting behavior, see Figure 5.2. Firstly, many configurations, according to the measurement data, perform better when running in parallel with other threads. This is surprising because the code running is the same; since, however, the functions running in parallel access the same memory bank, there probably are memory access conflicts, thus increasing the execution time. Hence, we certainly do not expect to observe a decrease of the function's ET.

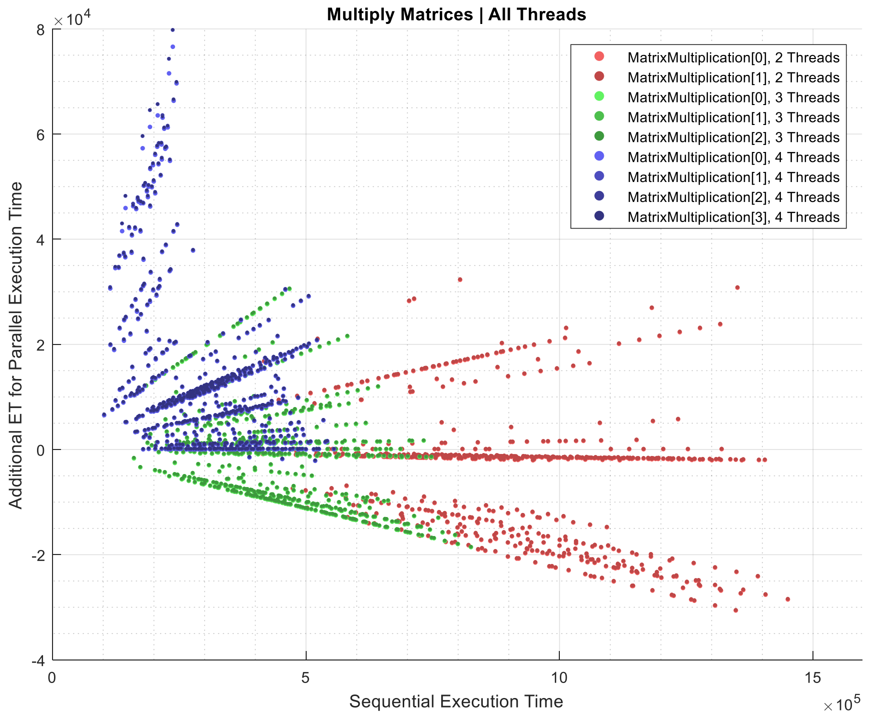

Figure 5.2 ET measurements of the matrix multiplication function

From these two plots, and especially from Figure 5.2, we conclude that there are phenomena other than access collisions taking place on the MPPA which we are neither aware of nor are they modeled by any of our prediction models. Since, however, the measurements of the matrix generation function, especially for three and four threads, exhibit a behavior which suggests a relation between access conflicts and the parallel ET, we henceforth apply our models only to that function.

5.2.1 Applying the first model

Recalling equation (4.1), we now train the constant c_1 using the measurement points of the matrix generation function with three threads (green dots in Figure 5.1). Solving for c_1, we get

$$c_1 = \frac{\Delta t_k}{(T-1) \cdot t_{k,seq}} = \frac{t_{k,par} - t_{k,seq}}{(T-1) \cdot t_{k,seq}}.$$

We use the data from three threads, thus $T = 3$. For simplicity reasons, we take the median value, i.e. the horizontal line which partitions the points in Figure 5.3 into two equally sized sets. Noting the range on which the measurements are distributed, calculating the median value yields a coefficient which is not accurate for many points and, thus, also results in a bad prediction.

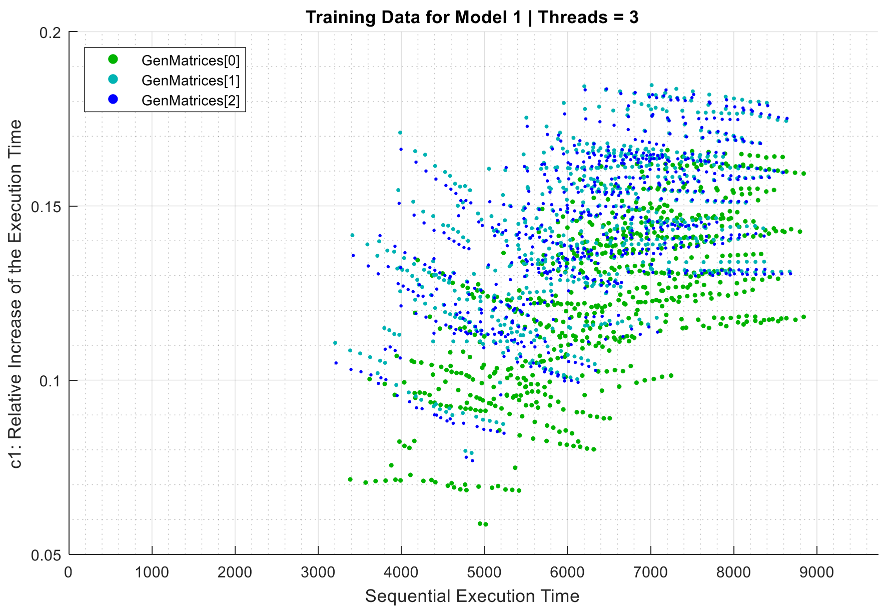

Figure 5.3 Training data for c_1

However, a different approach, like calculating a regression line, is not feasible either for the same reason. Nevertheless, we proceed and apply the constant to the data for two and four threads to find out whether our concerns are true, see Figure 5.4, which shows how accurate the prediction model works. Along the ordinate (y-axis), Figure 5.4 shows the difference between the measured and the predicted increase of the ET, i.e. $\Delta t_k - \Delta t_k^1$.

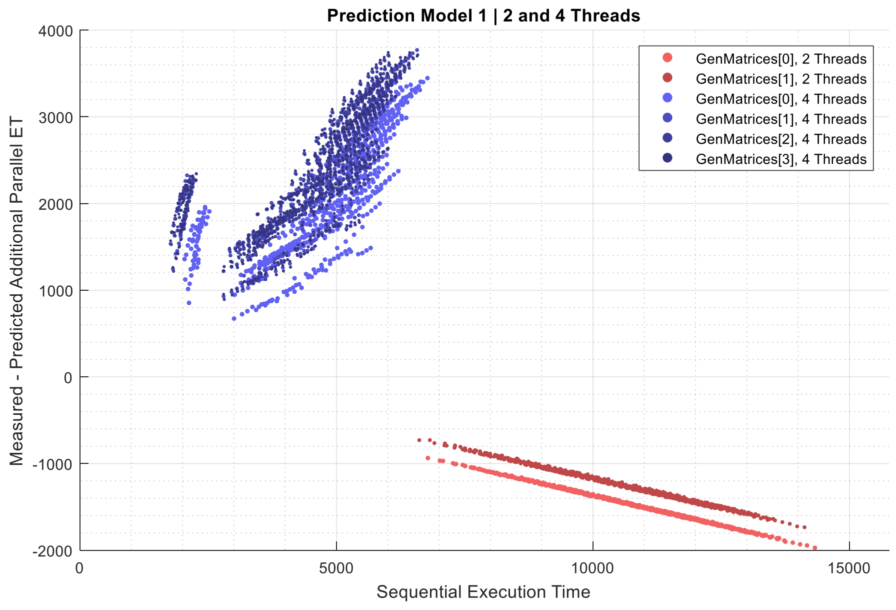

Figure 5.4 First model applied to the matrix generation function with two and four threads

Because our model directly relates the sequential ET with the increase in parallel execution, and because the matrix generation function with two threads does not exhibit the expected behavior, the model is too pessimistic about the parallel ET for two threads. For four threads, however, it is too pessimistic.

As it is the simplest of the models, we did not expect accurate results. For the derivation of the results of models two and three (equations (4.2) and (4.3), respectively) the steps we go through are the same, thus we refer the reader to the Appendix A, where all plots are shown.

5.2.2 Applying the second and third model

Also, we omit the data from two threads as we are aware that the models do not work because of the insubstantial additional ET for parallel execution.
Unlike the first model, the second and third are plotted against the number of memory accesses, because the main contributor to the model is no longer the sequential ET but the memory access count. As can be seen in Figure A.7 and Figure A.10, we again face the issue that the median value results in a bad prediction. Therefore, we again see inaccurate predictions of the increase of the parallel ET, though the range of both, $\Delta t_k - \Delta t_k^2$ and $\Delta t_k - \Delta t_k^3$, is smaller and closer to the objective value zero than that of $\Delta t_k - \Delta t_k^1$. Hence, compared to the first model, we do see an improvement of the prediction models which consider memory accesses.

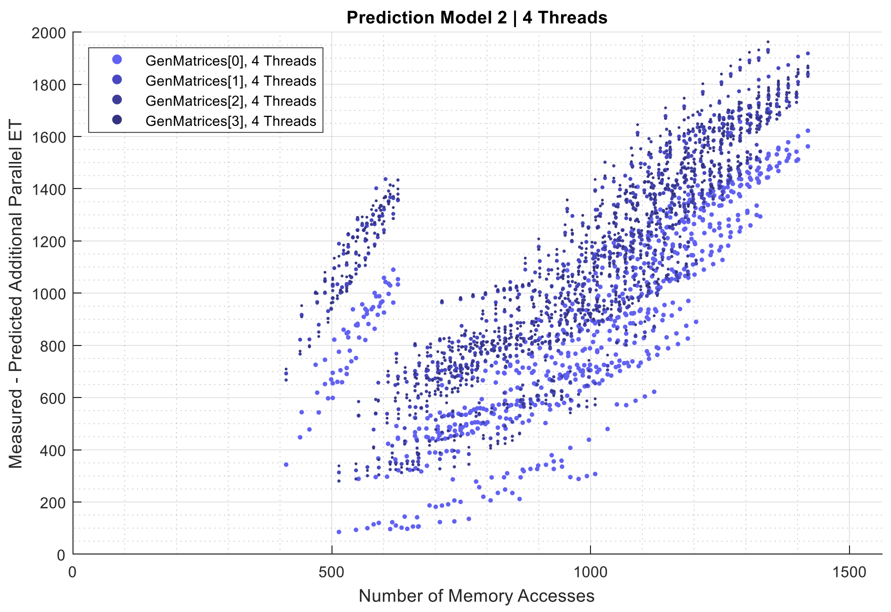

Figure 5.5 Second model applied to the matrix generation function with four threads

Comparing Figure 5.5 with Figure 5.6, it is difficult to state which model provides "better" results; the range of error is almost the same, both models almost equally pessimistic. However, we draw the attention to one detail: The data points in

Figure 5.5 seem to correlate along certain lines with different slopes. A few points – the ones to the left and below – are disconnected from most of the measurements. In the third model, that is, in Figure 5.6, these points are indistinguishable from all others. Furthermore, the predicted number of access collisions and the execution time prediction error clearly correlate along an individual line. We conclude from this that the third model manages to explain some effects the other two models do not; what is missing is a linear coefficient which would correct the prediction error. As the difference between the equations lies solely in how we predict the access conflicts, we are optimistic that the general approach is valid.

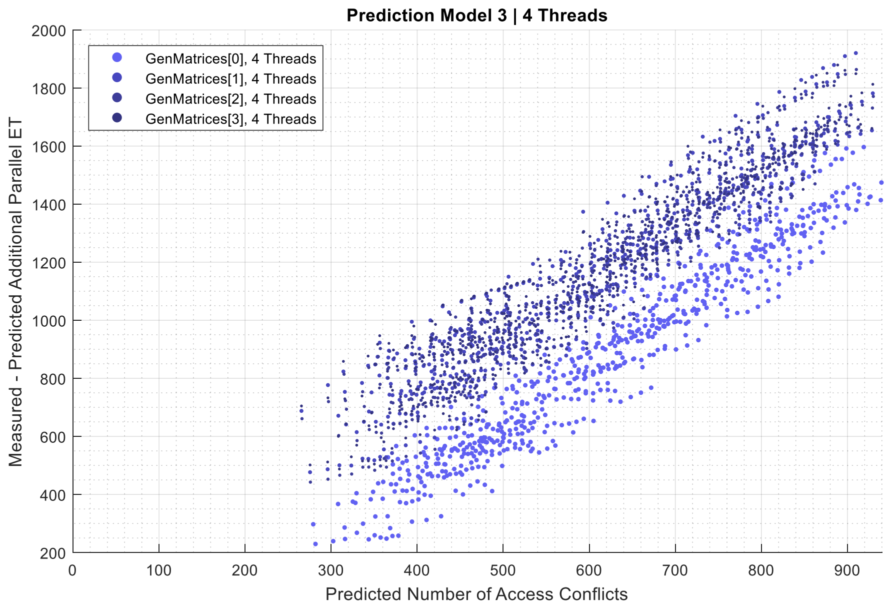

Figure 5.6 Third model applied to the matrix generation function with four threads

Chapter 6

Conclusions

The goal of this thesis was to find means and ways to determine the execution time of a function running in parallel based on its execution time without any interference. We identified three steps that are necessary, and which are likely needed to be performed more than once. That is, in case the verification of the model is not satisfactory, we return to step one to improve the tool, implemented as a compiler pass and, in step two, develop new probabilistic models. With the results presented in Chapter 5, this is undoubtedly necessary.

Furthermore, we clearly compromised on every step:

- The compiler pass generally provides useful and correct results for the programs we analyzed. However, it lacks the implementation of conditional branch analysis and the analysis how often every function is called. These two limitations need to be addressed, otherwise the code to analyze is implicitly assumed to be free of constructs the pass would fail on (like recursion or conditional branches).
  We laid the foundations on top of which further analyses can be performed. The concept to create a tree structure to determine the memory access count is a generic approach and allows extensions to be implemented without the need to redesign existing parts of the pass.
- We proposed three models, of which the third is the most advanced and promising. Nevertheless, even the third model did not provide accurate predictions. One reason is certainly the amount of training data we used, which is too little. Realizing that the parallel execution time of a single function is sometimes shorter than the sequential, and that there is fundamentally different behavior between different number of threads running in parallel, this suggests that there are many processes more we do not understand.

However, it must not be the goal to find this out yet. As a next step, we propose to stick to the derived models, and to collect data with more threads, hoping to find more correlations of the kind the matrix generation function with three and four parallel threads exhibits. If this is the case, we can make a clearer statement about the practicability of the models. Because the training data is rather uncorrelated, we question the approach how we train the model parameters. This is, of course, also an indicator that the models we derived are impractical. In case of the third model, we propose to consider a different probabilistic distribution of the memory accesses (or to derive a fourth model). Possibly, it helps to extend the analysis pass such that information about how memory accesses are distributed over time are obtained, or to find, at least, parts in programs where there are many memory accesses in few lines (burst detection).

- Lastly, we also simplified the verification process. We mentioned already that more training data is necessary for the models. Additionally, the models must be applied to other programs. This also helps to locate, for instance, the source of the behavior of the matrix multiplication function; the more measurements we get, the less is the impact of odd behavior.

Despite these compromises, we can draw a positive overall conclusion. Partly, because the access analysis tool is stable and working and partly, because the verification of the third model hints at an essentially correct approach. Also, we were aware that we need to go through the steps presented in section 1.2 several times to achieve an accurate and reliable prediction. The results obtained so far emphasize this need.

Appendix A

Result data

On the following pages is every plot relevant for the evaluation of the probabilistic models we derived in section 4.2. For each function in a thread, there are 660 different measurements, i.e. 660 different matrix configurations.

A.1 Measurements for matrix multiplication and matrix generation functions with two, three and four threads

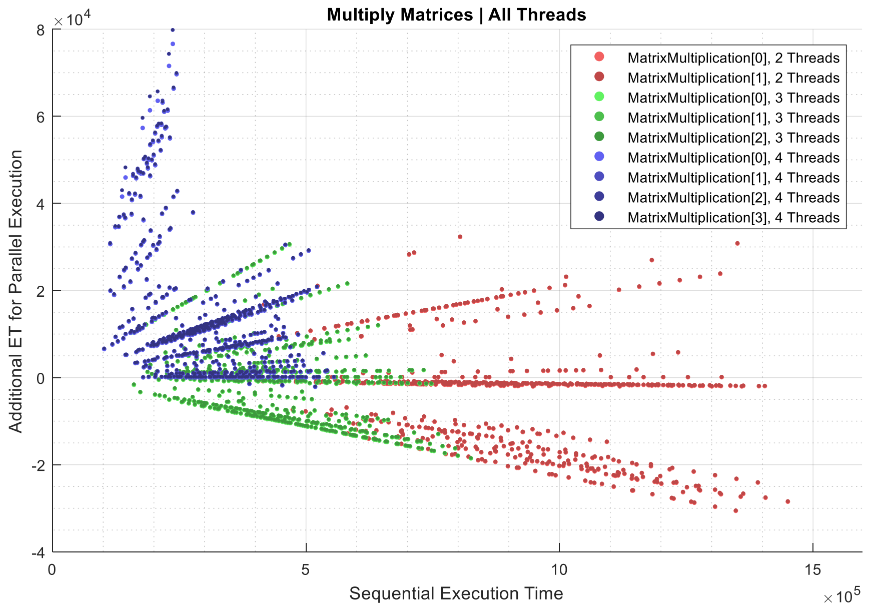

Figure A.1 Execution times for the matrix multiplication function

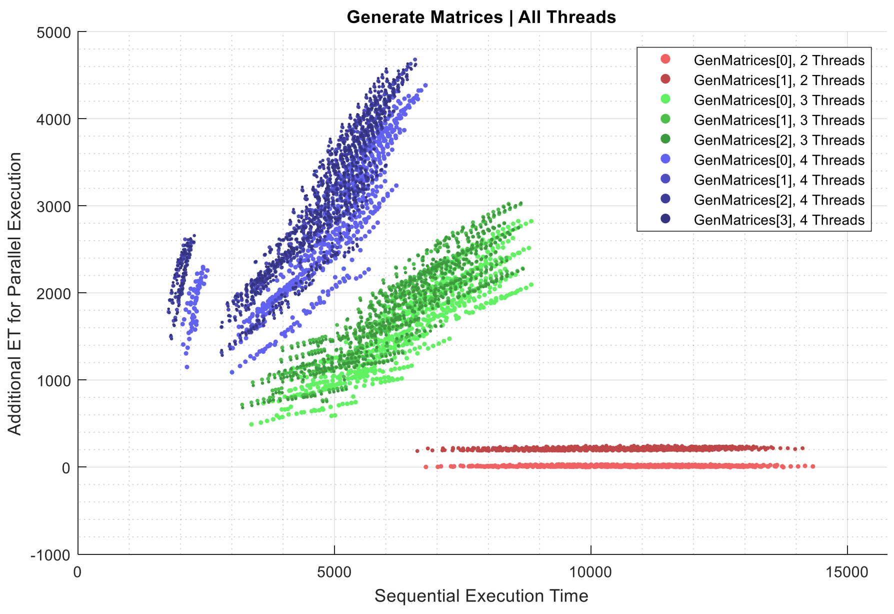

Figure A.2 Execution times for the matrix generation function

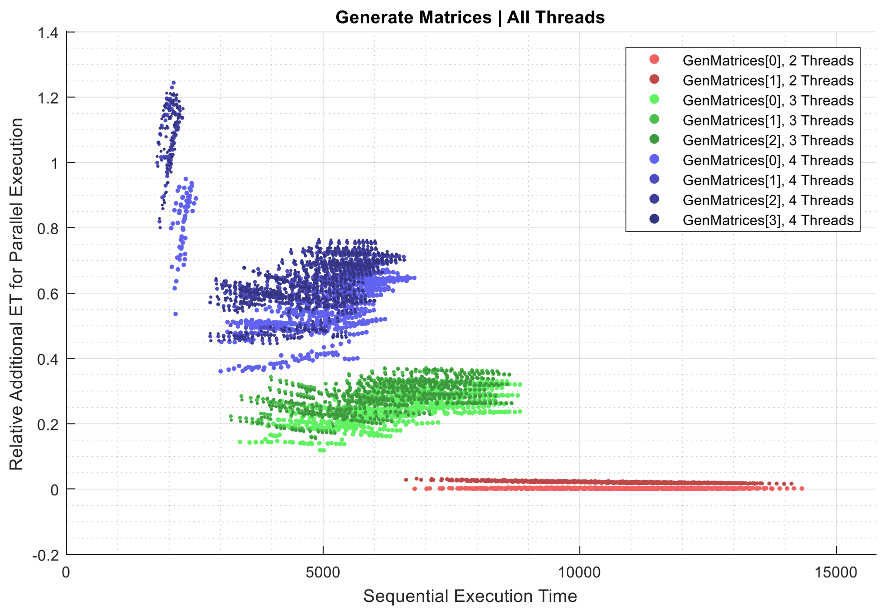

Figure A.3 Relative increase in the execution time for the matrix generation function

A.2 First model

From (4.1),

$$c_1 = \frac{\Delta t_k}{(T-1) \cdot t_{k,seq}}.$$

k is a placeholder for all configurations of every function of all threads.

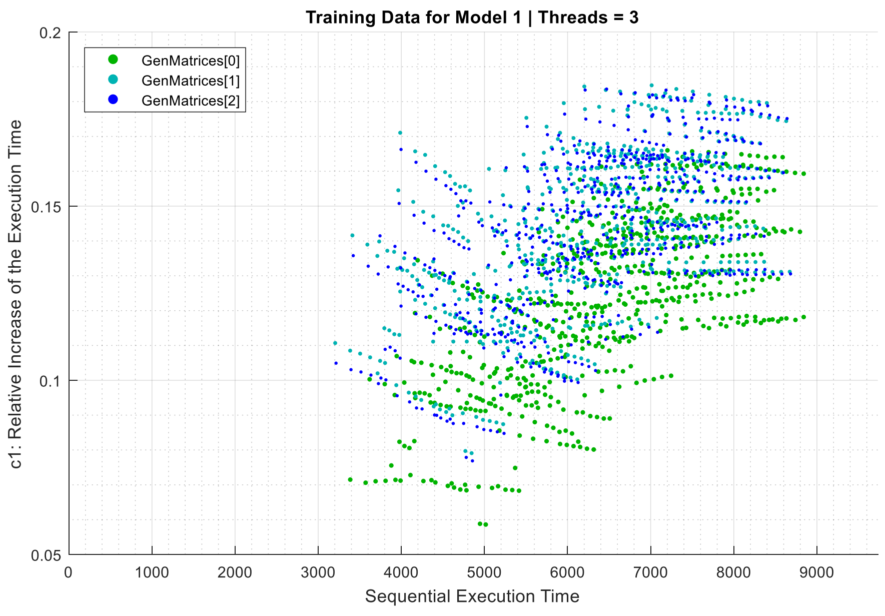

Figure A.4 Training data for model one

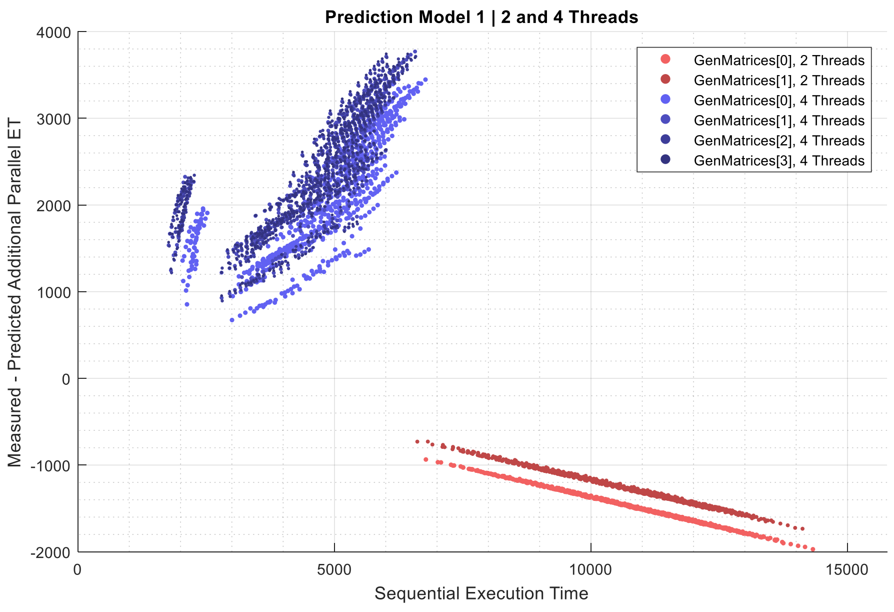

Figure A.5 Model one applied to functions with two and four of them running in parallel

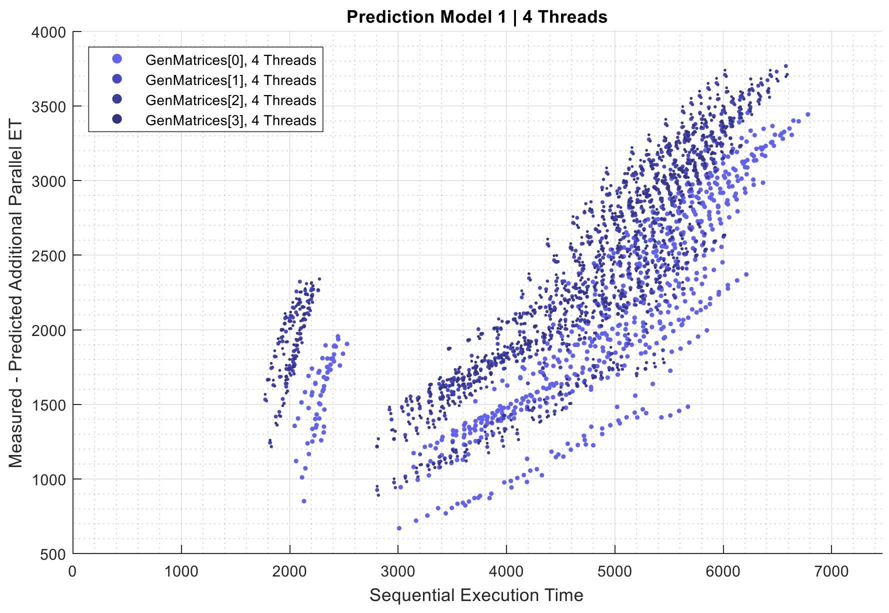

Figure A.6 Model one applied to functions with four of them running in parallel

A.3 Second model

From (4.2),

$$c_2 = \frac{\Delta t_k}{(T-1) \cdot N_k}.$$

k is a placeholder for all configurations of every function of all threads.

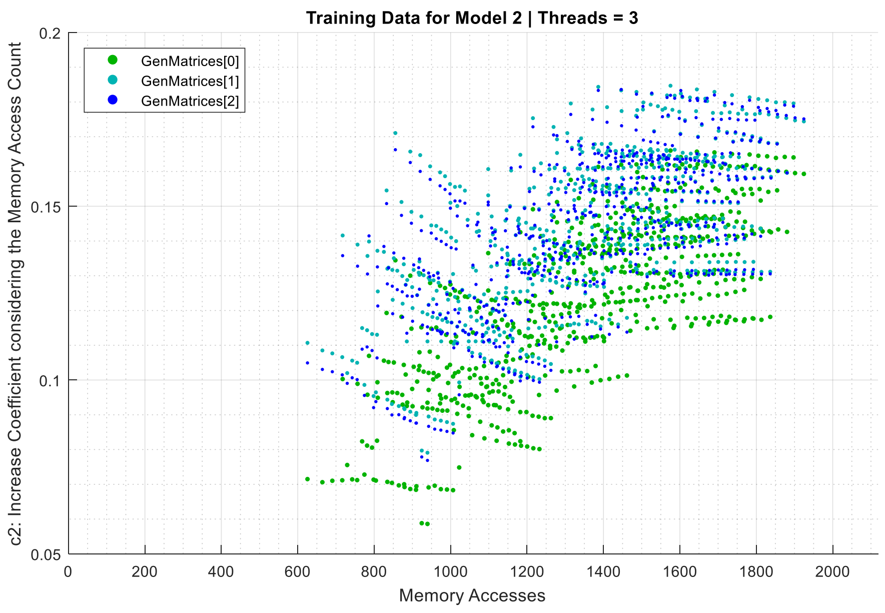

Figure A.7 Training data for model two

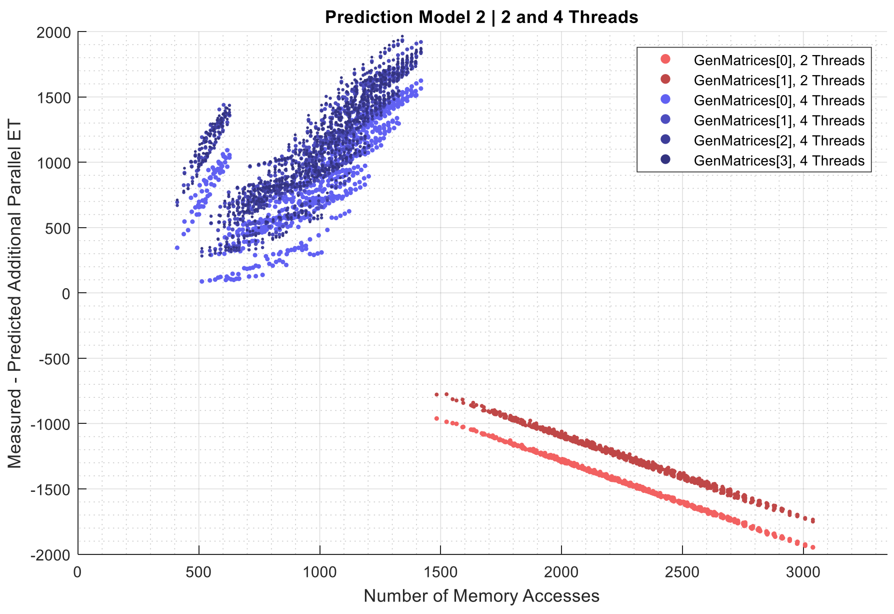

Figure A.8 Model two applied to functions with two and four of them running in parallel

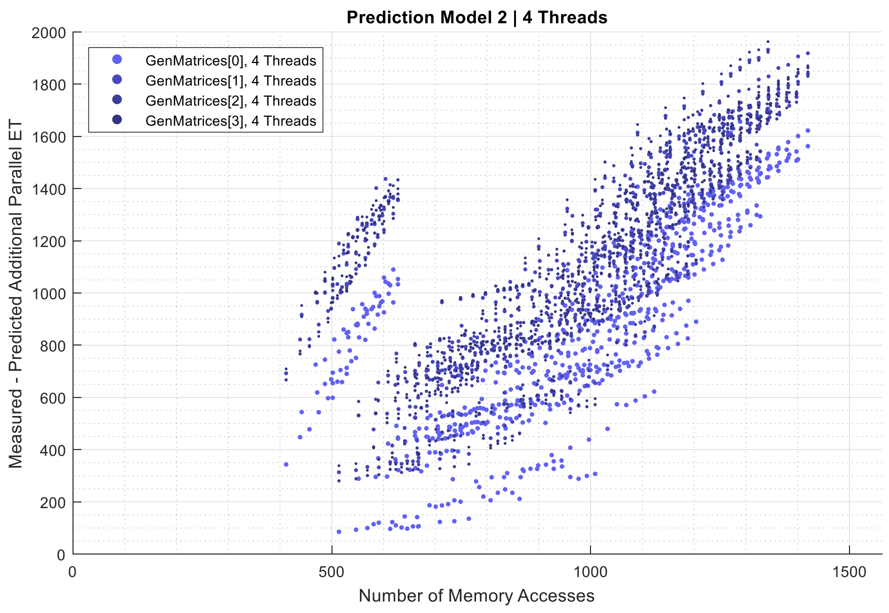

Figure A.9 Model two applied to functions with four of them running in parallel

A.4 Third model

From (4.3),

$$c_3 = \frac{\Delta t_k}{N_k \cdot \sum_{\substack{i=1 \\ i \neq k}}^{n} \frac{N_i}{t_{i,seq}}}.$$

k is a placeholder for all configurations of every function of all threads.

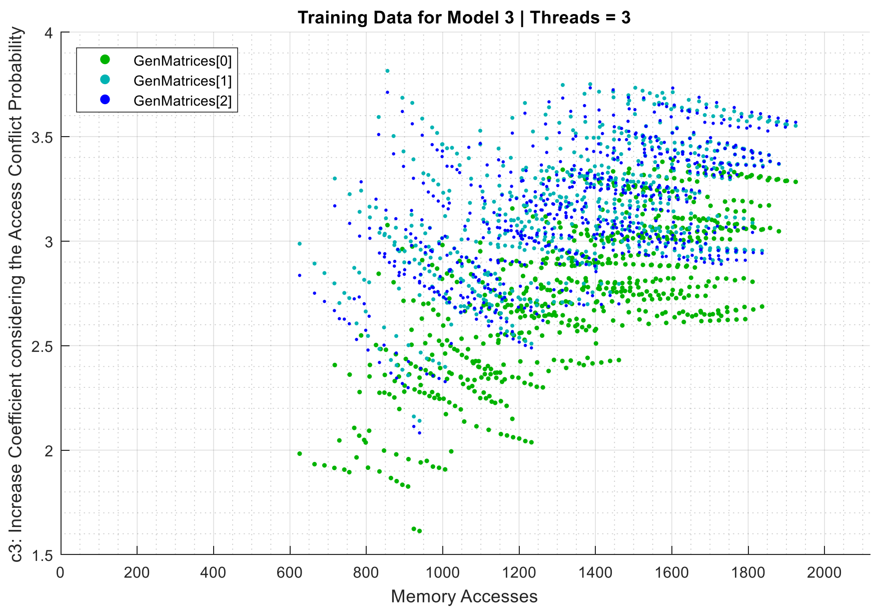

Figure A.10 Training data for model three

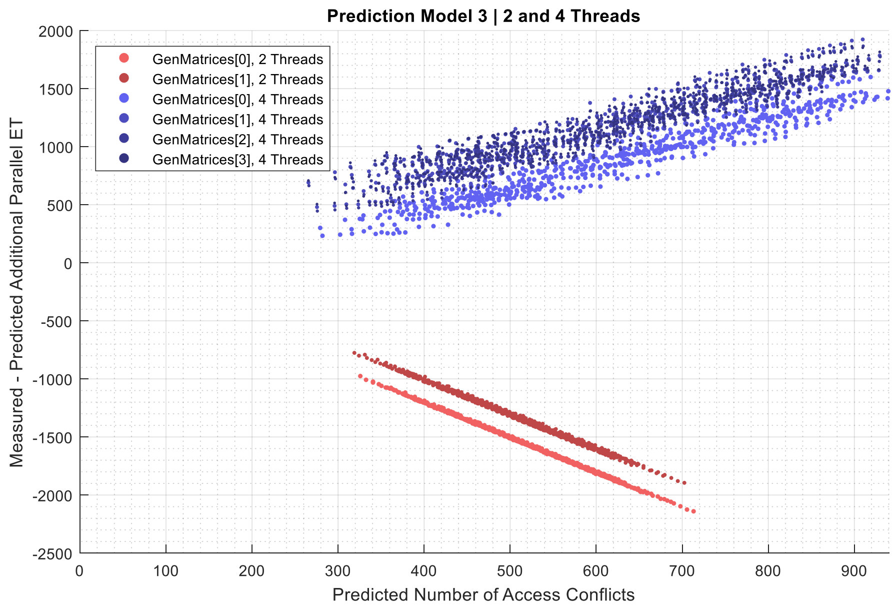

Figure A.11 Model three applied to functions with two and four of them running in parallel

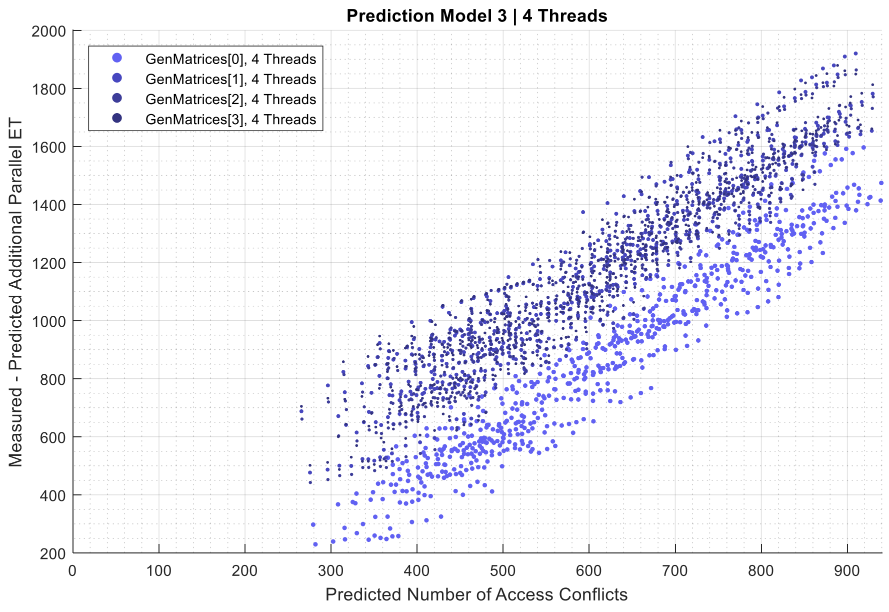

Figure A.12 Model three applied to functions with four of them running in parallel

Bibliography

[1] LLVM Project, «The LLVM Compiler Infrastructure,» 2017. [Online]. Available: http://llvm.org/. [Zugriff am 20 November 2017].

[2] C. Lattner, «LLVM,» in *The Architecture of Open Source Applications*, lulu.com, 2011, pp. 155-170.

[3] C. Lattner, «LLVM: An Infrastructure for Multi-Stage Optimization,» Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, 2002.

[4] K. Cooper und L. Torczon, Engineering a Compiler, Elsevier, 2011.

[5] R. I. Davis, A. Burns und D. Griffin, «On the Meaning of pWCET Distributions and their use in Schedulability Analysis,» in *Real-Time Scheduling Open Problems Seminar*, 2017.

[6] LLVM Project, "LLVM's Analysis and Transform Passes," 3 January 2018. [Online]. Available: https://llvm.org/docs/Passes.html. [Accessed 5 January 2018].

[7] S. M. a. B. Natarajan, «Compiler synthesized dynamic branch prediction,» in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, 1996, pp. 153-164.

[8] A. (Oak), «Adding Metadata to Instructions in LLVM IR | Stack Overflow,» 24 December 2013. [Online]. Available: https://stackoverflow.com/questions/13425794/adding-metadata-to-instructions-in-llvm-ir. [Zugriff am 10 January 2018].

[9] LLVM Project, «LLVM Language Reference Manual | LLVM 7 Documentation,» 09 January 2018. [Online]. Available: https://llvm.org/docs/LangRef.html. [Zugriff am 10 January 2018].

[10] LLVM Project, «Getting Started with the LLVM System,» 05 09 2017. [Online]. Available: http://releases.llvm.org/5.0.0/docs/GettingStarted.html.

[11] LLVM Project, «Writing an LLVM Pass - LLVM 6 documentation,» 26 10 2017. [Online]. Available: https://llvm.org/docs/WritingAnLLVMPass.html. [Zugriff am 30 10 2017].

[12] A. Sampson, «Adrian Sampson: Run an LLVM Pass Automatically with Clang,» 20 April 2013. [Online]. Available: http://www.cs.cornell.edu/~asampson/blog/clangpass.html. [Zugriff am 30 Oktober 2017].

[13] LLVM Project, «LLVM Programmer's Manual,» 5 12 2017. [Online]. Available: http://llvm.org/docs/ProgrammersManual.html. [Zugriff am 16 12 2017].