University of Passau
Faculty of Computer Science and Mathematics

Master's thesis

Adjustable Family-based Performance Measurement

Christian Kapfhammer

submitted at: 11th August 2017

First examiner:
Prof. Dr.-Ing. Sven Apel

Second examiner:
Prof. Christian Lengauer, Ph.D.

Tutor:
Florian Sattler

Abstract

A Software Product Line enables the creation of configurable systems. The base product can be enhanced by selecting a set of configurable features such that the user is able to construct multiple products with specific properties. Since a feature extends the functionality of the base code, additional code is added to the product and influences the performance of the system. A Software Product Line usually contains more than one configuration option, i.e. multiple options that affect the performance.

Thus, we analyse the influence of each feature on the performance. To do this, we extend the work of Florian Garbe who already modified the tool Hercules by inserting certain functions to measure the execution time of each feature block. Afterwards, the results were used to predict the performance of other possible configurations of the SQLite case study [1].

The results of our measurements are useful because it shows the impact of each feature. Unfortunately, increasing the number of measurements also increases the produced overhead of the measurement functions such that the overhead surpasses the actual execution time of the original code. Hence, we are going to improve the performance measurement by introducing several algorithms that determine which feature blocks should actually be measured. We apply each algorithm to the SQLite case study. At the end, we compare all results with each other and with the results of Florian Garbe.

Acknowledgements

First, I thank my supervisors Prof. Dr.-Ing. Sven Apel and Prof. Dr.-Ing. Norbert Siegmund for giving me the opportunity to work besides them. They recruited me as working student to have a sneak peek into performance measuring.

Furthermore, I also want to thank Florian Sattler for tutoring me and for many fruitful discussions and feedback during my work on this thesis.

Finally, I thank my student colleague and writer of the former master thesis Florian Garbe for helping me understand the framework TypeChef and its extension Hercules.

Contents

Abstract **i**

Acknowledgements **iii**

1 Introduction **1**
1.1 Objectives . . . 2
1.2 Structure . . . 2

2 Background **3**
2.1 Software Product Lines . . . 3
2.2 Feature and Feature Model . . . 4
2.3 Variability with C Preprocessor . . . 6
2.4 TypeChef and Hercules . . . 7
2.5 Statistic methods . . . 9

3 Approach **11**
3.1 Block Coverage . . . 11
3.2 Granularity . . . 13
3.2.1 General approach . . . 13
3.2.2 Special influences on the performance . . . 14
3.2.3 Metrics for granularity . . . 18
3.3 Limitations . . . 21

4 Evaluation **25**
4.1 Test system specifications . . . 25
4.2 SQLite Case Study . . . 25
4.2.1 SQLite TH3 Test Suite Setup . . . 26
4.2.2 Adjustments of the test setup . . . 27
4.2.3 Calculated score distribution and filter properties . . . 27
4.3 Comparison between results . . . 30
4.3.1 Measurements and overhead . . . 31
4.3.2 Prediction results . . . 34

5 Concluding Remarks **39**
5.1 Conclusion 39
5.2 Future work 40
5.3 Related work 41

A Appendix **43**
A.1 Modifiers of each metric 43
A.2 Selected modifiers for case study 45
A.3 Additional prediction results data 46

Bibliography **49**

List of Figures

2.2.1 Feature diagram of the Car SPL 4
2.2.2 Feature model in DIMACS format 5
2.4.1 Process of performance measuring, schema of Florian Garbe [1] . 7
2.4.2 Converting variability from source code into an AST 8
2.4.3 Insertion of performance functions 9
2.5.1 Data sets with lines and pearson coefficient 10

3.1.1 Block coverage example 12
3.2.1 Current transformation of Hercules 13
3.2.2 Influence of conditional statements 15
3.2.3 Influence of loops to blocks 15
3.2.4 Interruptions in loops with breaks 16
3.2.5 Interruptions in loops with continues 17
3.2.6 Example for score calculation 19
3.3.1 Disjunction with specialization 21
3.3.2 Two blocks with the same condition 22

4.2.1 SQLite case study setup, schema of Florian Garbe [1] 26
4.2.2 Score distribution of bin scores 28
4.2.3 Score distribution of weighting statements 29
4.2.4 Performance distribution of blocks 30
4.3.1 Comparison of measurements with bin score 31
4.3.2 Comparison of overhead with bin score 31
4.3.3 Comparison of measurements with weighting statements 32
4.3.4 Comparison of overhead with statement weighting 32
4.3.5 Comparison of measurements with performance filtering 33
4.3.6 Comparison of overhead with performance filtering 33
4.3.7 Comparison of prediction errors 36
4.3.8 Comparison of prediction errors including deviation 37

List of Tables

4.1 Number of blocks in each bin 28

A.1 Median percent errors in prediction results 47
A.2 Median percent errors in prediction results including deviation . 47

Chapter 1

Introduction

Product Lines are well-known approaches in many industries such as in car manifacturing. But the way products were manifactured in the past differs from nowadays approach. Products were only designed individually for each customer. As time went on, society changed. More and more people were able to afford buying products and the era of mass production arose. The car manufactorer company Ford was the first to utilize the concept of production lines. Although this way of production was cheaper than before, the standardized products did not fulfill the needs of the individual customer. Therefore, production lines were combined with mass customization which allowed the production of individualized products [2].

Creating invidual products by reusing components can also be applied to software systems, calling this process Software Product Lines (SPL). SPLs are software systems which are highly configurable and can be tailored to customer needs. The implementation of variability depends on the domain of the SPL. The variability of `C` SPLs is realized by the preprocessor directives which are not oblivious to the underlying programming language. Unfortunately, there is one aspect in which SPLs are hard to design. That is their performance. The reason for this problem is the dependance of the performance on the selected features. The amount of possible product versions of an SPL, called variants, grows exponentially the more options are available to configure the SPL. Thus, applying traditional analysis methods to every possible variant is not practicable [3].

Therefore, new analysis methods, called family-based analysis, are applied to achieve our goal. These methods provide results in a fraction of time compared to sequential analyses [4]. But, before these methods can be used, the `C` code has to be transformed. By applying the principle of configuration lifting [5] we can convert compile-time variability into run-time variablity, i.e. the conditional directives used by the preprocessor are transformed into `if`-statements. The presence conditions of the `if`-statements are constructed by global variables

resembling the configuration options of the preprocessor. Thus, we result in one product containing every variant of the SPL.

As we want to determine the performance of a variant of a SPL, we have to find a way to calculate the performance of each feature. Variability encoding already replaces conditional preprocessor directives with `if`-statements. So, we can enhance its functionality by adding functions to measure the performance of each configuration option. Therefore, every single feature is measured. The results can be used to predict the performance of the remaining configurations [1].

Nevertheless, this raises the question if truly every feature block has to be measured to predict the performance of a product line variant. If a feature is measured and does not influence the performance at all, the overhead of the measurement increases. The idea is to remove as many measuring functions without raising the error percentage of the predictions. To reach this goal, we have to define when exactly a feature block is not measured. The key factors in this task are the feature block's contents and its surroundings. As there are multiple ways to rate the features, we present different ways to estimate the feature block's complexity.

1.1 Objectives

Currently, Hercules adds measurement functions before and after every feature block to collect performance data in the code. However, these added functions also produce a significant overhead that impacts the performance of the program and thereby the acccuracy of the measurements. The objective of this thesis is to improve the measurement by reducing the overall overhead. We achieve this by rating each feature block regarding its influence on performance and only adding measurement functions to relevant blocks. We are going to look at several options how to rate each feature block and determine which one is the most feasible by comparing each of their results on the SQLite case study.

1.2 Structure

The thesis is structured as follows. In Chapter 2, we explain basic terms, definitions, and summarize important facts about background aspects. Chapter 3 introduces improvements to Hercules, calculating the block coverage and multiple agorithms to filter the measurement functions. We also discuss further problems and limitations we encountered while working with the program. Chapter 4 explains the general evaluation process as well as the case study SQLite. In Chapter 5, we conclude our work, give ideas how our work could be further improved, and discuss related topics.

Chapter 2

Background

In this section, we introduce multiple concepts, definitions, and explain the major concept based on the example of a car. We begin by defining Software Product Lines as well as other related concepts. Lastly, we give a brief overview about the functionalities of the frameworks TypeChef and Hercules.

2.1 Software Product Lines

A *Software Product Line* (SPL) is a set of similar software-based systems that are created from a set of common core components and use a shared set of features that satisfy the needs of a certain market segment [6]. In this process, these software components are reused to create multiple versions of a product. Selecting a subset of features determines which software components are used in the product. A single product version is refered to as a *variant*. While the initial process of planning and management is not free, SPLs show multiple advantages in the creation of specific products. Reusing core aspects results in reduced costs of development and maintenance [7]. As the customer has specific needs that need to be fulfilled, SPLs enable the simple customization of products which are tailored to the needs of the customer and the current market.

We use this general definition to specify our preprocessor-based `C` Software Product Lines. The preprocessor-free code resembles our shared set of software components. It is used in every product of this product line. The preprocessor also uses code segments that are depending on optional program features. These program features add functionality to our product (see Section 2.2). The selection of features is described by the arguments of the preprocessor and create a variant of the software product line.

2.2 Feature and Feature Model

There are several ways how the term *feature* is defined as it depends on the context in which the term is used [8]. In our case, features are the core aspects of a product line. They represent the requirements and show similarities as well as differences between product variants. Thus, features are used as specification of product variants. To generate a product variant, the user has to choose a set of desired features. This set is also called a *configuration*.

If a configuration of a program has been chosen, the program variant can be generated. But not every configuration can be used for this purpose. For example, a configuration may contain two features like `WINDOWS` and `UNIX` that exclude each other. Therefore, a *feature model* can be used to check the validity of a configuration. Feature models usually appear in one of the two following types: the first uses multiple propositional formulas where a configuration is valid if all formulas are fulfilled. The second one uses a structure called *feature diagram* to determine valid feature combinations.

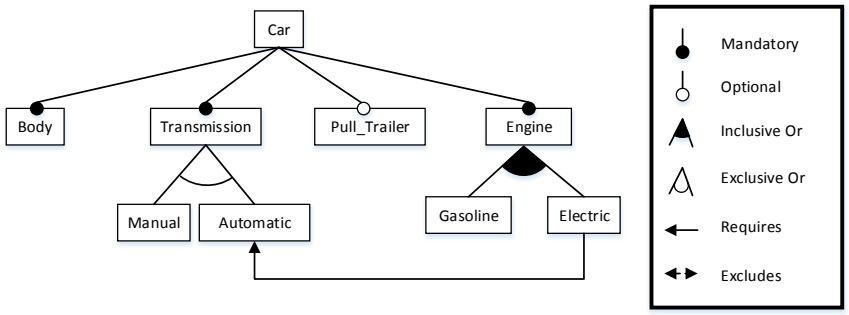


Figure 2.2.1: Feature diagram of the Car SPL

A *feature diagram* is a compact representation of all possible product variants and visualizes which products can be generated [9]. It can be used to determine valid configurations of a product. Figure 2.2.1 shows the feature diagram containing multiple features. In this example, we are going to use a car as an SPL. This car should have a body, an engine and a way to change the gear. As an additional feature our car may be able to pull a trailer. We are going to use this example in the rest of this thesis to explain all issues.
Mandatory features always have to be selected if the corresponding parent is also selected. In this case, `Body`, `Transmission`, and `Engine` are present in all possible configurations. Optional features on the other hand may or may not be selected. For example, we may add the feature `Pull_Trailer` to our `Car` product. Both, mandatory and optional features, can only be selected if the corresponding parent features are selected. In other words, it is not possible to

select `Gasoline` without its parent `Engine`.
These two abstractions are not sufficient to visualize complex product variants. There is also an exclusive OR and an inclusive OR which can be used to enforce at least one or exactly one feature. In this case, either `Manual` or `Automatic` can be selected for the behavior of the gear. In the other case, `Gasoline` or `Electric` can be selected for the engine in the system. However, there is also a possibility to select both of them. There are two arrows to illustrate further dependencies between two features , thus reducing the amount of valid configurations. Here, the feature `Automatic` needs to be selected if the feature `Electric` is going to be used.
If it is needed, there is also the possibility to add custom propositional formulas to the feature model. These constraints also have to be fulfilled to get a valid configuration. Thus, we can set an upper limit to the number of possible configurations for our software product line. In this example, a valid configuration of our feature model could be `(Body, Transmission, Automatic, Engine, Gasoline, Electric)` while an invalid selection of features could be `(Transmission, Manual, Automatic, Engine, Pull_Trailer, Electric)` due to the missing feature `Body` and the invalid selection of `Manual` and `Automatic`.

Of course, this visualized form of a feature model is not optimal for checking the validity of a configuration. Instead, we should use the other mentioned form, a list of propositional formulas. Within this model, each feature represents a literal which equals either `true` or `false` depending on the chosen configuration [9]. A given configuration is valid if and only if all propositional formulas of the feature model are fulfilled. This feature model is useful regarding the automated validity checking of configurations [7]. Furthermore, the propositional formulas can be described in different formats. When operating with TypeChef the user has to give the program a feature model in the DIMACS format [10]. In this format, all features are listed and consecutively numbered in the beginning followed by the clauses and expressions in the Conjunctive Normal Form (CNF) format.

```
1  c 1 Body
2  c 2 Transmission
3  c 3 Manual
4  c 4 Automatic
5  ...
6  c 9 Electric
7  p cnf 9 9
8  3 4
9  -3 -4
10 -3 2
11 -4 2
12 ...
```

Figure 2.2.2: Feature model in DIMACS format

Figure 2.2.2 shows the feature model of Figure 2.2.1 in its DIMACS format. As we can see, our 9 features are labeled at the beginning of the file. Furthermore,

we list all relations between the features in the CNF form. Either `Manual` or `Automatic` may be selected, but not both at the same time. Thus, we add clauses that if the feature `Manual` is deactivated, the feature `Automatic` is selected (see line 8) and vice versa (see line 9).

2.3 Variability with C Preprocessor

There are two types of technologies which integrate variability into a program: language-based variability mechanisms and tool-driven variability mechanisms [7]. On the one hand, language-based variability mechanisms try to implement variability by using available concepts in programming languages like design patterns or frameworks. On the other hand, tool-driven variability mechanisms use external tools to infuse variability into the code. Build systems, version-control systems, but also preprocessors are part of this technique. In this thesis, we have a closer look at the C preprocessor.

The C preprocessor, also called CPP, is a macro preprocessor that extends the capabilities of the standard C programming language. It allows the developer to enrich the source code with commands which instruct the compiler to chose between different code snippets at compile time. This kind of variability is also known as *compile-time variability*. Because the CPP annotations are oblivious to the structure of the programming language, it can be inserted at any level of the source code [11]. The CPP enables the utilization of three specific directives: file inclusion, text substitution and conditional compilation. `#include` allows the inclusion of a file. `#define` defines a macro which can be used in the code. Regarding conditional compilation there are several ways to use this directive. The directive `#ifdef` checks if the specified identifier is defined, while `#if` and `#elif` are used for checking arithmetic expressions. Using the function `defined`, the directive `#ifdef` can be replaced by the `#if` directive. For example, the expression `#ifdef Body` checks the same condition as `#if defined(Body)`. In combination with the directives `#else` and `#endif`, the syntax of the conditional directives resembles the usage of conditional expressions in most of the programming languages.

The conditional directives are our main focus in this thesis. The features of an SPL may be used as presence conditions of the conditional directives. In this thesis, the content between the conditional CPP directives is referenced as a *feature block*. A feature block consists of multiple lines of code that share the same presence condition. The start of the block is represented by an `#if` or `#ifdef`. Afterwards the block can either be ended with an `#endif` or continued with an `#elseif` or `#else` resulting in new feature blocks with other conditions. A feature block is often represented by its condition used in the conditional directives of the CPP. Although, a program may contain more than one block with the same condition. In case of nested blocks, the conditions of the outer blocks still have to be fulfilled in the inner blocks.

2.4 TypeChef and Hercules

TypeChef[1] is a research project that can be used for parsing and type-checking processor-based product lines in C [12]. The tool analyzes variability caused by the CPP's `#ifdef` directive in the source code. During the analysis the source code is transformed in multiple steps, as Figure 2.4.1 shows [1]. During all its steps the results never lose their variability-awareness.

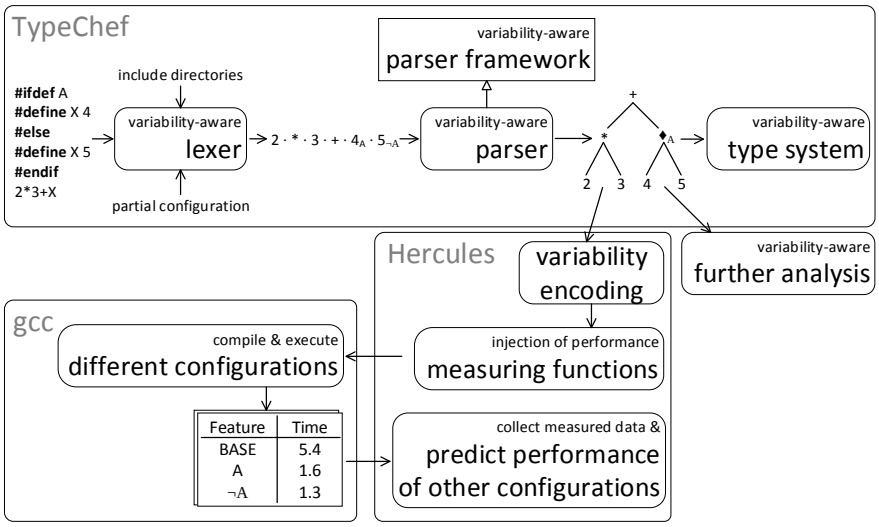


Figure 2.4.1: Process of performance measuring, schema of Florian Garbe [1]

The framework begins its analysis with lexing in which TypeChef partially evaluates preprocessor directives. In this step, included directories are inlined into the code and all macros are being replaced by their corresponding definition. However, conditional directives are left untouched, thus keeping the variability of the code. The lexing process also divides the input code in different tokens and annotates each token with a presence condition.
After obtaining the conditional token stream from the lexer, TypeChef parses the stream to create a syntax tree from this information. During the parsing process the variability-aware parser framework enforces disciplined annotations. Disciplined annotations are declarations, definitions and directives that include a statement inside a function or fields inside a `union` or `struct`. The reason for the enforcement is the need of TypeChef to convert the variability from the token level to the abstract syntax tree (AST), creating a variability aware AST. This data structure contains all information about the preprocessor-variability in the source code, mapping all structures of the source code into AST elements. Figure 2.4.2 shows, the condition of the `if`-statement in (a) depends on the

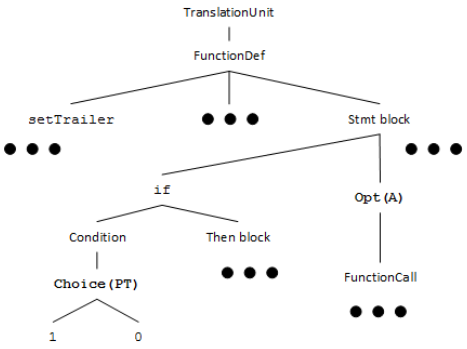
[1] `https://ckaestne.github.io/TypeChef/` (visited: 2017-02-20)

```
char* getTrailer() {
  if (
#ifdef Pull_Trailer
    1
#else
    0
#endif
  ) {
    return "Trailer";
  }

#ifdef Automatic
  displayError("No trailer available.");
#endif

  return null;
}
```

(a) Code variability



(b) (Simplified) AST variability

Figure 2.4.2: Converting variability from source code into an AST

`Choice` node with the condition `Pull_Trailer` and the single statement calling the function *displayError* is listed as `Opt` node with the condition `Automatic` in (b).

All in all, TypeChef is a variability-aware parsing tool to analyze and transform the variability of C code which is created with preprocessor directives. For our cause, there is a further extension to this framework named Hercules[2]. Hercules transforms compile-time variability into run-time variability by using if-statements, renamings and duplications. The code is transformed in such a way that it is possible to choose which code is executed while running the programm. To be able to measure the performance of the features in a program, Florian Garbe added the functionality to insert measurement functions. There are two kinds of measurement functions: one is starting the measurement under a specific context, the other one ends the last called measurement. In this thesis, these functions are denoted as *perf_before* and *perf_after*. These functions are placed at the beginning and at the end of each block. Additionally, there are statements that can leave the measured blocks, for example a `goto`-statement, for which additional ending functions are placed right before this statement. In case of `return`-statements while performance measuring, a *perf_return* is inserted containing the returning value and a call of *perf_after* as parameter. This is necessary because the returning value may have further influences on the performance of the program.
Figure 2.4.3 shows the transformation process of Hercules. (a) displays the initial code and (b) the results of the process. The conditional directive in line 7 is being transformed into an `if`-statement. The condition of the directive is described by global variables representing the used features. At the borders of the block in line 7-11 the measurement functions `perf_before` and `perf_after`

[2] `https://github.com/joliebig/Hercules` (visited: 2017-02-20)

```
1  void driveCar() {
2    driving = true;
3
4    while(driving) {
5      moveCar();
6
7  #if defined(Gasoline) && defined(Electric)
8
9      useFuel(getChosenFuel());
10
11
12 #elif defined(Gasoline)
13
14     useGasoline();
15
16
17 #else
18
19     useElectricity();
20
21 #endif
22
23     if(fuel == 0) {
24       driving = false;
25     }
26   }
27 }
```

(a) Initial code

```
void driveCar() {
  driving = true;

  while(driving) {
    moveCar();

    if (id2i_gasoline && id2i_electritc) {
      perf_before("Gasoline && Electric");
      useFuel(getChosenFuel());
      perf_after();
    }
    if (id2i_gasoline && !id2i_electritc) {
      perf_before("Gasoline && !Electric");
      useGasoline();
      perf_after();
    }
    if (!id2i_gasoline && id2i_electritc) {
      perf_before("!Gasoline && Electric");
      useElectricity();
      perf_after();
    }

    if(fuel == 0) {
      driving = false;
    }
  }
}
```

(b) Transformed code

Figure 2.4.3: Insertion of performance functions

are inserted which handle the measurement process. The block's content in line 9 is placed between the two measurement functions. This process is also repeated for the blocks in the lines 12-16 and 17-21.

2.5 Statistic methods

Later we need to calculate scores for the blocks and measure their performance. As we want to investigate the connection between scores and performances, we take a look at correlation between the calculated value and the measured performance. The *Pearson correlation*[3] calculates a coefficient which describes the linear correlation of two variables.

$$r = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2} \sum_{i=1}^{n} y_i^2}$$

If we visualize this principle in a graph, this means that we have a set of n data points $(x_1, y_1), ..., (x_n, y_n)$ that are spread around the plane. The Pearson correlation tries to draw a line through the data points as good as possible. The calculated coefficient r describes how big the variation of the data points around the line is. The coefficient range is between 1 and -1.

[3] `onlinestatbook.com/2/describing_bivariate_data/bivariate.html` (visited: 2017-06-22)

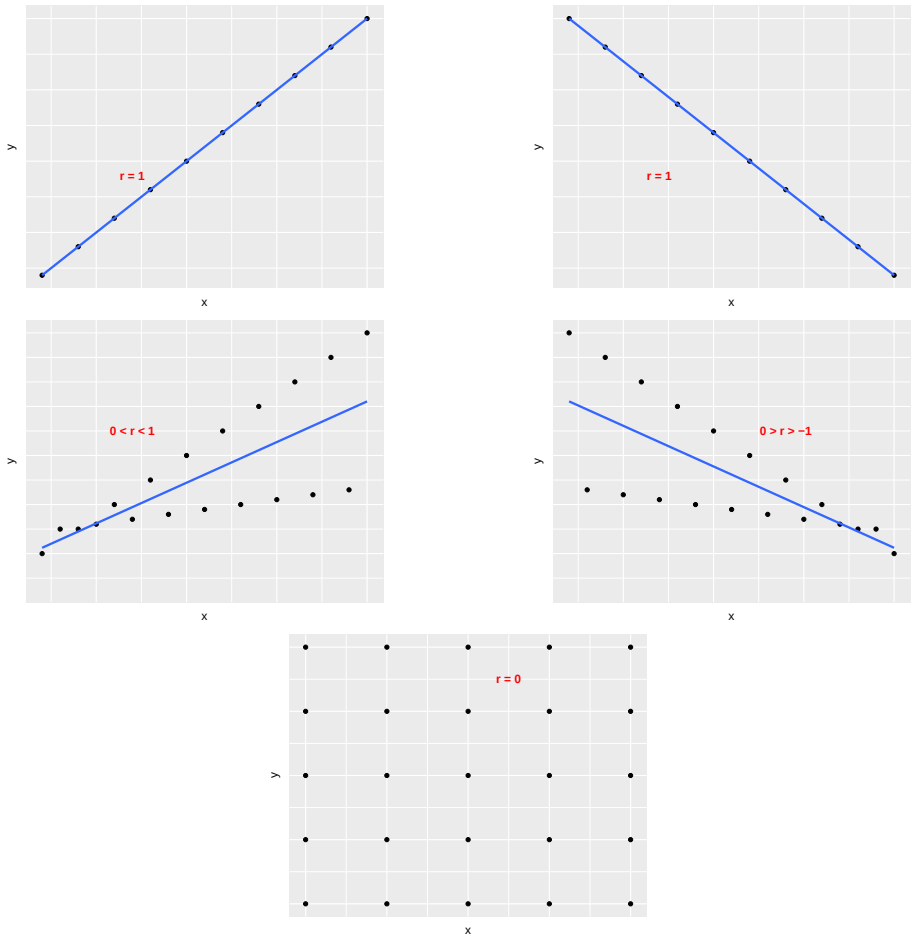


Figure 2.5.1: Data sets with lines and pearson coefficient

Figure 2.5.1 shows different data sets and the presents the correlation coefficient. The signum of the coefficient describes the direction of the line. Positive values resemble a growth of the line while negative coefficients show a decrease of the line.

Chapter 3

Approach

In this section, we discuss multiple concepts to improve the functionality in HERCULES. We introduce the principle of block coverage to calculate configurations and have a closer look at the granularity of our measurement functions. Besides all the improvements of the measurement we also discuss which problems we encountered and how they restrain our work.

3.1 Block Coverage

Configurations are one of the essential aspects of our evaluation. Choosing a random valid configuration might result in a program with low functionality and thus in poor evaluation measurement. Our goal is to calculate configurations that transform the program in a way such that most of the code is executed. With this appraoch we make sure that every block in the code is measured at least once. By using the idea of *block coverage*, we construct configurations that aid finding better configurations suitable for testing.

The main idea of block coverage is that every block is covered by at least one configuration. We are going to extend this idea by covering as many blocks as possible in one single configuration and thus, try to reduce the amount of needed configurations to cover all blocks. If one configuration is not enough, other configurations are created to cover the rest of the blocks.
To achieve this goal there are several steps to take. By using the AST created by TYPECHEF we can easily iterate through all elements of our program. During the iteration we can check if an element of the AST contains a condition which needs to be fulfilled so that the corresponding block can be executed. Due to the structure of the AST we iterate from the top to the bottom of the program. As a result, all calculated configurations of previous blocks are used in the next blocks. In case a condition of an block cannot be merged with any previously calculated configurations, a new configuration is created. Within nested blocks there has to be no extra calculations to guarantee the outer condition because

the conditions of the inner blocks already contain the condition of their outer elements.
After iterating through the AST and calculating all configurations we have to check if the selected configurations are valid. In some cases, it could be possible that some features are selected without their parents. Furthermore, mandatory features also have to be included in those configurations if their requirements are fulfilled. If these features are included into the calculated partial configurations, we end up with a set of configurations that covers every possible block in a program. It is important to mention that these configurations may not be optimal for measuring the performance of the program.
Checking a given configuration follows the same process. We also iterate through the AST from top to bottom and keep track of the conditions of the elements. Instead of creating configurations, we count the number of encountered blocks and the number of fulfilled blocks.

The following example Figure 3.1.1 illustrates the strategy of the algorithm in which the configurations are calculated.

```
void startEngine() {
  playEngineSounds();

#ifdef Manual
  initializeManualGear();
#endif
#ifdef Automatic
  initializeAutomaticGear();
#endif
#ifdef Electric
  playQuietEngineSounds();
#endif

  actuallyStartEngine();
}

void startCar() {
  startEngine();

#ifdef Pull_Trailer
  setTrailer(true);
#endif
}
```

Figure 3.1.1: Block coverage example

In the beginning, we create a default configuration {`(True)`} at which the other features are concatenated. We reach our first `#ifdef` and concatenate the feature `Manual` to our only configuration resulting in the feature set {`(Manual)`}. Regarding the next block we cannot combine the feature `Automatic` with the feature `Manual`. Thus, we create a new configuration resulting in the configuration set {`(Manual)`, `(Automatic)`}. The block with the condition `Electric` has no conflicts with the previously calculated configurations, so we concatenate this feature to each configuration. After doing the same with the block `Pull_Trailer` in the next function definition the resulting configuration set is {`(Manual,`

`Electric, Pull_Trailer)`, {`(Automatic, Eletric, Pull_Trailer)`}. These configurations are not valid yet because they do not contain all necessary parent features (see Figure 2.2.1), meaning we have to add further features to the configurations to make them valid. After adding the missing mandatory features we end up with the final configuration set {`(Manual, Electric, Pull_Trailer, Body, Transmission, Engine)`, `(Automatic, Electric, Pull_Trailer, Body, Transmission, Engine)`}.

3.2 Granularity

When a valid configuration is chosen, it is used to determine which feature blocks are executed and measured. The measurement of each feature is done by inserting measurement functions whenever a block starts or ends. Simply put, blocks that have up to no influence on the performance of the program are also measured. As a consequence, this creates unnecessary overhead which increases the execution time of our measurement up to the point that the overhead time even surpasses the actual execution time.

```
double getCurrentGasoline() {
#ifdef Gasoline

  return amountGasoline;


#else

  return 0.0;
#endif
}
```

(a) Initial code

```
double getCurrentGasoline() {
  if (id2i_gasoline) {
    perf_before("Gasoline");
    perf_return(
        amountGasoline, perf_after());
  }
  if ((! id2i_gasoline)) {
    perf_before("!Gasoline");
    perf_return(0.0, perf_after());
  }
}
```

(b) Transformed code

Figure 3.2.1: Current transformation of Hercules

In Figure 3.2.1 (a) and (b), we simply return the amount of gasoline if the corresponding feature `Gasoline` is defined. Returning a single value does not affect the performance of the program very much. If we measure such blocks of code, we add unnecessary overhead to the program. Thus, there has to be a way to differ between complex blocks and non-complex blocks, so we only add measurement functions where they are needed. The idea of granularity is to measure only the feature blocks which influence the performance of the program in at least a specific rate. The general process how we achieve this goal is described in the next section.

3.2.1 General approach

The goal is to measure the code of feature blocks which have a certain influence on the performance. This means we have to check if measurement functions have to be added to a block while transforming the code. To achieve this goal

we have to carry out multiple steps. Our initial data is the AST of the `C` file and a threshold to filter the ignored statements.

1. Determine blocks in AST.
2. Assign statements to blocks.
3. Calculate the score of each block.
4. Filter blocks by using threshold.

In the beginning, we have to determine the used blocks of the program and label them with an alias. This way, we can easily refer to a block by using its created alias even if the condition of two blocks are identical. While labeling the blocks, the statements are assigned to its block in which they are used. A statement is used in one block only, i.e. if a statement is within a constellation of nested blocks, the statement is assigned to the most inner block. The third step contains the calculation of scores of each block. The values of the scores depend on the used metric (see Section 3.2.3). The calculated score is going to be adjusted by the specified modifiers (see Appendix A.1). In the last step, we can determine which blocks should be filtered and should not get any measurement functions by specifying a threshold. The statements of the filtered blocks are given to HERCULES. While transforming the code, HERCULES checks if the block gets any measurement functions by checking the list of filtered statements.

3.2.2 Special influences on the performance

Although this seems like a simple task, there are multiple special cases that need to be addressed. Even if a block has a low score on its own, it does not mean that this block needs to be ignored immediately. We are going to discuss which problems may occur while iterating through the AST and how these can be solved.

Conditionals

`if`-statements are used in nearly every programming language and help to perform different computations depending on the specified condition. If a condition is never fulfilled, the corresponding branch is never executed. This raises the question in which way this influences the measurement of our blocks. If a block highly impacts the performance and is rarely executed, unnecessary overhead is added.
In Figure 3.2.2 (a) and (b), starting an engine is complex. But it is only executed if a key is inserted into the car. This has an influence on the measured block `Engine` because this decreases the chance that the code impacting the performance of the block is executed. Increasing the number of branches decreases the chance even further that the branch with the complex code is executed. The same problem occurs with `switch`-statements. If a `switch`-statement lists a large number of `case`-statements and only one of them contains code that

```
void startCar() {
#ifdef Engine

    if (keyInserted) {
        startEngine();
    }

#endif
}
```

(a) Initial code

```
void startCar() {
    if (id2i_engine) {
        perf_before("Engine");
        if (keyInserted) {
            startEngine();
        }
        perf_after();
    }
}
```

(b) Transformed code

Figure 3.2.2: Influence of conditional statements

increases the execution time of the program, we can not assure that the block is reached. Thus, the number of branches/`case`-statements is used to modify the content's score such statements.

Loops

A block containing a single line of code does not inflict any influences to the performance of a program. But before we ignore this block, we have to analyze its surroundings. Executing the code once might not have a big impact on the performance, but if it is repeated a large number of times, the influence of this single block may rise.

```
void decreaseFuel() {
    while(driving) {
#ifdef Gasoline

        fuelGasoline--;

#endif
    }
}
```

(a) Initial code

```
void decreaseFuel() {
    while(driving) {
        if(id2i_gasoline) {
            perf_before("Gasoline");
            fuelGasoline--;
            perf_after();
        }
    }
}
```

(b) Transformed code

Figure 3.2.3: Influence of loops to blocks

Even if the code is not very complex and does not contain any special constructs, the block still might influence the program's performance as we see in Figure 3.2.3 (a) and (b). Executing the decrementation once does not show any affection on the performance. But in combination with the `while`-loop surrounding the block `Gasoline` the execution time of the block increases. This problem can also be transferred to the `for` and `do-while`-loops analogously. As we do not know the exact number of iterations of the loop most of the time, we have to correct the score of each statement inside a loop in some way (see Appendix A.1).

Control flow irregulations

Now, we could generally assume that every measuring function in a loop can stay in the code. But there are multiple inappropriate ways to get out of a loop. We cannot guarantee that loops are exited the same way every time. If only one iteration is interrupted, the complete loop may be interrupted or we jump to a different part of the program. This can be problematic in many ways. The following examples show how `break`, `continue` and `goto` influence the measuring of the feature performances.
The keyword `break` is used to jump out of a loop when it is executed. If there are nested loops, only the most inner loop is interrupted in which the keyword is executed. Using the example of Figure 3.2.3, we can construct a show case in which this functionality influences the performance measurement.

```
void decreaseFuel() {
    while(driving) {
#ifdef Gasoline

        fuelGasoline--;

#endif
        break;
    }
}
```

(a) Initial code

```
void decreaseFuel() {
    while(driving) {
        if(id2i_gasoline) {
            perf_before("Gasoline");
            fuelGasoline--;
            perf_after();
        }
        break;
    }
}
```

(b) Transformed code

Figure 3.2.4: Interruptions in loops with breaks

As Figure 3.2.4 (a) and (b) show, inserting the keyword `break` in the `while`-loop changes the context of the block completely. We might think that the block `Gasoline` costs a lot of time if it is contained in a loop. In reality, the block is executed only once, the `break`-statement exits the `while`-loop and the program continues with the code after the loop. In that way, the usage of `break` negates the influences of the loop on the block such that only a single line of code is measured.
We have a similar case with `continue`. Although we do not interrupt the complete loop, we are still able to end some iterations of it. Thus, there is a similar example like in Figure 3.2.4.
We see in Figure 3.2.5 (a) and (b) that the feature measurement takes place once. If the car is currently not moving, we enter the `if`-block that contains a `continue`-statement. In this way, we only measure the block `Gasoline` if the car is moving. This results in getting about the same effects as with `break`.
The last keyword that needs to be discussed is `goto`. It can be used to jump to labels which can be inserted at any part of the scope. Thus, `goto` shows the same effects we analyzed with `break` and `continue`.

```
void decreaseFuel() {
    while(driving) {
        if (notMoving) {
            continue;
        }
#ifdef Gasoline

        fuelGasoline--;

#endif
    }
}
```

(a) Initial code

```
void decreaseFuel() {
    while(driving) {
        if (notMoving) {
            continue;
        }
        if(id2i_gasoline) {
            perf_before("Gasoline");
            fuelGasoline--;
            perf_after();
        }
    }
}
```

(b) Transformed code

Figure 3.2.5: Interruptions in loops with continues

Using one of these keywords results in ignoring the rest of the code afterwards. Therefore, we have to act accordingly. But if we encounter one of these keywords, we don't stop the calculations at this point. We rather adjust the score of the structure containing the keywords. The structure is either a loop or a function because `break` and `continue`-statements are only contained in loops and `goto`-statements may occur anywhere. The adjustement of the score depends on the type of keyword. `break` and `goto` decrease the score of the code more than `continue` because they exit the loop completely or jump to another part of the structure. `continue` only stops one iteration, so there is still a chance to execute the block's code in the loop. As there is no indication to which part of the code the keyword `goto` leads, there will be no any special calculations for this.

Function calls

Even if a block contains only one line of code, its content still plays an important role. The execution of this code may lead to other parts of the program that are unknown at the point of the granularity calculation. Using functions is one of these cases. If a function is called in a block, it may lead to other complex calculations. Thus, it needs to be measured. This depends on the called function. The function itself can be defined before or after calling it. This means that we also have to know the general influence of a function as well. Therefore, we are also going to calculate the score of each function defined in our program.
Calling a function may lead to further function calls resulting in a chain reaction of accumulated functions. We may be able to extend this idea by returning to the source and creating a recursion. Unfortunately, there is no way to determine how many times the recursion functions are called.

3.2.3 Metrics for granularity

There are multiple ways to rate the blocks in a program. Choosing two different metrics may result in different score values. We are going to regard possible metrics which can be used to calculate the score. The general idea about score is that it should show the influence of the block on the program's performance: the higher the score, the bigger the impact of a feature on the performance of the program.

Bin scores

There are already multiple algorithms that rate a specific object in a certain topic. For example, the Building Energy Asset Score[1] assesses the physical and structural energy efficiency of buildings. This score is a value between 1 and 10 and depends on various factors like the location and the type of the building.
We can adapt this idea and also assess our blocks by their contents. The idea is to pick certain categories in which each block is rated. We call this type of score *bin score*. Each category is also rated by a value between 1 and 10. There are certin structures that may impact the block's performance. Thus, we choose the following categories:

1. `if`-statements
2. `switch`-statements
3. Loops
4. Function calls
5. Control flow irregulations

The first category takes a look at the `if`-statements inside the blocks. Here, the important factors are the number of `if`-statements and how many branches each statement has. An `if`-statement can be used to execute only specific code pieces, i.e. if there is a huge number of `if`-statements in the block, there is a high chance that this code is not executed. A general `if`-statement has two branches, one in which the conditon is fulfilled and the other one in which the condition is not fulfilled. The number of branches increases if there are `else-if`-statements. Additional branches decrease the probability that the statements of a branch are executed. Thus, this category has to be weighted negatively.
The same reasoning can be applied to `switch`-statements with one exception. `case`-statements without a `break`-statement continue with the next `case`-statement. Apart from that, this category is also weighted negatively.
As mentioned in Section 3.2.2, loops and function calls may increase the performance of the program. Unfortunately, there is no indication how often loops and, if existent, recursions are executed. Nevertheless, a high number of loops

[1] `https://energy.gov/eere/buildings/building-energy-asset-score` (visited: 2017-06-27)

and function calls should still be weighted positively. Regarding function calls, we are calculating bin scores for them as well to estimate their general impact on the performance. These scores are used when evaluating the function calls within the blocks.
The last category rates the block regarding control flow irregulatons. As they interrupt the normal flow of programm, a huge number of such statements should logically be weighted negatively.
All in all, each block is rated according to these categories. In the end, all ratings have to be combined and result in a score ranging from 1 to 10. The influence of each category can be adjusted by the specified modifiers (see Appendix A.1).

```
1  void updateGear() {
2    if (currentSpeed == 0) {
3      currentGear = 0;
4    } else if (currentSpeed < 25) {
5      currentGear = 1;
6    } else if (currentSpeed < 75) {
7      currentGear = 2;
8    } else {
9      currentGear = 3;
10   }
11 }
12
13 void brake() {
14   slowlyDecreaseSpeed();
15
16 #ifdef Automatic
17   while(carIsBraking) {
18     currentDisplayText = "Now braking automatically";
19
20     updateGear();
21   }
22
23   currentDisplayText = "Stopped braking";
24 #endif
25 }
```

Figure 3.2.6: Example for score calculation

We present an example in Figure 3.2.6 that shows the score calculation in detail. In this example, we use the same modifiers as in our later case study (see Appendix A.2) and we particularly have a look at the function *updateGear* and the block `Automatic`. First, we rate the function *updateGear*. This function only contains one `if`-statement with 4 branches, which is rated negatively since a lot of branches implicate that code which affects the performance may not be executed. Thus, we rate the `if`-bin of this function with the value 9. As for the other bins, there are corresponding statements that are used for these bins. Therefore, the score of the `switch`-bin and the bin for control flow irregulations is 10. The bins for function calls and loops both get a score 1. As we set our modifiers to specific values, the bin score of the function equals a bin score of 3. Next, we calcuate the score of the block `Automatic`. Since there are no `if`-statements, `switch`-statements and control flow irregulations, the corresponding bins are rated with the value 10. The block `Automatic` contains exactly one `while`-loop. But since it is just one loop, the score of this bin stays at 1.

Nevertheless, we still have a function call inside the block. Since we have a score for this function, we consider that score in our calculation and get a score of 6. In the end, the block `Automatic` has a score of 5.

Weighting statements

If we have a closer look into the contents of the blocks, we may calculate more accurate scores for the blocks. Therefore, we weight the occuring statements in each block. This is done by simply iterating through the AST and increasing the score of the current blocks whenever we encounter a statement. Initially, we start with a weight of 1 for each statement. But the weight is being adjusted whenever we encounter a special structure (see Appendix A.1). If we encounter nested blocks, the statements of the inner blocks also increase the score of the outer blocks and the function containing the blocks. Nevertheless, there are some statements which do not increase the score, namely empty statements and control flow irregulations, i.e. `break`, `continue` and `goto`. For function calls, we save the location in which the call occurs and the weight that modifies the function call's weight. We calculate a temporary score for each block by adding the scores of nested blocks to the outer blocks. These scores are used to estimate a score for the functions. The last step handles the function calls in the program. If a function call occurs in a block, we get the score of this function and iterate through further function calls. Each returned value is multiplied by the function call's saved weight. In case of recursions, we calculate the set of functions that form the recursion, add the scores of each function in this set together and handle all calls out of the recursions appropriately. The final scores contain their own statements, the scores of the nested blocks and their function calls.
To illustrate the process of this metric, we use the example in Figure 3.2.6. We set the general weight for loops to 2 while every other modifier will be not specialized further. In this example, we want to calculate the score of the block `Automatic`. As this block is not in any loop, there are no special weight modifiers. After iterating through the AST once we have the following scores. Since line 17 starts a loop and counts as statement as well, it increases the score of the block. We chose a general loop weight of 2. Thus, the score of the block `Automatic` is increased by this value. The statement in line 18 is an assignment with an original weight of 1. As this statement is inside a loop, we multiply it by the loop's modifier and increase the block's score by 2 once again. The function call in line 20 does not affect the score yet. Lastly, The statement in line 23 is outside the loop and increases the score of the block `Automatic` by 1. This leaves us with a temporary score of 5. While iterating through the AST, we calculated a score for the function `updateGear` as well. This score consists of the statements in line 2 to 9. These statements form an `if`-structure. This reduces the weight of each statement within as well as the conditions as well. In total, there are 4 branches in this `if`-structure and the weight of all statements from line 2 to 9 is set to 0.25. Therefore, the score of the function `updateGear` is 2. In the last step, we resolve all function calls in blocks. In this example, we

have only one function call in the block `Automatic`. Since the function call is within a loop, we multiply the score of the function `updateGear` by 2 and add it to the current score of the block. Finally, the algorithm ends and the block `Automatic` has a score of 9.

Performance filtering

As we want to measure only the blocks which majorly influence the performance, a logical option is filtering the blocks by their performance. Of course, this metric does not work until we actually measure each block at least once. If the needed data is available, the process of this metric is simple. Each block gets a score of either 0 if the block should be filtered or 1 if the block should be measured. Rather than setting a threshold for the score we use this threshold as minimal execution time a block should have. In this way, we only have to check if the block's execution time is greater than the specified threshold.

3.3 Limitations

During the development of the improvements we encountered several special cases which limit the rating of the blocks. In this section, we list all code limitations with a corresponding explanation and example.
In order to determine the blocks of the code, we have to look at the conditions of statements in the AST. This is the only way to determine the blocks of the given code. A block begins if the condition of the current statement is not a subset of the previous statement's condition. There is one case in which the distinction of the block limits is impossible.

```
void printFuelAmount() {
#if defined(Gasoline)||defined(Electric)
  print("Current fuel state:");

#if defined(Gasoline)
  print("Gasoline: " + getFuelGasoline());
#endif
#if defined(Electric)
  print("Electric: " + getFuelElectric());
#endif

  print("Fuel printed");
#endif
}
```

(a)

```
void printFuelAmount() {
#if defined(Gasoline)||defined(Electric)
  print("Current fuel state:");
#endif
#if defined(Gasoline)
  print("Gasoline: " + getFuelGasoline());
#endif
#if defined(Electric)
  print("Electric: " + getFuelElectric());
#endif
#if defined(Gasoline)||defined(Electric)
  print("Fuel printed");
#endif
}
```

(b)

Figure 3.3.1: Disjunction with specialization

Both code segments (a) and (b) in Figure 3.3.1 have the same AST in which we use a disjunction and specialization combined. But we clearly see that they do not have the same code structure. In (a), the disjunction block contains the

specialization, while in (b) all blocks are on the same level. TypeChef simplifies the condition of the inner statements of (a) and transforms the condition $(Gasoline \vee Electric) \wedge Gasoline$ into the simplified form $Gasoline$, resulting in an identical structure of the ASTs. Logically, this makes sense. But regarding the calculation of the block scores, these are two different cases because the statements of the specialization in (a) influence the score of the disjunction block. To avoid this case, we have to use the code style in (b).

In C code, a block has two specific points that define the start and the end. The CPP directives are not available when using only the AST. So we have to determine the block boundaries by examining the condition change of the statements. Whenever the current statement has a different condition regarding the previous statement's condition, we know that the current block changed. If the condition c_1 of the current statement is no subset of the previous statement's condition c_2, the block with the previous condition c_2 ended and a new one began with the condition c_1. This raises the question what happens when a block is followed by a block with the same condition.

```
void changeGearAutomatic() {
#ifdef Automatic
  updateGear(getCurrentVelocity());
#endif

#ifdef Automatic
  updateGearView(getCurrentGear());
#endif
}
```

(a)

```
void changeGearAutomatic() {
#ifdef Automatic
  updateGear(getCurrentVelocity());



  updateGearView(getCurrentGear());
#endif
}
```

(b)

Figure 3.3.2: Two blocks with the same condition

In Figure 3.3.2, code segment (a) divides both updates when changing the gear while in (b) both updates are in the same block. Even if the functionality is divided into two blocks with the same condition, we merge both blocks together which results in less needed measurement functions. The inclusion of measurement functions can be enforced if an empty statement is placed between the two blocks.

Lastly, there are multiple factors that can affect the performance of the program and are not directly visible in the code itself. We already mentioned loops and recursions. In the metrics of this thesis, these two structures are considered in the scores and are estimated by the modifiers. But we do not know exactly how long they last. If the user has to give input in this case, we cannot determine the iteration length of the structure. Regarding function calls and recursions, the usage of different parameters in calls may also affect the performance metrics. A potential recursion can be ended immediately if certain parameters are used. As these are special cases that refer to the logic of the code, we do not consider these cases in the metrics. Futhermore, function pointers can be used at any

place of the code. If the referenced function is executed, we cannot determine which function exactly is executed. In this case, we are using a default value for function calls which are unknown or not defined. Nevertheless, this is also an estimation as there is no information apart of the initialization which function is used. The worst case is that either an empty function or a recursion are referenced by the pointer. These cases are a fraction of what we cannot detect.

Chapter 4

Evaluation

Now that we are able to give every block a score, we can also determine the execution time of every block and further combinations. With this information we can determine which blocks claim the most execution time. In the following section, we take a look at the resulting data we obtain while calculating our scores and measuring our blocks. Furthermore, we are going to compare the previous results of Florian Garbe with the filtered results.

4.1 Test system specifications

As we want to compare our results with the ones of Florian Garbe, we should use the same test system. The experiments were executed on a cluster system consisting of 17 nodes with an Intel Xeon E-5 2690v2 @ 3.00 GHz with 10 cores each. The system was equipped with 64 GB RAM per CPU and running Ubuntu 16.04 as the operationg system. Each test scenario was executed on its own node such that there were no disturbances that might influence the measurements.

4.2 SQLite Case Study

SQLITE[1] is a database product line with 93 configuration options. It is the most widely deployed database in the world and can be tested using the TH3 test suite[2]. Using these two concepts combined, we are able to use SQLite on Hercules with some limitations.
As we are going to compare our results with the ones of Florian Garbe, we use the same setup for our measurements. Therefore, we are going to use the SQLite case study as well.

[1] `https://www.sqlite.org/` (visited: 2017-05-13)
[2] `https://www.sqlite.org/th3/` (visited: 2017-05-13)

4.2.1 SQLite TH3 Test Suite Setup

In this section, we briefly explain our setup of the SQLite TH3 test suite. For the end results we measured the execution times on our cluster system three times for each metric and calculated the average performance times to increase the accuracy of our measurements.

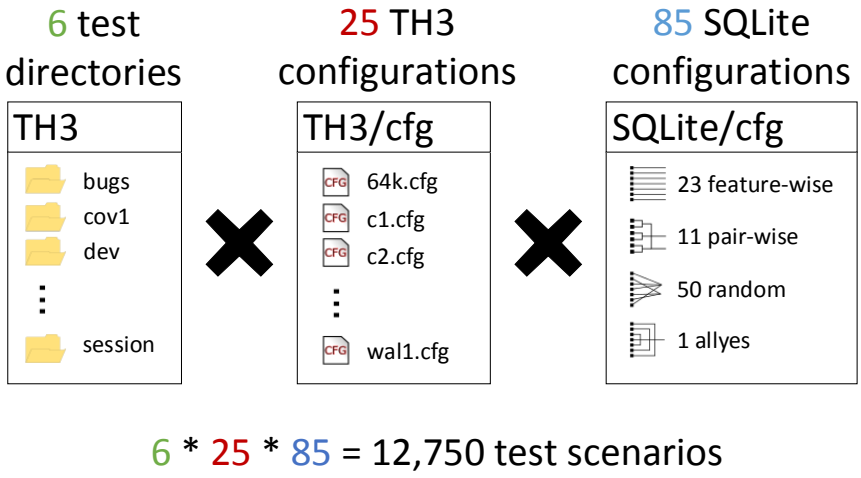


Figure 4.2.1: SQLite case study setup, schema of Florian Garbe [1]

The test setup consists of 3 initial parts. First, we have the TH3 directories we are going to test. Each directory contains multiple `.test` files which are converted into `C` files. Linked with SQLite, they execute all given tests. After certain modifications and restrictions to the test setup (see Section 4.2.2), we are left with 6 directories containing 1 to 355 `.test` files.
Next, the TH3 test suite itself has 25 different configurations that transform the code of the directories in different ways. These configurations should not be mixed up with the SQLite configurations. Combining each directory with each TH3 configuration results in 300 different transformations.
Lastly, there are the configurations for SQLite. Unfortunately, it is difficult to measure all possible configurations and there is no thorough feature model for SQLite. Thus, we are focusing on a subset of 23 specific features. These features are used to create configurations that are used on the 300 transformations. The configurations are divided into four different groups:

- **Feature-wise**: We generate 23 different feature-wise configurations consisting of one enabled feature and as little as possible other features to create a valid configuration.

- **Pair-wise**: These configurations were generated with the SPLCA[3] tool. For this type of configuration, we generate all possible pairs of the 23 features. The generated configurations cover multiple pairs at the same time, resulting in 11 pair-wise configurations.
- **Random**: Features are randomly selected to create 50 different configurations. As there are 23 features, we generate a random binary number between 0 and 2^{23} and map it to a configuration. Duplicate or illegal configurations due to the feature model are discarded beforehand.
- **Allyes**: All 23 features are selected resulting in a single configuration.

4.2.2 Adjustments of the test setup

After we talked about the general test setup, we take a look at further limitations before evaluating our results. Florian Garbe already modified the test setup in multiple ways [1]. Originally, there are 26 TH3 configurations. But one of them never executed any test cases as it is for testing a DOS related filename scheme and is not compatible with the UNIX setup.
Furthermore, the original TH3 test suite contains 10 directories. The *stress* directory was excluded as it contains tests with a very high run time. Excluding this directory resulted in several SQLite out of memory errors in two directories. To remove these errors and also reduce the amount of `.test` files per directory, one of the directories was partitioned once and the other one had to be partitioned twice. Florian Garbe also excluded 3 further directories since their execution time is less than 2 ms. Thus, they are not useful for the measurements.
Lastly, we focus on the code itself. We already mentioned in Section 3.3 that we are not able to determine every block if a specific code style is used. We encountered such cases in code of the test suite. As we saw in Figure 3.3.1, a disjunction with an inner specialization is hard to recognize and even hinders the filtering of blocks. In the `C` code of SQLite, we encountered two cases where this code style was used. Thus, we refactored the cases such that we can work with the code correctly without generating any errors while the functionality of the code remains the same. Furthermore, there was also one case in which the filtering did not work due to a global variable. The variable changed its value according to the selected features and thus, formed its own block. This resulted in a filtered measurement function which should not be removed. Therefore, we had to add an additional statement such that the functionality of the code is not affected and the measurement function stays.

4.2.3 Calculated score distribution and filter properties

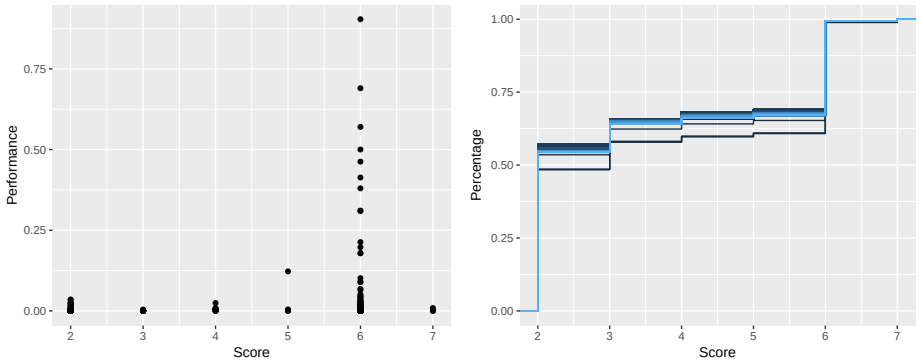
As mentioned in Section 3.2.3, we developed three different possibilities to calculate the score of each block. This results in multiple choices what threshold

[3] `http://martinfjohansen.com/splcatool/` (visited: 2017-05-19)

we may choose. If the value of the threshold is guessed without any previous measurements, the results might turn out as we want. Therefore, we executed all algorithms but did not filter any blocks. By giving each block a unique name we measured the performance of each block. In this way, we encountered over 1700 different blocks. Unfortunately, not all blocks could be measured with the given test setup of SQLite. Nevertheless, we are going to present the calculated results in this section as well as choose which threshold we are going to use for each algorithm. The displayed performance of the blocks is their mean performance value as they were measured multiple times. We removed 5 blocks from the graphics because their mean performance values are much higher than the other values. Regarding their scores, the blocks are not filtered when using the specified thresholds. The used values for the modifiers of each metric can be found in Appendix A.2.

Bin scores

In Section 3.2.3, we described the process how we divide the blocks into different bins. Now, we sort each block into its corresponding bin and discuss the calculated results.



(a) Score distribution (b) EDF

Figure 4.2.2: Score distribution of bin scores

Bin	2	3	4	5	6	7
#Block	295	52	11	5	171	2

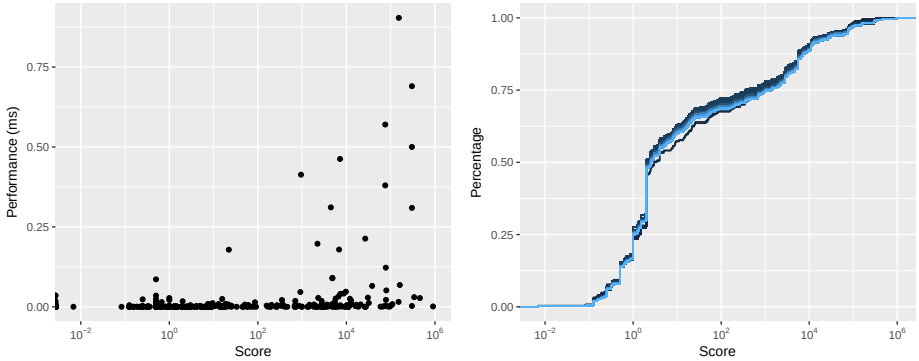
Table 4.1: Number of blocks in each bin

Figure 4.2.2 (a) shows each bin with its blocks and their mean performance values. Table 4.1 shows the number of blocks contained in each bin. Most of the blocks are contained in bin 2 while blocks with high performance are sorted into bin 6. But there are also blocks with low performance in this bin. We noticed that one block with a high execution time was rated with a low score.

This is the case because this block calls a function *fsync*. This function has not many statements and any special structures to increase the scores of any bins. The high execution time is occurs because the function writes data into a file. Thus, we adjusted the score of this block such that it is not filtered.
Futhermore, we had a closer look at the causes of each block's score. Almost all blocks have the highest score in control flow irregulations as most of them do not contain any of these kind of statements. Rating the `switch`-statements give similar results. There are ~ 10 blocks containing `switch`-statements with a large number of `case`-statements. Furthermore, there are not many blocks that contain loops. Most of the time blocks are placed inside loops, but in this metric this case is ignored. The main cause of the high scores are the function calls and recursions due to their high amount and the size of the recursions. Calculating the correlation coefficient for this score distribution, we get the value 0.24.
In Figure 4.2.2 (b), we see the empirical distribution function of every transformation when using bin scores. It shows the percentage of all previous block's performances in relation to the overall execution time. The percentage increases in bin 2 as it contains most of the blocks of all bins. As for the filtering, we have to choose at least the bin 2 to filter any blocks. We see that the performance percentage highly increases when we reach the score 3. Thus, we are going to ignore all blocks that have a lower bin score than 3. In total, we are filtering 295 blocks this way.

Weighting statements

Compared to the bin scores, weighting each statement of each block results in a variety of score values. As the bin score metric is a more general approach than this metric, it is expected that the scores are more distributed than in the previous metric.



(a) Score distribution (b) EDF

Figure 4.2.3: Score distribution of weighting statements

In Figure 4.2.3 (a), the scores are spread across the whole score-axis. Blocks with a high average performance get a high score as the performance rises. But

blocks with low performance values may also get a high score due to our limited possibilites to analyse the code's logic. Comparing the Pearson correlation coefficient of this score distribution to the one with bin scores, the coefficient is higher with a value of 0.32.
As in the bin metric, Figure 4.2.3 (b) shows the empirical distribution function of every transformation when weighting statements. It shows the same as in Figure 4.2.2 (b), whereas the scores are more distributed in this case. We see an increase in percentage at the value 2. This point has the biggest influence on the program's performance. This is why we want to measure it. Therefore, the score 2 is used as threshold such that the algorithm filters 223 blocks.

Performance filtering

Unlike in the other two metrics, we do not calculate an actual score for each block because we filter the blocks by their performance. Therefore, no modifiers are used in this metric and we take a look at the performance distribution.

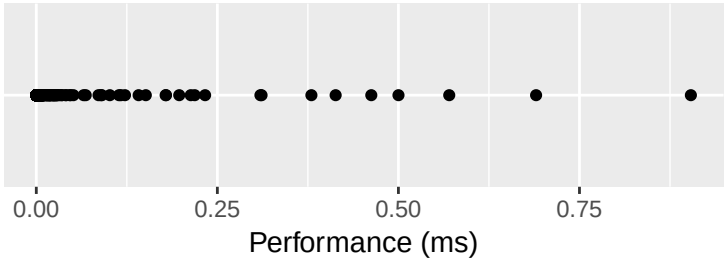


Figure 4.2.4: Performance distribution of blocks

The average performance value for each single block varies as Figure 4.2.4 shows. Most of the performance values are very small. This data was obtained by labeling each measurement function with the corresponding block alias that is the same in each transformation and measuring the performance by using the test setup. As we are interested in the blocks with high performance, we have to pick a threshold with a low value. In this case, the value of the threshold, we are going to use, is 0.000125 ms and 261 blocks are going to be ignored.

4.3 Comparison between results

Applying each metric with its corresponding filter threshold transforms the original source code in different ways and may ignore other blocks, whereas other metrics would accept them. Of course, this brings different results in measurements and predictions during our 12,750 test scenarios. This section presents the differences between these results and compares each with the original data presented by Florian Garbe.

4.3.1 Measurements and overhead

The introduced metrics filter blocks that should not be measured. Thus, the number of measurements and the amount of overhead should both decrease if we use any metric. These values depend on the type of used metric. We are going to look at the data we obtained while measuring the case study.

Bin score

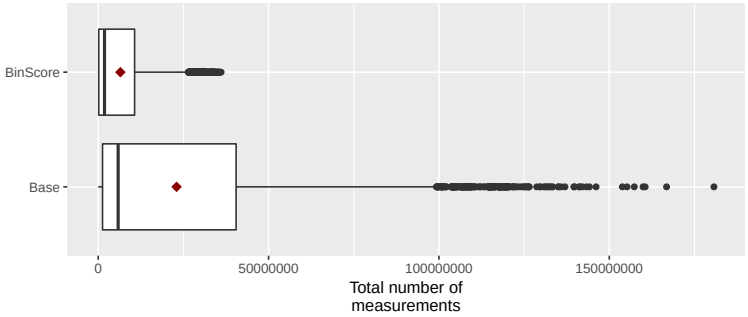


Figure 4.3.1: Comparison of measurements with bin score

Calling the measurement function `perf_before` increases the number of measurements during the runtime. Figure 4.3.1 shows a comparison of the measurements between the non-filtered and filtered version of the code. Originally, the value varied between 180 and 180,712,831. By filtering 295 blocks we reduced the number of measurements severely to a range between 51 and 36,061,770. The mean is depicted by the squared point and decreased from 23,005,422 to 6,511,300.

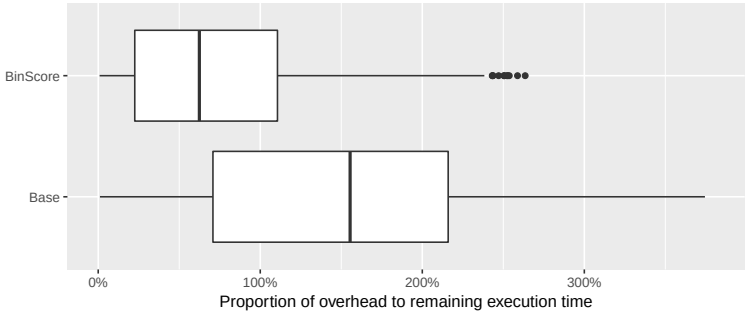


Figure 4.3.2: Comparison of overhead with bin score

Next, we have a look at the overhead produced by the measurement functions. As we reduced the overall number of measurements, the logical conclusion is that

the overhead decreases as well. In Figure 4.3.2, we compare the proportion of the measured overhead to the remaining execution time. The maximum percentage drops from 374% to 263%. The execution time for the measurements exceeds the execution time of the TH3 test code when no filtering is enabled. But Figure 4.3.2 shows that the median declined from 155% to 68%.

Weighting statements

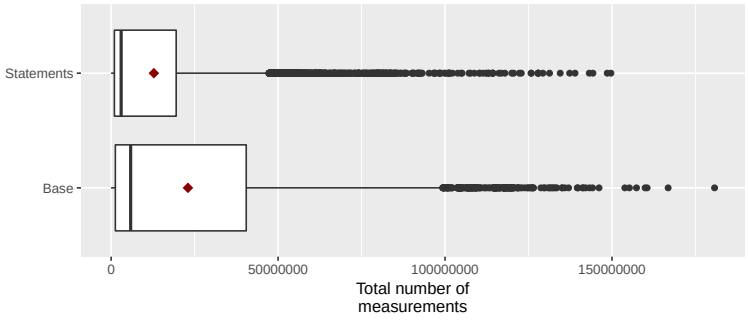


Figure 4.3.3: Comparison of measurements with weighting statements

Applying another metric may provide other results. Figure 4.3.3 presents the total number of measurements when using the statement weighting as filtering argument. Since this metric rates the blocks by its contents and its surroundings, the filtered blocks differ from the ones when using bin scores. All in all, the number of measurements still decreases and ranges from 153 to 148,569,335. As for the mean, Figure 4.3.3 shows that the red dot dropped from 23,005,422 to 12,804,930.

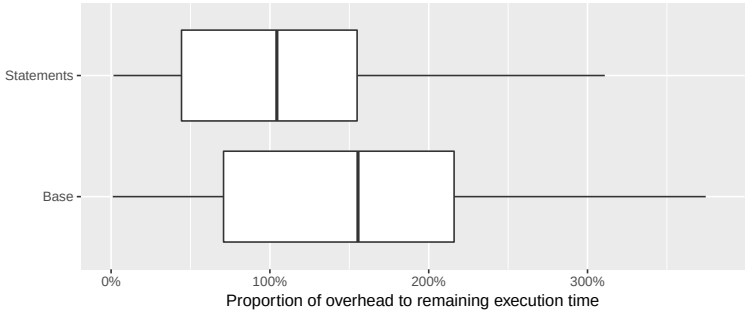


Figure 4.3.4: Comparison of overhead with statement weighting

Even if the number of measurements did not decrease as much as with bin scores, the overhead is still adjusted accordingly. Thus, we reduced the median

of the overhead proportion from 155% to 103% , as shown in Figure 4.3.4. The values of the percentages range from 1% to 311%.

Performance filtering

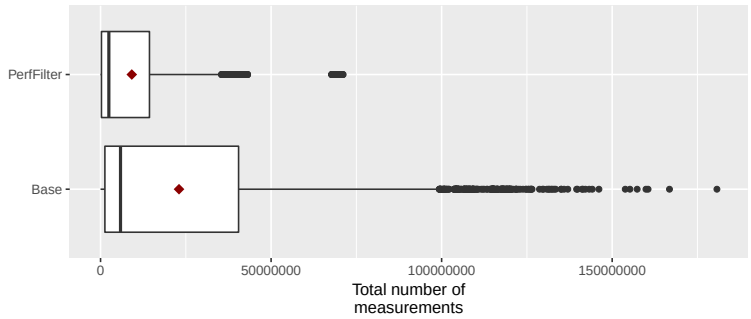


Figure 4.3.5: Comparison of measurements with performance filtering

At last, we take a look at the results when filtering by performance in Figure 4.3.5. In this case, we filtered 261 blocks and the range of the measurements varies between 79 and 71,127,686. We notice that there are some cases in which the number of measurements is over 60,000,000. But, they are below 50,000,000 most of the time. The mean also moved from 23,005,422 to 9,156,436.

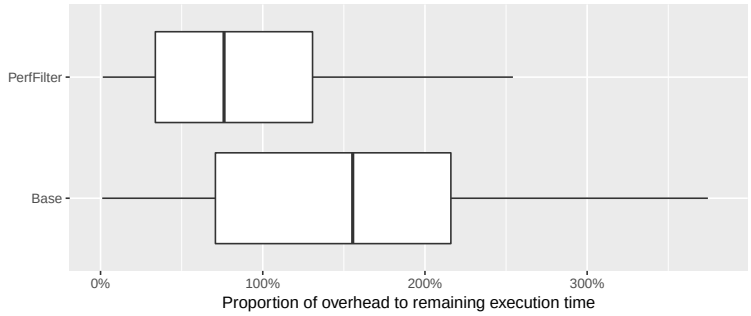


Figure 4.3.6: Comparison of overhead with performance filtering

Regarding the overhead, lesser measurement functions result in lower overhead percentages in the proportion comparison. Figure 4.3.6 shows the comparison of the overhead percentages between the base implementation and the filtered version. We see that the median dropped from 155% to 76%. In comparison to the base implementation, the filtering causes the maximum overhead percentage to decrease from 374% to 254%.

4.3.2 Prediction results

Removing measurement functions from the code may reduce the total number of measurements and the amount of overhead produced by such functions. The measurement functions are used in such a way that if an inner measurement is removed, the corresponding time is added to the outer measurement. The total measurement might turn out more inaccurate than before. In this manner, we are going to compare the prediction results.
We already have detailed information about our 85 SQLite configurations. Florian Garbe made cross predictions for the different configuration modes, i.e. the data of a configuration group is used to predict the performance of other groups [1]. The time of the prediction for a configuration is compared to the time the performance simulator takes when executed with the corresponding configuration. Therefore, we are going to use the same formula:

$$percent\ error = \frac{|predicted\ value - expected\ value|}{expected\ value}$$

The results of his research are that the *allyes* configuration is the worst configuration for predictions while the *random* configurations yield the best prediction data. Although, this was only possible by including the variance to the percent error. We have to keep that in mind while we compare his results with ours in Figure 4.3.7. This figure compares the percent error of the predictions between the measurement without filtering to every introduced metric. We compute the percent error by comparing the average prediction time to the actual time the performance measuring of that configuration needs. If there are multiple configurations for a prediction, the result is the average of all individual predictions of this group. A perfect prediction is reached when the percent error equals 0.

Using the *allyes* configuration as source of our predictions provides the worst predictions. Florian Garbe discovered that a possible reason is related to the feature `SQLITE_NO_SYNC`. Deactivating this feature drastically increases the execution time of the program. Since the *allyes* configuration activates this feature and has no further information about `SQLITE_NO_SYNC`, the predictions are accordingly bad. This is still the case when applying metrics to the case study. In Section 4.2.3, we briefly mentioned certain blocks with high execution times which are not filtered for this reason. One of these blocks is executed when deactivating the feature `SQLITE_NO_SYNC` and causes this behavior. Furthermore, we see that the median in percent error rises in every case. Bin scores and filtering by performance provide larger maximum percent errors than the base implementation or the filtering by statements. However, we proclaimed that this kind of prediction is not practical. Thus, we analyze the other prediction modes.

In general, the predictions are more accurate than before even though some outliers have a higher error value than in the base implementation. In most of the cases, the upper quartile is decreased. We are able to see this phenomenon when predicting pairwise with featurewise and vice versa. Nevertheless, there are also cases which have a worse predction than in the unfiltered version of the case study as we can see in *pairwise predicts allyes*.

The percent errors of the metric measurements vary. The results depend on the examined prediction mode. The reason for this behavior is the different block filtering of each metric. For example, while performance filtering removes measurement functions such that the mode *pairwise predicts featurewise* provides more accurate predictions than others, weighting statements give lower percent errors in the mode *featurewise predicts allyes*. Bin scores usually lower the upper quartile but also have a higher median than the other metrics.

During our measurements we also measured the standard deviation of each configuration. Figure 4.3.8 displays the same as the previous figure with the additional influence of the variance and we are able to spot the same changes. While predicting the performance of the *allyes* configuration, we noticed that the number of outliers increases when applying the metrics on the case study. We still have to be careful regarding the *random* predictions since the variance is very high in this configuration group. High variance values reduce our percent errors but they also increase the inaccuracy of our predictions [1]. Even if the results of the *pairwise* predictions are improved when including the variance, they still have high variance values as high as the *random* predictions.

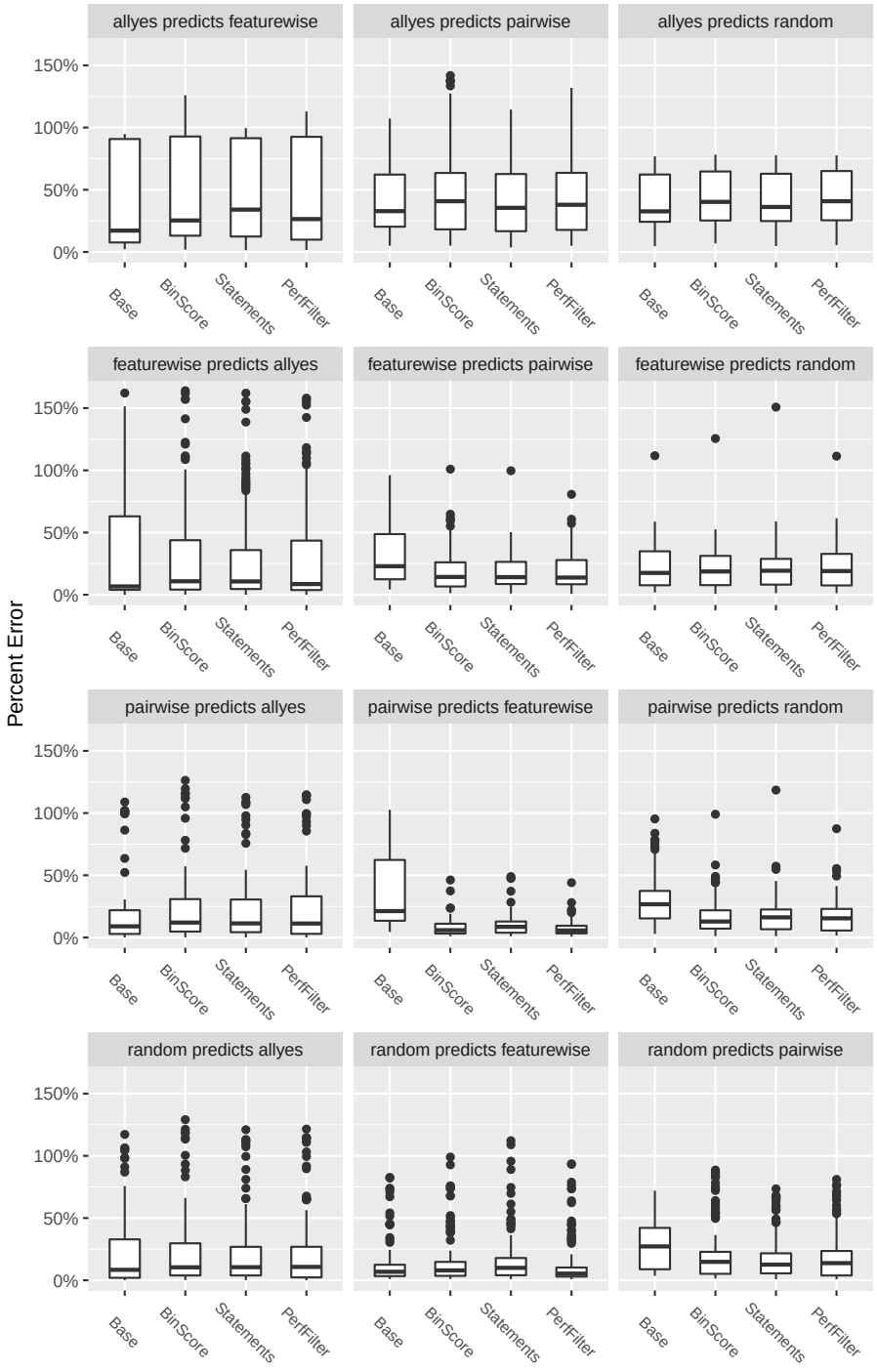


Figure 4.3.7: Comparison of prediction errors

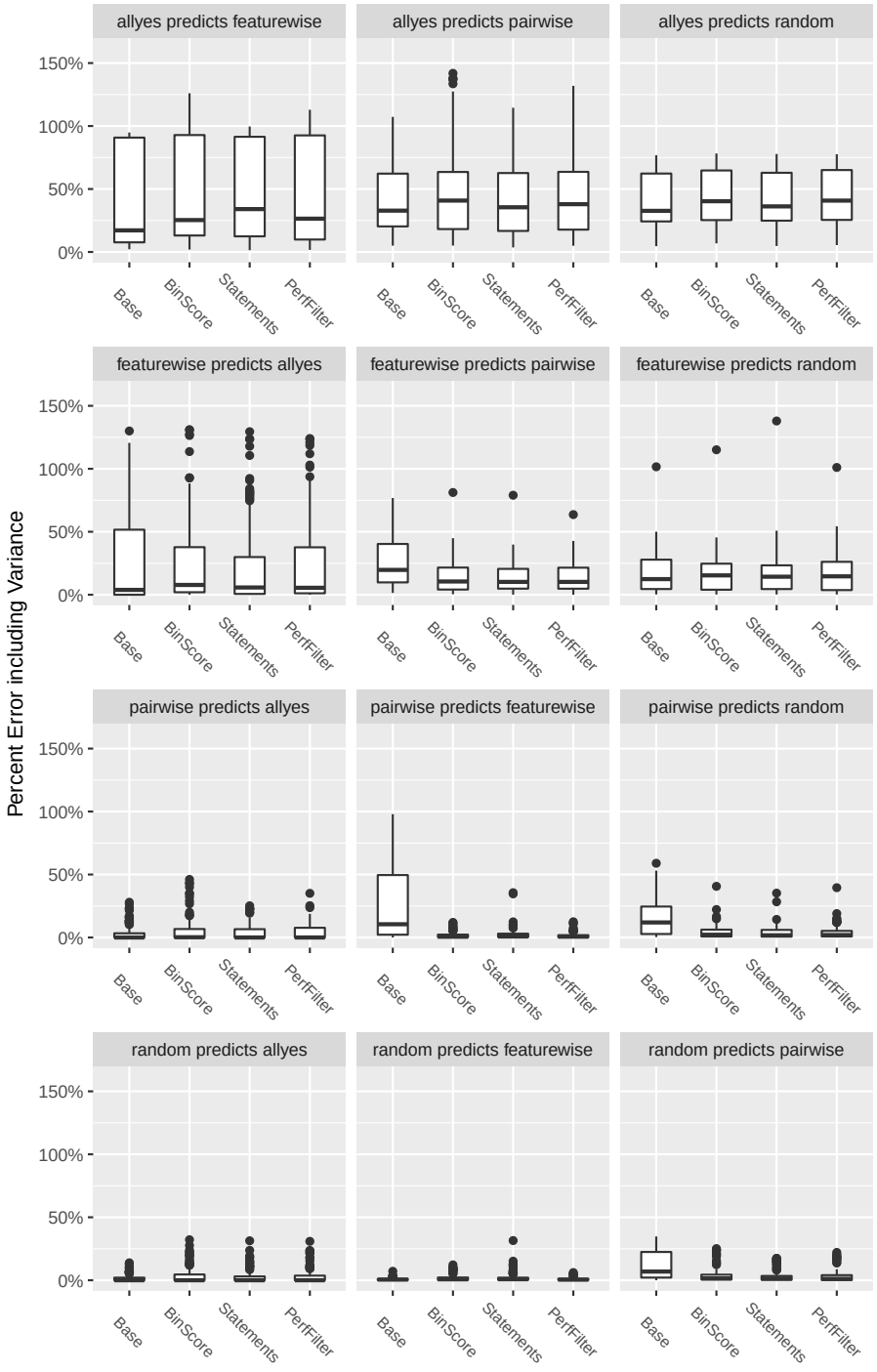


Figure 4.3.8: Comparison of prediction errors including deviation

Chapter 5

Concluding Remarks

This section contains the final conclusion of this thesis. We summarize our results and discuss them as well as the challenges which occured during this thesis. Furthermore, we give an outlook on topics for future work in this research direction. Finally, a list of related work is mentioned.

5.1 Conclusion

Hercules transforms the `C` preprocessor directives in the code in a way such that the variants of a SPL can be chosen during run-time. The execution time of each block can be measured by measurement functions that are inserted whenever the block is entered or exited. A huge number of such functions creates a lot of overhead, which exceeds the actual execution time of the code. In this thesis, we present multiple metrics to filter statements that do not have a huge impact on the performance of the program.

Regarding the calculated scores, the results vary. On the one hand the bin scores rate the blocks and put them into 10 bins, while on the other hand the statement weighting provides no restrictions for the maximum score value. Nevertheless, there are multiple factors that affect the performance of the block we have no information about. We provide modifiers to estimate the impact of their influences. Unfortunately, this is not enough as most of the code depends on the current state of the program.

The prediction results change in different ways as we work with the case study SQLite. Using no filtering of blocks, the *allyes* configuration is the worst configuration for performance predictions of other configurations. This fact holds true for any introduced metric. Comparing the results to the data without any filtering, the percent error even increases in every metric when predicting with the *allyes* configuration. However, we reduced the percent error of most of the predictions by applying our filtering algorithms. In consideration of the devia-

tion, the *random* configuration predictions still provide the best prediction data.

All in all, we reached our goal to reduce the amount of overhead caused by the measurement functions without drastically decreasing the accuracy of our predictions. However, an open issue is that we cannot determine the block's influence on the program's performance by analyzing the code alone. Additional information about each block might aid us in this task.

5.2 Future work

As we started working on methods how to filter blocks in the code, we determined several topics for further future work. First, there are multiple possibilities how to filter blocks that were not explored in our research. Due to the similar processes of each metric it is simple to enhance the current framework of this work by *more metrics*. A possible metric is filtering by feature interaction degree. The degree declares the amount of different features in an interaction. In this case, the specified threshold for the algorithm represents the maximum feature interactions degree. Another metric could specify a list of features. Therefore, all blocks containing these features should be measured. As we see, there are still several open filtering possibilities.

Next, the third introduced metric filters the blocks of the code by their performance. This data was obtained by temporarily labeling the blocks by an alias and then use this alias in the measurement functions. In this case study, we simply created a name by adding an incrementing number to the end of the block's condition. Thus, we could link each similar block in every transformation of the source code. Unfortunately, this process does not work on every possible transformation in other case studies because the number of blocks with the same condition could differ in two different transformation of another case study. This might result in an alias that refers to two different blocks. Thus, a *general process of labeling blocks* should be used instead. But finding such a process proves to be difficult as we only work with the AST. The AST does not have any information about the source code and how the blocks are structured originally.

Furthermore, we introduced two metrics in this thesis which only use the AST of the original code. In an additional metric, we also use previously measured performance information to filter blocks that do not need to be measured. We can use this process in the other two metrics and, thus, *use additional information* about the code to improve the current metrics. For example, when weighting statements we already consider `if`-statements and `switch`-statements by counting the branches they create and adjusting the weight of all further statements. But there is no information in the code if a branch is ever executed or if it is always executed. Thus, we adjust the weight of the containing statements equally. Providing execution probabilities for each branch increases the accuracy of the

scores.

In this thesis, we had a look at the relation between the calculated score and the performance values. Now, the question arises if there are other factors that might have a relation with the performance of a SPL. The selection status of a feature might not only affect the performance of a variant but also its *energy consumption*. Observing the energy management when selecting a feature shows the influence on the energy consumption of the overall SPL.

Lastly, the measurement approach can be evaluated by applying them on *further case studies*. In our current case study SQLite, there is more code inside the preprocessor directives than in the code base. To be exact, it exceeds 50% of the code base. But regarding the research of Florian Garbe, it is a suboptimal target for the performance measuring and prediction approach. Unfortunately, finding viable case studies is proving to be difficult as specific requirements must be met. TypeChef needs to compile the code and has to be able to check the type correctness successfully. A well-thought-out and detailed feature model already helps solving several issues, but it also needs to be in the right form. Additionally, Hercules has to transform the `C` code such that all measurement functions are inserted at the right place. If even one of these functions is misplaced, the measurement is not going to work.

5.3 Related work

The original motivation for this topic arose from the prior work from Post et. al about configuration lifting [5] and the paper from Siegmund et. al about family-based performance measurement [13]. The idea of configuration lifting enabled the variability encoding as we explained above. The motivation of measuring the performance of `C` SPL was developed when working with Siegmund et. al. In their paper, they were working with JAVA SPL and a tool chain based on FEATUREHOUSE[1]. The difference between their approach and ours is that the variability is only contained in functions while the variability of the CPP may occur in any level of the code. The results of their evaluation on 5 different test systems show that the average prediction accuracy is placed at 98%. In contrast to a brute-force approach, only a fraction of the effort is needed. Additionally, they also verified that an increasing number of features in an SPL also increases the saved time by the measurements while also decreasing the amount of configurations needed for a 100% code coverage when family-based performance measurements are used.

Calculating bin scores was inspired by the U.S. Department of Energy's Building Energy Asset Score[2]. The Asset Scoring tool is a web-based evaluation tool

[1] `http://www.infosun.fim.uni-passau.de/spl/apel/fh/` (visited: 2017-07-18)

[2] `https://energy.gov/eere/buildings/building-energy-asset-score` (visited: 2017-06-27)

that assesses the physical and structural energy efficiency of commercial and multifamily residential buildings [14]. To rate a building, the user has to insert specific input about the building such as the roof type and its location. In this way, buildings can be compared with each other and helps to determine possibilities where to invest in energy efficiency upgrades.

Finally, there are other tools that estimate the performance of a SPL. Kwon et. al developed a framework called Mantis that predicts the performance of Android applications [15]. The framework consists of multiple steps in which a given program, program schemes and input by the user are used to generate a function to approximate the program's execution time that only uses a specific subset of features. After some adjustments the result is a predictor that calculates the performance of smartphone applications. They verify in their evaluation that Mantis achieved an accuracy with a prediction error of 5%. Furthermore, Mantis was applied on three hardware platforms and generated predictors that provided accurate estimations.

Appendix A

Appendix

A.1 Modifiers of each metric

As we mentioned in Section 3.2.1 and Section 3.2.3, there are mutliple modifiers which modify the score calculation each metric. Each modifier emphasizes which structure of the code is more or less important. In this section we are going to list all of them as well as explain the purpose of each metric:

Bin scores

- **if_weight**: Weight for the category regarding `if`-statements.
- **switch_weight**: Weight for the category regarding `switch`-statements.
- **loop_weight**: Weight for the category regarding loops.
- **function_call_weight**: Weight for the category regarding function calls.
- **control_flow_weight**: Weight for the category regarding constrol flow irregulations

All weights are used to weight each category bin. It does not matter if the sum of all values does not equal 1. The result is adjusted accordingly.

Weighting statements

- **loop_weight**: General weight for loops. Multiplies the score within loops by this factor if no specific value is specified.
- **for_weight**: Weight for `for`-loops. Multiplies the score within `for`-loops by this factor.
- **while_weight**: Weight for `while`-loops. Multiplies the score within `while`-loops by this factor.

- **do_weight**: Weight for `do-while`-loops. Multiplies the score within `do-while`-loops by this factor.
- **control_flow_weight**: General for control flow irregulations. If a `break`-statement, `continue`-statement or `goto`-statement occurs in the code, the score is multiplied by this factor if no specific value is specified.
- **break_weight**: Weight for `break`s. If a `break` occurs within a loop, the score of the loop is multiplied by this factor.
- **continue_weight**: Weight for `continue`s. If a `continue` occurs within a loop, the score of the loop is multiplied by this factor.
- **goto_weight**: Weight for `goto`s. If a `goto` occurs within a loop, the score of the loop is multiplied by this factor. `goto`s outside of loops are modifying the score of the functions. The factor is adjusted nevertheless.
- **recursive_weight**: Weight for recursions. Multiplies the score of a recursion by this value if a recursive function is called in the code.
- **function_call_weight**: Weight for function calls. Multiplies the score of a function call by this value.
- **default_function_weight**: Default weight for functions. If there is no score for a function, this weight is used.

Furthermore, we introduce two more options to modify the score of function calls when weighting the statements. First, we may predefine the score of functions if needed. The specified score is used if the function is called and does not trigger any calculations regarding further function calls or recursions. By using function offsets, the specified functions can be prioritized by adding the specified value to the function's score. Thus, we are able to choose which functions are capable of having a high execution time. The reason for those two modifications is that we are not able to determine the complexity of every function. For example, the process of saving data is simple and does not require a lot of statements. But depending on the size of the data, the execution time might be high or low. In this case, we increase the offset of the called function such that it gets a higher score at the end.

A.2 Selected modifiers for case study

In this section, we list the modifier values we applied on our metrics while working on the case study and provide explanations why these values were chosen.

Bin scores

- **if_weight** = 0.1
- **switch_weight** = 0.1
- **loop_weight** = 0.35
- **function_call_weight** = 0.4
- **control_flow_weight** = 0.05

We apply these modifiers in dependence on their impact on the performance. Since a huge usage of `if`-statements, `switch`-statements and control flow irregulations decrease the score of the corresponding bin, the bins get the highst score if there is none of such statements in the blocks. This does not necessarily mean that the block has a huge impact on the performance of the program. In this case, loops and function calls are rated higher as they impact the performance of the block the most.

Weighting statements

- **loop_weight** = 2.0
- **break_weight** = 0.75
- **continue_weight** = 0.8
- **goto_weight** = 0.5
- **recursive_weight** = 50.0
- **function_call_weight** = 1.0
- **default_function_weight** = 5.0
- **function offsets:**
  - sqlite3MemGetMemsys3: 200.0
  - sqlite3MemGetMemsys5: 200.0
  - sqlite3ExplainSelect: 10000.0
  - sqlite3CodeSubselect: 100.0
  - sqlite3ExprCodeIN: 1000.0
  - sqlite3Fts3Init: 2000.0

- sqlite3CodeSubselect: 20.0
- sqlite3FkCheck: 500.0
- sqlite3FkActions: 50.0
- sqlite3ExplainExpr: 10000.0
- fsync: 10000.0

In the SQLite case study, we noticed that the impact of the loops is not very high. But they still have an impact on the score, especially when working with control flow irregulations. Furthermore, we detected that most of the blocks containing a recursion have a high average execution time resulting in a high modifier. As pointed out in Section 3.3, we cannot the determine which functions are called when using function pointers. In rare cases, we encountered some function pointers in the code. Nevertheless, the blocks containing these pointers did not affect the performance that much and thus, we estimated the weight for such function calls with the value 5.
Regarding the function offsets, we used these values to adjust the scores of recursions since the corresponding functions forming the recursions are small and were rated lower than expected. The function *fsync* gets a higher score as it saves data. This function is used in the code once within a block with the condition `!SQLITE_NO_SYNC`. This block has the highest execution time of all blocks because it saves a huge amount of data at some time in the case study.

A.3 Additional prediction results data

In this section, we list further data we obtained during our performance measurements in Section 4.3.2. Table A.1 and Table A.2 displays the median percent errors of Figure 4.3.7 and Figure 4.3.8 in each prediction mode.

Prediction Mode	Base	BinScore	Statements	PerfFilter
allyes predicts featurewise	17.1652%	25.3429%	34.0535%	26.4442%
allyes predicts pairwise	32.8288%	40.8156%	35.4929%	37.9424%
allyes predicts random	32.6553%	40.2931%	36.1894%	40.7844%
featurewise predicts allyes	6.6792%	10.8297%	10.6933%	8.5836%
featurewise predicts pairwise	22.9337%	14.2921%	14.1508%	13.8089%
featurewise predicts random	17.5408%	18.7019%	19.2858%	19.0143%
pairwise predicts allyes	8.9712%	11.9944%	11.3233%	11.2236%
pairwise predicts featurewise	21.2108%	5.8749%	8.5811%	5.6302%
pairwise predicts random	26.7113%	12.8566%	16.1893%	15.4229%
random predicts allyes	8.3794%	10.3854%	10.4863%	10.7338%
random predicts featurewise	6.8197%	7.9143%	9.9717%	8.3315%
random predicts pairwise	27.1846%	14.7906%	12.569%	13.791%

Table A.1: Median percent errors in prediction results

Prediction Mode	Base	BinScore	Statements	PerfFilter
allyes predicts featurewise	17.1652%	25.3429%	34.0535%	26.4442%
allyes predicts pairwise	32.8288%	40.8156%	35.4929%	37.9424%
allyes predicts random	32.6553%	40.2931%	36.1894%	40.7844%
featurewise predicts allyes	3.8557%	7.771%	5.7601%	5.5327%
featurewise predicts pairwise	19.7201%	10.5259%	10.1783%	10.1848%
featurewise predicts random	12.3986%	15.3717%	14.3138%	14.6099%
pairwise predicts allyes	0%	0.2026%	0%	0%
pairwise predicts featurewise	10.5188%	0.6315%	0.828 %	0.6772%
pairwise predicts random	11.9474%	2.1916%	1.7537%	1.9615%
random predicts allyes	0%	0%	0%	0%
random predicts featurewise	0.5256%	0.7435%	0.8293%	0.4675%
random predicts pairwise	6.8947%	1.4933%	1.3415%	1.1658%

Table A.2: Median percent errors in prediction results including deviation

Bibliography

[1] Florian Garbe. Performance measurement of c software product lines. Master's thesis, University of Passau, Germany, Bavaria, Passau, 2017.

[2] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[3] Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo G. Giarrusso, Sven Apel, and Sergiy S. Kolesnikov. Scalable prediction of non-functional properties in software product lines. In *Proceedings of the 15th International Software Product Line Conference (SPLC)*, pages 160–169, Los Alamitos, CA, 8 2011. IEEE Computer Society. **Best Paper Award**.

[4] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1):6:1–6:45, June 2014.

[5] Hendrik Post and Carsten Sinz. Configuration lifting: Verification meets software configuration. In *ASE*, pages 347–350. IEEE Computer Society, 2008.

[6] Paul C. Clements and Linda Northrop. *Software Product Lines: Practices and Patterns.* SEI Series in Software Engineering. Addison-Wesley, August 2001.

[7] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation.* Springer, 2013.

[8] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. *What's in a Feature: A Requirements Engineering Perspective*, pages 16–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[9] Don Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Conference on Software Product Lines*, SPLC'05, pages 7–20, Berlin, Heidelberg, 2005. Springer-Verlag.

[10] DIMACS challenge. Satisfiability. Suggested format. ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/, 1993.

[11] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 191–202, New York, NY, USA, 2011. ACM.

[12] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. Typechef: Toward type checking #ifdef variability in c. In *Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development*, FOSD '10, pages 25–32, New York, NY, USA, 2010. ACM.

[13] Norbert Siegmund, Alexander von Rhein, and Sven Apel. Family-based performance measurement. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, pages 95–104, New York, NY, USA, 2013. ACM.

[14] Na Wang, Supriya Goel, Atefe Makhmalbaf, and Nicholas Long. Development of building energy asset rating using stock modelling in the usa. *Journal of Building Performance Simulation*, 0(0):1–15, 0.

[15] Yongin Kwon, Sangmin Lee, Hayoon Yi, Donghyun Kwon, Seungjun Yang, Byung-Gon Chun, Ling Huang, Petros Maniatis, Mayur Naik, and Yunheung Paek. Mantis: Automatic performance prediction for smartphone applications. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 297–308, Berkeley, CA, USA, 2013. USENIX Association.

Eidesstaatliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtliche oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Passau, den 11. August 2017 _____