University of Passau

Department of Informatics and Mathematics

Master's Thesis

Performance Measurement of C Software Product Lines

Author:

Florian Garbe

March 15, 2017

Advisors:

Prof. Dr.-Ing. Sven Apel

Chair of Software Engineering

Prof. Christian Lengauer, Ph.D.

Chair of Programming

Garbe, Florian:
Performance Measurement of C Software Product Lines
Master's Thesis, University of Passau, 2017.

Abstract

Software Product Line (SPL) are the answer to the rising demand for configurable and cross-platform systems. For such a system with just 33 configurable features, there are already more possible derivable variants than humans on our planet. Maintaining and analyzing these highly configurable software systems can be a difficult task. It is not uncommon that performance-related issues, especially in the maintenance phase, are a major risk to the longevity of a project. Variability encoding is the transformation of compile-time variability into load-time variability and is a technique that can be applied to SPL for further analysis. This work introduces the combination of performance measuring functions and variability encoding in order to gain performance related information for individual features and compositions of features. We also propose a method to make performance predictions program configurations by utilizing the previously mentioned feature data that was obtained through analyzing other configurations.

AST Abstract Syntax Tree

CNF Conjunctive Normal Form

CPP C Preprocessor

FID Feature-Interaction Degree

FM Feature Model

FTH Feature-Time Hashmap

PE Percent Error

SPL Software Product Line

Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

Importance of performance

Performance is often a very important factor for software systems, especially for embedded systems [Hen08]. A well optimized software system has less restrictions towards the required hardware components and saves (battery) power. Often, the customer uses these design parameters to rank the different service or software providers according to their results. "93 percent of the performance-related project-affecting issues were concentrated in the 30 percent of systems that were deemed to be most problematic" [WV00]. Furthermore pinpointing the parts of a software system, that are most suitable for conducting performance related improvements, can be difficult.

SPL introduction

These concerns increase in complexity for the Software Product Line (SPL) domain. SPL are highly configurable systems that can be customized in different aspects, e.g., to deploy a product for different platforms. However in the domain of `C` SPL the variability is implemented in the form of preprocessor directives which are oblivious to the underlying programming language. In order to use the code or reuse existing verification tools these preprocessor annotations have to be resolved. The amount of derivable variants for a SPL scales exponentially with the amount of configuration options and, as a consequence traditional analysis methods applied to each derivable variant is not feasible [SRK+11].

Family-based approach

In order to tackle these issues new analysis methods have been explored, so called family-based analysis [TAK+12, LvRK+13, KvRE+12]. Their research demonstrates that family-based or variability-aware analysis are able to produce results in the fraction of time compared to sequential analysis of each variant. Furthermore, the variability-aware results are complete, in contrast to other analysis methods that only look at a small subset of the whole configuration space.

Variability encoding

Post et al. introduce the idea of variability encoding as a method to reuse Microsoft's Static Driver Verifier for device drivers that are implemented as SPLs [PS08]. They lay out rules to transform the compile-time variability into run-time variability, e.g., configuration-dependent execution of a statement is transformed into a standard `C IF` statement. The preprocessor configuration options are encoded as global variables and used as conditions in the previously mentioned `IF`

statements. The output of variability encoding is a so called *meta product* or *simulator* and these simulators are able to find bugs in existing systems [vR16, PS08].

Performance measuring

Looking back at the initial problem, we now want to manipulate the variability encoding process to automatically conduct performance measurements for the different configuration options. Since variability encoding already transforms conditional preprocessor directives into standard `IF` statements it seems reasonable to add performance measuring functionality at that point. The results can then be analyzed to judge the impact that the selection of a configuration option has on the product as a whole, which can be very useful during development or maintenance or even for the users.

Performance prediction

The results of these performance measurements are gained by executing the simulator in a certain configuration, or in multiple configurations. This data can also be used to make performance predictions for one of the remaining configurations. The accuracy of these predictions depend on the compatibility of the configurations or the code-coverage of the initial data set. A perfect example scenario is a SPL that consists of optional and independent configuration options that all add functionality. Conduct performance measurements for the configuration with all options turned on and use these results to make accurate predictions for all the other combinations of selectable options. This saves a lot of effort for medium and large scale SPL compared to individually analyzing each derivable variant.

1.1 Objective of this Thesis

The main objective of this thesis is to introduce the concept of combining variability encoding with performance measuring. Since this is a novelty approach in the `C` SPL domain, we highlight the potential shortcomings of our methods to discuss possible improvements. Next, we explain how to utilize the measured performance data for one or multiple SPL configurations to predict the performance of another configuration.

We implemented this approach in the Hercules project, which is an extension to the variability-aware parsing framework TypeChef (see Chapter 2). We decided to use two different case studies to apply our performance analysis and to conduct different performance predictions. One of our test systems is a small basic model of an elevator and the other one is the real-world SPL SQLite. These two vastly diverse target systems will help us answer the following research questions:

RQ1 What is the ideal scenario for our approach towards performance measurements for SPLs?

RQ2 How is the execution time distributed? Does the annotation-free code base use up most of it?

RQ3 What efforts have to be made in order to make accurate performance predictions?

RQ4 How do different groups of configurations perform in predicting other configurations?

1.2 Structure of the Thesis

Chapter 2 first introduces different concepts in the preprocessor-based SPL domain in order to properly explain variability encoding. Chapter 3 then highlights how we use the variability encoding process to weave in performance measuring functions. These functions are easily exchangeable so it is important to explain their current implementation in detail to nurture ideas for improvements. Chapter 4 focuses on the results of applying our approach to different case studies. Finally Chapter 5, Chapter 6 and Chapter 7 complete our thesis with a conclusion, as well as future work and related work.

2. Background

In this section we will briefly introduce different concepts and terminology. First we explain Software Product Lines (SPLs) and other concepts that are related to them. Last we establish the core asset that the thesis is based on, variability encoding.

2.1 Software Product Lines

> Clements et al. [CN+99]: "A software product line is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."

The above definition from Clements et al. describes the important aspects of SPLs. Their goal is to reduce development and maintenance efforts by applying strategic reuse of core aspects of a product and to offer customization in order to be able to market variants of the product to different customer groups or different hardware systems.

We will now specify this general explanation of SPL for our context of preprocessor-based `C` SPLs. On the one hand the common set of core assets equates to the shared code base of the product line which is basically the preprocessor-free code. On the other hand the code segments that are part of conditional preprocessor directives resemble the code parts that belong to these optional program features. A program feature is a unit that encapsulates program functionality, more in Section 2.3. Compiler arguments or build tools can then be utilized to decide the selection status of each individual feature and derive a variant of the software product line.

2.2 Introduction of our Example SPL: Calculator

Throughout this thesis, we will explain different topics and provide specific examples on the basis of a hypothetical *Calculator* SPL. The general concept and goals of this product line are to implement a basic calculator. The most basic variant of the

Calculator is only able to display numbers and operations like addition and division are implemented as features. This Calculator product line will be constantly used for demonstrations in this thesis. However, these examples and snippets from the Calculator system are not supposed to be complete or represent functional code. Instead they are used as basic examples and reduced to the parts that are necessary for the explanations.

2.3 Features and Feature Models

In the previous section we already mentioned features in the context of software product lines and will now introduce them. First we will look at different definitions for features that can be found in publications:

> Zave [Zav99, Bat05]: A feature is an increment in program functionality.

> Kang et al. [KCH+90]: A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems.

> Kästner et al. [KA13]: A feature is a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option.

These are three different ways to characterize features in the software domain. The first two definitions focus on features as general concepts in the software domain whereas the third definition is much more specific and applicable to our use of the term *feature*.

Since we are mostly working on the source code level the term *feature* in the context of this thesis will be used when we talk about the conditions inside the conditional preprocessor directives. These conditions can either be just basic feature names as in `#ifdef OS_UNIX` or feature expressions which utilize boolean operators, e.g. `#if defined (OS_WIN) || defined (OS_OTHER)`. We will refer to these conditions as either *feature (expression)*, *context* or *presence condition*. Selecting a feature in the compilation process will then include all the different preprocessor directives scrambled across the whole code base that this feature selection satisfies.

Most product lines have limitations and restrictions for the feature selection process, the so called Feature Model (FM). A FM is a structure that defines valid feature combinations. Visual representations of FMs are tree-like structures [Bat05]where the nodes can have different relationships to each other. Example relations:

- *Mandatory* and *optional* relation between parent and child nodes.
- *Requires* and *exclude* relation between any two nodes.
- *Alternative* or select *at least 1* group relation between parent and its child nodes.

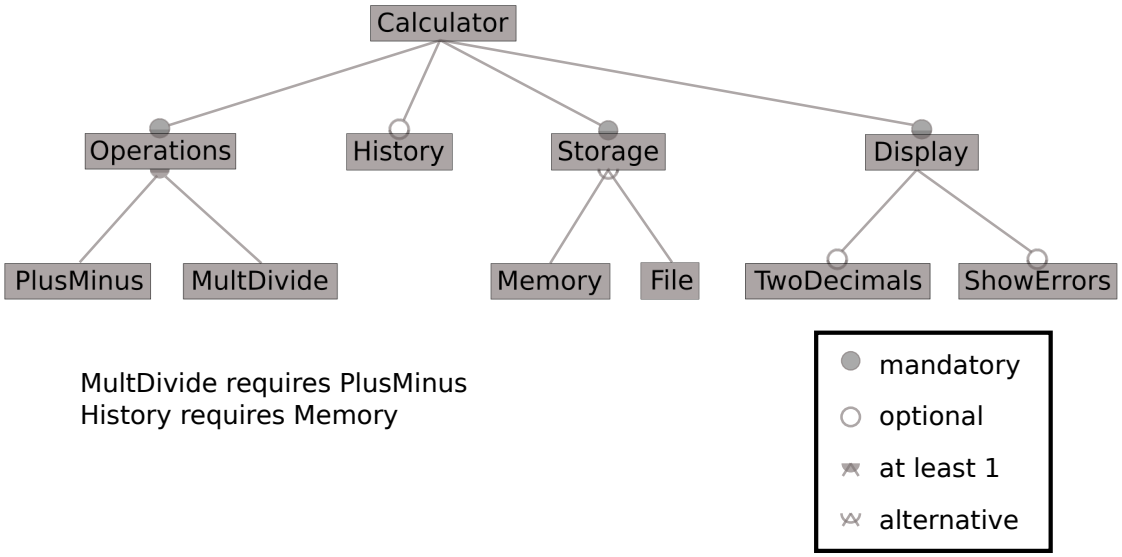

Figure 2.1: Feature model for the Calculator SPL

These are just a few example relations and the visualizations and relations can differ greatly between different publications. In Figure 2.1 we briefly illustrated the FM of the Calculator SPL. In order to generate a valid Calculator variant we have to fulfill the requirements of the FM, for example, we have to select all mandatory features, we can only choose one of the features `Memory` or `File` or the selection of the feature `MultDivide` also requires `PlusMinus` to be selected.

The previously explained visualization of the FM can also either be developed as a propositional formula instead or turned into one. Each feature corresponds to a boolean variable where true indicates that the feature has been selected or enabled and false otherwise. This propositional formula can be used for automated checks and tasks related to the feature model [MWC09, MWCC08, TBK09]. Checking if a chosen configuration `A` is valid for a FM `M` for example requires a satisfiability check `isSat(M && A)`. This propositional formula can be saved in various different formats. The variability-aware parsing framework TypeChef[1] we are using is able to parse FM descriptions in the DIMACS[2] format which is basically a list of terms followed by clauses & expressions in the Conjunctive Normal Form (CNF) format. The DIMACS excerpt shows how the formulas for the optional feature `History` look like when taking into account that `History` requires `Memory` and `Memory` is part of an alternative group together with the feature `File`.

```
c 1 History
c 2 File
c 3 Memory
...
c 12 ShowErrors
p cnf 12 5
-2 -1
-2 -3
2 3
...
```

DIMACS excerpt

[1] https://github.com/ckaestne/TypeChef
[2] http://www.domagoj-babic.com/uploads/ResearchProjects/Spear/dimacs-cnf.pdf

2.4 Variability in C SPL

In this thesis our focus lies on variability in `C` SPL implemented with the C Preprocessor (CPP). The CPP is a powerful text-based preprocessing tool that extends the capabilities of the standard `C` programming language. The CPP has seen widespread usage since it's launch over 4 decades ago. However, since the CPP annotations are oblivious of the underlying structure of the programming language, it is difficult for developers and analysis tools to cope with CPP directives [SC92, EBN02, LKA11, Pad09, VB03]. Code that uses nested `#ifdef` statements for conditional compilation can be very hard to understand and maintain and existing analysis and verification tools for `C` software are not able to handle preprocessor directives. The consequences of the latter is that in order to use these existing tools the code has to be preprocessed in all possible variants or new variability-aware tools have to be developed from scratch. Deriving all possible products however is not feasible for most software systems.

The CPP allows developers to utilize several new tools in their software systems: text substitution with `#define` macros, conditional compilation with `#ifdef` directives and file inclusion with `#include` directives. The variability-aware parsing framework TypeChef parses the source code and deals with these preprocessor directives accordingly. The `#define` and `#include` directives are resolved by textual substitution of their macro or file contents before TypeChef starts the parsing process. TypeChef will then create a variability-aware Abstract Syntax Tree (AST) in which the variability expressed via conditional preprocessor directives `#if`, `#elif`, `#else`, `#ifdef` `#ifndef`, `#else` is preserved by utilizing `Choice` and `Opt` nodes.

```
1  void divisionByZero(
2  #ifdef TWODEC
3    float
4  #else
5    double
6  #endif
7      dividend) {
8    if (
9    #ifdef SHOWERROR
10       1
11   #else
12       0
13   #endif
14   ) {
15     displayError(
16       dividend);
17   }
18   throwError();
19 }
```

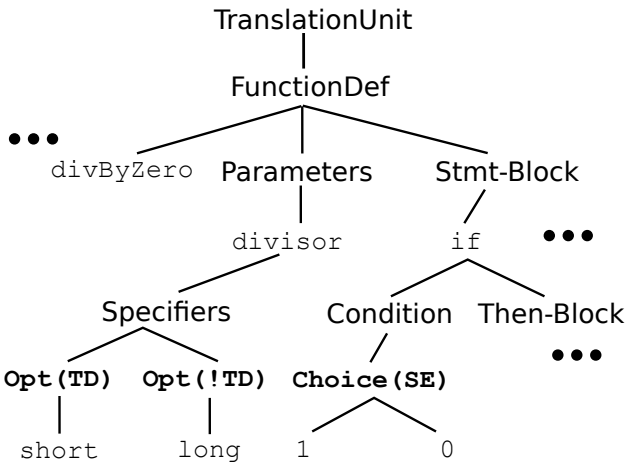

Figure 2.2: Expressing variability in the AST

Developers can now implement new variability-aware analyses which utilize this AST data structure. Kenner et al. developed TypeChef as a framework for variability-aware type checking [KKHL10] however their framework has been used for other variability-aware tasks, as well. Liebig et al. have developed Morpheus, a refactoring tool for preprocessor based SPL [LJG+15] and presented results of type checking and liveness analysis [LvRK+13].

Going back to our Calculator SPL, Figure 2.2 depicts a code snippet on the left side and the complying variability-aware AST representation on the right side. This example shows how variability is preserved by TypeChef AST generation process. The function parameter `dividend` in line 7 has two different specifiers `float` and `double`, which depend on the selection of the feature `TWODEC`. In the AST the two possible specifiers are represented by `Opt` nodes. `Opt` nodes are basically tuples with 2 entries: a feature expression and an AST element, e.g. `Opt<!TD, LongSpecifier>`

Besides `Opt` nodes TypeChef also uses `Choice` nodes to express variability. The general concept of the choice calculus was presented by Erwig et al. in his publication "The Choice Calculus: A Representation for Software Variation" [EW11]. In TypeChef these `Choice` nodes are implemented as a tree-like data structure that has 3 data fields: a feature expression and two child nodes. If the feature of the feature expression is selected then the first child node will be evaluated, otherwise the second child node will be chosen. The two child nodes can either be `Choice` nodes or `One` nodes, where the `One` node is a terminal node that only holds a leaf AST element. This makes it possible to have nested `Choice` nodes.

2.5 Variability Encoding

The following section introduces the most important core aspect for our performance measuring process: variability encoding. We will talk about the general concept of variability encoding and go into details specifically when applied to the `C` programming language by providing examples. At last we will highlight shortcomings and problematic aspects we encountered as we applied variability encoding on real-world product lines.

2.5.1 Introduction

As we have seen in the previous sections, `C` SPL heavily utilize external language tools like the CPP to express variability. These CPP annotations however have to be resolved before compilation since they are not a part of the standard `C` language. The objective of variability encoding is to transform these annotations, that the programming language itself is oblivious to, into standard programming constructs without changing the behavior of the software system. These transformations create a new *meta product* which encapsulates the behavior of all different variants of the software product line. This meta product is also called *product simulator*.

2.5.2 Approach

Our general approach towards variability encoding in the domain of C software systems is to first rename or rename & duplicate top level declarations inside `#ifdef`

directives and second transform the remaining `#ifdef` statements in functions to C-conform `IF` statements. The first practical advances have been made by Post et al. [PS08] in their paper Configuration Lifting: Verification meets software configuration where they manually execute the transformation process on a small Linux sound driver. Based on their idea we have developed the tool Hercules[3] under guidance and in heavy collaboration with (but not limited to) Jörg Liebig, Alexander von Rhein and Christian Kästner. Hercules is implemented as an extension to the variability-aware parsing framework TypeChef and automates the transformation from compile-time variability expressed via CPP directives to run-time variability by utilizing renamings, duplications and IF statements. The FM is used in order to avoid unnecessary code duplications by checking satisfiability of every computed variant that has to be created by the duplication process. The feature selection state is encoded in the form of global variables, one for each distinct #ifdef NAME.

```
1  #ifdef TWODEC
2    float
3  #else
4    double
5  #endif
6    result;
7    // ...
8  #ifdef SHOWERROR
9    displayError(result);
10 #endif
```

a) Original source code

```
1  float _TD_result;
2  double _NTD_result;
3  // ...
4  if (opt.SHOWERROR) {
5    if (opt.TWODEC) {
6      displayError(_TD_result);
7    } else {
8      displayError(_NTD_result);
9    }
10 }
```

b) Meta product code

Figure 2.3: Code before and after variability encoding

2.5.3 Example

Taking a look at Figure 2.3 we can see how the transformation of this example looks like. The original source code is on the left side and the result of our variability encoding can be found on the right side. The variable `result` in a) lines 1-6 has two derivable variants, one with the specifier `float` and the other one with specifier `double`. As we previously mentioned dealing with variable declarations requires us to duplicate code and the example shows that there are now two new definitions `float _TD_result` and `double _NTD_result` in b) lines 1-2. On the other hand the optional function call of `displayError` is embedded into an `IF` statement which uses our feature selection variable `opt.SHOWERROR`. As a consequence of replacing the original declaration of `result` with two new declarations we also have to create new `IF` statements every time `result` has been used in the original source code, see lines 5 and 7 b).

[3] https://github.com/joliebig/Hercules

2.5.4 Goal

The newly created *meta product* can be used with traditional verification tools, which are not variability aware, and the current configuration can be switched on the fly by changing the global feature variables. To give an example, model checking tools can potentially analyze the result of variability encoding and explore the whole configuration space at once [vR16].

2.5.5 Behavior Preservation

The crucial property of the variability encoding transformation has to be behavior preservation. In order to draw meaningful conclusions about the properties of the original SPL by analyzing its *meta product* the behavior must not change. Alexander von Rhein [vR16] has developed a formal correctness proof for the behavior preserving properties of variability encoding for the language Featherweight Java, a functional subset of the Java programming language. Since the `C` programming language includes complex language mechanics such as `switch-case` statements, `goto` statements, `enums` and `structs` a formal correctness proof is not feasible. However, we were still able to utilize Hercules and apply transformations to real-world product lines such as BusyBox, SQLite and a few Linux drivers.

2.5.6 Shortcomings

As previously mentioned, we were able to apply variability encoding with Hercules to real-word product lines. However, we have also found problematic cases were our approach fails. Sometimes these cases can be fixed by manually rewriting the source code and sometimes the amount of derivable variants for a certain programming constructs makes the transformation infeasible. If Hercules encounters an extreme pattern with many possible variants it will not transform that element and manual intervention, if possible, is required.

```
1  static const char * const azCompileOpt[] = {
2  #define CTIMEOPT_VAL_(opt) #opt
3  #define CTIMEOPT_VAL(opt) CTIMEOPT_VAL_(opt)
4  #ifdef SQLITE_32BIT_ROWID
5    "32BIT_ROWID",
6  #endif
7  #ifdef SQLITE_4_BYTE_ALIGNED_MALLOC
8    "4_BYTE_ALIGNED_MALLOC",
9  #endif
10 #ifdef SQLITE_CASE_SENSITIVE_LIKE
11   "CASE_SENSITIVE_LIKE",
12 #endif
13 #ifdef SQLITE_CHECK_PAGES
14   "CHECK_PAGES",
15 #endif
16 #ifdef SQLITE_COVERAGE_TEST
17   "COVERAGE_TEST",
18 #endif
19 ... // 100 additional #ifdef directives
```

Listing 2.1: Exponential explosion of variability in SQLite

Variant explosion

We encountered these patterns in product lines like `BusyBox` and `SQLite` [LvRK+13]. Listing 2.1 depicts an example for this pattern that was taken from SQLite: a variable declaration that contains a total of 105 different `#ifdef` directives. This leads to an exceptional large number of variants of `azCompileOpt[]` even when taking into account that some variants are not valid according to the FM of SQLite. To make things worse all usages of `azCompileOpt[]` have to be duplicated as well. But in this case we were able to create a function that assigns the correct content to `azCompileOpt[]` by using 105 `IF` statements that append the content of the complying `#ifdef` statement to the current contents, similar to a StringBuilder. However, this solution cannot be applied to `enums` and `structs` which can theoretically be implemented in similar fashion with an explosion of variants.

Setup problems

TypeChef and Hercules are still work in progress and there are still significant limitations that prevent applying them to real-world systems [KKHL10, vR16]. Hercules can only start the transformation process if TypeChef is able to parse the source code and there are no type errors. Any type error found by TypeChef would immediately propagate into the meta product created by the variability encoding process and as a consequence the meta product cannot be compiled. Most real-world systems do not provide a thorough FM and if features are not compatible together TypeChef will immediately find these type errors. In this case the FM has to be forged and tweaked manually.

3. Approach

In this section we talk about our approach towards performance measuring for `C` Software Product Lines (SPLs) by utilizing variability encoding.

3.1 Overview

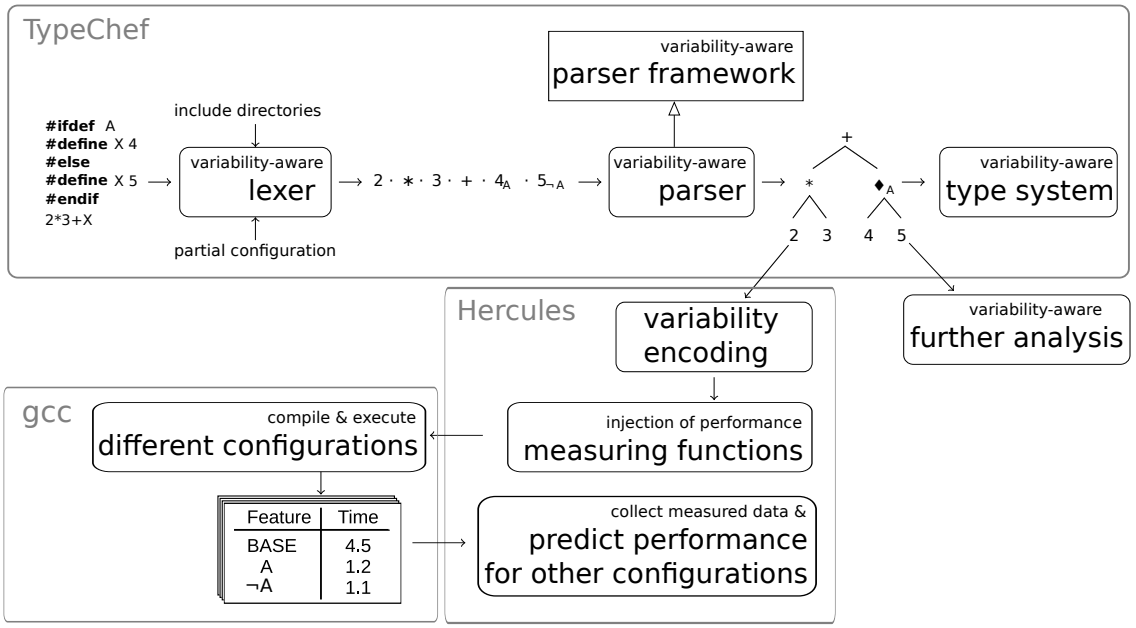

Figure 3.1: Overview for our Performance Measuring Process

Figure 3.1 depicts our general idea behind our performance measuring and prediction process. The top part breaks down the different steps that TypeChef has to execute in order to generate the variability-aware Abstract Syntax Tree (AST) [KKHL10]. Next, the bottom half shows how Hercules utilizes the AST to generate the so called *meta product* or *product simulator* through variability encoding and the next steps that are needed for generating performance measurements. For the remainder

of this thesis we will refer to the result of variability encoding as *product simulator* and to the result of the injection of measuring functions as *performance simulator*. The following sections provide a detailed explanation of these individual procedures.

3.2 Combining Variability Encoding with Performance Measuring

Section 2.5 shows how code can be manipulated to transform compile-time variability to run-time variability. One of the code transformation techniques is to transform optional or variable statements to `IF` statements. This is where the performance measuring process starts. Every time one of these statements is turned into a new `IF` statement we add two function calls, one at the beginning of `IF` and one at the end. We can now measure the difference in the timestamps between these two function calls, which resembles the execution time for this `IF` statement. The injection of these functions is pretty straight forward for most of the affected transformations. In Section 3.4 we will briefly talk about our approach towards dealing with more complex programming structures.

It is important to highlight that our current implementation for these performance measuring functions is easily interchangeable and can be manipulated to add further improvements because their definition is part of a separate file that is loaded via `#include`. We will now explain their implementation:

Algorithm 1: Implementation of the performance measuring function perf_before

```
1  Stack<String> context_stack;
2  Stack<Integer> time_stack;
3  Stack<Boolean> is_new_context;
4  Hashmap<String, Integer> context_times;
5  function perf_before (context);
   Input : String representation of associated context
6  Integer before_outer = getTime;
7  time_stack.push(before_outer);
8  if context ∉ context_stack then
9  |  context_stack.push(context);
10 |  is_new_context.push(TRUE);
11 else
12 |  is_new_context.push(FALSE);
13 end
14 Integer before_inner = getTime;
15 time_stack.push(before_inner);
```

The first function, inserted at the start of each `IF` statement that was created by variability encoding, is called *perf_before* and is responsible for multiple things. This function requires a `char*` argument that resembles the representation of the context that is also part of the `IF` condition. The general overview can be seen in algorithm 1.

Calling `perf_before` first generates an immediate time stamp `before_outer`, that is used to measure the measurement overhead itself later on. Next, it puts the label of the previous `#ifdef` statement on top of the *context stack* of `#ifdef` labels but only if the context is not already present in the current *context stack*, see lines 8-13. This is to avoid recursive function calls or loops to unnecessarily affect the *context_stack*. An example for this is optional code in context `X64` calls a function that also has optional code parts under context `X64`. In the same lines we also keep track of the information whether the context is a new addition to `context_stack`, or not by adding it to *is_new_context* for later use. In a scenario of nested `#ifdefs` we are able to produce the absolute context of the current statement by combining all `#ifdef` labels from the *context stack* with the boolean & operator. The last step is to generate a second time stamp *before_inner* and throw both *before* time stamps onto the *time_stack* so we can use this information for our second function *perf_after*.

Algorithm 2: Implementation of the performance measuring function perf_after

```
Stack<String> context_stack;
Stack<Integer> time_stack;
Stack<Boolean> is_new_context;
Hashmap<String, Integer> context_times;
// ...
16 function perf_after;
17 Integer after_inner = getTime;
18 Integer context_time = after_inner - time_stack.pop;
19 Integer new_context_time = context_time;
20 String assembled_context = getContext(context_stack);
21 if is_new_context.pop then
22 |   context_stack.pop;
23 end
24 if assembledContext ∈ context_times then
25 |   new_context_time += context_times.get(assembled_context);
26 end
27 context_times.put(assembled_context, new_context_time);
28 Integer before_outer = time_stack.pop;
29 Integer after_outer = getTime;
30 Integer measurement_overhead = (after_outer - before_outer) - context_time;
```

algorithm 2 shows the general idea behind the *perf_after* implementation. The first obligation of the `perf_after` function in line 16 is to also generate a time stamp called *after_inner*. This time stamp is used with *before_inner* to compute the elapsed time between the end of *perf_before* and the beginning of *perf_after* in the very next line. *context_time* then resembles the execution time of the statements between our injected function calls. This information is put into a `Hashmap[String,Double]`, where we add the current time for the combination of features in our *time_stack* to previously measured execution time for this feature combination. The last step is once again to take a time stamp *after_outer*. These *outer* time stamps are used

during post processing to account for the execution time of our injected performance measuring functions.

We also injected similar functions at the start and end of the `MAIN` function which measure the time for the context `BASE`. This way the program itself generates a hashmap, which contains the sum of all execution times for each context and combination of nested features of statements that have been executed in the given configuration. Each different configuration generates a different hashmap, unless two configurations have exactly the same executable code. However, in that case the execution times should still be different because they fluctuate. This hashmap has to be post processed to account for multiple nested measurements as explained in Section 3.6. We will refer to this hashmap as the Feature-Time Hashmap (FTH).

3.3 Example Software Product Line

Before we go into details about how we utilize the information in the resulting FTH let us take a look at what actually happens to the code in our calculator product line.

Listing 3.1 shows how the code for multiplication looks after variability encoding and injection of our performance measuring functions *perf_before* and *perf_after*. The `#ifdef` and `#else` directives from Listing 3.1 (a) Line 5 and 11 have been transformed to `IF` statements, Listing 3.1 (b) Line 5 and 11 respectively. Additionally the performance functions have been inserted at the start and end at each of the generated `IF` statements.

Execution of the calculator product line with features `TWODEC` and `HISTORY` selected will now automatically measure the time the code of each of these features needs to be executed when calling function `multiply`. Figure 3.2 shows how the *context stack* progresses during the execution of the program. # is used as a separator between features or feature expressions in nested `#ifdef` directives and M# is an abbreviation for MultDivide#. The following line numbers are in reference to Listing 3.1 b). Since the function `multiply` itself is defined in an `#ifdef` directive with identifier `MULTDIVIDE` the function call also has to be inside a similar `#ifdef` directive. This is why the initial state of the context stack already contains the information for the feature `MULTDIVIDE`. Each execution of our *perf_before* function then adds a new feature or logical combination of features to the *context stack*. The first execution in line 7 pushes `TWODEC` on the stack. On the other hand calling *perf_after* removes the last feature of the *context stack*. In line 11 the feature `TWODEC` is discarded and in line 14 the new feature `HISTORY` is added.

```
1  #ifdef MULTDIVIDE
2  double multiply(double a,
3      double b) {
4    double res = a*b;
5    double rf;
6  #ifdef TWODEC
7    // 2 decimal places
8    rf = 100.0;
9    res = round(res*rf)
10     / rf;
11
12 #endif
13 #ifdef HISTORY
14
15   saveResult(res);
16
17 #endif
18   return res;
19 }
20 #endif
```

a) Original source code

```
1
2  double multiply(double a,
3      double b) {
4    double res = a*b;
5    double rf;
6    if (opt.twodec) {
7      perf_before(twodec);
8      rf = 100.0;
9      res = round(res*rf)
10       / rf;
11     perf_after();
12   }
13   if (opt.history) {
14     perf_before(history);
15     saveResult(res);
16     perf_after();
17   }
18   return res;
19 }
20
```

b) Performance measuring code

Listing 3.1: Injection of performance measuring functions

One of the requirements of successfully generating performance measurements is that each call of `perf_before` is followed by the corresponding call of `perf_after` because otherwise the information inside the *context stack* would be corrupted. This will be discussed in further detail in Section 3.4.

The end result of executing our program is a hashmap, FTH, which accumulates the execution time of statements tied to their context in the form of features or combinations of features. Table 3.1 shows what the hashmap looks like after one multiplication with the calculator product line and features `MultDivide`, TWODEC and HISTORY enabled. It is important to note that the time measured for `MultDivide` includes the execution time of the other two features, which are nested inside the `#ifdef` directive from the function call of `multiply`.

After line 1
MULTDIVIDE

After line 7
M#TWODEC MULTDIVIDE

After line 13
M#SAVEHIST MULTDIVIDE

After line 16
MULTDIVIDE

Figure 3.2: Context stack progression

feature	time (seconds)
MULTDIVIDE	10
M#TWODEC	2
M#HISTORY	4

Table 3.1: Feature time hashmap

3.4 Dealing with Control Flow Irregularities

When applying our approach to real world product lines we quickly noticed that it's not as simple as inserting a function to start the measurement and a corresponding function to end that measurement. In Section 3.3 we mentioned that it's necessary that each performance measuring starting function is at some point followed by its corresponding measuring ending function in order to avoid corrupting the *context stack*. However, the `C` programming language contains multiple programming constructs that make it possible to skip the corresponding measurement ending function call: `break, continue, goto, return` statements. These statements cause control flow irregularities meaning that they have the ability to prevent the execution of a time measurement ending function after its corresponding measurement starting function has already been called.

In order to tackle this issue we add additional measurement ending calls as necessary before these statements. This is done in a top down traversal of the AST of the program after applying variability encoding where we have to keep track of the number of currently active time measurements when encountering certain programming elements. A measurement is considered as active after calling the function `perf_before` and before its corresponding `perf_after` function is called. Multiple active measurements happen when the original source code contained nested `#ifdef` directives.

```
1  // Case statements and breaks
2      case 29:
3          if (!omit_pragma && !omit_pager_pragmas) {
4            ❶ perf_before("!omit_pragma && !omit_pager_pragmas");
5              if ((! zRight)) {
6                  returnSingleInt(pParse, "synchronous", (pDb->safety_level - 1));
7              } else {
8                  if ((! db->autoCommit)) {
9                      sqlite3ErrorMsg(pParse, "Safety level may not be changed
10                             inside a transaction");
11                 } else {
12                     (pDb->safety_level = (getSafetyLevel(zRight, 0, 1) + 1));
13                     setAllPagerFlags(db);
14                 }
15             }
16           ❌ perf_after();
17             break;
18           ❶ perf_after();
19         } else {
20             // more code
21         }
22         break;
23     case 30:
24       // more code
```

Listing 3.2: Handling control flow irregularities in SQLite: `break`

For `return` statements we calculate the difference between active time measurements before executing the `return` call and active time measurements when entering the body of the function that the `return` belongs to. Afterwards an additional amount

of `perf_after` function calls is added in front of the `return` statement. The exact amount depends on the previously mentioned difference. `goto` statements are handled in a similar way and for `continue` and `break` statements we keep track of the difference in active measurements to the associated loop (or `case` statement for `break`). This way we ensure that previously started measurements are properly finalized before jumping to a different part of the program code.

Listing 3.2 and Listing 3.3 depict examples from our SQLite case study. Code parts which were not relevant to the following explanation were left out. The injected performance measuring functions from Section 3.2 are annotated with numbers, e.g., Listing 3.2 line 4 and 18 are annotated with the same number which indicates that these are corresponding performance starting and ending functions. The additional performance ending function calls from Section 3.4 are annotated with the letters xyz.

First, we take a look at Listing 3.2 where parts of a `switch` statement can be seen. The `IF` statement in line 3 is generated from an `#ifdef` directive in the original source code. The next line then starts the performance measuring process and it is considered active from line 5 to line 18 where the `if` statement ends. However, since there is a `break` statement in line 17 the corresponding `perf_after` function will never be executed. This is why according to our previous explanation we are adding one additional `perf_after` call to properly terminate the performance measuring for the feature *!omit_pragma && !omit_pager_pragmas* because there is one additional active measurement when executing the `break` statement compared to the start of the `case` statement in line 2.

```
1  // Functions and return
2  static int btreeCreateTable(Btree *p , int *piTable , int createTabFlags )  {
3      int rc;
4      // more code
5      if (!omit_autovacuum) {
6        ❷ perf_before("OMIT_AUTOVACUUM");
7          rc = allocateBtreePage(pBt, (&pPageMove), (&pgnoMove), pgnoRoot, 1);
8          // more code
9          if (rc == 0) {
10       ⓨ perf_after();
11           return rc;
12         }
13         if (((id2i_sqlite_coverage_test ) )) {
14           ❸ id2iperf_time_before_counter("SQLITE_COVERAGE_TEST", 628);
15             if (rc != 0) {
16                 releasePage(pPage);
17               ⓩ perf_after(); perf_after();
18                 return rc;
19             }
20           ❸ perf_after();
21           }
22       ❷ perf_after();
23     }
24 }
```

Listing 3.3: Handling control flow irregularities in SQLite: `return`

In our other example Listing 3.3 the focus is on `return` statements. There are two places in which variability encoding generated `if` statements, lines 30 and 38. Before executing the return in line 36 there is 1 active time measurement from line 31 and so we have to add one `perf_after` call annotated with ❾. The `return` statement in line 42 the situation is different. Both measurements from lines 30 and 38 are considered active and this is why we have to add two `perf_after` function calls in line 42❷ before executing the `return` statement.

3.5 Feature Interactions

We have already mentioned examples where we feature interactions occur but did not explain them yet. Feature interactions in the context of this thesis describes either a direct nesting of conditional preprocessor directives in the code itself or a nesting via function calls. We use a delimiter symbol # to differentiate between nesting of `#ifdefs` and the preprocessor expressions themselves. The so called Feature-Interaction Degree (FID) for a given feature combination in the *context stack* equals the amount of # that occur in the combination.

Listing 3.4 presents different cases of feature interaction. First of all, lines 12-13 are nested `#ifdefs` and therefore the performance measurement in line 14 has a FID of 1. Second, line 4 is a preprocessor directive and executes a function call in line 6. Hence the performance measurement for line 22 also has a FID of 1: `PLUSMINUS#HISTORY`. In contrast, the feature expression inside the `random` function in line 27 et seqq. has FID of 0 since it is not a feature interaction by itself. These interactions are generated automatically during execution of the performance simulator by checking the *context stack*.

```
1  int first, second, result;
2  switch (op) {
3  #ifdef PLUSMINUS
4    case '+':
5      result = add(first,
6        second);
7      break;
8  #endif
9  // ...
10 }
11 #ifdef SHOWERRORS
12   #ifdef TWODEC
13     errors = round(errors);
14   #endif
15   showErrors(errors);
16 #endif
17 }
```

a) Calling add

```
19 int add(int a, int b) {
20   int result = a + b;
21 #ifdef HISTORY
22   archiveInHistory(result);
23 #endif
24   return result;
25 }
26
27 int random() {
28 #ifdef TWODEC && HISTORY
29   seed = getNewSeed();
30 #endif
31   // ...
32 }
```

b) add function implementation

Listing 3.4: Feature interaction example

3.6 Post Processing of Measurements

The previously generated FTH has to be processed to yield useful information. The `C` implementation of our performance measuring functions is intended to be very basic to not affect the overall program run time too much by introducing complex time measuring and collecting algorithms. Since we are working with time stamps and nested performance measurements we need to subtract the time of an inner measuring from its direct predecessor. Therefore, we use a delimiter symbol that allows us to differentiate between a nesting of feature `#ifdef A` in feature `#ifdef B` from their composition `#if defined A && defined B`. Looking back at Table 3.1 generated from the code in Listing 3.1 the final numbers for `MultDivide` changes and the resulting FTH after post processing can be seen in Table 3.2.

feature	time (seconds)
MULTDIVIDE	4
M#TWODEC	2
M#HISTORY	4

Table 3.2: FTH from Table 3.1 after Post Processing

This information by itself can already provide some useful insight into how the program execution time is divided between the different features that were executed in the run under the given program configuration. In order to generate performance predictions for the other possible program configurations we can use the previously generated FTH and filter out all the features that are incompatible with the new configuration. Feature `A` and feature `B` are considered incompatible when their composition `A && B` is not satisfiable in the context of the feature model for that product line. The FTH only provides valuable information after post processing its information. For that reason in the remainder of this thesis the usage of FTH implies it has already been post processed.

3.7 Collection of Prediction Information

Since it is very unlikely that one configuration can cover the whole code basis because of restrictions in the Feature Model (FM) or mutually exclusive code parts from `#ifdef` and `#else` preprocessor directives, we implemented a way to collect the information from multiple FTHs generated over several program runs, each with a different program configuration. If configuration *one* includes the performance measurements for feature `x64` and configuration *two* includes the mutually exclusive measurements for `x86` we can combine the data from both runs to gain information.

However, these FTHs can contain the same feature entries with different execution times across different configurations. This can happen when there are data-flow dependencies across the different features. It is important to note that this data dependence between two features occur when one feature manipulates data inside its preprocessor directive and another feature accesses that data in its own directive without being nested in the previous one. If the same feature entries have different measured times across multiple FTHs, we compute the average of these execution times, the standard deviation, and populate a new summarized FTH with these values. If a feature entry is unique to a configuration it is added to the summarized FTH without further changes. We anticipate that users with insightful knowledge

about the configuration options of their product line can come up with a small subset of configurations which can be used to create a solid basis for multiple prediction scenarios.

Table 3.3: Combining information of multiple FTHs

feature	time (seconds)				
	config 1	config 2	config 3	avg.	std dev.
MULTDIVIDE	4	-	5	4.5	0.5
TWODEC	-	2.2	2.4	2.3	0.08
HISTORY	1	1.1	4.2	2.1	1.29

Table 3.3 shows how we combine performance measurement information from multiple FTH. In this example there are 3 different configurations that were used and they all have different times for features in their FTH. Most of the feature times are similar however `HISTORY` in config 3 needs a lot more execution time compared to the two previous configurations. This is the case because there is a dataflow dependency between `HISTORY` if `TWODEC` and `MULTDIVIDE` are both selected. This causes `HISTORY` to have a very high standard deviation of `1.29` seconds, which is about 61% of its average time `2.1` seconds. Using this data set as a basis to compute performance predictions for other configurations can lead to a high variance because of the uncertainty in the feature `HISTORY`.

$$p = \sum_{e \in F} \begin{cases} e, & \text{if } e \in SAT(e\&c\&fm) \\ 0, & \text{otherwise} \end{cases}$$

where:

p = predicted data
e = entry in FTH
F = FTH
SAT = function to check satisfiability of a boolean expression
c = configuration to generate prediction for
fm = FM

Figure 3.3: Generating prediction data

In order to make predictions we utilize the data from the combination of multiple FTHs, as seen in Figure 3.3. Although the predicted data consists of two different numbers, the average and the deviation. In the end both are just the sum of all entries in FTH that are still considered satisfiable when combined with the FM and the new configuration that is to be predicted.

3.8 Limitations and Problematic Aspects

Now that we have explained our general approach towards performance measuring & prediction for software product lines it is time to look at potential shortcomings:

meta product performance difference

First of all the meta product of a product line encapsulates the properties of all derivable variants. A widespread usage of the C Preprocessor (CPP) is implementing code for different hardware architectures, e.g. to choose between different types for variables like `int` and `long`. In order to utilize these variable types in the meta product we have to resort to code duplications. But these code duplications can cause an increase in memory consumption and negatively affect the performance of the meta product compared to the original product line, see Section 4.3.6.

no granularity control

The time measurements are injected in every former `#ifdef` directive even when the number of statements inside the `#ifdef` directive is low or the execution time of the code is completely negligible. These measurements cause additional measurement overhead which can negatively affect the accuracy of our predictions. Even when accounting for the measurement time by using inner and outer timestamps there will always be a call to `free` for variables after the last outer time stamp is generated. The execution time of these performance concluding statements like `free` is not part of the calculated measurement overhead. Instead they are included in the time of a previous outer measurement.

mutually exclusive code

The next potential problem is that our approach can only measure the execution time of actually executed code in one configuration at a time. The consequence is that mutually exclusive code from `#ifdef` and `#else` directives can never be measured in a single program run. In these cases it is important to come up with a sound list of configurations and combine their results as mentioned in Section 3.7

variance because of data dependency

When dealing with multiple configurations the time measured for equal features can differ greatly. The consequence is a high standard deviation for that particular feature in the FTH and as a consequence an increased variance for the performance prediction. The reason for the difference in observed execution time of the same features across different configurations can be data dependency: if feature `A` affects the upper limit of a computational loop that is used by feature `B` in an unrelated preprocessor directive, then the time of feature `B` is increased in configurations where feature `A` is also selected.

The last two paragraphs show that the selection of configurations which are measured and used as basis for the prediction is crucial. In the next section we will take a thorough look at a few different prediction scenarios we selected and how they affect the result of our prediction.

4. Evaluation

As previously mentioned, the data we get from measuring the performance in a configuration can be used in many different ways. The results display the measured execution times for each feature and combinations of features, that are nested in conditional preprocessor directives. We can also collect the information of one or several different configurations and use that data to predict the performance for other configurations. In the following section we take a look at two different case studies and present possible answers to the research questions that were posed in the introduction.

4.1 Test System Specifications

Our experiments were conducted by one of the following computer setups:

DS The desktop system is a personal desktop machine that is powered by an Intel i7-4790K@4.0GHz with 4 cores & 8 threads, 16Gb DDR3 RAM that runs on Ubuntu 14.04.

CS The cluster system consists of 17 nodes, each with an Intel Xeon E-5 2690v2@3.0 GHz with 10 cores, 20 hyper-threads, 64 GB RAM that run on Ubuntu 16.04.

4.2 Elevator Case Study

The Elevator or Lift system has been designed by Plath and Ryan[PR01] as a basic model of an elevator that can be extended by different features, e.g. `TWOTHIRDSFULL` will not accept new elevator calls after reaching over $\frac{2}{3}$ of its capacity since it is unlikely to be able to accept more passengers. This software system has been used in research to reason about feature properties, interactions, analysis strategies and more[AvRW+13, SvRA14].

The Elevator system has 6 features and the feature model can be seen in Figure 4.1. The feature `Base` is mandatory, all other features are optional except for

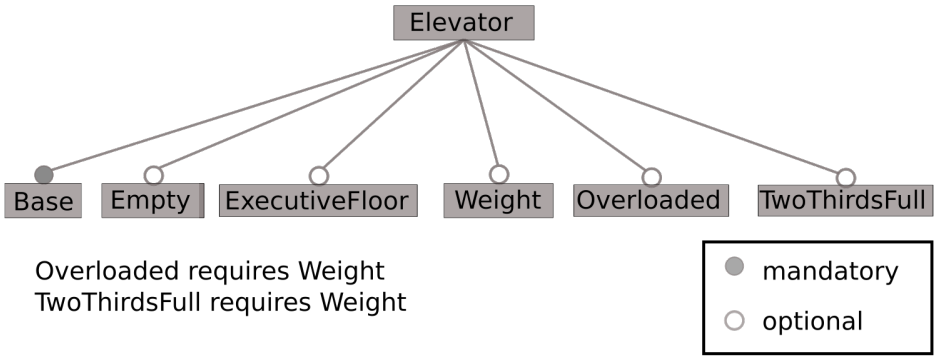

Figure 4.1: Elevator FM visualized

$$((Overloaded \implies Weight) \;\&\&\; (Tworthirdsfull \implies Weight)) \;\&\&\; Base$$

Figure 4.2: Elevator FM as propositional formula

two implications for the features `Overloaded` and `TwoThirdsFull`. The propositional formula for this Feature Model (FM) can be seen in Figure 4.2. According to this FM there are 20 valid configurations for the ELEVATOR system. All times were measured on our system **DS**.

This case study is very close to the ideal scenario for our approach and will help us answer **RQ1**. ELEVATOR does not contain any feature interactions in the form of nested `#ifdefs` in the code or via function calls. All regular features are implemented in isolated conditional `#ifdefs` without any mutually exclusive code parts and it is possible to select all features together, which covers the whole code base. These are the perfect terms for our approach. The only downside of the ELEVATOR case study is the code that belongs to the features has a very low execution time.

```
1  void enterElevator(int p) {
2    enterElevator__before__weight(p);
3      #ifdef WEIGHT
4          usleep(100);
5          weight = weight + getWeight(p);
6      #endif
7  }
8  // ... other Code
9  void leaveElevator__before__empty(int p) {
10     leaveElevator__before__weight(p);
11     #ifdef WEIGHT
12         weight = weight - getWeight(p);
13     #endif
14 }
```

Listing 4.1: Implementation for feature `Weight` in ELEVATOR

feature	time (ms)	NoM
ExecutiveFloor	80.038330	2500
Weight	30.349609	400
Empty	20.287354	200
Overloaded	0.209473	1302
TwoThirdsFull	0.070557	1202
Base	6.579589	1

Figure 4.3: Performance Measuring results in ELEVATOR's allyes configuration

feature	time (ms)
ExecutiveFloor	79.332507
Weight	30.962723
Empty	20.771696
Overloaded	1.498675
TwoThirdsFull	-0.245845
Overloaded&ExecutiveFloor	-1.822054
TwoThirdsFull&Overloaded	0.420249
Base	0.473498

Figure 4.4: SPLCONQUEROR results using all configurations

Listing 4.1 shows the implementation for the `Weight` feature. We have artificially increased the execution time for this feature by adding a `sleep` call in line 4. If not for the sleep call the feature only performs a simple addition or subtraction and this causes increased fluctuations in measured execution times. This also reduces the effect that the measurement overhead has on the overall observed execution times.

4.2.1 Performance Measuring

We have applied our performance measuring approach and created a variant simulator that is able to produce different Feature-Time Hashmap (FTH). The configuration for the variant simulator can be exchanged by editing the values of the global feature variables in the external configuration file. For this case study the *allyes* configuration, where all 6 features are selected, covers all parts of the source code.

There are data dependencies across the features, e.g. `Weight` changes the value for the elevator and `Overloaded` uses this information to judge if the elevator is too crowded. However, these dependencies occur in all valid configurations where `Overloaded` is selected since according to the FM it requires feature `Weight`. Additionally, the fact that `Weight` changes an `int` value that is used in a comparison by `Overloaded` does not affect the execution time of that comparison.

The results after post processing Section 3.6 can be seen in Figure 4.3. Although only the `BASE` feature has undergone changes since all the other features do not have any nested features inside them. A total of 5605 measurements (see column *NoM*). Looking at the `Base` feature we immediately notice that the execution time is significantly higher compared to the other features `Overloaded` and `TwoThirdsFull` although none execute any `sleep` statements. This is one of the downsides of our approach, since all measurements outside of the `Base` feature are nested inside the measurement for `Base` the last `free` call that is executed at the end of every measurement is included in the overlying `BASE` feature.

We have measured each configuration of ELEVATOR 10 times and computed their average execution times. Using these numbers in SPL CONQUEROR[1], a machine-learning library for measuring and predicting performance, we can generate a detailed breakdown of how features and feature combinations affect the performance

[1] http://www.infosun.fim.uni-passau.de/se/projects/splconqueror/

of the Elevator system, seen in Figure 4.4. Comparing these results, which were generated by executing 20 variants, to our performance measuring results in Figure 4.3, which was generated in a single execution of the *allyes* configuration, we can see that the times for the first three features are very close. The other times however are not comparable, possibly because of data dependencies or inaccuracy for measurements below 2 ms.

4.2.2 Overhead calculation flaws

As the results for the `BASE` feature in Section 4.2.1 show, there is a flaw in our approach that affects the performance measurements. Looking back at algorithm 2 in Section 3.2 this flaw is not included. As our performance measuring functions are implemented in `C`, we have to deallocate a variable after conducting the final measurement that computes the internally calculated overhead.

```
1 void perf_after() {
2   double after_inner = getTime();
3   time_struct* t = (time_struct*) pop(&time_stack);
4   // Omitted code
5   double measurement_overhead = getTime()
6       - before_outer - context_time;
7   // store overhead
8   free(t);
9 }
```

Listing 4.2: Deallocation in perf_after affects preceding measurements

The code in Figure 4.2 is a more accurate representation of our implementation. Line 5 executes the final measurement for that feature and calculates the internal overhead. But the `time_struct* t` has to be deallocated and the time needed for this deallocation (+ storing overhead) is attributed to the preceding measurement. Starting the measurement for `BASE` is the first statement inside the `main` function. All of the other performance measurements are nested inside `BASE` or other features. Consequently all measurements, except for `BASE`, are attributed additional execution time from their nested successor measurements. We will talk about potential solutions for this flaw in Chapter 6.

4.2.3 Performance Prediction

We can now utilize the numbers generated from a single execution of the *allyes* configuration from the previous section to make predictions about the performance of other configurations. We already computed the variant times for all 20 configurations of Elevator in the previous section in order to generate results for SPLConqueror.

Table 4.1 shows the 20 different configurations using abbreviations for the features and their execution times compared to the predicted time using our performance prediction approach. To account for incorrectly assigning a `free` call to our time

configuration	variant time	predicted time	percent error
B	0.460100	0.460100*	-
B&EM	20.7212925	20.747454	0.13%
B&W	31.0513020	30.809709	0.78%
B&W&T	31.131196	30.880266	0.81%
B&W&T&O	32.6761007	31.089739	4.86%
B&W&O	32.7689886	31.019182	5.34%
B&W&EM&T	52.3653030	51.167620	2.29%
B&W&EM	52.7860880	51.097063	3.20%
B&W&EM&O	53.693986	51.306536	4.45%
B&W&EM&T&O	54.4927120	51.377093	5.72%
B&EX	80.3105116	80.498430	0.23%
B&EM&EX	100.610495	100.785784	0.17%
B&W&T&EX	110.584903	110.918596	0.30%
B&W&EX	110.721207	110.848039	0.11%
B&W&T&EX&O	110.799599	111.128069	0.30%
B&W&EX&O	110.803103	111.057512	0.23%
B&W&EM&T&EX	130.88851	131.20595	0.24%
B&W&EM&T&EX&O	131.033087	131.415423	0.29%
B&W&EM&EX&O	131.037807	131.344866	0.23%
B&W&EM&EX	131.394696	131.135393	0.20%

Table 4.1: Comparing all variants to our prediction

for the `Base` feature we will instead use the time that was observed for the `BASE` configuration as is stated in the first data row and noted with a '*'. There are 6 outliers with a Percent Error (PE)[2] of over 1% but overall the numbers are pretty similar. Reasoning about the distribution of execution times between features and the shared code base according to **RQ2** does not make sense for this case study, since we artificially tinkered with these execution times. The majority of the time is spent inside optional features.

The effort that is required for this approach is applying variability encoding to the source code and then executing a configuration or multiple configurations in order to generate data used for the prediction of other configurations. For Elevator the efforts according to **RQ3** are as follows: ∼600 ms for variability encoding and 133 ms for executing the performance simulator in the *allyes* configuration. If we add up the execution times of the 20 variants that alone amouints to over 1500 ms. Case studies with more features and many more derivable variants will further increase this gap. However, with more features and possible feature interactions it is possible that measuring one *allyes* configuration will not produce good predictions for all configurations.

4.3 SQLite Case Study

Our next case study is the real-world software system SQLite[3]. It is a highly-configurable database product line and is considered the most widespread database

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

[3] https://www.sqlite.org/

system worldwide [LJG+15]. SQLite has 93 different configuration options in its amalgamation version 3.8.1 and can be tested by its extensive TH3 test suite[4].

Our variability approach has been examined for SQLite towards behavior preservation and the amount of overhead in the form of additional Abstract Syntax Tree (AST) nodes in the meta product vs the original source code [vR16]. The general conclusion is that SQLite is highly compatible for variability encoding with a few restrictions.

The SQLite case study combined with the TH3 test suite are considered black box systems. Without any further insider knowledge and because of the sheer complexity we argue that this software system is not an ideal scenario according to **RQ1**.

4.3.1 SQLite TH3 Test Suite Setup

We will now explain the setup we used for testing our approach with the SQLite case study. The execution times were generated on our cluster system **CS** as average times from 10 runs.

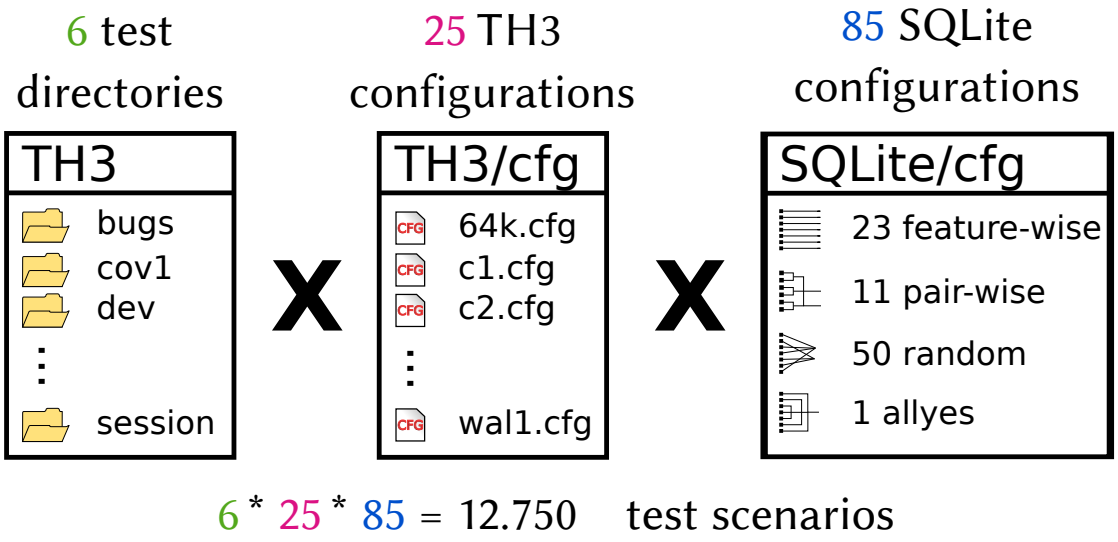

Figure 4.5: SQLite TH3 test setup

First, we need to explain how the TH3 test suite operates. At its core is a script that converts one `.test` file or multiple files in a given directory into a C file that executes all given tests if linked with SQLite. The SQLite amalgamation version[5] is a concatenation of all the source files required to embed SQLite. After our modifications to the test suite setup (see Section 4.3.2) we are left with 6 test directories, and each consists of 1 to 355 `.test` files.

Second, the TH3 test suite itself is configurable. In order to avoid confusion of SQLite and TH3 configurations we will refer to the latter as TH3 *configs*. There are 25 different TH3 configs in our modified setup and each defines a set of different

[4]https://www.sqlite.org/th3/
[5]https://www.sqlite.org/amalgamation.html

properties (P). The previously mentioned `.test` files declare a set of `REQUIRED` (R) and `DISALLOWED` (D) properties and during execution these tests are skipped for a property ϕ if $\phi \in P \wedge \phi \in D$ or if $\phi \notin P \wedge \phi \in R$ and executed otherwise.

Last, are the configurations of SQLite itself. Since it is infeasible to analyze all possible configurations and because we had no thorough FM, we decided to focus on a subset of 23 features, the *focus features*. We then created four different groups of configurations for these features:

Feature-wise Each configuration consists of one enabled focus feature and as little as possible other features, which can be required according to the FM. This leaves us with 23 different feature-wise configurations.

Pair-wise These configurations have been generated with the SPLCA tool[6]. The basic idea is to generate all possible pairs for the focus features and generate configurations, which cover multiple pairs at the same time. SPLCA has generated 11 pair-wise configurations.

Random We generated 50 different configurations by mapping randomly generated binary numbers between 0 and 2^{23} to a configuration. Duplicate and invalid configurations according to the FM have already been discarded during the generation process.

Allyes We have already mentioned the allyes configuration in previous sections. For this configuration we simply enabled all 23 focus features.

4.3.2 Test Setup Modifications and Restrictions

After establishing the general concept of the `TH3` test suite we now talk about the details for the modifications and restrictions that were applied. It is very important to note that the numbers in the following section are comparisons in execution times between our meta product from variability encoding and our meta product after integrating the performance measuring functions. We decided to compare these two because the meta product itself has already a very different performance compared to the software variant that is tested. We will talk about this further in Section 4.3.6.

During our testing approach we noticed that one of the 26 TH3 configs never executes any test cases. This configuration is responsible for testing a DOS related filename scheme and is not compatible with our UNIX setup, so we excluded it. We also excluded two `.test` files because one contains an array declaration that is loaded with over 100 `#ifdefs` across nearly 3000 lines of code and the second test file performs time and date tests for which we cannot verify the behavior preservation between variant and meta product. For us to verify behavior preservation, we require a binary test result *success* or *failure*. Some tests however deviate from this scheme by utilizing information that is not constant for different test executions, e.g. current system time or memory consumption.

[6] http://martinfjohansen.com/splcatool/

Next, the TH3 test suite originally consists of 10 folders that contain `.test` files. We have excluded the *stress* folder since it contains tests with a very high run time. These stress tests can verify robustness by loading a corrupted database, simulate a power-failure in the middle of a transaction and more. After excluding the stress folder we ran into SQLite out of memory errors during execution of our meta product in two directories. We decided to create partitions for these two folders and thus reduce the amount of `.test` files per folder. One directory was partitioned once and the other one had to be partitioned twice. We further excluded 3 more folders which only consist of 1-2 `.test` files each because their execution time is too low (<2 ms) to be useful. In summary we excluded four folders and created three new folders to balance out the test load.

Last, to generate compatible test scenarios across the different SQLite configurations we excluded all `.test` files that are not executed in all our SQLite configurations for a given combination of TH3 config & test directory and added them to a list of shared tests. With 12 test directories and 25 TH3 configs this equals 300 different shared test lists. We reason that comparing execution times from testing different SQLite configurations does not make sense when both configurations perform different test cases.

We revalidated the results after applying these restrictions and filtered one more folder where the execution time was very low (<2 ms), this is mostly because no test cases or just a few quick test cases remained. We also had to remove two folders which produced a lot of TH3 test suite errors. In summary we are left with 6 test directories, 25 TH3 configs and 85 SQLite configurations, as presented in Figure 4.5. All of these three can be combined in any arbitrary way and thus there are 12750 different test scenarios.

4.3.3 Statistics for the Measuring Process

First, we start by presenting various statistics about our approach before the next sections illustrate our results from analyzing the performance of SQLite.

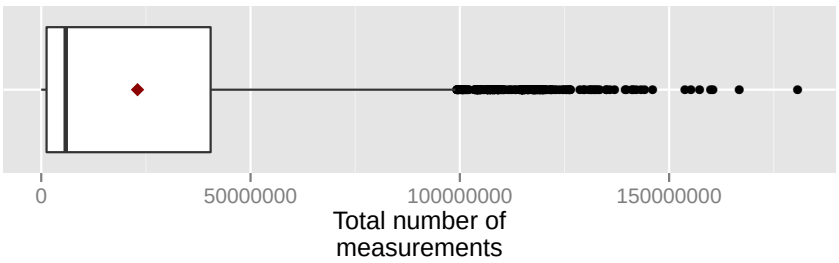

Figure 4.6: Amount of function calls to our measuring functions in SQLite

The injected function `perf_before` tracks the amounts of measurements that have been started for each context. We will now take a look at the total amount of measurements that have been performed across all of the 12750 different test scenarios. Figure 4.6 shows that these numbers vary greatly and take on anywhere between 180 to 180712831 measurements. The red dot resembles the mean amount of measurements at 23005422. This is something that has to be kept in mind, we already

saw that the Elevator product line was influenced greatly by the overhead the `BASE` feature.

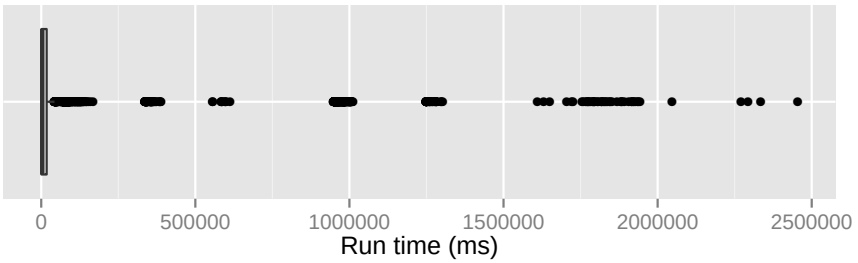

Figure 4.7: Runtime distribution for performance measuring in SQLite

Second, we will look at the distribution of the execution times for performance measuring across all test scenarios, seen in Figure 4.7. The average time is just over 55 seconds and the longest run took about 41 minutes. This time does not include the overhead for the data and time stamp management that we subtract during the execution.

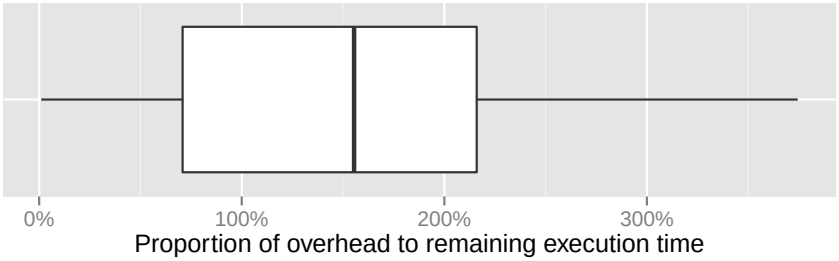

Figure 4.8: Percentage of overhead compared to the previous execution time in SQLite

Figure 4.8 shows the percentage of internally calculated overhead vs the remaining execution time of the TH3 test code. As we already saw in the previous paragraphs the performance functions are called up to hundred million times and this fact is reflected in Figure 4.8. In the majority of cases the execution of a test scenario needs more time for the measurements then for the execution of the test code itself.

4.3.4 Performance Results

In order to answer **RQ2** we have gathered data about feature interactions for every test scenario and generated different figures to show how the execution times are distributed for the different groups of Feature-Interaction Degrees (FIDs).

The maximum amount of FID across all test scenarios is 6 and a concrete example for this case can be seen in Figure 4.9. This interaction occurred through a combination of nested `#ifdefs` and function calls inside `#ifdefs`. However, Figure 4.10 shows that a FID of 3 or more interactions take up a very insignificant proportion of the total execution time. On the other side interactions of degree 2 have a wide margin of impact on the total execution time.

feature interaction stack
VDBE_PROFILE
!SQLITE_OMIT_SUBQUERY
!SQLITE_OMIT_OR_OPTIMIZATION
SQLITE_OMIT_FOREIGN_KEY
!SQLITE_COVERAGE_TEST
!SQLITE_OMIT_EXPLAIN
SQLITE_ENABLE_FTS4 || SQLITE_ENABLE_FTS3

Figure 4.9: Example feature interaction degree of 6 in SQLite

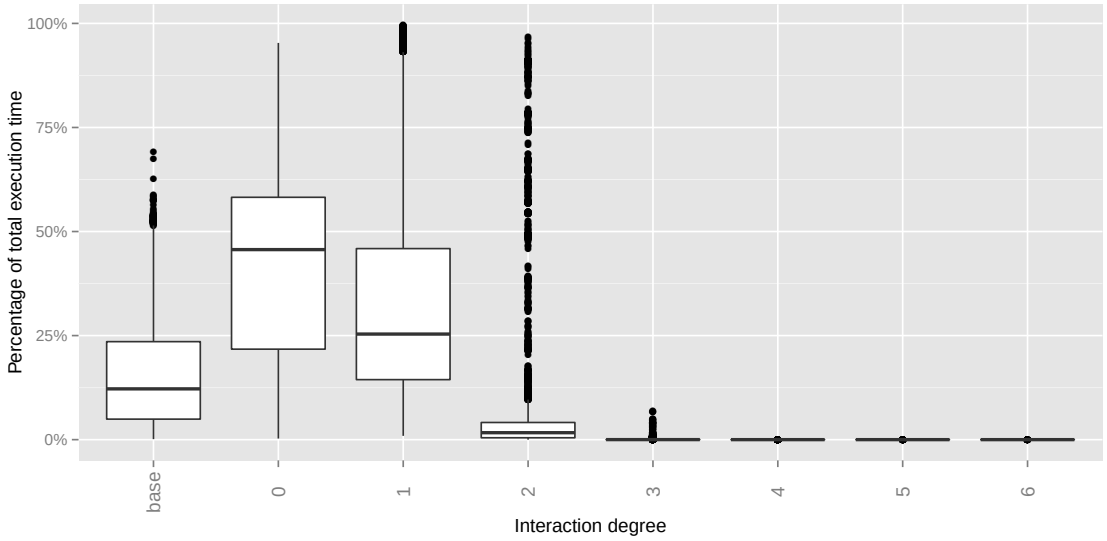

Figure 4.10: Execution time distributed among different degrees of feature interactions

Grouping up the previous data by our four different configuration groups in Figure 4.11 shows that the configurations have a great effect on the distribution. The *allyes* configuration group consists of one configuration and is very stable across all TH3 configurations and test directories. Most of the run time is spent in context without any interactions (FID of 0), followed by one interaction (FID of 1) and the `BASE` code. The `BASE` code is not part of any FID and is categorized separately.

This changes drastically for the *feature-wise* configurations, which are very inconsistent across all FIDs. There are quite a few cases where the vast majority of the execution time is used up by a FID of 2 and this also occurs in the *pair-wise* and *random* configurations. The latter two also have a very identical distribution. Removing 4 specific TH3 configs from the data set removes most outliers for interaction degree 2. Figure 4.12 depicts how this removal affects the *pair-wise* configurations. It seems that the tests related to these configurations focus on testing computationally intensive code in feature interactions of second degree.

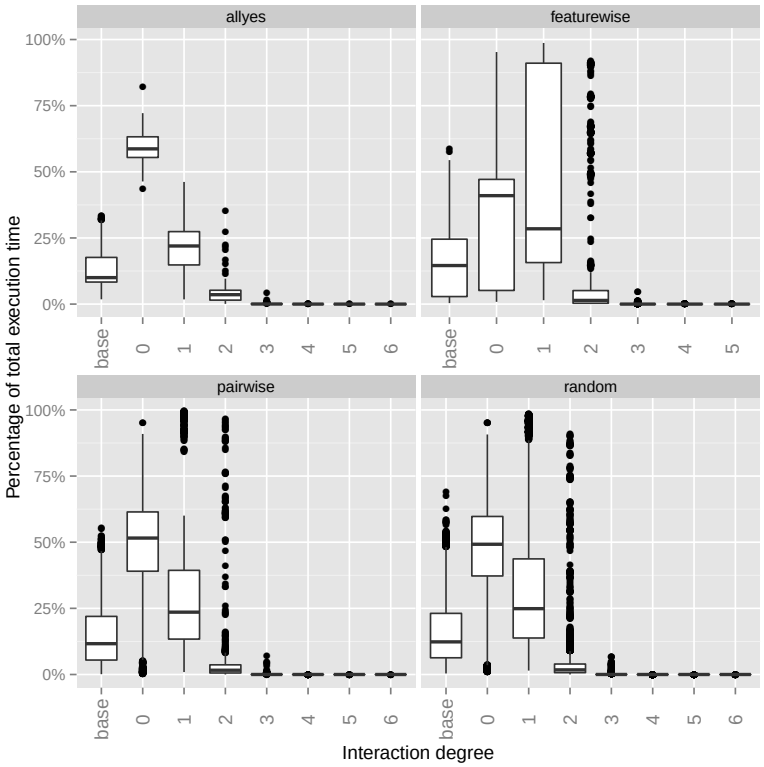

Figure 4.11: Execution time distributed among degrees of feature interactions, grouped by configuration mode

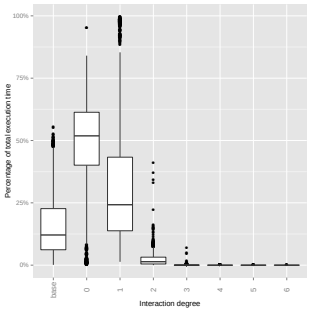

Figure 4.12: Pair-wise distribution after removing 4 TH3 configurations

In the end the data can be grouped in various ways but without in-depth knowledge about SQLite, TH3 and their configurations these data remain observations. To answer **RQ2**, it is interesting that the annotation-free code is on average in third place ranked by proportion of total time consumed. Context without interactions and context with one interaction consume more time for SQLite.

4.3.5 Prediction Results

For this section the data generated during our performance analysis is used to generate performance predictions. Since we already generated detailed numbers for each of our 85 SQLite configurations the next step is to make cross predictions for the different configuration modes. These results will show which configuration group is suited for prediction other configurations, see **RQ4** and also which groups are not compatible and produce predictions that are off the mark.

In order to get more accurate numbers we compared the time of a prediction for configuration ϕ to the time that the performance simulator actually takes when executed in configuration ϕ. In practice the goal has to be to compare the prediction to the actual execution time of that variant, but since our measurement process assigns deallocation calls to the preceding time measurement, this is a compromise. See Chapter 6 for possible improvements.

Figure 4.13 displays the different cross predictions[7]. The PE^2 is computed by taking the average prediction time and comparing it to the actual time that the performance measuring of that configuration needs. The result of a prediction for a configuration group with multiple configurations is then again the average for all individual predictions of that group. A PE of 0 indicates a perfect prediction. The different modes show diverse compatibility to each other. The *allyes* configuration makes the worst predictions, especially *allyes predicts feature-wise*. A reason for this is that the *allyes* configuration enables the feature `SQLITE_NO_SYNC`, which is only enabled in one of the 23 feature-wise configurations. Further these 22 configurations that have disabled `SQLITE_NO_SYNC` all have an execution time of over 3100 ms, whereas the remaining feature-wise configuration has a run time of $\sim$ 90 ms. The *allyes* configuration with about $\sim$170 ms execution time does not have any measurement information for the `SQLITE_NO_SYNC` feature and thus all of its predictions for configurations that include this feature are off the mark.

Figure 4.14 now displays the same information from the previous paragraph but it includes the deviation that is computed for the predictions. Since we have access to the actual results the performance error including deviation is computed as seen in Figure 4.15. Examples for this calculation can be seen in Figure A.1. Especially, example b) highlights how a high variance instantly produces a perfect result which is not applicable if you do not know the expected values. The results for *random predicts* have improved substantially by including the deviation because they have the highest deviation when compared to the average time. As a result they cannot be considered good candidates for predictions. Their high deviation leads to an inaccurate prediction result. See Section A.2 for predictions grouped by test directory.

[7] We excluded one combination of TH3 config and test directory since it can produce a percentage error of $\sim$6 and messes up the y scale of the boxplots

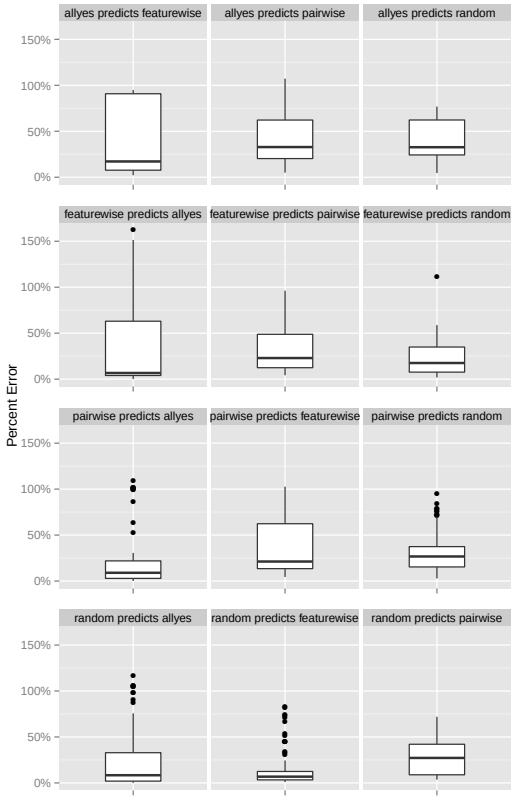

Figure 4.13: Percent error in cross predictions for the different configuration modes

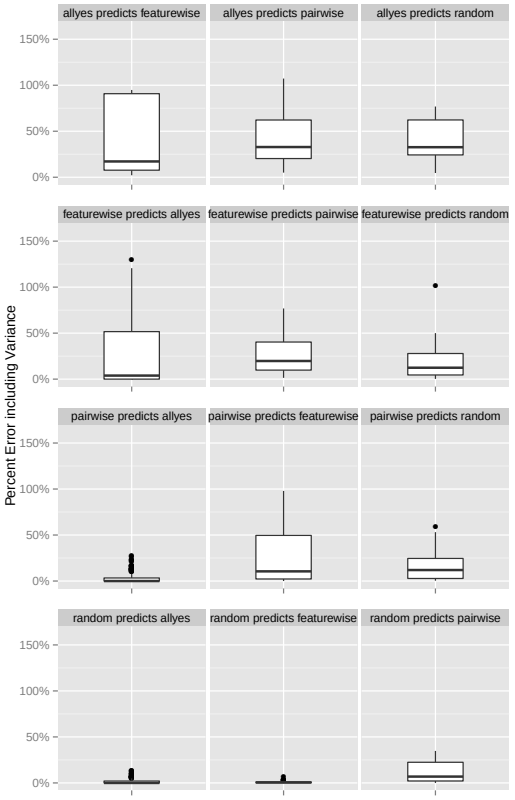

Figure 4.14: Percent error in cross predictions for the different configuration modes including deviation

4.3.6 Comparing Execution Times

Section 3.8 already mentioned that the performance can vary between the meta product (without performance measurements) and the derivable variants. Figure 4.16 shows the PE when comparing the execution times. Overall the times are relatively close but there are some outliers which deviate by nearly 90%. We reason that these differences vary between different case studies and depend on the way that the conditional preprocessor directives are utilized. The closer this overall number is the better the prediction results. As a consequence, we suggest avoiding case studies, where the meta product performance deviates a lot from the software variants.

$$PEV(a, v, r) = \begin{cases} PE(a + v), & \text{if } a + v < r \\ PE(a - v), & \text{if } a - v > r \\ 0, & \text{otherwise, since } r \in]a - v, a + v[\end{cases}$$

where:

PE = function to compute PE
a = predicted average
d = predicted deviation
r = actual result

Figure 4.15: Taking deviation into consideration for calculation of PE

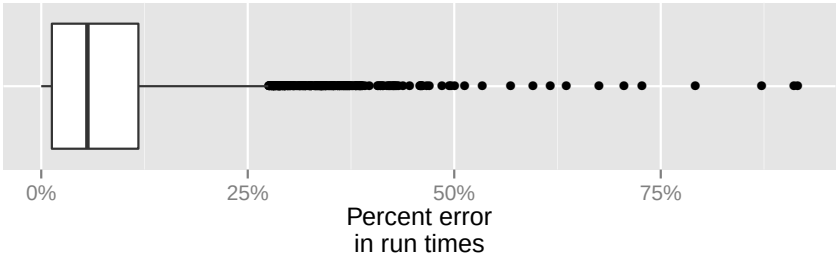

Figure 4.16: Comparing execution times: meta product vs variant

5. Conclusion

The amounf of derivable variants for a configurable systems scales exponentially with its number of configuration options. Thus they pose to be a tough problem to solve for analysis and verification tools. In this thesis we present the combination of performance measuring and variability encoding in order to analyze the performance of Software Product Lines (SPLs).

Our case studies show that the results vary. For the Elevator case study we are able to provide very accurate results by analyzing just one configuration in our simulator. Elevator helps us to characterize the properties that represent a good target SPL for our approach:

- No mutually-exclusive conditional preprocessor directives.
- Feature Model (FM) with very few restrictions, which allows us to select all features and gather a lot of information from a single execution.
- The different features do not have any grave performance-altering data dependencies.

The second case study, SQLite, shows that our approach is applicable even to complex systems. SQLite in its amalgamation version has 93 configurable features spread over 180.000 lines of code [sql]. **RQ2** posed the question of time distribution and SQLite has shown, that in contrast to our expectations, the most execution time is used by features with a Feature-Interaction Degree (FID) of zero or one and the shared code `BASE` only comes in third place. On the other hand it has shown that the code inside preprocessor directives with a FID of 2 or more does not have a significant impact on the performance.

As far as performance prediction in SQLite is concerned we received varying results. The *allyes* configuration is the worst configuration for predicting the performance of other configurations. We found out that a certain group of features in the form of `SQLITE_OMIT_SOMETHING` contain a lot of executable routines whenever they are

turned off. Using the randomly generated configuration set yields the best prediction data when taking the deviation into account. However, the random configuration set consists of 50 different configurations which is more than all the other configuration groups combined. Consequently the *random* configuration predictions require the most effort and also have a high variance. Overall the predictions for SQLite vary, but to make final verdict about its compatibility with our performance measuring approach, further knowledge about its configurations and test suite is required.

6. Future Work

Granularity control

The possibilities for future work are manifold. First of all we measure the performance of every preprocessor directive, even if it only contains one negligible calculation. The effort to perform the measurement for such a statement is multiple orders of magnitudes higher than the statement execution itself. This further increases the run time of the performance simulator and as SQLite has shown with an average of 23 million performance measurements the internally calculated overhead is already surpassing the execution time of the remaining code. Adding different metrics for granularity control to avoid unnecessary performance measurements can potentially reduce the overhead without affecting the accuracy of the measurements.

Measurements in separate thread

Second, in order to lessen the effect that these performance measurements have on the remaining execution we advise revisiting their implementation and conduct these measurements in a separate processor thread. Our internal overhead calculations do not take the very last memory deallocation from a `free` call into account, instead it's execution time is assigned to the execution time of the preceding measurement. For features without any Feature-Interaction Degree (FID) this preceding measurement is the measurement for `BASE`. The Elevator case study has already shown that the numbers for the features are very accurate but the numbers for `BASE` are off by about ∼6 ms because it includes 5604 `free` calls from its nested performance measurements.

Deviations in feature times

Third, making accurate predictions requires the execution times for a feature ϕ to be constant across all configurations that include ϕ. We already mentioned data dependencies as a possible example, where the time measured for a feature ϕ is affected by a different feature χ, e.g. χ increases the upper limit of a loop. If this manipulation in χ is part of an unrelated (not nested) preprocessor directive the measurements for ϕ is affected. If a Software Product Line (SPL) utilizes the conditional preprocessor directives in this way, our approach is not able to generate accurate predictions.

Code coverage conf. set

Fourth, in order to measure the whole code base in regards to mutually exclusive preprocessor directives we advise finding a small set of configurations that covers all

parts of the source code. The so called *code coverage* set of configurations should be the best candidate to generate performance data for predictions. The closer this heuristic is to generating the minimal set of configurations the less effort has to be made for performance predictions.

More case studies

Last, in order to properly judge our approach it needs to be applied to more case studies. Liebig et al. showed that SQLite is one of the very few exceptions, in an analysis of 40 SPL, where the code inside the preprocessor directives exceeds 50% of the code base. SQLite is a complex software system and with that degree of variability, we categorize it as a suboptimal target for our performance measuring and prediction approach. However, new case studies have to fulfill certain requirements before they can be utilized. The parsing framework TypeChef has to be able to not only successfully parse these new SPLs, but also to type check them without any errors. Though, providing a detailed Feature Model (FM) is able to solve most of these issues. The next step is for Hercules to generate compilable `C` code, that does not have any active, un-terminated performance measurements at the end of the program execution.

7. Related Work

Our thesis is influenced by prior work from Post et al. about configuration lifting [PS08] and the paper of Siegmund et al. "Family-Based Performance Measurement" [SvRA14]. While configuration lifting is the motivation behind variability encoding, the collaboration with Siegmund et al. has started the idea of performance measuring in the context of `C` Software Product Lines (SPLs). The background for the work of Siegmund et al. are Java SPL and their tool chain is based FeatureHouse[1], a software-artifact composition framework [AKL09]. In contrast to our approach, the variability occurs only on a function level, whereas C Preprocessor (CPP) directives can be used in any arbitrary way. Siegmund et al. apply their approach to 5 different test systems and obtain a prediction accuracy of 98%, on average. They also verify, that their approach requires only fraction of the effort, that a brute-force approach requires. One of the crucial results is that the time saved by using of family-based performance measurements increases with the number of features in a SPL, but also decreases with the amount of configurations, that are required to gain 100% code coverage.

Regarding family-based analysis methods, in general they are an exciting development that displays promising results [TAK+12, SvRA14, LvRK+13, KvRE+12, NKN14]. The exact implementation for these approaches differs from the software domain and analysis context that they are applied to. However, the core concept remains the same: employ the similarities between the different products in order to save analysis efforts, since the shared code base (ideally) takes up the vast majority of the source code.

An empirical study by Ernst et al. analyzes 26 packages with 1.4 million lines of `C` code [EBN02] and shows that conditional preprocessor directives, on average, account for 4% of the total code. Liebig et al. analyzed the preprocessor usage in 40 different SPLs [LAL+10]. They come to the conclusion that variability correlates with the size of the project and the `#ifdef` nesting is used sparsely.

[1] http://www.infosun.fim.uni-passau.de/spl/apel/fh/

Finally, performance related publications can be split up into different categories. First, the machine-learning or data-set training approach[GSKA16, SK01], which benefits immensely from domain knowledge as Grebhan et al. show [GSKA16]. Using this knowledge reduces the training efforts, however, they stress that this knowledge needs to be precise or the results lose accuracy. Second, the measurement-based approach, that is used by Siegmund et al. [SRK+11, SKK+12]. They gather performance data per feature by measuring the influence of each feature and all related *deltas*. Deltas are a way to express the influence that a feature `A` has on another feature `B`. The feature-specific data and appropriate deltas can then be used to calculate the performance for different configurations. Weyuker et al. discuss their experience with performance testing [WV00]. They stress the importance of performance-related testing and argue, that performance is a very important and project-affecting property. Further, they mention that the test case generation or generation strategies for measuring performance is a crucial criteria for all measurement-based approaches.

A. Appendix

A.1 Percent Error calculation example

This is just a brief example that did not fit into Section 4.3.5 for a calculation of the Percent Error (PE) with deviation.

Example a)

$$p = 65.3ms + 7.7ms$$
$$r = 75.0ms$$
$$PEV(65.3, 7.7, 75.0) = PE(73.0, 75) = \frac{|73.0 - 75|}{75} = 2.66\%$$

Example b)

$$p = 29.3ms + 19.0ms$$
$$r = 47.4ms$$
$$PEV(29.3, 19.0, 47.4) = 0\%$$

Figure A.1: Example calculations for PE with deviation

A.2 Prediction results continued

The following Figure A.2 and Figure A.3 present the results from Section 4.3.5 for the cross predictions grouped by TH3 test directory.

A.3 Prediction result table excerpt

Table A.1 is an excerpt of data gained from the prediction process. The **IDs** range from 0 to 299 because the initial numbering included all 12 TH3 test directories and together with its 25 configs we generated 300 different test scenarios. The unusable results were omitted afterwards.

Since we did not only predict the performance, but also measure the performance of all configurations, we are able to generate a lot more data about the differences between our prediction and the result. E.g., **MPSFD** displays the impact of the

difference in measured times for the shared features between the prediction Feature-Time Hashmap (FTH) and the result FTH as a proportion of the total difference between the predicted time and the actual time. The figures in previous sections about performance predictions for SQLite have been generated from this table.

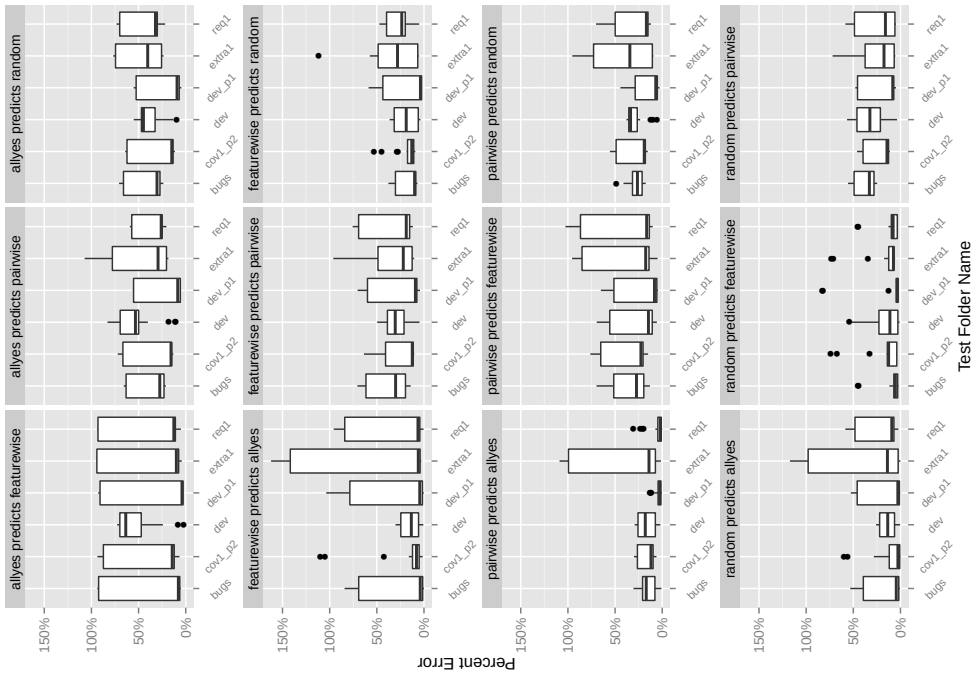

Figure A.2: Percent error in cross predictions for the different configuration modes

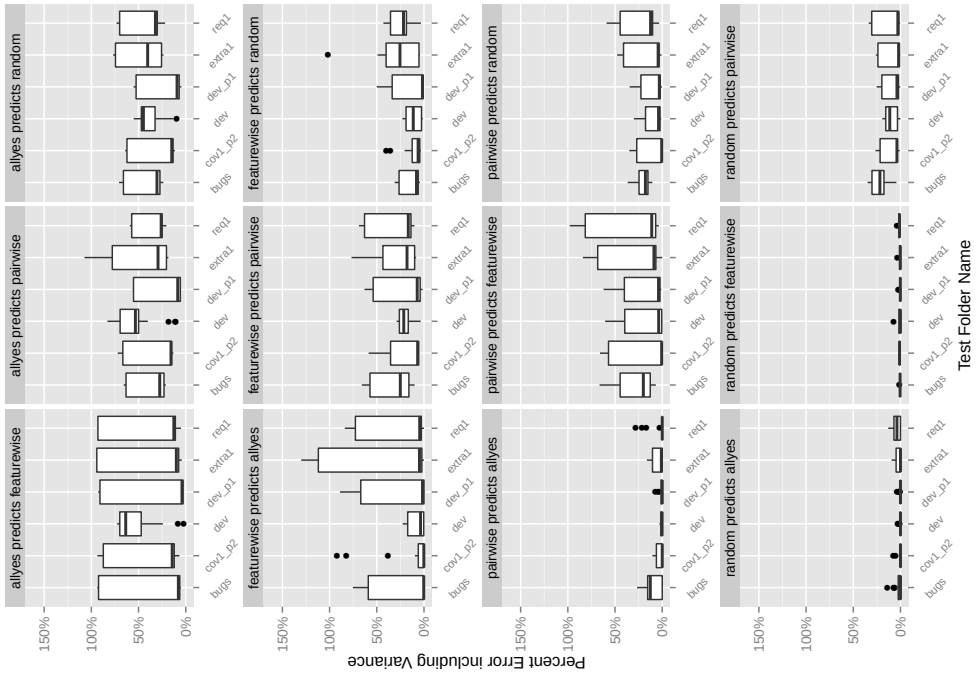

Figure A.3: Percent error in cross predictions for the different configuration modes including deviation

ID	IM	PM	PE	PED	DP	MPTP	MPTR	MPSFD	MPSFDD
10	pairwise	allyes	0.207184	0.168329	0.04901	0.014898	0.001458	-0.193745	-0.176822
11	pairwise	allyes	0.177823	0.13293	0.054603	0.014655	0.001605	-0.164773	-0.145056
12	pairwise	allyes	0.089712	0	0.114893	0.01423	0.000844	-0.076326	-0.083381
235	featurewise	random	0.176358	0.159393	0.019849	0.173136	0.006296	0.039298	0.034812
236	featurewise	random	0.198365	0.186149	0.016944	0.174804	0.006151	0.038666	0.033707
237	featurewise	random	0.473009	0.430184	0.035791	0.028087	0.000009	0.500204	0.460544
24	random	featurewise	0.444728	0.00032	0.421064	0.00008	0.018624	0.463425	0.009282
75	random	featurewise	0.090581	0.008933	0.226134	0.000094	0.001686	0.089256	0.014736
76	random	featurewise	0.037834	0.005807	0.049937	0	0	0.037833	0.010379

ID: test scenario ID from 0-299; **IM**: input mode; **PM**: predict mode; **PE**: percent error; **PED**: percent error incl. deviation; **DP**: deviation percentage; **MPTP**: mean percentage of time only in prediction; **MPTR**: mean percentage of time only in result; **MPSFD**: mean percentage of shared feature difference; **MPSFDD**: mean percentage of shared feature difference incl. deviation;

Table A.1: Excerpt from the prediction data

Bibliography

[AKL09] Sven Apel, Christian Kastner, and Christian Lengauer. Featurehouse: Language-independent, automated software composition. In *Proceedings of the 31st International Conference on Software Engineering*, pages 221–231. IEEE Computer Society, 2009. (cited on Page 43)

[AvRW+13] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: Case studies and experiments. *Proc. of ICSE. IEEE*, 2013. (cited on Page 25)

[Bat05] Don Batory. *Feature Models, Grammars, and Propositional Formulas*, pages 7–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. (cited on Page 6)

[CN+99] Paul Clements, Linda Northrop, et al. A framework for software product line practice. *SEI Interactive*, 2(3), 1999. (cited on Page 5)

[EBN02] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of c preprocessor use. *Software Engineering, IEEE Transactions on*, 28(12):1146–1170, 2002. (cited on Page 8 and 43)

[EW11] Martin Erwig and Eric Walkingshaw. The choice calculus: A representation for software variation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(1):6, 2011. (cited on Page 9)

[GSKA16] Alexander Grebhahn, Norbert Siegmund, Harald Köstler, and Sven Apel. Performance prediction of multigrid-solver configurations. In *Software for Exascale Computing-SPPEXA 2013-2015*, pages 69–88. Springer, 2016. (cited on Page 44)

[Hen08] Thomas A Henzinger. Two challenges in embedded systems design: predictability and robustness. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 366(1881):3727–3736, 2008. (cited on Page 1)

[KA13] Christian Kästner and Sven Apel. Feature-oriented software development. In *Generative and Transformational Techniques in Software Engineering IV*, pages 346–382. Springer Berlin Heidelberg, 2013. (cited on Page 6)

[KCH+90] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990. (cited on Page 6)

[KKHL10] 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*, pages 25–32. ACM, 2010. (cited on Page 9, 12, and 13)

[KvRE+12] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward variability-aware testing. 2012. (cited on Page 1 and 43)

[LAL+10] Jörg Liebig, Sven Apel, Christian Lengauer, C Kastner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 105–114. IEEE, 2010. (cited on Page 43)

[LJG+15] Jörg Liebig, Andreas Janker, Florian Garbe, Sven Apel, and Christian Lengauer. Morpheus: Variability-aware refactoring in the wild. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 380–391. IEEE Press, 2015. (cited on Page 9 and 30)

[LKA11] 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*, pages 191–202. ACM, 2011. (cited on Page 8)

[LvRK+13] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable analysis of variable software. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 81–91. ACM, 2013. (cited on Page 1, 9, 12, and 43)

[MWC09] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. Sat-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*, pages 231–240. Carnegie Mellon University, 2009. (cited on Page 7)

[MWCC08] Marcilio Mendonca, Andrzej Wasowski, Krzysztof Czarnecki, and Donald Cowan. Efficient compilation techniques for large scale feature models. In *Proceedings of the 7th international conference on Generative programming and component engineering*, pages 13–22. ACM, 2008. (cited on Page 7)

[NKN14] Hung Viet Nguyen, Christian Kästner, and Tien N Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 907–918. ACM, 2014. (cited on Page 43)

[Pad09] Yoann Padioleau. Parsing c/c++ code without pre-processing. In *Compiler Construction*, pages 109–125. Springer, 2009. (cited on Page 8)

[PR01] Malte Plath and Mark Ryan. Feature integration using a feature construct. *Science of Computer Programming*, 41(1):53–84, 2001. (cited on Page 25)

[PS08] Hendrik Post and Carsten Sinz. Configuration lifting: Verification meets software configuration. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 347–350. IEEE, 2008. (cited on Page 1, 2, 10, and 43)

[SC92] Henry Spencer and Geoff Collyer. # ifdef considered harmful, or portability experience with c news. 1992. (cited on Page 8)

[SK01] Martin Shepperd and Gada Kadoda. Using simulation to evaluate prediction techniques [for software]. In *Software Metrics Symposium, 2001. METRICS 2001. Proceedings. Seventh International*, pages 349–359. IEEE, 2001. (cited on Page 44)

[SKK+12] Norbert Siegmund, Sergiy S Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 167–177. IEEE, 2012. (cited on Page 44)

[sql] The SQLite amalgamation. https://www.sqlite.org/amalgamation.html. Accessed: 2017-03-14. (cited on Page 39)

[SRK+11] Norbert Siegmund, Marko Rosenmuller, Christian Kastner, Paolo G Giarrusso, Sven Apel, and Sergiy S Kolesnikov. Scalable prediction of non-functional properties in software product lines. In *Software Product Line Conference (SPLC), 2011 15th International*, pages 160–169. IEEE, 2011. (cited on Page 1 and 44)

[SvRA14] Norbert Siegmund, Alexander von Rhein, and Sven Apel. Family-based performance measurement. *ACM SIGPLAN Notices*, 49(3):95–104, 2014. (cited on Page 25 and 43)

[TAK+12] Thomas Thüm, Sven Apel, Christian Kästner, Martin Kuhlemann, Ina Schaefer, and Gunter Saake. Analysis strategies for software product lines. *School of Computer Science, University of Magdeburg, Germany, Tech. Rep. FIN-004-2012*, 2012. (cited on Page 1 and 43)

[TBK09] Thomas Thum, Don Batory, and Christian Kastner. Reasoning about edits to feature models. In *Proceedings of the 31st International Conference on Software Engineering*, pages 254–264. IEEE Computer Society, 2009. (cited on Page 7)

[VB03] László Vidács and Árpád Beszédes. Opening up the c/c++ preprocessor black box. In *Proceedings of the Eight Symposium on Programming Languages and Software Tools (SPLST)*, 2003. (cited on Page 8)

[vR16] Alexander von Rhein. *Analysis Strategies for Configurable Systems*. PhD thesis, Universität Passau, 2016. (cited on Page 2, 11, 12, and 30)

[WV00] Elaine J Weyuker and Filippos I Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *IEEE transactions on software engineering*, 26(12):1147–1156, 2000. (cited on Page 1 and 44)

[Zav99] Pamela Zave. Faq sheet on feature interaction. *Link:* http://www.research.att.com/~pamela/faq.html, 1999. (cited on Page 6)

Eidesstattliche Erklärung:

Hiermit versichere ich an Eides statt, dass ich diese Masterarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Name

Passau, den 15. März 2017