Feature Interactions in Configurable Software Systems

Sergiy Kolesnikov

February 4, 2019

Dissertation zur Erlangung des Doktorgrades
der Naturwissenschaften (Dr. rer. nat.)
eingereicht an der Fakultät für Informatik und Mathematik
der Universität Passau

Dissertation submitted to
the Department of Informatics and Mathematics of
the University of Passau
in Partial Fulfillment of Obtaining
the Degree of a Doctor in the Domain of Science

Betreuer / Advisor:
Prof. Dr. Sven Apel

Externer Gutachter / External Examiner:
Prof. Vittorio Cortellessa, Ph.D.

Die Dissertation wurde eingereicht am / The dissertation was submitted on:
2019-02-06

Unterschrift / Signature:

Sergiy Kolesnikov

Abstract

Software has become an important part of our life. Therefore, the number of different applications scenarios and user requirements of software systems grows rapidly. To satisfy these requirements, software vendors build configurable software systems that can be tailored to diverse needs without rebuilding them from scratch, which reduces costs and development time.

Despite considerable advances in software engineering, which allow building high-quality configurable software systems, some challenges remain. One of these challenges is the *feature interaction problem* that arises when parts (features), from which a configurable system is composed, interact in unexpected ways, and inadvertently change the behavior or quality attributes (such as performance) of the system.

The goal of this dissertation is to systematically study the nature of feature interactions, their causes, their influence on performance of configurable systems, and, based on empirical results, suggest ways of improving techniques for detecting and predicting feature interactions.

More specifically, we compared and evaluated different strategies for the analysis of configurable software systems. The results of our evaluation complement empirical data from previous work about how different analysis strategies for configurable software systems compare with respect to different aspects, such as performance. These results shall be used to develop effective and scalable techniques and tools for analysis of configurable software including feature-interaction detection and prediction techniques and tools.

Technically, we used a machine-learning technique to quantify the influence of feature interactions on performance of real-world configurable systems. We studied the characteristics of interactions that have the largest influence on performance and found that interactions among few features have higher influence than interactions among many features. With a growing number of interacting features, the influence of the corresponding interactions decreases consistently. This implies that interactions involving multiple features can be ignored in practice because of their marginal influence on performance. We also investigated the causes of the interactions and were able to identify several patterns that link these interactions to the architecture of the systems: For example, we found that if a data processing system consisted of multiple features that processed the same data in sequence then these features interacted. The identified patterns can help to anticipate performance interactions already at an early development stage when a system's architecture is designed.

Furthermore, considering that control-flow interactions (observable at the level of control flow among features) are easier to detect than performance interactions (externally observable through measuring performance of different combinations of features), we conducted a case study on two configurable systems. In this case study, we investigated a possible relation among control-flow feature interactions and performance feature interactions. We also discussed how this relation can be exploited by interaction detection and performance prediction techniques to make them more time efficient and precise. Our case study on two real-world configurable systems revealed that a relation indeed exists, and we were able to show how it can be used to reduce the search space of possibly existing performance interactions. The study can serve as a blueprint for further studies that can rely on our conceptual framework for investigating relations among external and internal interactions.

Overall, the contribution of this dissertation consists of scientific and technical insights, practical tool implementations, empirical evaluations, and case studies that advance the current state of research in the area of feature interactions in configurable software systems. In particular, we provide insights into the causes of feature interactions and their influence on performance of real-world configurable systems (e.g., interaction patterns, decreasing influence of interactions with growing number of involved features). Our results also suggest ways of improving techniques for detecting and predicting feature interactions (e.g., ignoring interactions among multiple features, reducing the search space based on relations among interactions).

Acknowledgements

This dissertation would not have been possible without the support of many people. No matter how large or small the involvement of each of them was, it is the whole and not parts that made this dissertation possible.

I want to thank my wife Katja for her relentless support, words of wisdom at the right moment, and always being there for me. Our son Luka came into this world when I just started writing down this dissertation and he gave me a lot of helpful input, not directly related to the topic, but nevertheless very positive and inspiring. I am also grateful to my parents who laid a foundation for all my achievements in life.

I would like to express my gratitude to my advisor Sven Apel. I got to know Sven as a student and have been always admiring his openness to students and colleagues, his capacity to absorb unbelievable amounts of information on a vast number of topics, and the ability to effectively share the knowledge that he generated from that information. Sven's personality was one of the main reasons why I gladly accepted the offer to become part of his research group. His support, patience, and advice are invaluable.

I would like to thank Christian Kästner whose special way of providing feedback helped me to learn writing research papers. I also would like to thank Vittorio Cortellessa who kindly accepted the invitation to be my external examiner, knowing how much work it would mean.

Furthermore, I want to thank my colleagues and students who were involved in different activities regarding this dissertation: Alexander Denk, Alexander Grebhahn, Claus Hunsen, Christian Kaltenecker, Olaf Leßenich, Jörg Liebig, Judith Roth, Semah Senkaya, Norbert Siegmund, Stefan Sobernig, and Andreas Stahlbauer. I also want to thank Eva Reichhart for her organizational and administrative support. And a special thanks goes to Alexander von Rhein with whom we shared an office for a long time and had a lot of interesting discussions on a variety of topics.

Contents

List of Figures **vii**

List of Tables **ix**

1 Introduction **1**

1.1 Motivation . . . 1
1.2 Goals . . . 5
1.3 Contributions and Key Results . . . 6
1.4 Outline . . . 7

2 Background **9**

2.1 Configurable Software Systems . . . 9
2.1.1 Notes on Terminology . . . 11
2.1.2 Variability Modeling . . . 12
2.1.3 Variability Implementation . . . 14
2.2 Feature Interactions . . . 21
2.3 Analysis of Configurable Systems . . . 26
2.4 Performance Prediction of Configurable Systems . . . 29
2.4.1 Performance-Influence Models . . . 29
2.4.2 Multiple Linear Regression . . . 32
2.4.3 Learning Performance-Influence Models . . . 33
2.5 Summary . . . 35

3 A Comparison of Analysis Strategies **37**
3.1 Product-Line Type Checking . . . 38
3.1.1 Examples of Type Errors . . . 39
3.1.2 Type Checking Product Lines . . . 40
3.1.3 Product-based Strategy . . . 40
3.1.4 Feature-based Strategy . . . 41
3.1.5 Family-based Strategy . . . 43
3.1.6 Summary . . . 44
3.2 Motivation and Hypotheses . . . 44
3.3 Empirical Evaluation . . . 45
3.3.1 Subject Systems . . . 47
3.3.2 Measurement Procedure . . . 47
3.3.3 Results . . . 48
3.3.4 Discussion . . . 52
3.4 Threats to Validity . . . 56
3.5 Related Work . . . 56
3.6 Summary . . . 57

4 Tradeoffs in Modeling Performance **59**
4.1 Motivation and Research Questions . . . 61
4.1.1 Use Cases of Influence Models . . . 63
4.1.2 Tradeoffs in Machine Learning . . . 65
4.1.3 Research Questions . . . 66
4.2 Empirical Study . . . 67
4.2.1 Learning Performance-Influence Models . . . 67
4.2.2 Measurement Procedure . . . 67
4.2.2.1 Measuring Model Properties . . . 67
4.2.2.2 Measuring Tradeoffs . . . 68
4.2.3 Subject Systems and Experimental Setup . . . 69
4.2.4 Results . . . 71
4.2.5 Discussion . . . 71
4.3 Threats to Validity . . . 81
4.4 Related Work . . . 82
4.5 Summary . . . 84

5 On the Relation of Feature Interactions **87**
5.1 Visibility of Feature Interactions . . . 88
5.1.1 External Feature Interactions . . . 89
5.1.2 Internal Feature Interactions . . . 90
5.2 Examples of Relations among Interactions . . . 92
5.2.1 Control-Flow Interactions (Internal, Operational) . . . 92

5.2.2 Performance Interactions (External, Non-functional) . . . 94
5.2.3 Relating Control-Flow and Performance Interactions . . . 95
5.3 Research Questions and Conceptual Framework . . . 96
5.3.1 Research Method . . . 96
5.3.2 Identifying Control-Flow Interactions . . . 97
5.3.3 Identifying Performance Interactions . . . 98
5.3.4 Relating Control-Flow and Performance Interactions . . . 98
5.3.5 Predicting Performance Interactions . . . 99
5.3.6 Subject Systems . . . 100
5.4 Results . . . 103
5.4.1 Performance Interactions . . . 103
5.4.2 Control-Flow Interactions . . . 106
5.4.3 Relating Interactions . . . 108
5.4.4 Predicting Performance Interactions . . . 112
5.5 Discussion . . . 113
5.6 Threats to Validity . . . 116
5.7 Related Work . . . 117
5.8 Summary . . . 118

6 Conclusion 119

Bibliography 123

Appendix 134
A.1 Influences of Interactions . . . 134
A.2 Materials Presented to the Interviewees . . . 138

List of Figures

1.1 Summary of our contributions. . . . 6

2.1 Variability diagram of the example file archiver. . . . 13
2.2 Prototypical implementation of the file archiver using feature-oriented programming. . . . 18
2.3 A configuration with DEFLATE, Encrypt, Hash, and File enabled. 19
2.4 Prototypical implementation of the file archiver using preprocessor. . . . 20
2.5 Compression ratios for the example. . . . 23
2.6 Example of a trivial performance-influence model. . . . 31
2.7 Example of an alternative performance-influence model. . . . 31
2.8 Example of an alternative performance-influence model with interaction terms. . . . 32

3.1 Examples of type errors. . . . 38
3.2 Feature model of the example product line. . . . 39
3.3 Steps of product-line type checking. . . . 40
3.4 Syntax tree of the example product line. . . . 43
3.5 Error message of the family-based type checker. . . . 44
3.6 Example of a variable type hierarchy. . . . 46
3.7 Extended feature model of the example product line. . . . 47
3.8 Type-checking times for each subject system. . . . 51

4.1 Two examples of performance-influence models. . . . 62
4.2 Example tradeoff curves and corresponding AUC values. . . . 68
4.3 Time–error tradeoff. . . . 72

4.4 Time–size dependency. 73
4.5 Size–error tradeoff.p 74
4.6 Shares of interactions of different size in performance-influence models and their influence on the prediction error. 77

5.1 Interactions in the audio streaming system. 93
5.2 mbedTLS: counts of features in control-flow interactions. 107
5.3 SQLite: counts of features in control-flow interactions. 107

List of Tables

2.1 Terms in configurable systems and software product lines domains. 11
2.2 Configuration options of the example file archiver. 12
2.3 Benchmark results for the example DBMS. 30

3.1 Conceptual comparison of the three type-checking strategies. . . 44
3.2 Overview of the subject systems. 48
3.3 Measurement results for each subject system and type-checking strategy. 50
3.4 Break-even points of the superiority of the family-based strategy for the subject systems. 53

4.1 Subject systems. 69

5.1 A lists of interacting features for the example. 95
5.2 Performance interactions and their influences on performance of the systems in seconds. 104
5.3 Performance interactions, the number of the control-flow interactions related to them, and the mean value of the corresponding Jaccard indexes. 109
5.4 Control-flow interactions, the number of the related performance interactions, and the mean value of the corresponding Jaccard indexes. Control-flow interactions without related performance interactions are not listed. 111
5.5 Precision and recall for the item set predictor. 114

A.1 A list of the most influential configuration options and interactions. 134

CHAPTER 1

Introduction

1.1 Motivation

Configurable software systems. In the time of the ongoing digital revolution, software plays an essential role in more and more areas of economy and society. Transportation and communication infrastructure, manufacturing and financial sector, all from national defense and security to entertainment industry, depend on software. As a result, we observe an explosive growth of different requirements, regulations, and application scenarios that software vendors have to consider while developing their software products. A software product with a single functionality is rarely enough nowadays to satisfy all customers. Similarly as a coffee machine that makes only coffee lungo is rarely enough, because there are also those delicious[1] espresso, cappuccino, late macchiato. That is, a customer will probably be unwilling to buy multiple coffee machines to make each of these beverages. To satisfy different customer needs, vendors introduce new *configuration options* (a.k.a. *features*) for their coffee machines, such as the ability to make multiple coffee beverages, milk frother, auto-brew mode, and adjustable grinder and coffee dosage. Based on these configuration options, vendors provide a line of diverse products from which a customer can choose: from simple budget variants with a few features to high-end variants with all bells and whistles included. In other words, coffee machine vendors design their machines as *configurable systems*, with the goal of satisfying diverse customer requirements, considering different application scenarios. Software engineering

[1] To my taste, at least.

companies are not much different in this respect: They design *configurable software systems*, pursuing the same goals.

Software product lines. Starting in 1960's, gaining momentum in 1990's, the continuous research and development in the domain of configurable systems culminated in the appearance of a new engineering discipline: *software product line engineering.* This discipline is concerned with all aspects of producing configurable software systems; it describes how scientific and technological knowledge and methods can be systematically applied in the production [PBvD05]. Adoption of the software product-line approach promises different benefits among which [Ape+13c, p. 8]:

Tailor-made software. A product-line approach facilitates tailoring products to individual customers, such that a set of differently tailored products is produced.

Low production costs. Since each tailored product is not developed from scratch, but rather composed from reusable parts, which were developed upfront, or is reconfigured to fit new requirements, the costs per product per customer are reduced.

Reduced development time. Composing products from reusable parts results in reduced development time. Even if a customer requests new functionality, building an new reusable part (with the requested functionality) for an existing well-designed product is faster than building the product from scratch.

Improved product quality. Standardized and reusable parts, which are composed to products, can be checked and tested systematically in different products and use cases, leading to more stable and reliable parts and, as a consequence, more stable and reliable products.

There are multiple cases of successful adoption of the software product line approach in industry [PBvD05, p. 413; SPL17]. Among the adopters are companies such as Boeing, Hewlett-Packard, LG, Philips, Bosch, and Siemens. These companies report on cost reduction, shorter development time, and quality improvement achieved by introducing the software product line approach. For example, Bosch successfully applied a software product line approach to develop a reference architecture for driver assistance systems (e.g., parking pilot) that was configurable, easy to integrate, highly performing, and therefore could be used in different contexts for different car manufacturers. As a result, Bosch was able to deliver high quality, low-price products that were adjusted to the individual needs of the car manufacturers.

Challenges of configurable systems. Along with the evident benefits of configurability and its successful adoption in industry, new problems have emerged, such as the *exponential explosion of configuration spaces* [Ape+13c, p. 247] and the *feature-interaction problem* [Bru05]. The *configuration space* is a set of all configurations of a configurable system. For example, by equipping one model of a coffee machine with the Milk Frother feature and another model not, a vendor creates two configurations of the coffee machine. A customer owing one of these machines can additionally adjust the grinder by setting the adjustment wheel to, say, 10 different positions, which results in 10 different coffee machine configurations. So, in total, we have 2 (models) times 10 (grinder positions) equals 20 different configurations. An additional configuration option, such as auto-brew mode, will double the number of configurations and increase it to 40 (4 models times 10 grinder positions). That is, the growth rate of the configurations' number is exponential in the number of configuration options. If the vendor will keep adding configuration options, the configuration space will explode. For example, the LINUX[2] kernel is one of the most prominent configurable software systems that has approximately 13 000 configuration options [Pas+15]. Even if we assume that from these configuration options only 300 can be switched on or off independent from each other, the size of the configuration space will still be in the order of 2^{300}. For comparison, the number of all atoms in the currently known, observable universe is estimated to be approximately 2^{265}. This unimaginable number of LINUX configurations makes it difficult for its developers to establish guarantees about properties of the system: Will all configuration successfully boot without a crash? Will any configuration fit in some predefined amount of the main memory? These questions cannot be answered in feasible time by examining every single LINUX configuration in the system's huge configuration space one by one. Legacy techniques for testing, type checking, static analysis, verification, or program analysis in general are well established for individual system configurations, but they do not scale to configurable systems. To cope with the problem of exponential explosion of configuration spaces new, *variability-aware* techniques for analysis of configurable systems were proposed [Lie15; Tar13; vRhe16].

A key problem of the analyses of configurable systems is that configuration options (or features, in terms of software product-line engineering) may interact with each other in an unexpected ways. A *feature interaction* occurs when the behavior of one feature is influenced by the presence of another feature (or a set of other features) [Bow+89]. Due to a feature interaction, a feature that functions well in one configuration may fail when combined with other features in a different configuration. Often, the interaction cannot be deduced

[2] https://www.kernel.org/

easily from the behaviors of the individual features involved, which hinders the understanding of the system's behavior as a whole.

The feature interaction problem struck the telecommunication industry in the late 1980 [Bow+89]. A classic example from this domain is the inadvertent interaction between the Call Forwarding and Call Waiting features of a telephony system [Cal+03]. Call Forwarding forwards calls to another phone-number if the recipient's line is busy. Call Waiting notifies the recipient on a busy line about incoming calls and allows the recipient to switch among the calls. Both features function correctly if only one of them is enabled in a given configuration. Enabling both features in one configuration leads to a feature interaction, such that only one of the features can fulfill its function: a call on a busy line can be either forwarded to another recipient or put in a waiting queue, but not both at the same time. This is an example of a feature interaction that infringes a *functional property* of the system, because the system fails to fulfill a specified *functionality*.

The next example describes a feature interaction in a configurable database management system that has two features: Compression and Encryption [Sie12, p. 22]. As the names suggest, Compression compresses the data stored in the database, and Encryption encrypts the data. While measuring the performance of configurations with only Compression or only Encryption enabled, we determine that the configurations can store a given amount of data in 20 seconds or 40 seconds respectively. Based on this measurements, we may assume that enabling both features will result in a configuration that stores the same amount of data in 60 seconds (20 seconds for compression plus 40 seconds for encryption). However, the actual measurement is 50 seconds (10 seconds less than expected). This is an effect of a feature interaction between Compression and Encryption: encrypting compressed data is faster than uncompressed, because the compressed data is smaller. This interaction does not prevent the features from fulfilling their functions, but it changes the qualitative aspect of the system (the speed of storing data) in an unexpected way. In other words, this interactions influence a *non-functional property* of the system [Sie12, p. 13].

To establish guaranties about functional and non-functional properties of a configurable system, it is imperative to detect feature interactions that influence these properties. A key problem, however, is that any combination of features in a configurable system may potentially interact. A system with n features has $\frac{n(n-1)}{2}$ pairs of features that may potentially interact; and the number of potential interactions with $k > 2$ features is $\binom{n}{k}$. For a reasonably large configurable system, it is practically impossible to inspect all possible combination of features to detect feature interactions, because the combinations are exponentially many in the number of features. So, the exponential explosion

problem strikes here again.

1.2 Goals

For the feature interaction problem to be solved, the nature of feature interactions needs to be explored systematically and comprehensively: Which feature interactions occur in real-world systems? What causes these interactions and what influence on the properties of a configurable system (such as performance) do the interactions have? Are all of the interactions difficult to detect or can certain kinds of them be detected easier than the others? Are there any relations between different kinds of interactions in a configurable system? Can we predict feature interactions of one kind based on the knowledge about feature interactions of another kind? These are the questions that we addressed in this dissertation. Answering them will contribute to solving the feature interaction problem that, in turn, will allow to produce more reliable, maintainable, and performant configurable systems.

In our dissertation, we first explore the configuration space explosion problem, which is the key obstacle to making analyses of configurable systems efficient and scalable. The goal is to compare different *variability-aware analysis strategies* regarding their time efficiency, scalability, and completeness of the analysis; and identify a strategy that is superior to other strategies under discussion. This is an important prerequisite for our work, because efficient and scalable variability-aware analysis strategies are the foundation for many analyses of configurable systems including feature-interaction detection and techniques that we applied in our thesis.

Then we focus on a particular type of analysis of configurable systems, namely the detection of feature interactions. To detect feature-interactions influencing non-functional properties of configurable systems, *performance-influence models* can be used. But, for performance-influence models to be generally applicable and useful in practice, they should have low prediction error, produce models of a comprehensible size, and it should be possible to construct these models in feasible time. Our goal here is to evaluate performance-influence models with respect to these properties and determine if accurate and also simple performance-influence models can be learned in feasible time. Furthermore, in the course of the evaluation of the feature models, we aim at gaining insights in the nature of feature interactions by investigating the cause of the influence of feature interactions on performance of the subject systems.

Finally, we discuss different kinds of feature interactions and classify them into *external* and *internal feature interactions*. Since internal feature interactions are generally easier to detect than external ones, we investigate a possible

relation between these to kinds of interactions with the ultimate goal to use this relation for predicting external interactions based on internal ones.

1.3 Contributions and Key Results

Guided by the goals described in Section 1.2, our work resulted in contributions that we summarize in Figure 1.1. The figure depicts different areas of research addressed in this dissertation, how they facilitate each other, and what our main contributions to these areas are. Most contributions were published in journals, conferences, or workshops that count among top-tier venues in our field of research (such as, SoSyM, GPCE, and FOSD). In particular, our contributions are the following:

- We implemented a type checker for feature-oriented, JAVA-based product lines that supports family-based, feature-based, and product-based analysis strategies. This is the first time that all three analysis strategies have been integrated within a single tool.
- We evaluated and compared the three type-checking strategies on a set of 12 subject product lines with regard to different aspects, such as the ability to detect different kinds of type errors and the quality of the provided information about errors. Most notably, we found that the family-based strategy is the most efficient strategy for all of them.
- Using a machine-learning technique based on multiple linear regression,

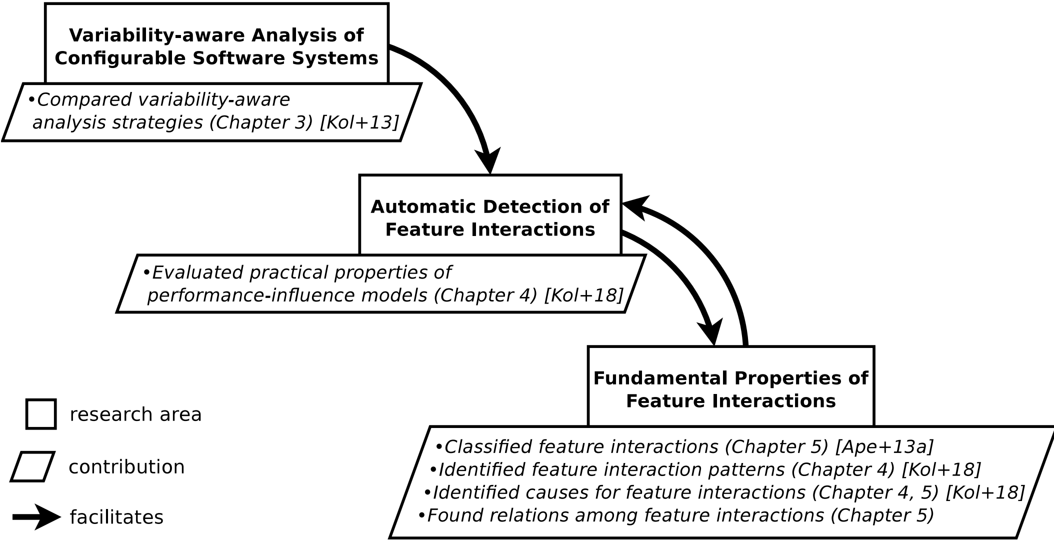


Figure 1.1: Research areas to which we contributed, their dependencies, and a summary of our main contributions to these areas.

we systematically studied and analyzed the tradeoffs among prediction error, model size, and computation time of performance-influence models for 10 real-world highly configurable systems from different domains. We found that the low influences of these tradeoffs allow us to build models that fit typical use cases, such as program comprehension and performance prediction.

- We investigated the causes for the configuration options and their interactions having the observed influences on the systems' performance, and we identified reoccurring patterns in the systems' architecture and in the dependencies among configuration options that explain these influences.
- We found that with the growing number of interacting features the influence of the corresponding interactions consistently decreased. That is, interactions involving multiple features can be ignored in practice because of their marginal influence on performance.
- We discussed and classified feature interactions according to their visibility in two classes: internal and external interactions. Using this classification, we defined a relation between internal and external interactions based on the features these interactions concern.
- We defined a conceptual framework for exploring the relation between internal and external interactions and, in a first case study of this kind, using two real-world highly configurable subject systems, we explored and confirmed the relation between internal and external feature interactions.
- By exploiting the found relation, performance prediction techniques based on machine learning and relying on sampling for building a training dataset can make sampling more focused on the configurations that potentially include interacting features, which may improve their prediction accuracy.

1.4 Outline

Chapter 2 lays the foundation for this dissertation by introducing basic terms, central concepts and definitions.

Chapter 3 presents an empirical study that compares product-based, feature-based, and family-based strategies for analysis of configurable systems with respect to the analysis time and completeness. It is imperative that we collect as much empirical evidence as possible to ensure the applicability of these strategies in variability-aware program-analysis tools, which we also apply in this dissertation.

Chapter 4 presents an empirical study on the tradeoffs in modeling performance of configurable systems. Performance-influence models describe the performance behavior of a configurable system as a whole and quantify

the influence of individual configuration options and their interactions on the system's performance. The results describe which interactions have the highest influence on performance and show how decisions about architecture of a system may lead to feature interactions.

Chapter 5 presents a case study on the relation of external and internal feature interactions in two configurable systems. The case study identifies a relation between control-flow and performance interactions and suggests how this relation can be used to improve performance prediction techniques for configurable systems.

Chapter 6 summarizes the dissertation.

CHAPTER 2

Background

This chapter shares material with the following publication: S. Apel et al. "Exploring Feature Interactions in the Wild: The New Feature-interaction Challenge". In: *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 2013, pp. 1–8 [Ape+13a]

In this chapter we lay the foundation for our dissertation by introducing the basic terms and central concepts used in the further discussion. We begin by introducing the notions of configurable systems, variability modeling, and implementation in Section 2.1; then we discuss feature interactions and analysis of configurable systems in general in Sections 2.2 and 2.3; finally we describe a technique for predicting performance of configurable systems using automatic feature interaction detection in Section 2.4. This chapter is not meant as a full featured literature review, therefore, for a deeper discussion of the topics, we refer the reader to the corresponding literature: configurable software systems [Ape+13c; CE00]; feature interactions [Ape+13d; Bow+89; JZ98]; analysis of configurable systems [Thü+14; vRhe16]; performance prediction of configurable systems [Guo+13; Sie+12; Zha+15].

2.1 Configurable Software Systems

A *configurable software system* or simply a *configurable system* is a software system that is explicitly built with the goal of satisfying different requirements regarding its functionality, non-functional properties (such as, performance or maintenance effort), operational environment (e.g., software and hardware

platform), or user group (e.g., novice of experienced users). This goal is achieved by introducing *variability* in the software system (Section 2.1.3), that is by implementing several alternatives of the same functionality or by making a functionality optional.

For example, several different algorithms with the same functionality but different resources requirements can be implemented in a configurable system. One algorithm may be CPU intensive and another main memory intensive. Depending on the CPU power and amount of the main memory provided by a particular operational environment, a more suitable algorithm implementation can be selected by a user. By selecting a concrete algorithm implementation the user configures the system and *resolves* the variability. During this *configuration process*, the user resolves the variability in the configurable software system based on the functional, non-functional, and operation environment requirements. The result of the configuration process is a *configuration* of the system. Therefore, a configurable software system can also be seen as a set of configurations—a *configuration space*—that satisfy different sets of requirements.

In Section 2.1.3 we discuss different implementation mechanisms of the software variability, but on a more abstract level variability in a configurable system can be described by a set of configuration options. A *configuration option* (a.k.a. *feature*) is a characteristic or end-user-visible behavior of a software system [Ape+13c, p. 14]. It describes what is in common and what is different among the configurations of a configurable system. By assigning a concrete value to a configuration option a user decides on the corresponding characteristic of behavior and resolves the variability that is denoted by the configuration option. For example, for the CPU intensive algorithm from our example we can have a configuration option CPU-intensive. By *enabling* (i.e., assigning value 1) or *disabling* (i.e., assigning value 0) (also *setting*/*unsetting*, *selecting*/*deselecting*) this configuration option the user can prescribe if the CPU intensive algorithm must be used by the system or not. This type of configuration options is called *binary* because it can be assigned only two values (enabled or disabled). Another type of configuration options is *numeric* configuration options that can be assigned a value from a predefined numeric interval. For example, for the memory intensive algorithm we can have a numeric configuration option MaxMem that defines the maximum amount of the main memory (e.g., in megabytes) that can be used by the algorithm.

Similarly to Siegmund et al. [Sie+15], we formally model a configurable system as a pair $(\mathcal{O}, \mathcal{C})$, where $\mathcal{O}$ is a set of configuration options and $\mathcal{C}$ is a configuration space. A configuration $c \in \mathcal{C}$ is then a function $c : \mathcal{O} \to \mathbb{R}$ that maps every configuration option to a real value. If a binary configuration

option $o \in \mathcal{O}$ is enabled in a configuration $c \in \mathcal{C}$, then $c(o) = 1$, and if it is disabled then $c(o) = 0$. A a numeric configuration option is mapped by c to a corresponding (user-)selected numeric value.

2.1.1 Notes on Terminology

The notions *configurable system*, *configuration*, and *configuration option* are closely related to the notions *product line*, *variant*, and *feature* that come from the domain of feature-oriented software product-line engineering [Ape+13c; BSR04; CE00; Kan+90]. Software product-line engineering incorporates not only the technical but also an economic perspective on the development of configurable software systems. It promises to increase the quality and to reduce the costs of developing a configurable software system [Big98; PBvD05]. In Table 2.1 we summarize the terms from the configurable systems and software product-line domains that denote equivalent concepts.

Sometimes, the term *highly configurable system* is used to describe certain configurable systems [Tar13; vRhe+15]. There is no a precise definition for a *highly* configurable system. Informally, we describe a configurable software system as highly configurable if an analysis task cannot be accomplished in feasible time by dealing with all configurations separately, one by one. For example, we cannot run a test suite on every configuration of the LINUX kernel, because even the exact number of the configurations can only be approximated and cannot be exactly computed in feasible time [Lie15, p. 61], not speaking about testing every single configuration. So, the difference between a configurable and highly configurable system depends on the concrete task that a user wants to accomplish with the system (for example, determine the fastest configuration, or run a test suit against each configuration) and how large the corresponding configuration space is. For the approaches and techniques that we discuss in this dissertation, the difference between configurable and highly configurable systems is irrelevant.

Table 2.1: Terms describing equivalent concepts in the configurable systems and software product lines domains.

Configurable Software System	Software Product Line
configuration	variant/product
configuration process	feature selection
configuration option	feature

Table 2.2: Configuration options of the example file archiver.

Option	Description
DEFLATE	Very fast compression algorithm with low compression rate
LZMA	Very slow compression algorithm with high compression rate
BZIP2	A compression algorithm representing a compromise between DEFLATE and LZMA regarding speed and compression rate
Hash	Cryptographic hash function implementation
Encrypt	Encryption functionality
File	File output for the data
Stdout	The standard output for the data

2.1.2 Variability Modeling

To introduce basic concepts and to demonstrate how they fit together, we use a configurable file archiver program as an example. The main purpose of the file archiver program is to combine multiple files in one file and to compress this file using one of the three alternative compression methods (DEFLATE, BZIP2, LZMA). The file archiver also provides two optional functionalities: Encrypt and Hash. Encrypt is used to encrypt data and Hash is used to ensure the integrity of compressed data. Hash is also used by Encrypt to implement authentication and integrity of the encrypted data. The compressed data can be written to a file using the File functionality, to the standard output using the Stdout functionality, or to both outputs simultaneously. The functionalities can be enabled or disabled using corresponding configuration options that we summarize in Table 2.2

Listing the configuration options of a configurable system can give an overview of the provided functionalities, but it is not enough to describe its variability, because there may be dependencies among configuration options. For example, the file archiver can apply only one compression method to data at a time. That is, the three compression methods constitute an alternative group of configuration options. Furthermore, we can enable or disable the encryption functionality at will. That is, Encrypt is optional, but depends on the Hash functionality (for the authentication and integrity-check implementation). Finally, File and Stdout are two ways of outputting the compressed data. Both of them can be used at the same time.

To formally describe the dependencies among configuration options of a configurable system *variability models* are used. A variability model can be represented by a *variability diagram* as shown in Figure 2.1. A variability diagram is a treelike structure with the root node representing a configurable

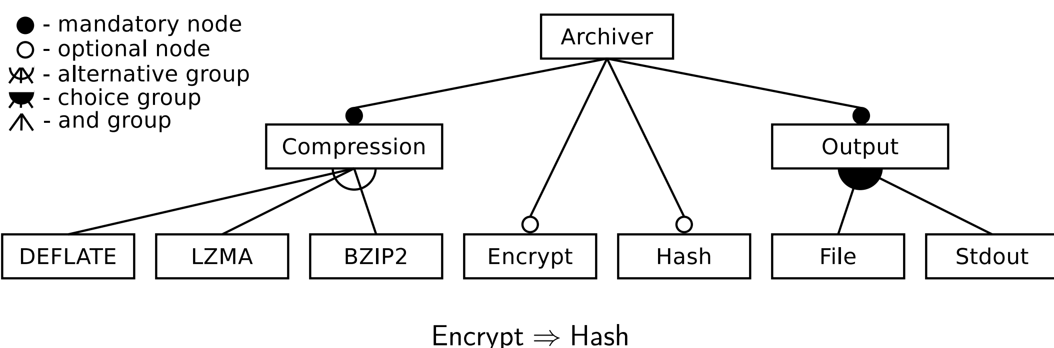


Figure 2.1: Variability diagram of the example file archiver.

system, leaf nodes representing the configuration options of the system and the inner nodes used for grouping. There is no a standard for variability diagrams and there are variants in which inner nodes can represent configuration options too. Additional graphical elements on the nodes are used to express dependencies among configuration options. A half circle under a parent node (e.g., Copmpression) denotes that the child nodes constitute an *alternative group*: exactly one of the configuration option in the group can be set for a given configuration. A filled half circle under a parent node denotes that the child nodes constitute a *choice group*: several or all of the configuration options in the group can be set for a given configuration. No circles under a parent node denote that the child nodes constitute an *and group*: all of the configuration options in the group must be set for a given configuration (except for the options represented by optional nodes). A circle above a node denotes an *optional node*, that is, the configuration option can be freely included into or excluded from a given configuration. A filled circle above a node denotes a *mandatory node*, that is, the configuration option that must be included in all configurations in which the configuration option denoted by the parent node is also selected. For example, for a configuration of the file archiver a compression method and at least one of the output methods must be included.

Not all dependencies among configuration options can be expressed through the tree structure of a variability diagram. Such *cross-tree constraints* can be expressed through propositional formulas that are supplemented to a variability diagram. For example, the propositional formula Encrypt ⇒ Hash in Figure 2.1 denotes that functionality of Encrypt depends on Hash and that Encrypt can be enabled only in combination with Hash.

Similar to the cross-tree constraints, any other dependency among configuration options in a variability diagram can be expressed as a propositional formula by interpreting feature names as propositional variables and expressing dependencies among features through logical operators [Ape+13c, p. 30]. For example,

the choice group of the output methods is formalized as $\mathsf{Output} \Leftrightarrow (\mathsf{File} \vee \mathsf{Stdout})$. The complete semantic of the variability diagram in Figure 2.1 is formalized by the following propositional formula:

$$
\begin{aligned}
&\mathsf{Program}\\
&\wedge (\mathsf{Compression} \Leftrightarrow \mathsf{DEFLATE} \vee \mathsf{LZMA} \vee \mathsf{BZIP2})\\
&\wedge \neg(\mathsf{DEFLATE} \wedge \mathsf{LZMA} \wedge \mathsf{BZIP2})\\
&\wedge \neg(\mathsf{DEFLATE} \wedge \mathsf{LZMA})\\
&\wedge \neg(\mathsf{DEFLATE} \wedge \mathsf{BZIP2})\\
&\wedge \neg(\mathsf{LZMA} \wedge \mathsf{BZIP2})\\
&\wedge (\mathsf{Program} \Leftarrow \mathsf{Encrypt})\\
&\wedge (\mathsf{Output} \Leftrightarrow \mathsf{File} \vee \mathsf{Stdout})
\end{aligned}
$$

Variability diagrams are a convenient graphical way to communicate a variability model of a configurable software system to a user or any other stakeholder. Propositional formulas may be less convenient for comprehension, but they let us apply mathematical procedures and automated tools such as SAT solvers to reason about and analyze variability models [BSR10; Kas+09; TBK09].

2.1.3 Variability Implementation

To increase generalizability of the results of our dissertation, we use subject system that apply different techniques to implement variability. Next, we describe these techniques and their properties.

There are multiple techniques to implement variability in a software system, but everyone of these techniques deals with the following three questions [Ape+13c, pp. 50–54]:

1. How variability is represented in the code?
2. What technique is used to implement and resolve variability?
3. At what time (in the software life-cycle) the variability is resolved?

How is variability represented in the code? There are two approaches for representing variability in the code that are widely used in practice: *annotation-based* [KAK08; Lie+10] and *composition-based* [AL08; Bat04; Kic+97] approaches.

In the annotation-based approach the code parts that are responsible for a particular functionality are annotated (marked) in a way prescribed by

the annotation language. An example of such annotation language is the C preprocessor. C-preprocessor language is used to make annotations directly in the source code of a program. When the variability is resolved (for example, during the compilation of the code, in the case of the C preprocessor) the annotated code is either included or not in the configuration, depending on if the corresponding configuration options is enabled or disabled. The annotation-based approach are rather popular because of their ease of use and availability for many programming languages. Its downsides are reduced code readability, lack of modularity, and error proneness [Ape+13c, p. 53].

In the composition-based approach, different functionalities are implemented in physically separated composable units. One of the widely used techniques that apply the composition-based approach is *software frameworks*. Different functionalities are implemented as plugins that can be combined to form different configurations of the system. Composition-based approaches are more complex in their implementation and may have a steeper learning curve for a programmer, but on the positive side they promote modularity of the code [Ape+13c, p. 53].

What techniques are used to implement and resolve variability? There are two techniques for implementing and resolving variability (that is, for creating a configuration of a configurable system): *language-based* [ALS08; BSR04; Kic+97; Pre97] and *tool-based approaches* [GS03; KA09; SGC07].

The language-based approach uses mechanisms provided by the programming language to implement and to resolve variability. A classical example of this approach is run-time configuration options that use variables and control flow statements to implement variable code. Based on the value of the variable a control flow statement prescribes at run-time which code blocks (implementing units of functionality) are executed and which not. Since the same language mechanisms are used to manage variability and to implement functionality of the software, the language-based approach makes it easy for developers to implement and reason about the configurable system. On the downside, the code responsible for variability blends in with the code responsible for the functionality of the system, which may complicate debugging, refactoring, and other maintenance activities.

The tool-based approach uses specialized external tools to manage variability. Tools that are widely used in practice are preprocessors and build systems [Ape+13c, pp. 107–127]. Preprocessors, such C preprocessor CPP, provide directives to remove code fragments before compilation based on user defined conditions. Build systems orchestrate all activities concerned with producing an executable of a software program, such as running generators

for code and documentation, executing tests, and compiling code. A build system is usually configured by build script that describes which code artifacts should be compiled and which not. This way, code artifacts (e.g., source files) that implement different functionalities can be included into or excluded from compilation, which effectively implements a variability management mechanism. Using build systems clearly separates the code of the software from the variability management infrastructure. In general, tool-based approaches introduce new tools that must be mastered and consistently used by the developers on the everyday basis, which may complicate the development and maintenance processes.

At what time (in the software life-cycle) the variability is resolved? Variability can be resolved at compile time, load time, or run time [Ape+13c, p. 50]. If variability is resolved at compile time then the functionalities associated with disabled configuration options are excluded from the code before compilation. An example of compile-time variability are techniques based on preprocessors. Compile-time variability has an advantage of being more memory efficient, because the unused functionality is not included in the compiled binary. The decisions taken at the compile time about which configuration options are enabled or disabled cannot be revised without recompilation. Therefore, if a user wants to add or remove some functionality it has to stop the running instance of the program, recompile it, and run the program again. This may be unacceptable for use cases for which program downtimes cannot be tolerated.

In the case of the load-time variability, variability is resolved when the program is started (i.e. loaded into the main memory). The configuration process can be implemented using configuration files that store information about enabled and disabled options. When the program is started it reads the configuration file and disables or enables functionalities according to the settings in the file. The compiled binary of the program contains code for *all* configuration options of the configurable system, so that any combination of them can be selected at the load time. Therefore, load-time variability may be less memory efficient than the compile-time variability. Security vulnerabilities existing in the disabled functionality may still pose a threat when using load-time variability, because the corresponding code is still present in the main memory when the program is executed and the vulnarability may be exploited. Nevertheless, load-time variability allows changing configuration of a system without recompilation which can save time. It may also be the only way to reconfigure the system if its source code is not available.

Run-time variability is very similar to the load-time variability. The only difference is that configuration options may be disabled or enabled not only at

load time, but also later during the program's execution. This allows changing program configuration without stopping the execution and prevents downtime.

Prototypical Implementations of the File Archiver Example

To illustrate the discussed variability mechanisms, we developed two prototypical JAVA-based implementations of the file archiver example: feature oriented and preprocessor-based. Both implementations consist of a single class `Archiver` that has four methods: `readFile()` reads data from a file, `process()` processes the data, `writeData()` writes out the processed data, `main()` calls the former three methods in order. There are two variants of the `writeData()`. Each variant of the method implements one of the ways of outputting the processed data (i.e., file, standard output, both). The choice of the output implementations is controlled through configuration options: if File is enabled then the data is written to a file, if Stdout is enabled then the data is written to the standard output. If both options are enabled, the data is written to the both outputs. Method `process()`, which is responsible for compressing the data, has three implementations, because the data can be archived using three alternative compression methods as described in the variability diagram in Figure 2.1. The corresponding configuration options are DEFLATE, LZMA, and BZIP2. Optionally, the compressed data may be encrypted and supplemented with its hash, which is controlled through configuration options Encrypt and Hash.

Feature-oriented implementation. Figure 2.2 illustrates the use of a feature-oriented programming technique that implements each configuration option in a separate *feature module* (further simply module). Each module adds the corresponding functionality (e.g., encryption or hashing) to the `Archiver` class. To create a configuration, the modules are automatically composed in a user-defined order. The compositions of the modules is done according to the *superimposition rules* [AL08] that prescribe how exactly different language units (e.g., classes and methods) from different modules must be combined. The composed code is then compiled to an executable. Figure 2.3 illustrates the code of a configuration in which configuration options DEFLATE, Encrypt, Hash, and File are enabled and features LZMA, BZIP2, Stdout are disabled. That is the configuration uses DEFLATE compression method to compress the data, than it encrypts the data, supplements it with a hash, and writes it out to a file. According to this configuration, the modules that implement the required functionality were composed. Since all methods in the corresponding modules belong to the same class `Archiver`, they are all combined in one class with the same name. The modules corresponding to options DEFLATE, Encrypt,

Feature Base
```
class Archiver {
   void main(Path in, Path out) {
      byte[] inData = readFile(in);
      byte[] outData = process(data);
      writeData(outData, out);
   }
   byte[] readFile(byte[] in) {
      ... // Read data from the file
   }
}
```

Feature Hash
```
class Archiver {
   byte[] process(byte[] data) {
      byte[] tmpData = original(data);
      byte[] digest = hash(tmpData);
      ... // Append the hash to the data
   }
   byte[] hash(byte[] data) {
      ... // Calculate the hash of data
   }
}
```

Feature File
```
class Archiver {
   void writeData(byte[] data, Path out) {
      ... // Write data to the output file
   }
}
```

Feature Encrypt
```
class Archiver {
   byte[] process(byte[] data) {
      byte[] tmpData = original(data);
      byte[] digest = hash(data);
      ... // Encrypt tmpData using digest
   }
}
```

Feature Stdout
```
class Archiver {
   void writeData(byte[] data, Path out) {
      ... // Write data to the stdout
   }
}
```

Feature DEFLATE (LZMA, BZIP2)
```
class Archiver {
   byte[] process(byte[] data) {
      ... // Compress data using DEFLATE
   }
}
```

Figure 2.2: Prototypical implementation of the file archiver using feature-oriented programming.

Hash *refine* `process` method, which is responsible for processing the input data. *Refining a method* means that each module implements only that part of the method for which it is responsible. So, these implementations must be superimposed to implement the complete processing of the input data. The order in which the implementation are superimposed corresponds to the order of the module composition. Technically, the superimposition of the methods is implemented using a special method call `original()` that is used to call the next `process`-implementation in the order of the composition.

With respect to the classification of the variability mechanisms, the feature-oriented programming technique is composition-based, since it uses separate modules to implement different units of functionality. It is also a language-based technique, because it requires an extension of the Java language with additional keywords (e.g., `original()` method calls) and with an ability to store multiple classes with the same name in one package (e.g., multiple `Archiver` classes in different modules). Nevertheless, the technique also requires an external composition tool with the knowledge of the composition

```
class Archiver {
   void main(Path in, Path out) {
      byte[] inData = readFile(in);
      byte[] outData = process_Hash(data);
      writeData_File(outData, out);
   }

   byte[] readFile(byte[] in) {
      ... // Read data from the file
   }

   void writeData_File(byte[] data, Path out) {
      ... // Write data to the output file
   }

   byte[] process_Hash(byte[] data) {
      byte[] tmpData = process_Encrypt(data);
      byte[] digest = hash(tmpData);
      ... // Append the hash to the data
   }

   byte[] hash(byte[] data) {
      ... // Calculate the hash of data
   }

   byte[] process_Encrypt(byte[] data) {
      byte[] tmpData = process_DEFLATE(data);
      byte[] digest = hash(data);
      ... // Encrypt tmpData using digest
   }

   byte[] process_DEFLATE(byte[] data) {
      ... // Compress data using DEFLATE
   }
}
```

Figure 2.3: A configuration with DEFLATE, Encrypt, Hash, and File enabled.

order and superimposition rules for composing the modules. Therefore, this technique also has a tool-based nature. Usually, the modules are composed right before compilation, so the variability is resolved before the compilation, which corresponds to a compile-time variability implementation.

Preprocessor-based implementation. Figure 2.4a illustrates a preprocessor-based technique which uses annotations,[1] such as `#ifdef Hash`, to mark parts of the code implementing different functionalities, that is, belonging to different configuration options. To create an executable of a configuration, the annotated code parts that belong to disabled configuration options are deleted and the remaining code is compiled. Figure 2.4b illustrates the code of a configuration with configuration options DEFLATE, Encrypt, Hash, and File

[1] In this example, C-preprocessor annotations.

```
class Archiver {
   void main(Path in, Path out) {
      byte[] inData = readFile(in);
   #ifdef Hash || Encrypt
      byte[] digest = hash(outData);
   #ifdef Hash
      ... // Append the hash to the data
   #endif
   #ifdef DEFLATE
      ... // Compress data using DEFLATE
   #elif defined LZMA
      ... // Compress data using LZMA
   #elif defined BZIP2
      ... // Compress data using BZIP2
   #endif
   #ifdef Encrypt
      ... // Encrypt data using digest
   #endif
      writeData(outData, out);
   }
   byte[] readFile(byte[] in) {
      ... // Read data from the file
   }
   #ifdef Hash || Encrypt
   byte[] hash(byte[] data) {
      ... // Calculate the hash of data
   }
   #endif
   void writeData(byte[] data, Path out) {
   #ifdef File
      ... // Write data to the output file
   #else
      ... // Write data to the stdout
   #endif
   }
}
```

(a) Prototypical implementation.

```
class Archiver {
   void main(Path in, Path out) {
      byte[] inData = readFile(in);

      byte[] digest = hash(outData);

      ... // Append the hash to the data


      ... // Compress data using DEFLATE






      ... // Encrypt data using digest

      writeData(outData, out);
   }
   byte[] readFile(byte[] in) {
      ... // Read data from the file
   }

   byte[] hash(byte[] data) {
      ... // Calculate the hash of data
   }

   void writeData(byte[] data, Path out) {

      ... // Write data to the output file



   }
}
```

(b) A configuration with DEFLATE, Encrypt, Hash, and File enabled.

Figure 2.4: Prototypical implementation of the file archiver using preprocessor.

enabled. The code that was marked as belonging to these configuration options using annotations is included in this configuration. Configuration options LZMA, BZIP2, and Stdout are disabled in this configuration and, consequently, the corresponding code was removed by the C-preprocessor (preserving newlines for easier error reporting, therefore the gaps in the code). The preprocessed code of the configuration can be now compiled with a C-compiler for producing an executable for the configuration.

The preprocessor-base technique is annotation-based, because it uses C-preprocessor annotations to mark code belonging to different configuration options. Since the technique uses a external tool (preprocessor) to exclude code of disabled features, the technique is also tool-based. The preprocessor is usually ran just before the compilation, so the technique counts to compile-time variability implementation.

2.2 Feature Interactions

In this section we discuss the notion of *feature interaction* that is one of the central concepts in our dissertation. Note that in this section we do not differentiate between features and configuration options and regard these two terms synonymous.

Features are units of functionality that define behavior of a configurable system. But the problem is that if we understand how the individual features behave in isolation we may still not be able to understand how they behave in combination (i.e., in a given configuration), because features can interact with each other. That is we cannot reason about the behavior of the system just as about a sum of behaviors of its features. We must also consider possible feature interactions to establish guaranties about functional and non-functional properties of the configurable system.

Apel et al. [Ape+13c] define feature interaction as follows:

> A *feature interaction* between two or more features is an emergent behavior that cannot be easily deduced from the behaviors associated with the individual features involved.

Although, there are *desired* interactions that are required to facilitate those functionalities of a system that need an interoperation of multiple features. In the file archiver example in Figure 2.1, the interaction between features Hash and Encrypt is a desired one and facilitates the proper encryption of data.

The problematic ones are the *undesired* or *inadvertent feature interactions* that occur when a feature influences the behavior of another feature in an

unexpected way [Ape+13c]. For an example of an inadvertent feature interaction and its influence on the behavior of a configurable system, consider the compression ratio of the file archiver in our example. The compression ratio measures how good an archiving program is in reducing the size of the input data. It is defined as follows:

$$Compression\ Ratio = \frac{Uncompressed\,Size}{Compressed\,Size} \cdot 100$$

The more efficient a compression algorithm the higher compression ration it can achieve.

One could assume that the compression ratio of our archiving program depends on the input data and is determined by the behavior of the selected compression algorithm, for example, DEFLATE or LZMA. In fact, the behavior also depends on the Encryption feature, because encryption can defeat the heuristics used by the compressions algorithms, as we describe next. The heuristic of the DEFLATE algorithm assumes that the input data have repeated occurrences of strings and achieves data compression by replacing these repeated occurrences with references to a dictionary [Sal04, p. 230]. These references are smaller in size than the replaced strings, therefore, the overall size of the data is reduced. The assumption about repeated occurrences of strings holds for most of the data representing text. That is why plain text documents can be compressed with high compression ratio.

Encryption algorithms make data indistinguishable from random data[2] and effectively removes any repeating patterns in it. Therefore, the assumption of the DEFLATE algorithm about repeating patterns does not hold for encrypted text anymore and it cannot compress the data. In other words, if both features DEFLATE and Encrypt are enabled *and* the input data is encrypted before compression, then the compression ratio will be unexpectedly lower than it is with Encryption disabled.

We demonstrate the influence of this interaction on compression ratio in an experiment with real compression algorithm DEFLATE (implemented in Zip[3] v. 3.0) and encryption algorithm AES256 (implemented in GPG[4] v. 1.4.20). As input data we used a file with 170 KB of plain text in UTF-8 encoding.[5] We processed the input data using Zip and PGP three times: 1) we only compressed the data; 2) we only encrypted the data; 3) we encrypted and then compressed the encrypted data.

[2] As long as the decryption key is not available.
[3] http://www.info-zip.org/
[4] https://www.gnupg.org/
[5] `https://www.gutenberg.org/files/11/11-0.txt`

In Figure 2.5, we illustrate the compression ratio (in %) for the three peaces of data that we got after processing: the compressed data (C), the encrypted data (E), and the encrypted and then compressed (E·C) data. Simply compressing the input data resulted in a compression ratio of 278 %, (the C bar). That is, the size of the compressed data is almost 3 times smaller than that of the uncompressed data. Simply encrypting the input data resulted in a compression ration of 100 % (the E bar). That is, the encrypted data has the same size as the input data. Encrypting and compressing the input data resulted in a compression ration of 100 %, too, which may be surprising at the first glance, because we would expect a higher compression ratio. But, as we discussed above, this is due to an interaction between compression and encryption.

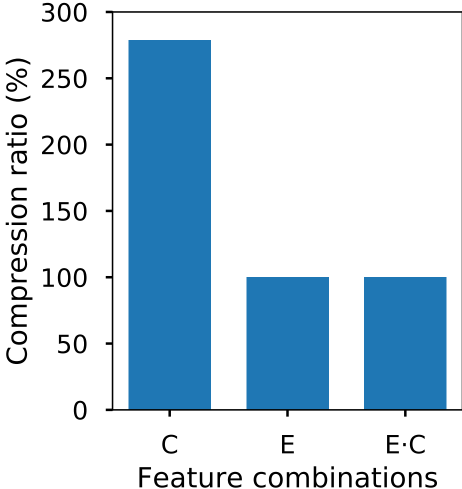


Figure 2.5: Compression ratios for encrypted data (E), compressed data (C), encrypted and then compressed data (E·C). An interaction of encryption and compression algorithms leads to an unwanted result: the compression ratio is the same as of the uncompressed data.

To resolve this feature interaction, we need to switch the order in which the input data are processed: we first have to compress the data and than encrypt it. The resolution of this interaction will result in the compression ratio of 278 % for the compressed and then encrypted data. What we have done right now is solving the feature-interaction problem for the file archiver, which consists in detecting and resolving (inadvertent) feature interactions among features [Ape+13c, p. 218].

One of the obstacles complicating the solving of the feature-interaction

problem in general is the number of potential feature interactions in a configurable system that may be exponential in the number of features as we discuss in the next section.

Formalizing Feature Interactions

Specifications. Talking about feature interactions requires talking about specifications, because to identify a feature interaction, one needs to be able to identify deviating and inadvertent properties or behaviors. A *specification* defines the expected behavior when features are combined. If a feature combination $f_1 \bullet \ldots \bullet f_n$, with '$\bullet$' denoting composition, satisfies specification ϕ, we write:

$$f_1 \bullet \ldots \bullet f_n \models \phi \tag{2.1}$$

Specifications may be concerned with individual features (stating their expectations and provisions; in this case, ϕ in (2.1) would be combined of multiple smaller specifications, associated with individual features) or combinations of features (e.g., stating properties that all feature combinations must exhibit) [Ape+13b]. In the remainder, we abstract over this difference, for simplicity.

Furthermore, depending on the kind of feature interaction, specifications are formulated more or less explicitly. The requirement that a system does not crash with a segmentation fault is very general and is often implicitly assumed; the same applies to other properties, such as the absence of null-pointer dereferences and data races. However, in other situations, specifications are much more explicitly formulated, using formalisms such as temporal logic or automata, for example, stating that a certain process terminates before another process or that adaptive cruise control does not disable the break system [Dom12].

Specifications are essential for the endeavor to understand feature interactions, as we explain next when discussing two properties along which we classify feature interactions.

Model of Feature Interactions Given a set of features (or configuration options) $\mathcal{F}$, predicate $interact_i^{\phi}(f_1, \ldots, f_n)$ denotes a feature interaction i that occurs in the subset $\{f_1, \ldots, f_n\} \subseteq \mathcal{F}$ of features (or configuration options) with respect to specification ϕ, and that the feature set is *minimal*—removing one feature from the set would deactivate interaction i. Accordingly, the only derivation

rule for *interact* is:

$$\frac{f_1 \bullet \ldots \bullet f_n \not\models \phi \qquad n > 1 \qquad \forall \{g_1, \ldots, g_k\} \subset \{f_1, \ldots, f_n\} \;:\; \neg \mathit{interact}_i^{\phi}(g_1, \ldots, g_k)}{\mathit{interact}_i^{\phi}(f_1, \ldots, f_n)} \tag{2.2}$$

This inference rule applies only to feature sets with two or more features, and it does not rule out that a given feature combination may contain multiple feature interactions of the same or different sizes. This is in line with Siegmund et al., who found that the presence of interactions of larger size is often implies the presence of corresponding interactions of smaller size [Sie$^+$12], which we model as distinct interactions $i_1, \ldots, i_m$.

As a design decision, we model interactions as violations of specifications. If features interact and satisfy a given specification, predicate *interact* does not hold. To model such desired interactions, one can adapt the specification such that it exposes the interaction—a case that we ignore, for simplicity.

Furthermore, our definition of feature interactions is centered around the presence of features, which is natural when reasoning about feature-oriented systems. Work on interaction testing also considers interactions that occur when one feature must be selected and another must be deselected [GC11; YCP06], which we do not consider for simplicity.

Finally, we do not consider constraints among features (e.g., that one feature requires or excludes another feature). Although constrains play an important role in modeling, managing, and analyzing variability [CE00], including them into our model would make it more complicated, but does not add anything substantial to our message. See Garvin et al. [GC11] and Siegmund et al. [Sie$^+$12] for examples of how constraints are incorporated in modeling and detecting feature interactions.

We denote an interaction between n features as $f_1 \cdot \ldots \cdot f_n$.

Size of Feature Interactions The *size* of a feature interaction $f_1 \cdot \ldots \cdot f_n$ is the number n of participating features (or configuration options):

$$\mathit{size}(f_1 \cdot \ldots \cdot f_n) = n \tag{2.3}$$

An interaction between two features is of size two, an interaction between three features is of size three; and so on.[6]

The number of potential interactions in a system can be exponential in the number of its set $\mathcal{F}$ of features (or configuration options):

$$2^{|\mathcal{F}|} - |\mathcal{F}| - 1 \in \mathcal{O}(2^{|\mathcal{F}|}) \tag{2.4}$$

[6]Other researchers also call an interactions among n features an n-way feature interaction [KKB08] or an n-th order interaction [Sie$^+$12].

Fortunately, the situation is not that bad in practice. The number of actual feature interactions that occur is likely to be much lower [CM06; Jay+07; Sie+13]—otherwise feature-based systems would not be practical. Motivated by this assumption, some researchers focus only of interactions between pairs of features, which is much more tractable:

$$\frac{|\mathcal{F}| \cdot (|\mathcal{F}| - 1)}{2} \in \mathcal{O}(|\mathcal{F}|^2) \tag{2.5}$$

Note that Equations 2.4 and 2.5 are approximations in the sense that they ignore that a given set of features may give rise to multiple different interactions of the same size. Still, they illustrate the nature of the feature-interaction problem very well.

A major problem in practice is that, for a given system and specification, it is not obvious which feature interactions *really* occur. Intuitively, the larger the size of a feature interaction, the harder the interaction is to detect. Feature interactions of size two (or three) can be simply detected by creating all pairs (or triples) of features [Ape+13d; CM06; Jay+07] and applying a proper interaction-detection technique (e.g., testing [GC11; Jay+07; Joh+12] or model checking [Ape+11; Ape+13d]). But, this way, one may miss interactions of larger sizes—these can be found *reliably* only by creating all possible feature combinations, which induces, again, an exponential effort.

One practical problem that is complicated by the potentially exponential number of feature interactions is predicting performance of configurable systems. To predict performance of a configurable system it is essential to know the influence of feature interactions in the system on performance (as we showed in the file archiver example). We discuss the problem of finding the influence of feature interactions on performance and predicting performance of configurable systems in Section 2.4 in more detail. But, first, we discuss how the problem of analyzing configurable software systems in general is approached.

2.3 Analysis of Configurable Systems

Among other techniques, we leverage *static program analysis* to analyze and study configurable systems. Next, we introduce this technique and also its different types that we apply in our dissertation.

Static program analysis is used to automatically analyze the behavior of software and, based on the analysis results, to optimize the software's behavior or to ensure its correctness [Wic+95]. The analysis is performed without executing the code of software, hence "static." For example, *type checking* is a lightweight static analysis technique for detecting a certain class of programming

errors (type errors) [Pie02, p. 4]. *Control-flow analysis* is another static analysis technique that determines an order in which parts of a program's source code are executed. This information can then be used for automatic optimization, for example, by identifying parts of code that can be executed in parallel [ASU86, p. 399].

The variability of configurable software introduces a new complexity dimension to the program analysis: a program analysis technique must analyze not a single program but a set of programs, that is, all configurations of a configurable system. A naïve strategy would be to analyze all configurations one by one, but, with a growing number of configuration options, this strategy will encounter the exponential explosion problem [Ape+13c, p. 247]. That is, testing configurations one by one does not scale in general. Therefore, several alternative analysis strategies have been developed that transform traditional analysis techniques into *variability-aware* techniques capable of analyzing configurable software efficiently. In this section, we introduce product-based, feature-based, and family-based strategies for analysis of configurable software.

Analysis Strategies

The main idea behind variability-aware analysis techniques is to incorporate the knowledge about the variability of an analyzed system into the analysis process to make the analysis scale to a large number of configurations. This is achieved through different *analysis strategies*. Next, we give an overview of the common analysis strategies. For a concrete application of the strategies using type checking as a concrete analysis technique see Section 3.1; for further in-depth discussion and real-world examples we refer the reader to the work by von Rhein [vRhe16].

Variant-based strategy. This strategy is a brute-force approach that applies a legacy (non variability aware) program analysis to every single configuration (i.e., variant) of a configurable system. The advantages of this strategy is that any off-the-shelf implementation of a program analysis can be directly applied to a configurable system without modifications. The disadvantages of this strategy is that it can be stretched to its limits very fast, because of the exponential explosion of the configuration space with respect to the number of the configuration options [Ape+13c, p. 247]. Therefore, it can only be effectively applied to configurable systems with relatively small configuration space.

Sampled-based strategy. This strategy is similar to the variant-based, but the analysis is applied not to all configuration, but rather to a subset (a sample)

of the configuration space. Therefore, the strategy scales better than the variant-based. Although, the scalability comes at a price, since the result of a sample-based analysis may not generalize to the entire configurable system and strongly depends on how representable the analyzed sample is for the whole configuration space. Consider, for example, a sound type checker that checks a sample consisting of one type-safe configuration. The type checker will accept this configuration and, consequently, the entire configurable system, no matter if the rest of the configurations is type-safe or unsafe.

Feature-based strategy. This strategy deals with a configurable systems not as a set of configurations, but as a set of features and analyses the features in isolation. This way the analysis escapes the combinatorial explosion and gains scalability. On the downside, since the features are analyzed in isolation, the relations among them cannot be taken into account by the analysis technique. These relations may provide valuable information, for example, in the case of feature-interaction detection.

Family-based strategy. This strategy deals with a configurable system as a whole without explicitly drawing the line between configurations or features. Technically, for a static program analysis, this means that not only the source code of a configurable system is analyzed as one unit, but also the corresponding variability implementation (for example, the preprocessor annotations, Section 2.1.3) and the variability model (Section 2.1). Including variability information into an analysis makes the analysis variability-aware and allows it to produce a result for the entire configurable system without analyzing each configuration individually. The produced result then holds for the entire configurable system and, based on the result, statements can be made about individual configurations. A family-based analysis is potentially faster than a variant-based, because it does not have to check a potentially exponential number of configurations, but it require a modification of a traditional program analysis and the corresponding tools to make them variability-aware.

Combined strategies. The introduced basic analysis strategies can be combined to form new strategies [vRhe16]. For example, we can think of a strategy that is a combination of the family-based and variant-based strategies: The corresponding analysis technique would partially resolve the variability of the configurable system (the variant-based part), effectively dividing it in several smaller configurable systems, which would still preserve some unresolved variability. Then each of these smaller configurable systems would be analyzed using the family-based strategy. Applying this combined strategy may be

advantageous if, for example, the variability implementation cannot be analyzed efficiently, as it is the case with undisciplined preprocessor annotations [LKA11]. Another use case is when the complexity of the family-based analysis is to high, so that it is feasible to divide the configurable system and run the analysis on its parts [vRhe16].

2.4 Performance Prediction of Configurable Systems

Performance is one of the important quality properties of software systems. For example, it is important for a Web server to serve client requests fast to guarantee that users do not wait too long for a page to load; for a data compression program it is important to maintain the compression rate expected by a user. It is relatively easy to check if a given program satisfies a performance requirement by simply benchmarking the program. To determine the performance of a configurable system we have to deal with two challenges: (1) the exponential explosion problem (Section 2.3), that is, simply benchmarking every configuration of the system does not generally scale; (2) the feature interaction problem (Section 2.2), that is, inadvertent feature interactions may influence the performance of a configuration in an unexpected way.

Next, we will describe a machine-learning technique that deals with these challenges and can automatically and efficiently learn performance-influence models for configurable systems [Sie+12]. The technique detects the influence of feature interactions on the performance of the system's configurations and represents this influence in the performance model. The performance-influence model then can be used to accurately predict performance of any configuration of a system without actually measuring it.

2.4.1 Performance-Influence Models

To introduce performance-influence models, we will use a simplified example of a configurable database management system (DBMS). The system has three configuration options: Compression (), Encryption (), and Localization (). If Compression is enabled, data is compressed before it is stored in the database. If Encryption is enabled, then the stored data is encrypted. Localization is responsible for switching between two different localizations of the system (i.e., the language of the user interface, times and dates representation). Base () represents the core functionality of the system, which is present in all configurations.

Table 2.3: Benchmark results for the example DBMS. For each configuration, represented by the symbols of the enabled options, the performance in MB/s is given.

Id	Configuration	Performance	Id	Configuration	Performance
1		100	5		80
2		90	6		90
3		70	7		70
4		100	8		80

With the three optional configuration options (, ,), there are eight possible configurations of the DBMS. It is safe to assume that the performance of these configurations, which we define as *write throughput* (measured in megabytes written per second (MB/s)), depends on the configuration options enabled in this configuration. For example, Compression and Encryption require time to compress or encrypt the data before it can be written out, which has negative influence on the performance of the DBMS. Localization on the other hand has no obvious influence on the write throughput of the DBMS. Table 2.3 summarizes the write throughput of an imaginary benchmark of each configuration.

Based on the whole-population[7] benchmark results, we can already model the performance of the DBMS, as shown in Figure 2.6. This trivial performance-influence model is a straightforward representation of the whole-population benchmark results in the form of a linear polynomial. Each *term* (i.e., summand of the polynomial) consists of two parts: a *constant* representing the performance of a configuration and a *variable* (e.g.,), which can have two values: 1 if the corresponding configuration is selected, and 0 if not. To predict the performance of, for example, configuration 5 (Table 2.3), we evaluate the model as follows:

$$100 \cdot 0 + 90 \cdot 0 + 70 \cdot 0 + 100 \cdot 0 + \underline{80 \cdot 1} + 90 \cdot 0 + 70 \cdot 0 + 80 \cdot 0 = 80$$

Since this model simply lists performance values for all system configurations, it has a perfect prediction accuracy. However, modeling performance this way requires benchmarking every system's configuration, which does not generally scale. Moreover, this model does not give much insight into what the influence of individual configuration options or their interactions on performance is.

Figure 2.7 shows an alternative performance-influence model of the example DBMS, which is similar to the previous model, but now the variables in

[7] That is, one that includes all configurations of a configurable system.

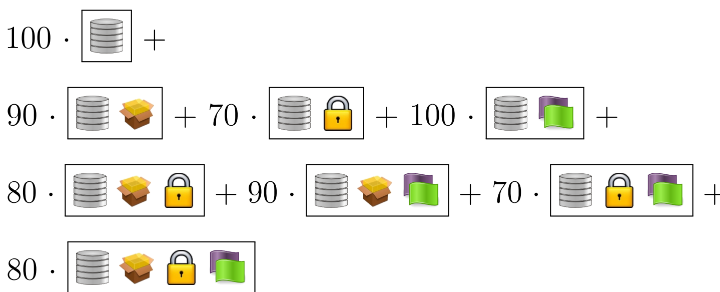


Figure 2.6: A trivial performance-influence model for the example DBMS, based on a whole-population benchmark. A variable (e.g.,) represents a system's configuration.

the terms (e.g.,) represent individual configuration options and not entire configurations. The corresponding coefficients of the terms represent the influence of these individual configuration options on the performance of the DBMS. Using this model, we can predict the performance of configuration 5 (Table 2.3) as follows: $100 \cdot 1 - 10 \cdot 1 - 30 \cdot 1 = 100 - 10 - 30 = 60$

$$100 \cdot \quad - 10 \cdot \quad - 30 \cdot$$

Figure 2.7: An alternative performance-influence model for the example DBMS, base on the performance influences of individual configuration options. A variable (e.g.,) represents a configuration option.

Comparing the two models, we observe that the second one contains terms that represent individual configuration options instead of whole configurations. Moreover, configuration options that do not have influence on performance (such as Localization) are omitted. Therefore, if it were possible to determine performance influences of individual configuration options, we could build a performance model that would accurately predict performance of all configurations. This model also denotes the influences of individual options on performance and, by interpreting the coefficients, we conclude that encryption reduces the performance of the DBMS more than compression.

However, the second model is an oversimplification, because it does not take feature interactions into account. Compression and Encryption interact, since encrypting compressed data is faster than encrypting the same data uncompressed. Not accounting for this interaction in the model results in a low prediction accuracy: the predicted value of 60 MB/s for configuration 5 has an error of 25 % compared to the measured value of 80 MB/s.

Figure 2.8 shows an performance-influence model that, besides the individual influences of configuration options, also accounts for the influences of feature

interactions (the last term). This model combines the advantages of the former two models: it is accurate and it is based on the performance influences of individual configuration options and their interactions. Moreover, such model can be learned using benchmarking results for only a fraction of all system's configuration, which makes the learning process scalable.

$$100 \cdot \quad - 10 \cdot \quad - 30 \cdot \quad + 20 \cdot \quad \cdot$$

Figure 2.8: An alternative performance-influence model for the example system with an interaction term. A variable represents a configuration option (e.g.,) or an interaction (e.g., ·).

Based on the formalization of configurable systems (Section 2.1), we define a performance-influence model as a function $\Pi : \mathcal{C} \rightarrow \mathbb{R}$ that maps each configuration $c \in \mathcal{C}$ to a performance value (as suggested by Siegmund et al. [Sie+15]). Further, let C denote the Compression option and E denote the Encryption option; the formal representation of the performance-influence model in Figure 2.8 is then

$$\Pi(c) = 100 - 10 \cdot c(C) - 30 \cdot c(E) + 20 \cdot c(C) \cdot c(E) \tag{2.6}$$

2.4.2 Multiple Linear Regression

Learning a performance-influence model that accurately predicts performance of any system's configuration, such as the model in (2.6), can be encoded as a *multiple linear regression* problem. Next, we discuss how it is done.

Multiple linear regression is a supervised statistical learning approach that consists in predicting a quantitative response Y based on *multiple* predictor variables $X_1 \dots X_p$ [HTF09, p. 11]. It assumes that there is a linear relationship between $X_1 \dots X_p$ and Y that can be written mathematically as

$$Y = \beta_0 + \beta_1 X_1 + \dots + \beta_p X_p$$

The values of predictor variables $X_1 \dots X_p$ are known and the values of *coefficients* (or *parameters*) $\beta_0, \dots, \beta_p$ are unknown and must be estimated (Section 2.4.3). Based on the estimates a predictor $\hat{y}$ of Y can be calculated as

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 X_1 + \dots + \hat{\beta}_p X_p$$

Typically, we use historical measurements consisting of measurements of $X_1 \dots X_p$ and Y for a number n of observations $(x_1, y_1), \dots, (x_n, y_n)$ to estimate the coefficients $\beta_0, \dots, \beta_p$. The goal is to obtain the coefficient estimates

$\hat{\beta}_0, \ldots, \hat{\beta}_p$, so that $y_i \approx \hat{\beta}_0 + \hat{\beta}_1 x_{1i} + \cdots + \hat{\beta}_p X_{pi}$ for $i = 1, \ldots, n$. In other words we want to minimize the difference $y_i - \hat{y}_i$ for all $i = 1, \ldots, n$. Consequently, finding the coefficient estimates boils down to a minimization problem that has an exact analytical solution [HTF09, p. 12] (which we do not discuss here any further for brevity).

We encode the problem of learning performance-influence model as a multiple linear regression problem by mapping values of configuration options and their interactions, for example $c(C)$, $c(E)$, and $c(C) \cdot c(E)$ in (2.6), to predictor variables $X_1 \ldots X_p$, and their influences on performance, for example -10, -30 in (2.6), to the coefficients $\beta_0, \ldots, \beta_p$. Y is then the performance value $\Pi(c)$ that is to be predicted.

2.4.3 Learning Performance-Influence Models

Next, we give an overview of an existing machine-learning algorithm that applies multiple linear regression to learn performance-influence models for configurable systems. The algorithm does not try to learn the influences of configuration options and their interactions all at once (which, again, may be exponentially many). Instead, it applies an iterative forward feature selection technique that prevents the exponential explosion of the number of the model terms. For a detailed description of the algorithm we refer the reader to the work by Siegmund et al. [Sie+15].

In Algorithm 1, we sketch the machine-learning algorithm. In a nutshell, the algorithm starts with an empty model and a set of candidates that can be included in the model to improve its accuracy. The set of candidates contains all configuration options and their combinations. In each iteration of the outer loop (Lines 2–20), the algorithm selects one candidate that satisfies a number of conditions best (defined through parameters) and add it to the model, so that the accuracy of the model increases. New candidates are added to the model until a termination condition is reached (e.g., the desired accuracy is reached or no more satisfying candidates exist).

Algorithm 1 is parameterized by parameters ***MinError*** and ***MinScore*** that are used to specify termination conditions. ***MinError*** specifies the prediction error the algorithm should strive for. As soon as the specified error is reached, the algorithm terminates (Line 20). The indirect effect of increasing ***MinError*** is that the learning algorithm terminates faster, because a model with lower accuracy can be learned with fewer iterations. This way, accuracy can be traded for learning time. Moreover, fewer iterations mean fewer candidates added to the model, so the complexity of the resulting model is reduced, too. Less complex models are also less susceptible to the negative effect of *overfitting*, which occurs when the models follow the noise (e.g., measurement errors) too

Algorithm 1 A sketch of the stepwise learning algorithm.

Input: *measurements*, *options*
Parameters: ***MinError***, ***MinScore***
Output: *model*

1: $selectedCandidates \leftarrow \varnothing$
2: **repeat**
3: $\quad (candidates, moreCandidates) \leftarrow$
4: $\qquad \textsc{constructCandidates}(options, \boldsymbol{MaxSize})$
5: $\quad bestCandidate \leftarrow$ **none**
6: $\quad error \leftarrow \infty$
7: $\quad lastError \leftarrow \infty$
8: $\quad$ **for all** $c \in candidates$ **do**
9: $\qquad model, modelError \leftarrow \textsc{learn}(selectedCandidates \cup \{c\}, measurements)$
10: $\qquad$ **if** $modelError < error$ **then**
11: $\qquad\quad error \leftarrow modelError$
12: $\qquad\quad bestCandidate \leftarrow c$
13: $\qquad$ **end if**
14: $\quad$ **end for**
15: $\quad score \leftarrow \textsc{computeScore}(lastError, error, bestCandidate)$
16: $\quad$ **if** $score \geq \boldsymbol{MinScore}$ **then**
17: $\qquad selectedCandidates \leftarrow selectedCandidates \cup \{bestCandidate\}$
18: $\qquad lastError \leftarrow error$
19: $\quad$ **end if**
20: **until** $(error \leq \boldsymbol{MinError}) \vee (score < \boldsymbol{MinScore} \wedge \neg moreCandidates)$
21: **return** model

closely [Jam+13, p. 22].

The algorithm decides whether a candidate should be added to the model by computing the difference in accuracy between a model with and without the candidate (Lines 15–19). We call this difference the *score* of the candidate. If the score is less than specified by ***MinScore***, the candidate is rejected. When we increase ***MinScore***, we increase the number of candidates that may be rejected, because the improvement in accuracy that they yield is lower than required. Rejecting these candidates reduces the accuracy of the resulting model, but it also reduces the complexity of the model (i.e., fewer terms are added to the model). Similar to ***MinError***, by increasing ***MinScore***, we trade accuracy for complexity.

2.5 Summary

In this chapter, we introduced the basic terms and central concepts on which we rely in this dissertation. We discussed the notions of configurable systems and feature interactions, then we introduced analysis strategies used to analyze configurable systems in general, and finally discussed performance-influence models used to predict performance of all configurations of a configurable system. We also mathematically formalized the introduced notions.

CHAPTER 3

A Comparison of Product-based, Feature-based, and Family-based Analysis Strategies

This chapter shares material with the following publication: S. Kolesnikov et al. "A comparison of product-based, feature-based, and family-based type checking". In: *Generative Programming: Concepts and Experiences, (GPCE)*. ACM, 2013, pp. 115–124 [Kol+13]

Analyzing configurable software systems is difficult, due to their inherent variability. Several variability-aware analysis strategies with different strengths and weaknesses have been proposed, in particular, product-based, feature-based, and family-based strategies (Section 2.3). Gaining more empirical evidence about how these strategies compare and to expand our knowledge about the applicability of these strategies is imperative, because in our dissertation we apply multiple variability-aware tools that rely on these strategies (such as TypeChef[1] and SPL Conqueror[2]).

Next, we compare product-based, feature-based, and family-based strategies in a controlled setting (i.e., by means of a common set of subject systems and the same analysis tool that implements all three strategies). As a concrete analysis technique, we choose type checking, as it has been used before in several studies on product-line analysis [Ape+10; DCB09; Käs+12; Lie+13]. We compare the strategies with regard to their ability to facilitate the detection of different kinds of type errors and the time required for type checking. We implemented the strategies as an extension of Fuji, an extensible compiler for

[1] http://fosd.net/TypeChef/

[2] http://fosd.net/SPLConqueror/

feature-oriented programming in JAVA [Ape+12]. Using FUJI, we compare the three strategies by applying them to 12 feature-oriented, JAVA-based product lines, from different application domains and of different sizes.

Overall, we found that the family-based strategy is superior in that it is complete (i.e., it was able to identify all type errors in the subject systems) and takes substantially less time for type checking than the other strategies. The feature-based strategy is also quite fast, compared to the product-based strategy, but incomplete. Based on the experimental results, we discuss a number of issues regarding the ability to detect and report errors, the role of optimization for family-based strategies, the influence of factors such as the size of a product line, and the trade-off between analysis coverage and time.

The implementation of the strategies (in the form of a FUJI compiler extension), the subject product lines, and the experimental data are available online.[3]

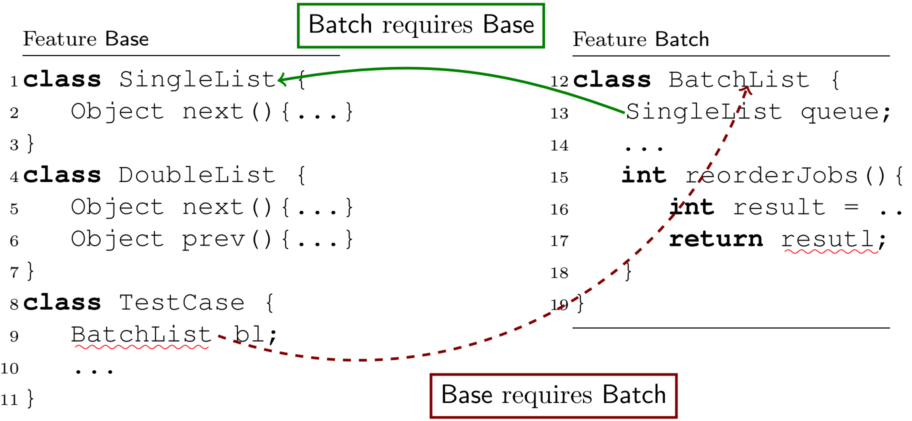


Figure 3.1: Examples of type errors: Two basic list implementations (feature Base) and a list for batch jobs (feature Batch); type errors are underlined; arrows denote references; the dashed arrow denotes a possibly dangling reference.

3.1 Product-Line Type Checking

By means of a simple product line and type checking as a concrete analysis technique, we give concrete examples of product-based, feature-based, and family-based strategies for product-line analysis.

[3] http://fosd.de/fuji/

3.1.1 Examples of Type Errors

The example in Figure 3.1 is a very simple product line of list data structures. It consists of two features: Base and Batch. The mandatory feature Base provides two basic implementations of lists: `SingleList` for singly-linked lists, and `DoubleList` for doubly-linked lists. It also provides a test class `TestCase`. The optional feature Batch provides a special list implementation `BatchList` for scheduling batch jobs that uses class `SingleList` of feature Base.

In Figure 3.2, we show the feature model (Section 2.1.2) of our example. It states that feature Base is mandatory (i.e., it must be present in every product) and feature Batch is optional (i.e., it may be present in a product). Consequently, our example consists of two valid products: {Base} and {Base, Batch} .

In Line 9 of Figure 3.1, feature Base refers to class `BatchList`. If we attempt to compile a product with Base and without Batch, we get a type error, because `BatchList` is declared only in Batch. This dangling reference is a simple example of an error that involves multiple features. The cause of this type error is an inconsistency between the feature model and the implementation of the product line: The feature model suggests that feature Base is independent from feature Batch, but the implementation requires these features to be selected together.

To resolve the inconsistency, we can make Batch mandatory or move the test case from Base to Batch. The key point is that, in large-scale product lines, such inconsistencies may go unnoticed for a long time and show up only late in the development cycle [Tar+11].

Another kind of type error is illustrated in Line 17 of Figure 3.1. There, the undeclared variable `resutl` is accessed. This type error is caused by a simple typo. It is an example of a *feature-local* error that can be discovered as soon as the affected feature is used in a product.

Next, we discuss which type errors can be detected by different type-checking strategies. We consider what information can be provided to a developer by a type checker to help to fix these errors. We also take a look at certain properties of the strategies that can influence type-checking performance.

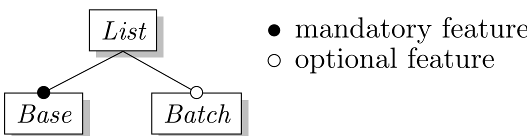


Figure 3.2: Feature model of the example product line.

3.1.2 Type Checking Product Lines

Our procedure for type checking of product lines consists of two steps, as illustrated in Figure 3.3. In the first step, *setup*, we parse the code of the considered features and compose it accordingly. In the second step, *checking*, we perform the actual type checks on the result of the first step. If a particular type-checking strategy cannot check the whole product line in a single run, the type-checking procedure is repeated. For example, the product-based strategy repeats the procedure for each product, the feature-based strategy repeats it for each feature, and only the family-based strategy checks all products simultaneously in a single run.

The performance of a type-checking strategy is the total time t required to check the entire product line:

$$t = \sum_{i=1}^{r} t_i^{\text{setup}} + t_i^{\text{checking}}$$

The value r is the number of type-checker runs needed to verify the complete product line; t_i^{setup} is the time used by the setup in run i, and t_i^{checking} is the time used by the checking step in run i. Based on this equation, we can derive the following possibilities for optimizing the type-checking procedure, of which the tree type-checking strategies make use to different extents:

- Minimize the number of type-checker runs r
- Minimize the setup time t_i^{setup}
- Minimize the checking time t_i^{checking}

3.1.3 Product-based Strategy

To ensure that every product of a product line is well typed, we can apply the product-based strategy. That is, we generate and check each product individually. This way, we find every type error in all products of the product line.

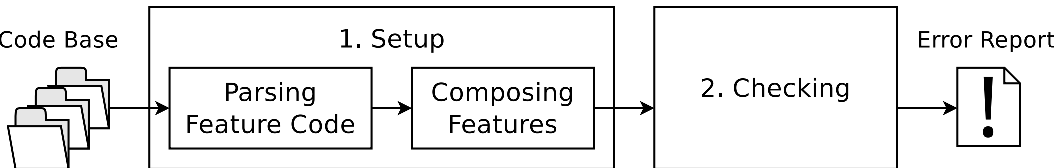


Figure 3.3: Steps of product-line type checking.

We can even use an off-the-shelf type checker (or compiler) for this task, because the individual products do not contain any compile-time variability. In our setting, the products are normal JAVA programs.

However, by generating and checking individual products, we lose information about the features the products are made of as well as about their dependencies. This makes it difficult to create meaningful error messages. The error messages for our example will only tell us that one product accesses an unknown type `BatchList`. A product-based type checker fails to identify the primary reason, namely, the false optionality of feature Batch. That is, the type checker cannot blame the feature that is responsible for the error, which is left to the user. This also applies to feature-local errors, such as for the undeclared variable `resutl`.

A major weakness of the product-based strategy is its poor scalability. With n optional and independent features, we have to repeat the type-checking procedure for each of the 2^n products. Therefore, the *upper bound* for the performance is

$$t = \sum_{i=1}^{2^n} t_i^{\text{setup}} + t_i^{\text{checking}} \qquad \text{(Product-based)}$$

The reason for the poor scalability are redundant analyses made in every step of the type-checking procedure (Figure 3.3). During the setup, we repeatedly parse and compose the same features again and again. During type checking, we repeat type checks that are similar among different products.

To avoid this redundancy, it is possible to parse the code of feature Base only once, because the corresponding parse tree is the same per product. Likewise, it is sufficient to perform type checks that concern code inside Base, such as type checking the body of method `next`, only once.

The remaining two type-checking strategies exploit this optimization potential, which we explain next.

3.1.4 Feature-based Strategy

Using the feature-based strategy, we check every feature of a product line individually. We assume that all types, declarations, and so on that a feature requires are available in all valid products. For example, if we check feature Base, then the feature-based type checker assumes that the required type `BatchList`, provided by feature Batch, is always available. Type `BatchList` becomes part of the feature's required interface. While the feature-based strategy may seem naive at first glance, it is motivated by open-world systems, in which not all features are known at development time [LKF02; LBL11].

Technically, we implement the required interface of a feature module using stubs. A *stub* is a bundle of JAVA interfaces and classes, possibly with member prototypes, that represent the types and members a feature requires from other features. Either stubs are provided by the developer to define the required interface, such as in HYPER/J [TOS02], or they are generated using tools, such as AHEAD [Tha+07] or FEATURESTUBBER.[4] To type check feature Base of our example, a stub containing an empty class named `BatchList` is needed, possibly with proper member declarations.

While checking a feature, the feature-based strategy does not know anything about other features. Consequently, a feature-based type checker cannot detect type errors that arise between features. In our example, it cannot detect the erroneous access to the missing type `BatchList`, because the type is provided by the stub (i.e., the type checker simply assumes that it will be provided by another feature). Only errors that are local to a feature can be detected by the feature-based type checker, as the undeclared variable `resutl`.

To guarantee that all products are well typed, feature-based type checking has to be supplemented with additional type checks during byte-code feature composition (which corresponds to linking in C). The result is a mixed *feature-product-based* strategy [Thü+12a].

Much like for the product-based strategy, we can use an off-the-shelf type checker for feature-based type checking, because a single feature complemented with stubs does not contain any compile-time variability.

In contrast to a product-based type checker, a feature-based type checker can provide sufficient information about feature-local errors, but it misses errors that arise from combinations of features.

Furthermore, the feature-based strategy requires one type-checker run for each feature. Thus, every feature is parsed and checked only once, and the number of the unnecessarily repeated actions is reduced compared to the product-based strategy. The strategy also completely avoids the feature-composition part of the setup (cf. Figure 3.3). To summarize, it utilizes the following optimization possibilities:

- The number of type-checker runs r is reduced from 2^n to n, where n is the number of features.
- The setup time t_i^{setup} is reduced by omitting feature composition.

Therefore, the performance of the feature-based strategy is

$$t = \sum_{i=1}^{n} t_i^{\text{setup}} + t_i^{\text{checking}} \qquad \text{(Feature-based)}$$

[4] http://fosd.de/featurebite/

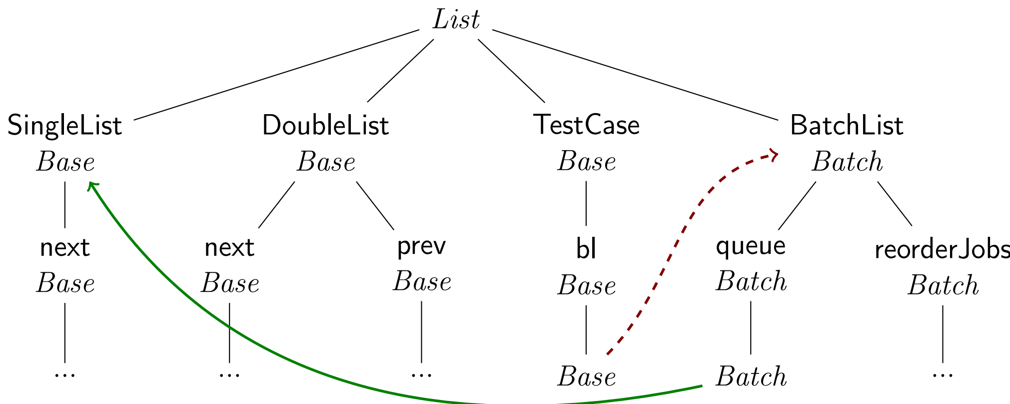


Figure 3.4: Syntax tree of the example product line. Every node knows the feature to which it belongs. Arrows denote references; the dashed arrow denotes a possibly dangling reference (cf. Figure 3.1).

3.1.5 Family-based Strategy

The family-based strategy analyses the code base of a product line as a whole. Hence, it can detect all type errors and guarantee that all products of a product line are well typed.

The variability of a product line has to be incorporated into the type-checking procedure, so that the type checker can take it properly into account. The key idea is to compose all features of the product line (even mutually exclusive ones), and to keep variability information in the syntax tree (i.e., which program element belongs to which feature and depends on which other features) [Tha+07]. This way, the syntax tree does not represent only a single product or feature, but the whole product line, as illustrated in Figure 3.4. A family-based type checker works on these enriched syntax trees and must be able to cope with variability. For this reason, we cannot use an off-the-shelf type checker for this task.

Figure 3.5 shows an example of an error message produced by our family-based type checker. The error message identifies the features participating in the type error. It shows in which feature and where exactly in its code the error occurs. Moreover, the error message describes the exact cause of the error, namely that feature Base requires feature Batch (because Base uses type `BatchList` introduced by Batch), but feature Batch is not present in all products in which Base is present; all these products contain the type error, which is useful information for debugging.

The family-based strategy parses and composes the code of all features in one run. Thus, no repetitive parsing or composition of the same source

```
Base/TestCase.java:9:
  Type Error: 1 optional target:
    Feature Base accesses the type
    (default package).BatchList of feature Batch
```

Figure 3.5: Error message of the family-based type checker.

Table 3.1: Conceptual comparison of the three type-checking strategies.

Strategy	Performance	Optimization	Error detection	Feature blaming	Tool reuse
Product-based	$\sum_{i=1}^{\mathbf{2^n}} t_i^{\text{setup}} + t_i^{\text{checking}}$	—	●	○	●
Feature-based	$\sum_{i=1}^{\mathbf{n}} t_i^{\text{setup}} + t_i^{\text{checking}}$	# runs, setup	◐	●	●
Family-based	$t^{\text{setup}} + t^{\text{checking}}$	# runs, checking	●	●	○

● possible ◐ partly possible ○ impossible

code is necessary. The resulting syntax tree represents the whole product line. Therefore, only one type-checking run is needed to cover the whole product line. To summarize, the strategy utilizes the following optimization possibilities:

- The number of type-checker runs r is reduced to the minimum of one.
- Furthermore, the time t^{checking} for type checking can be reduced by using caching, as we will explain in Section 3.3.4.

Therefore, the performance of the family-based strategy is

$$t = t^{\text{setup}} + t^{\text{checking}} \qquad \text{(Family-based)}$$

3.1.6 Summary

For a better overview, Table 3.1 summarizes the properties of the three strategies, regarding performance, optimization, the ability to find errors and to blame features, and the possibility to reuse off-the-shelf tools.

3.2 Motivation and Hypotheses

Our goal is to compare the three strategies quantitatively in terms of their performance. We believe that the family-based strategy outperforms the other

two, because it reduces the number of type-checker runs to one. This way, the strategy avoids unnecessarily repeated analysis operations during the setup and checking steps. Nevertheless, a family-based type checker has to take the whole variability of a product line into account. Therefore, the respective problem is more complex and may require more time for analysis. Moreover, family-based type checkers rely on SAT solvers to determine dependencies between features [Ape+10; Käs+12]. The corresponding SAT solver calls are expensive and may reduce the overall performance. Therefore, product-based type checking may be faster than family-based, especially, on product lines with a small number of products.

As for the feature-based strategy, it processes the same amount of code as the family-based strategy and completely avoids the composition part and ignores feature combinatorics. Thus, it is an open question whether it outperforms the family-based strategy when the analyzed product line has a small number of features. Of course, the potential win of the feature-based strategy would be at the expense of the number of type errors found, as it would detect only feature-local errors (see Section 3.1.4). To be able to detect errors occurring between features, we supplemented the strategy with additional type checks at the byte-code level, as explained in Section 3.1.4. These type checks run on the per-product basis when composing separately compiled feature modules, which corresponds, in fact, to a mixed feature-product-based strategy. For illustration, we also present the performance measurements for this mixed strategy.

Based on our considerations, we state the following three hypotheses that we address in the evaluation:

- **H.1:** The family-based strategy is superior to the feature-based and product-based strategies in terms of performance.

- **H.2:** As an exception to H.1, the product-based strategy is superior to the family-based one, if the analyzed product line has a relatively small number of products.

- **H.3:** As an exception to H.1, the feature-based strategy is superior to the family-based one, if the analyzed product line has a relatively small number of features.

3.3 Empirical Evaluation

We implemented the product-based, feature-based, and family-based type-checking strategies in a single type checker. The type checker is an extension

of a feature-oriented JAVA compiler FUJI.[5] It operates on the abstract syntax tree, built by the FUJI's parser, and extends the underlying JAVA type system to implement variability-aware type checks.

To calculate dependencies between features, our type checker uses a corresponding library from the FEATUREIDE project.[6] In FEATUREIDE, the problem of determining a dependency between features is reduced to a SAT problem and solved by querying an off-the-shelf SAT solver (SAT4J). FEATUREIDE implements a caching mechanism to reduce the response time in the case of multiple identical SAT solver queries (see Section 3.3.4, Caching).

It is important to note that our type checker covers many but not all JAVA type rules. In a nutshell, it checks all accesses to fields, methods, constructors, and types, and verifies that the accessed elements are present in all corresponding products. The possibly variable type hierarchy of the corresponding product line is considered too, because it can influence the presence or absence of program elements, such as fields or methods. For illustration, let us assume that we decided to add two *alternative* features to our example, as illustrated in the Figures 3.6 and 3.7. The new features refine feature Batch, and specify a new superclass for class `BatchList`. Consequently, the two alternative features define which methods are inherited by `BatchList`. If feature BatchSingle is selected, the superclass of `BatchList` is `SingleList`, and `BatchList` inherits method `next`. If feature BatchDouble is selected, the superclass of `BatchList` is `DoubleList`, and `BatchList` inherits the methods `next` and `prev`. Now, if `prev` is called on a `BatchList` object, the type checker has to determine in which feature combination method `prev` is inherited by

[5] `http://fosd.de/fuji/`
[6] `http://fosd.de/featureide/`

Feature BatchSingle

```
1 class BatchList extends SingleList {
2   ...
3 }
```

Feature BatchDouble

```
1 class BatchList extends DoubleList {
2   ...
3 }
```

Figure 3.6: Example of a variable type hierarchy: The choice between BatchSingle and BatchDouble defines the superclass of class `BatchList`.

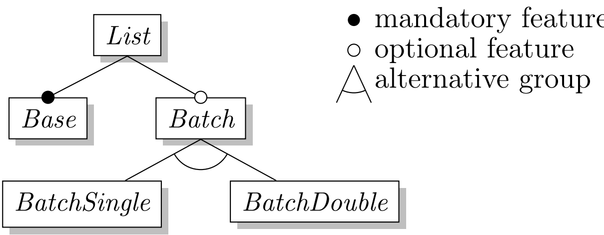


Figure 3.7: Feature model for the example product line, extended with the two new features BatchDouble and BatchSingle.

`BatchList` (in our case, the required combination is {Base, BatchDouble}).

Further type rules cover explicit and implicit casts, which also take a possibly variable type hierarchy into account. The rules for implicit casts cover assignment expressions involving two variables, assignments of a return value of a method call, parameter passing, and so on.

3.3.1 Subject Systems

We conducted the evaluation of the type-checking strategies on a set of 12 feature-oriented, Java-based product lines. The set has been collected and prepared before for benchmarking purposes and was used in several studies [AB11; Ape+12]. The subject systems belong to different application domains, and they are of different sizes: in terms of lines of code, number of features, and number of products. In Section 3.3.4, we use the three size categories to discuss the relation between the size of a product line and the corresponding performance of type checking. Table 3.2 summarizes relevant information about the systems.

3.3.2 Measurement Procedure

To compare the performance of the three type-checking strategies, we applied each strategy to each subject system, and we measured the time required by every step of the type-checking procedure (i.e., t^{setup} and t^{checking}). We repeated each measurement 10 times and took the average value to reduce measurement bias. The maximum relative standard error was 3.1 %, which we observed for family-based type checking of ZipMe. For the product-based strategy, we did not include the time needed to generate the configuration of each product, because this time was negligible compared to the time required for type checking. Likewise, for the feature-based strategy, we did not include the time used to generate stubs (Section 3.1.4), because stubs represent required interfaces and are part of the corresponding feature modules. We measured the performance

Table 3.2: Overview of the subject systems (LOC: number of lines of code; # F: number of features; # P: number of products).

System	Domain	LOC	# F	# P
EPL	Expression evaluation	304	12	425
GPL	Graph library	2855	25	156
GraphLib	Graph library	401	5	16
GUIDSL	Configuration tool	14 318	26	24
Notepad	Text editor	2193	10	512
PKJab	Chat client	4109	8	48
Prevayler	Persistence library	6185	6	32
Raroscope	Compression library	415	4	16
Sudoku	Game	1926	6	64
TankWar	Game	4845	38	2458
Violet	Model editor	10 866	88	$\approx 2^{88}$
ZipMe	compression library	5076	13	24

of the family-based type checker twice, with and without caching, to investigate the influence of the caching on the overall performance.

We instrumented the code of the Fuji compiler with calls to the timer of `ThreadMXBean`,[7] such that we measure only the CPU time consumed by the type-checker thread. This approach eliminates the influence of other concurrent tasks (e.g., garbage collection) on the measurement results. Furthermore, we did not measure the JVM startup time for each type-checking run, because the overhead can be avoided using special tools.[8]

We conducted all measurements on a workstation equipped with an Intel Xeon CPU (2.9 GHz) and 8 GB RAM, running Ubuntu 12.04 (64-Bit) and OpenJDK 7 (u21).

3.3.3 Results

In Table 3.3, we present the results of our measurements (in seconds). For each subject system and each strategy from our comparison, we show the setup time (t^{setup}), the checking time (t^{checking}), and the total time (i.e., the performance of the strategy, t). We also provide total times (t) for the feature-product-based strategy and the family-based strategy with caching disabled. For the feature-based strategy, we computed the speedups relative to the product-based strategy. For the family-based strategy, we computed the speedups relative

[7] `java.lang.management.ThreadMXBean` is part of the Java 7 API.
[8] `http://martiansoftware.com/nailgun/`

to the product-based strategy and the feature-based strategy. Note that we aborted the product-based measurements for VIOLET after checking 40 random products, because it was impossible to check all of the approximately 2^{88} products in reasonable time. We mark the corresponding values in the table with "X."

For comparison, we visualize the results in Figure 3.8 by means of bar plots. There is one bar plot per subject system, consisting of five stacked bars, divided into two groups (with different axes to compensate the considerable differences between the measured times). Each bar denotes the amount of time (in seconds) used by the corresponding type-checking strategy. The light gray part of each bar denotes the time required by the setup step; the white part denotes the time required by the check step. The crosses over the bars for VIOLET indicate that we aborted measurements at this point.

Note that, for the family-based strategy, there are two bars: the first, FM, denotes the performance of the strategy with SAT-solver caching *enabled*; the second, $FM^{\star}$, denotes the performance with SAT-solver caching *disabled.* $FT^{\star}$ denotes the performance of the feature-product-based strategy; its dark gray part denotes the time required by the byte-code feature composition (Section 3.1.4).

As we can see, the family-based strategy is the fastest for all subject systems. Compared to the product-based strategy, the minimum *speedup* of this strategy has been measured for GUIDSL, where it is 8.8 times faster. The maximum speedup of 745.3 has been measured for TANKWAR. As we could not check all products of VIOLET in reasonable time, we do not consider the corresponding speedups (they are likely to be much higher). Compared to the feature-based strategy, the speedup of the family-based strategy lies in between 1.7 and 6.5. The feature-based strategy is the second fastest. Its speedup compared to the product-based strategy lies in between 2.2 and 129.7.

Recall that the feature-based strategy finds only feature-local errors (Section 3.1.4). In our evaluation, it found no errors at all. The reason is that our subject systems have been used in many previous studies. Every single feature of the product lines was type checked as part of a product at least once. Therefore, all feature-local errors have already been detected and fixed. The other two strategies detected the same 556 unique type errors. These errors occurred between features and stayed undetected, because the corresponding feature combinations have been never considered by the developers and users of the systems.

The results support our first hypothesis H.1 (Section 3.2): the family-based strategy is superior to the other two strategies in terms of performance. Although quite apparent from Table 3.3, we still conducted statistical tests

Table 3.3: Measurement results for each subject system and type-checking strategy (in seconds). For each system and each strategy from our comparison, the setup time (t^{setup}), checking time (t^{checking}), and total time (t) are provided. We also provide total times (t) for the feature-product-based strategy and the family-based strategy with caching disabled. For the feature-based strategy, the speedups relative to the product-based strategy are provided. For the family-based strategy, the speedups relative to the product-based strategy and the feature-based strategy are provided. We rounded all values to one decimal place. ✗ We aborted the product-based measurements for VIOLET after checking 40 random products (cf. Section 3.3.3).

	Product			Feature			
	Time (seconds)			Time (seconds)			Speedup w.r.t.
System	t^{setup}	t^{checking}	t	t^{setup}	t^{checking}	t	Product
EPL	152.7	28.9	181.7	4.6	0.3	4.9	37.2
GPL	68.5	43.7	112.2	8.1	3.8	12	9.4
GRAPHLIB	6	2.8	8.8	1.8	0.5	2.4	3.7
GUIDSL	19.6	22.5	42.1	12.2	6.9	19.1	2.2
NOTEPAD	216.1	300.1	516.3	3.7	4.3	8	64.5
PKJAB	27.6	41.6	69.2	3.9	2.4	6.3	11
PREVAYLER	19.3	35.7	55.1	3	1.9	4.9	11.2
RAROSCOPE	6.4	3.1	9.5	1.9	0.4	2.3	4.1
SUDOKU	32	37.9	69.9	3	2.9	5.8	12
TANKWAR	1254.4	1326.5	2580.8	12.7	7.2	19.9	129.7
VIOLET	21.6✗	37.1✗	58.7✗	39.6	14.4	54	1.1✗
ZIPME	12.8	10.2	23	5.5	1.3	6.8	3.4

	Family					Feature-product	Family (no caching)
	Time (seconds)			Speedup w.r.t.		Time (seconds)	Time (seconds)
System	t^{setup}	t^{checking}	t	Product	Feature	t	t
EPL	0.4	0.4	0.8	240.3	6.5	90.5	6.4
GPL	0.6	1.7	2.3	48.9	5.2	84.2	798.2
GRAPHLIB	0.4	0.3	0.7	12.7	3.4	6.5	1.7
GUIDSL	1	3.7	4.8	8.8	4	55.9	2102.2
NOTEPAD	0.5	1.6	2.1	242.2	3.8	334	34.2
PKJAB	0.6	2	2.6	27.1	2.5	70.8	47.2
PREVAYLER	0.7	2.2	2.9	18.9	1.7	51.4	21
RAROSCOPE	0.4	0.3	0.7	13.4	3.2	7.8	1.6
SUDOKU	0.6	1.1	1.7	41.7	3.5	54.6	9
TANKWAR	0.6	2.9	3.5	745.3	5.7	2176.5	3200.9
VIOLET	0.8	10	10.8	5.4✗	5	100.2✗	32251.1
ZIPME	0.6	1	1.6	14.5	4.3	30.1	185.3

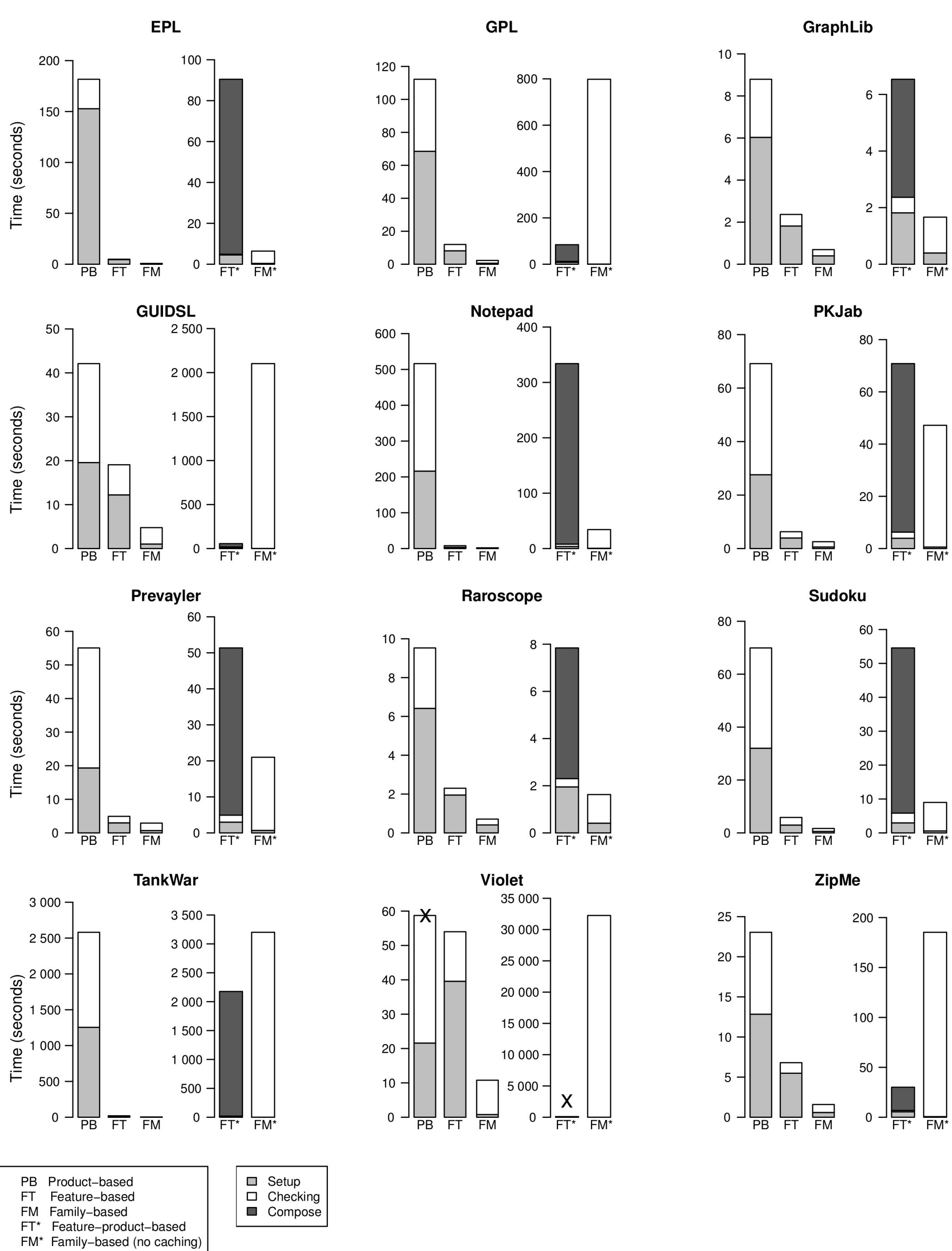


Figure 3.8: Type-checking times for each subject system—five bars per system. A bar denotes the time used by the corresponding type-checking strategy. Each step of the type-checking procedure (Section 3.1.2) is denoted by a different color inside a bar. The crosses over the bars for VIOLET indicate that we aborted the product-based measurement after checking 40 products (cf. Section 3.3.3).

to test all hypotheses. We used the Wilcoxon test, because the data are not normally distributed (according to a Shapiro-Wilk test). Though, we could use the non-parametric ANOVA, we decided to use the conservative double-test variant with Bonferroni correction, because it is more rigorous. For all performance comparisons, the p value is much smaller than 0.01.

We found no supporting evidence for hypothesis H.2 (i.e., product-based is superior on product lines with few products) and H.3 (i.e., feature-based is superior on product lines with few features), because, for none of our subject systems, the product-based or the feature-based strategy is superior to the family-based strategy, not even for very small product lines with few products and features (e.g., GraphLib and Raroscope).

A comparison of the results for the two variants of the family-based type checker shows a substantial influence of SAT-solver caching on performance.

Finally, the feature-product-based strategy is always slower than the family-based strategy (with caching) and the feature-based strategy. More interestingly, it is only in several cases slower than the product-based strategy (e.g., GUIDSL, PKJab, ZipMe), which indicates the benefits of separate feature compilation and byte-code feature composition.

3.3.4 Discussion

Next, we discuss the results of our measurements based on the size categories of Table 3.2 as well as regarding the implementation of our type checker. We use the product-based strategy as the base line, and compare it to the other strategies. We subdivided this section in three parts, one part for each strategy.

Product-based strategy. The measurements of the product-based strategy support our expectations about its poor scalability. As discussed in Section 3.1.3, this strategy induces considerable redundant work in every step of the type-checking procedure, while the number of the unneeded repetitions increases with the number of products. An extreme example is Violet, which we could not even check completely, because of the sheer amount of time required to generate and check all of the approximately 2^{88} products. Nevertheless, if developers have to use a standard (non-variability-aware) type checker, product lines with few products and relatively small code bases (e.g., Raroscope) can be checked in reasonable time.

Comparing the results for GUIDSL and ZipMe makes it apparent that it is insufficient to consider only the number of products when estimating the performance of the product-based strategy. Both product lines have the same number of products, but type checking GUIDSL lasts almost twice as long as

Table 3.4: Break-even points of the superiority of the family-based strategy for the subject systems. Total number of products # P and break-even points ↯B (i.e., the number of products whose cumulative analysis time exceeds the time needed by the family-based strategy to check *all* products of the product line).

System	# P	↯B	System	# P	↯B
EPL	425	2	Prevayler	32	2
GPL	156	4	Raroscope	16	2
GraphLib	16	2	Sudoku	64	2
GUIDSL	24	3	TankWar	2458	4
Notepad	512	3	Violet	$\approx 2^{88}$	8
PKJab	48	2	ZipMe	24	2

type checking ZipMe. The cause is the larger code base of GUIDSL, which is almost three times larger than the code base of ZipMe. Systems with a similar number of products and a similar size of the code base (e.g., GraphLib and Raroscope) have similar times.

Although not in the scope of our study, one could use sampling to speed up the product-based strategy [Guo+13; JHF12; OMR10; Sie+12; Sie+13]. This would render the analysis incomplete, but tractable, at least. To give an impression of how the family-based strategy performs in comparison to sampling, we computed the (average) number of products one has to check with a sample-based strategy to exceed the time needed for the family-based strategy. This number marks the break-even point, at which the family-based strategy is superior without question (recall sampling is incomplete). In Table 3.4, we list the break-even points for the subject systems in terms of this number (and the overall number of products checked by the product-based strategy). The results are clear: Only when checking a very small number of products (less than 5 %, on average), a sample-based strategy is faster. But these small numbers also mean that the coverage will be very low and does not satisfy state-of-the-art coverage criteria (e.g., pair-wise coverage [OMR10]). This observation is in line with previous results [Lie+13].

Feature-based strategy. A feature-based type checker parses and checks the code of each feature only once (cf. Section 3.1.4). The result of this optimization becomes apparent when we compare the performance of the product-based and feature-based strategies in the setup step. With a growing number of products the advantage of the feature-based strategy becomes more evident. Nevertheless, the feature-based strategy induces an overhead for every type-checker run that is caused by instantiating internal data structures and loading classes from the

Java run-time library. This overhead explains why setting up type checking for GUIDSL is only slightly faster using the feature-based type strategy, then using the product-based strategy. A peculiarity of GUIDSL that is responsible for this effect is that it has a relatively small number of products and more features than products.

The checking step of the feature-based strategy also consumes less time than that of the product-based strategy. Still, we have to keep in mind that the feature-based strategy is able to detect only feature-local errors (cf. Section 3.1.4). Our subject systems have been used in many previous studies and all eventual feature-local errors have already been fixed. Therefore, the feature-based strategy found no errors. The inability to detect the full range of errors is the main weakness of this strategy.

We used FeatureBite[9]—a tool developed by us—to perform supplementary type checks when composing individually compiled feature modules to products. These additional checks at the byte-code level find type errors that arise between features (cf. Section 3.1.4). This way, we can achieve the same level of type safety as with the other two type-checking strategies (all 556 errors are found). However, our evaluation demonstrates that attaining type safety by combining the feature-based and product-based strategy requires considerably more effort than using the feature-based strategy alone, which was to be expected. The interesting finding is that, in all subject product lines except GUIDSL, PKJab, and ZipMe, the feature-product-based strategy outperforms the product-based strategy.[10] The reason is that the number of products of these three product lines is relatively low compared to the number of their features, which outweighs the benefit of separate feature compilation. This result demonstrates that the intermediate steps of checking and compiling feature modules and composing them at the byte-code level can positively influence analysis performance.

Family-based strategy. The family-based strategy is the clear winner among the three strategies. It requires only one run to check all products of a product line. Consequently, it does not induce the overhead of feature-based type checking (i.e., repeated instantiation of data structures in each run) in the setup step. It also avoids the overhead of the product-based strategy (i.e., repeated type checks) in the checking step (cf. Section 3.1.5).

Furthermore, our results show that the family-based strategy outperforms also the feature-based strategy in the checking step, even though the feature-based strategy considers only features in isolation. We attribute this phe-

[9] `http://fosd.de/featurebite/`

[10] We do not consider Violet, as we could not check all its products.

nomenon to the same kind of overhead that the feature-based strategy induces in the setup step (i.e., repeated instantiation of data structures in each run). From Table 3.3, we can see that the advantage of the family-based strategy in the checking step increases with the number of features. For product lines with a small number of features (e.g., GraphLib, Prevayler, Raroscope, Sudoku), the family-based strategy is 1.7 to 3.5 times faster than the feature-based strategy. For product lines with larger numbers of features (e.g., GPL, GUIDSL, TankWar), the family-based strategy is 4.1 to 5.8 times faster than the feature-based strategy.

Caching. One property of our family-based type checker poses a principal boundary on its performance. The type checker reduces the problem of determining dependencies between features to a SAT problem (cf. Section 3.1.5). SAT is NP-complete, which renders family-based type checking NP-complete, as well (w.r.t. the number of features). Luckily, today's SAT solvers mitigate this theoretical boundary for practical problems. Nevertheless, the calls to a SAT solver are still expensive enough, so minimizing the number of such calls is always a good idea.

Our family-based type checker uses a caching mechanism. All queries of the type checker to the SAT solver are cached, and none of the queries is performed twice. As we can see from the measurements (the FM and FM* bars in each plot, Figure 3.8), the caching mechanism leads to a substantial speedup. This is due to the fact that the family-based type checker makes a considerable number of repeated, identical calls that involve the SAT solver.

A large number of features often means a more complex feature model and, consequently, more expensive SAT solver calls. A small number of products keeps the time needed for the product-based type checking relatively low. ZipMe and GUIDSL are such product-lines, and, as we can see in Figure 3.8, the family-based type checker without caching is slower than the product-based type checker.

The reasons for the success of caching is that the feature modules in our subject systems are relatively coarse-grained units, and checking them involves checking a large number of identical type, method, and field accesses. This may not be the case if a product line consists of many fine-grained features containing no or few identical accesses (e.g., as may be the case for preprocessor-based variability).

3.4 Threats to Validity

We implemented a substantial subset of type rules in FUJI, but not all type rules specified for the JAVA language. This threatens the internal validity of our study. However, the implemented rules cover a considerable number of language constructs and involve complex analyses of the possibly variable type hierarchy. We can safely assume that adding new type checking rules (e.g., checking access modifiers) will not change the overall picture substantially.

As often the case, the external validity of our study is affected by the choice of the subject product lines. In our evaluation, we used only product lines built with AHEAD/FUJI-style feature modules. The coarse-grained nature of these features is beneficial for the caching mechanism used in the family-based type checker (cf. Section 3.3.4, Caching). Although, we cannot draw sound conclusions for other kinds of feature implementations (e.g., based on the C preprocessor), previous work shows a similar picture, at least regarding the performance of the family-based strategy compared to the product-based strategy [Lie+13].

3.5 Related Work

Our classification of product-line analysis strategies is based on a recent survey by Thüm et al. [Thü+12a]. Beside the classification, the authors discuss the conceptual strengths and weaknesses of the individual strategies. Based on this survey, von Rhein et al. propose the Product-Line-Analysis model [vRhe+13] that describes a whole spectrum of possible combinations of product-line analysis strategies.

The family-based strategy has been applied to several analysis techniques, including type checking [Ape+10; DCB09; Käs+12; Tha+07], static analysis [Bod+13; Bra+13; Lie+13], model checking [Ape+11; Ape+13d; Cla+10; LTP09], performance measurement [SvRA13], and deductive verification [Thü+12b]. The feature-based strategy has been used before for type checking [AH10; BDS13] and verification [LKF02] of product lines. Product-based analyses with sampling have been used in the context of product-line testing [JHF12; OMR10] and performance prediction [Guo+13; Sie+12; Sie+13].

There are only few studies that compare product-line analysis strategies empirically. Two studies evaluated the performance of the family-based and product-based strategy in the context of product-line verification [Ape+13d; Cor+12]. Brabrand et al. compares the performance of the family-based and product-based strategy for static analysis [Bra+13]. For type checking, Liebig at al. evaluated the efficiency of several sample-based strategies, compared to

the family-based strategy [Lie+13]. While their results are in line with ours, we are the first who have implemented all three strategies in one tool and have evaluated them using the same subject systems and measurement procedure, so that all comparisons have been made in a controlled setting.

3.6 Summary

Since tools for analysis of configurable software rely on different product-line analysis strategies, it is imperative to gain more empirical evidence about how these strategies compare and to expand our knowledge about the applicability of these strategies. To this end, we compared the three product-line analysis strategies—product-based, feature-based, and family-based—in a controlled setting. In our evaluation, we used feature-oriented programming as an implementation technique and type checking as an analysis technique, although the big picture of our results may be transferable to other techniques. In particular, we compared the analysis performance, but we also addressed the ability to detect different kinds of errors, and the quality of the provided information about errors. Our evaluation is based on a feature-oriented compiler that we extended with the three type-checking strategies for this purpose, and a subject set of 12 feature-oriented, Java-based product lines.

A main result of our study is that the family-based strategy outperforms the other strategies for all subject systems in terms of analysis time. We identified its caching mechanism as the key factor for the success, as it substantially reduces the number of SAT-solver queries. At the same time, the family-based strategy is complete: it finds errors that are feature-local and that occur among several features (556 in total), which is not the case for the feature-based strategy. Furthermore, the family-based strategy provides the most comprehensive error messages, as it has all information on features and variability at its disposal, which is not the case for the other two strategies.

Although not being in the focus of our study, we found that pursuing a sampling-based strategy (checking only a tractable subset of products) would not change the big picture. For our subject systems, the break-even point, at which the family-based strategy becomes faster, is at very low numbers of products, which means that the corresponding analysis coverage of sampling is likely to be very small, compared to the family-based strategy, which achieves full coverage.

Surprisingly, the feature-based strategy is often slower than the family-based strategy, although it ignores feature interactions and is, consequently, incomplete. Combining the feature-based with the product-based strategy makes it complete, but is substantially slower. Interestingly, such a combined

strategy outperforms the plain product-based strategy in most cases in our experiments, which indicates that separate feature compilation and byte-code composition can have a positive effect on analysis performance.

Altogether, our evaluation complements valuable empirical data about how different analysis strategies for configurable systems compare with respect to completeness and performance. This information can be taken into account by the developers of different variability aware analysis techniques and tools among which are also techniques and tools for feature-interaction detection that we use in our dissertation.

CHAPTER 4

Tradeoffs in Modeling Performance of Configurable Systems

This chapter shares material with the following publication: S. Kolesnikov et al. "Tradeoffs in modeling performance of highly configurable software systems". In: *Software and Systems Modeling (SoSyM)* (Feb. 2018). Online first, pp. 1–19 [Kol+18]

One of the goals of our dissertation is to understand the influence of feature interactions (or interactions among configuration options) on performance of configurable systems. To accomplish this goal, we need to gather and analyze empirical data on interactions in configurable systems and their influences on performance. That is, our first steps should be (1) detecting interactions that have influence on performance for a set of real-world configurable systems and (2) quantifying the interactions' influences. Both can be done using performance-influence models (Section 2.4.1). Performance-influence models describe how individual configuration options and, what is especially important for this dissertation, how their interactions influence performance of a configurable system. For our and other techniques based on performance-influence models to be practically useful and generally applicable, performance-influence models should exhibit the following properties:

1. Low *prediction error* (i.e., be as accurate as possible), such that it accurately describes the influence of interactions on system's behavior,

2. Small *model size*, such that it is understandable by humans for a wide variety of tasks involving human judgment, such as program comprehension, but still clearly denotes interactions present in the system, and

3. Short *computation time*, such that constructing the model is feasible in practice.

It is well known in the machine-learning community that there are tradeoffs among prediction error, model size, and computation time [Dom00; Jam+13; SW11]. Hence, optimizing for one property may negatively influence the others. The goals of this chapter are the following: (1) to explore whether these tradeoffs are practically relevant for performance-influence models in the domain of configurable software systems and how significant they are; (2) to study if the interactions described by performance-influence models really exist in the subject systems, their causes and influence on performance.

In a nutshell, our results show that, although, the tradeoffs among the different model properties technically exist, their effect is surprisingly low, so that they have *effectively no negative influence for practical purposes*. Furthermore, by analyzing the source code and documentation of the subject systems, we were able to show that the identified interactions actually exist in the subject systems. We were also able to explain the causes for these interactions having the observed influences on the systems' performance. Moreover, we identified several interaction patterns across subject systems, such as dominant configuration options and data pipelines, that explain the influences of highly influential configuration options and interactions, and give further insights into the domain of configurable systems.

These results are important in several ways: First, they demonstrate that one learning approach can be used for different real-world application scenarios, which is crucial for practicality. Second, they demonstrate that the domain of configurable software systems exhibit specific properties (e.g., the distribution of interactions) that make circumventing the tradeoff problem possible, allowing researchers and practitioners to develop efficient learning approaches by concentrating on a few important configuration options and their low-order interactions (i.e., interactions involving only a small number of configuration options, in our case, two or three). Third, we confirm that the learning approach that we use effectively identifies real interactions in configurable systems. Finally, the identified interaction patterns can be used as anti-patterns and help prevent or at least to anticipate the possible presence of performance interactions already in the early stages of the configurable systems development when architectural decisions are made. In total, these results contribute to the goals of our dissertation by providing insights into the nature and properties of feature interactions that may help in detecting or preventing feature interactions.

Technically, we use a state-of-the-art machine-learning algorithm to automatically learn performance-influence models (Section 2.4.3). To this end,

we have studied the properties of models learned for a set of 10 real-world configurable software systems. Based on the results of this study, we analyze the tradeoffs among the properties of the models and discuss their applicability in common practical use cases.

All experimental data and analysis scripts are available on a supplementary website.[1]

4.1 Motivation and Research Questions

In the domain of configurable systems, we lack empirical understanding of how strong the tradeoffs among prediction error, model size, and computation time of performance-influence models are. That is, while learning performance-influence models for real-world configurable systems, we do not know for which combinations of these properties we are able to effectively optimize that, in turn, may negatively influence practicability and general applicability of performance-influence models. Our goal is to quantify these tradeoffs by means of a series of experiments and to gain insights in to the characteristics of the configuration spaces of configurable software systems, for example, such as the relevance of different kinds of interactions and their influences on performance.

To illustrate the properties of performance-influence models and the tradeoffs among them, we will use two simple models, as shown in Figure 4.1a. These models describe the request throughput (requests per second, req/s) of the Apache Web server for a fixed standard benchmark. They are slightly simplified versions of the real models that we learned during our evaluation. The variables in the models represent binary configuration options (Section 2.1) of the Web server (such as, AccessLog, HostnameLookups, etc.), which can be either enabled or disabled (values 0 or 1).

It is important to note that both models in Figure 4.1a describe the same system, but have different size and prediction error. The table in Figure 4.1b lists the actual performance measurements of the Web server next to the predictions for the corresponding configurations using either Model A or Model B. Both models describe the main effects of the configuration options strongly influencing the system: In its default configuration (with all options disabled), the server can process 1000 req/s. However, with option AccessLog enabled, the throughput is decreased by 250 req/s. Enabling option HostnameLookups decreases the throughput by further 150 req/s. Both models accurately describe the performance of the first two configurations with one or the other option enabled (configurations 1 and 2 in Figure 4.1b). The third configuration contains

[1] http://fosd.net/tradeoffs/

(a) Two performance-influence models for the Apache Web server.

Model A: $1000 - 250 \cdot \underline{A}\text{ccessLog} - 150 \cdot \underline{H}\text{ostnameLookups}$

Model B: $1000 - 250 \cdot \underline{A}\text{ccessLog} - 150 \cdot \underline{H}\text{ostnameLookups}$
$+ 100 \cdot \underline{A}\text{ccessLog} \cdot \underline{H}\text{ostnameLookups} + \ldots$
$+ 2 \cdot \underline{A}\text{ccessLog} \cdot \underline{E}\text{nableSendfile} \cdot \underline{K}\text{eepAlive}$
$+ 1 \cdot \underline{E}\text{nableSendfile} \cdot \underline{F}\text{ollowSymLinks} \cdot \text{Ha}\underline{n}\text{dle}$

(b) Performance values predicted by the models.

#	Configuration	Measured Value	Predicted Value Model A	Predicted Value Model B
1	A	750	750	750
2	H	850	850	850
3	H, ***T***	850	850	850
4	H, ***I***	950	850	850
5	A, H	700	600	700
6	A, E, K	752	750	752
7	A, E, F, n	751	750	751
⋮	⋮	⋮	⋮	⋮

Figure 4.1: Two examples of performance-influence models for the Apache Web server and the corresponding predicted performance values. The underlined letters in the option names are used as abbreviations in the table (e.g., A stands for AccessLog). The slanted letters in the table denote configuration options that are not covered by either model. The predicted values that match the actually measured values are shaded in green, those that do not match are shaded in red.

the configuration option TypeConfig, which is not covered by the two models. Nonetheless, both models predict the configuration's performance accurately, because the configuration option has no measurable influence on performance. In contrast, configuration option InMemory in the fourth configuration has a substantial influence on the performance, and its absence in both models leads to prediction errors.

The two models differ in how they characterize minor variations and interactions among configuration options. By the individual influences of 250 req/s and 150 req/s of the options AccessLog and HostnameLookups, we could expect a combined performance penalty of 400 req/s, when both options are enabled, and in fact the first, simpler model assumes that. In practice though, we observed that both options interact, and their combined penalty is only 300 req/s (see Figure 4.1b, configuration number 5). By studying the system's documentation, we found that both options partially use the same data retrieved from a request, so the data are retrieved only once, but used by both options, and this reuse results in a higher throughput. While the first model produces an inaccurate prediction by ignoring this interaction, the second model covers this interaction (term AccessLog·HostnameLookups) and yields accurate predictions for more configurations, but at the cost of a more complex model (5 model terms instead of 2) and increased computation time.

Clearly, interactions can be important for prediction accuracy, but not all interactions may have a substantial influence on performance. The second model includes several interaction terms that only slightly alter the predicted performance by 2 or 1 req/s (e.g., EnableSendfile·KeepAlive), resulting in a relative accuracy improvement of at most 0.3 % and 0.1 % (compare the predicted values for configurations number 6 and 7). If we are fine with accepting such small prediction errors, we could ignore these interactions and work with smaller and simpler models. Note that such small-influence terms may be an indication for model overfitting, that is, the model describes measurement noise more than actual influences, at the cost of significant computation time and complexity of models.

4.1.1 Use Cases of Influence Models – A Discussion with HPC Experts

Over the last few years, we have been creating and using performance-influence models for various applications in a number of domains, including Web servers, code-analysis tools, and high-performance computing (HPC).[2] In an attempt to better understand requirements and use cases of performance-influence

[2] http://www.exastencils.org/

models regarding prediction error, model size, and computation time, we had several discussions with four of our collaborators from the HPC domain. The discussions constitute a basis for a lightweight explanatory analysis rather than a deep study in itself. Still, the discussions are informative enough to guide our analysis. All four HPC experts develop, analyze, and work with performance-critical applications on an everyday basis in areas, such as image and signal processing, automatic code generation, and differential-equations solvers, having 10 to 20 years of experience in the corresponding areas. They are working with the following systems: DUNE [Bas+06], HSMGP [Kuc+13], HIPA^{CC} [Mem+12], and SaC [GS06] (DUNE and HSMGP being also subject systems in our experiments; see Section 4.2).

To anchor our discussions with concrete data, we learned performance-influence models for the systems with which the experts were deeply familiar. Specifically, we took models at an early, intermediate, and late stage of the incremental learning process (Section 4.2.1): The early models were smaller, but more inaccurate (like Model A in Figure 4.1) than those in the later stages of the learning process (like Model B in Figure 4.1). We presented the models[3] to the experts explaining their general structure and asked the following questions:

1. What are use cases for the presented performance-influence models that you can think of?
2. What are acceptable tradeoffs among prediction error, model size, and computation time of a model with respect to these use cases?

The use cases mentioned by the experts can be grouped in two categories:

1. *Performance prediction.* Performance-influence models can help stakeholders of a system find the system's optimal configuration (for a give setting). For example, the SaC compiler has a default configuration that may have suboptimal performance for some hardware platforms. Learning a performance-influence model for each target platform allows developers to find the optimal configuration for each of them.
2. *Program comprehension and debugging.* Among the important program comprehension tasks, the experts named (1) confirming or disproving existing assumptions about influences of individual configuration options and (2) gaining new insights and deeper understanding of the performance of the system. For example, the HIPA^{CC} expert was surprised to see that pixelsPerThread configuration option had only a small influence on system performance.

[3]The models used in the discussions can be found in the appendix on page 138.

All experts stated that the decision about which model to chose or which property to optimize would depend on the given use case. For example, a model with the lowest prediction error is needed for performance prediction, whereas the model size and the computation time would be less important. For confirming a theoretical assumption about the influence of a certain configuration option, a simple model without interaction terms that is fast to learn could suffice. Regarding the usefulness of performance-influence models for this use case, one expert stated: *"We assumed many things all the time and now we can actually see them."* All experts were ready to accept the model with the highest prediction error (among the provided models), which was smaller and therefore easier to comprehend. For gaining deeper insights, such as finding and debugging unexpected interactions, all experts said that they required a model with interaction terms, but still of a tractable size. One expert stated: *"The smaller model is readily comprehensible compared to the larger model. It is more graspable."* For debugging purposes, computation time becomes important too, and the experts were ready to accept a model with a high prediction error if they could save a considerable amount of computation time during debugging.

The use cases identified in our discussions map very well to the model properties that we study here. Therefore, we are confident that exploring the tradeoff space spanned by these properties has an immediate practical use.

4.1.2 Tradeoffs in Machine Learning

The tradeoffs between prediction error, model size, and computation time are well known in the machine-learning community: A key concept is the bias-variance tradeoff [Dom00; SW11], which refers to the tradeoff between the size and prediction error of a model. Bias refers to the prediction error one encounters for a model with a fixed size and all data that is available. That is, for a small and simple model, the bias error may be high, because the model potentially does not explain the observed data to a full extent. Variance refers to the sensitivity of the model to the noise in the training data (such as measurement errors). More complex and larger models tend to fit the noise in the learning set, so that one may encounter large prediction errors when the model is applied to new data. So, learning a larger model may reduce its prediction error, but, at the same time, may complicate its understandability, simply because of its large size. Therefore, one of the main goals in machine learning is to find the sweet spot between underfitting (i.e., too simplistic models) and overfitting (i.e., too complicated models). However, often the search for this sweet spot is primarily driven by the minimization of the prediction error and does not take the comprehensibility of the resulting model into account.

Researchers in software engineering often apply machine learning without specifically considering the possible effects of the tradeoffs, or they just optimize for one criterion (e.g., prediction error) until other criteria leave the acceptable value ranges. For example, genetic algorithms have been used for multi-objective optimization to find configurations of configurable systems that satisfy multiple quality requirements [Say+13]. However, they trade computation time for prediction error, because most of these configurations are not valid. Other approaches aim solely at reducing the prediction error using classification and regression trees [Guo+13; Sar+15], but produce models that are hard to comprehend for humans. The goal of the mentioned approaches was never to balance or even explore the tradeoffs, but to optimize only for one property and ignore the others. As pointed out by our experts (Section 4.1.1), such approaches are only of limited practicality, because a different use case may require a different approach with yet another tool. We aim at filling this gap.

Notably, recent research in the performance-engineering community recognized the importance of the tradeoffs. In their recent work, Brosig et al. [Bro+15] explore alternative stochastic performance-modeling approaches regarding several low-level properties, such as the capability of handling loops in the analyzed software system. While we concentrate on regression models and more general properties, their work clearly connects to ours in showing that different stochastic models are suitable for different use cases and that it is important to have this information before performance analysis.

4.1.3 Research Questions

Our overarching goal is to explore the tradeoffs during the learning process of performance-influence models and gain insights into the performance behavior of configurable systems. For the purpose of our study, we use a state-of-the-art learning technique that is based on multivariate linear regression learning and forward feature selection [CS14]. We specifically aim at answering two research questions:

- **RQ1:** How significant are the tradeoffs among prediction error, model size, and computation time of the performance-influence models of real-world configurable systems?

- **RQ2:** Can these tradeoffs be balanced, such that the resulting models can be applied in different use cases, as identified by our discussion with experts?

4.2 Empirical Study

To answer our two research questions, we conducted an empirical study in which we created and compared different performance-influence models for 10 real-world configurable software systems. Next, we describe how we learned performance-influence models for our subject systems and how we analyzed their properties.

4.2.1 Learning Performance-Influence Models

For our experiments, we use a learning algorithm based on multivariate linear regression and forward feature selection (Section 2.4.3). It has proved to be accurate and effective for learning performance-influence models of real-world configurable systems [Sie+15]. During the learning process, we learn increasingly accurate models and keep track of the prediction error, model size, and computation time of each intermediate model, so that we can study how the properties evolve and how significant the tradeoffs among them are. Note that it is not our primary goal and contribution to invent a new technique for learning performance-influence models, but to use an established technique to study and leverage the tradeoffs among the three properties.

4.2.2 Measurement Procedure

To answer our research questions, we need to quantify the prediction error, size, and computation time of performance-influence models and tradeoffs among them. For this purpose, we define a number of measures.

4.2.2.1 Measuring Model Properties

The *prediction error* of a performance-influence model Π is the mean relative prediction error over the set of system configurations $\mathcal{C}$:

$$error(\Pi, \mathcal{C}) = \frac{1}{|\mathcal{C}|} \sum_{c \in \mathcal{C}} \left| \frac{\Pi(c) - measure(c)}{measure(c)} \right|,$$

where $c \in \mathcal{C}$ is a system configuration, $measure(c)$ is the performance of the configuration actually measured, and $\Pi(c)$ is the performance of the configuration predicted by the model Π. For example, for Model B of Figure 4.1a and the set of configurations number 1 through 7 in Figure 4.1b, the prediction error is 0.03 (or 3 %), mainly because the model wrongly predicts the performance value for the configuration number 4.

We define the *model size* as the number of configuration options in every term of the model. The model size of a performance-influence model Π and its set $terms(\Pi)$ of terms is defined as follows:

$$modelSize(\Pi) = \sum_{t \in terms(\Pi)} size(t),$$

where $t \in terms(\Pi)$ is a term of the model Π and $size(t)$ is the number of configuration options in t. For example, Model B in Figure 4.1a has a size of 2, because it contains two terms and each term consists of only one configuration option.

The *computation time* of a model is equal to the CPU time used by the algorithm to learn the model.

4.2.2.2 Measuring Tradeoffs

To characterize the tradeoffs between the three properties quantitatively, we use the *Area Under the Curve* (AUC) measure. To calculate the AUC for a tradeoff between two properties, we plot one property against another and calculate the integral of the resulting curve. The integral value is the corresponding AUC. We normalize the property values in the range $[0, 1]$ before calculation, therefore, the corresponding AUC is a value in the same range.

Figure 4.2 illustrates three example tradeoff curves and the corresponding AUC values for different kinds of tradeoffs between computation time and prediction error properties.[4] If the two properties are in inverse relationship (Figure 4.2a), then a relatively large (small) positive change in one property always results in a relatively large (small) negative change in the other property.

[4]The tradeoffs for other property pairs are calculated in the same way.

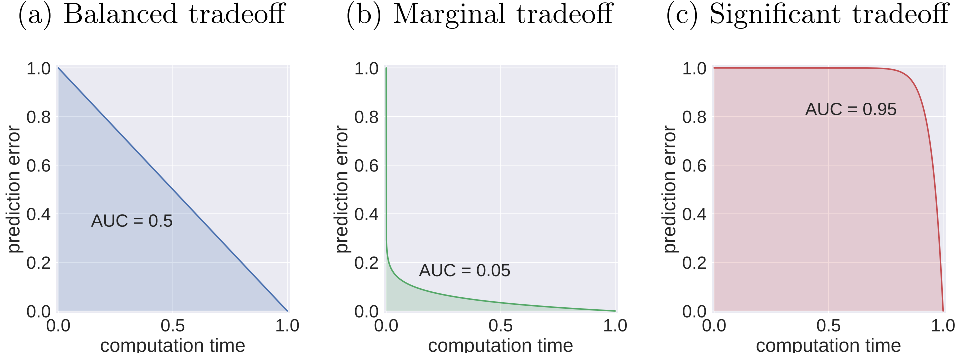


Figure 4.2: Example tradeoff curves and corresponding AUC values.

That is, the tradeoff between these two properties is balanced. The AUC value for a balanced tradeoff like this is close to 0.5.

AUC values that are smaller that 0.5 indicate a shift to a marginal tradeoff (Figure 4.2b), which is favorable in our setting: A small initial increase in computation time already leads to a large initial decrease of prediction error. Conversely, a large initial decrease in prediction error requires only a small increase in computation time. A marginal tradeoff would allow us to learn smaller and more accurate models faster.

AUC values that are larger than 0.5 indicate a shift to a significant tradeoff (Figure 4.2c), which is unfavorable in our setting: A large initial increase in computation time would lead to a small initial decrease in prediction error. A significant tradeoff means that we would have to invest much computation time or accept large model sizes if we want to learn a model with low prediction error.

By calculating the AUC values for each subject system and each pair of model properties, we can determine which kinds of tradeoffs—balanced, marginal, or significant—are present in the subject systems.

4.2.3 Subject Systems and Experimental Setup

Table 4.1: Subject systems; $|O|$: number of configuration options, $|C|$: number of configurations. The number of configurations is less than $|O|^2$ because of dependencies among configuration options.

System	**Domain**	$\|O\|$	$\|C\|$	**Performance metric**
AJStats	Static analysis	20	30 256	Analysis time
Apache	Web server	9	192	Response rate
BDB-C	DBMS	18	2 560	I/O time
BDB-J	DBMS	26	180	I/O time
Clasp	ASP solver	19	700	Solving time
DUNE	Stencil code	31	2 304	Solving time
HSMGP	Stencil code	32	3 456	Solving time
LLVM	Compiler	11	1 024	Optimization time
Lrzip	Archiving tool	19	432	Compression time
x264	Video codec	16	1 152	Encoding time

As subject systems, we selected 10 real-world configurable software systems of different sizes, complexities, and from different application domains, as summarized in Table 4.1. The systems differ in the number of configuration options

as well as in the number of resulting configurations. They are implemented in different programming languages and support configuration at compile time, load time, or both. We used the systems' documentation to determine which configuration options may have influence on performance.

For each subject system, we measured performance of *all valid*[5] *configurations* (whole-population analysis) using standard benchmarks for the respective domain. We repeated the measurements multiple times to control for the measurement noise (see Section 4.3, for more detail). Based on the benchmark data, we learned performance-influence models of the subject systems using the machine-learning algorithm described in Section 4.2.1. In practice, one would not measure all configurations but only a sample, due to time constraints when gathering a learning set for the machine-learning process, which is demonstrated by Siegmund et al. [Sie$^+$15]. However, for the purpose of our study, we were specifically interested in exploring the full range of tradeoffs, meaning that we were also interested in the maximum possible accuracy of the resulting performance-influence models (to see the maximum possible extent of the corresponding tradeoffs). So, we used the benchmark results for all configurations as the learning set. The usage of the largest possible learning set also neutralizes one of the possible reasons for overfitting: non-representative sampling of the learning set.

The learning procedure was conducted on a dedicated server with an Intel Xeon E5-2609, 2.5 GHz and 128 GB RAM, running Ubuntu 14.04. To obtain accurate models, but not to run the computation indefinitely, we terminated the learning procedure as soon as the score of the current candidate fell below 0.05 (see Section 4.2.1). From our experience, this ensures that we learn all actually existing performance influences, but largely avoid measurement errors manifesting in the model (i.e., overfitting). If we had continued learning, we would have essentially learned the measurement error.

After each iteration of the learning algorithm, we saved the current model (see Section 4.2.1) to study the evolution of the model properties. For each model, we calculated the prediction error, its size, and the computation time. To rule out the time measurement bias caused by warm-up effects and computation-setup overhead, we subtracted the time of the first learning round from the elapsed-time measurement. Considering that the initial learning rounds are the fastest, this subtraction does not introduce any relevant deviation from the actual computation time.

[5] Not all combinations of configuration options are valid system configurations, because of dependencies among the configuration options.

4.2.4 Results

For most subject systems, we obtained highly accurate models at the end of the learning procedure. The largest prediction error is 6.25 % for BDB-C. In Figure 4.3, we show how the prediction errors evolve during the learning process (the solid blue line). The AUC lies in the range between 0.07 and 0.29, indicating a marginal tradeoff between computation time and prediction error. That is, we may be able to to learn more accurate models faster.

The size of the learned models varies substantially from system to system: Among the models with the highest prediction accuracy, the smallest model has the size of 12 (BDB-J) and the largest model has the size of 544 (DUNE). In Figure 4.4, we show how the model size evolves for each system during learning.

Due to the dependency between computation time and model size, the tradeoff between model size and prediction error is similar to the tradeoff between computation time and prediction error, as we show in Figure 4.5, with similar AUC values between 0.13 and 0.29. As one would expect, more accurate models have larger sizes and, conversely, smaller models have a higher prediction error.

4.2.5 Discussion

Research Questions

Our results confirm that there is, as expected, a tradeoff between computation time and prediction error: investing more time reduces the prediction error. However, our results also show that this tradeoff is rather marginal, with AUC smaller than 0.3, for all systems. This insight is surprising and is good news for the domain of configurable systems, because it means that it is possible to efficiently learn relatively accurate models. In Figure 4.3, we can observe how the prediction error drops quickly to a certain level early on in the learning process, whereas later the accuracy improvement saturates.

Between model size and computation time, we observe a strong positive dependency instead of a tradeoff. This result was to be expected due to the incremental nature of our algorithm, which monotonically increases model size by learning an additional term in each round. In fact, most machine learning mechanisms operate iteratively to incrementally approximate an optimal solution, simply out of necessity to handle the complexity of the huge search space (for $|o|$ options, there are $|o|$ possible main influences, $|o| \cdot (|o| - 1)/2$ possible pairwise interactions, and an exponential number of higher-order interactions among more than two options). With this huge search space, it is generally not feasible to use an exact, analytical approach.

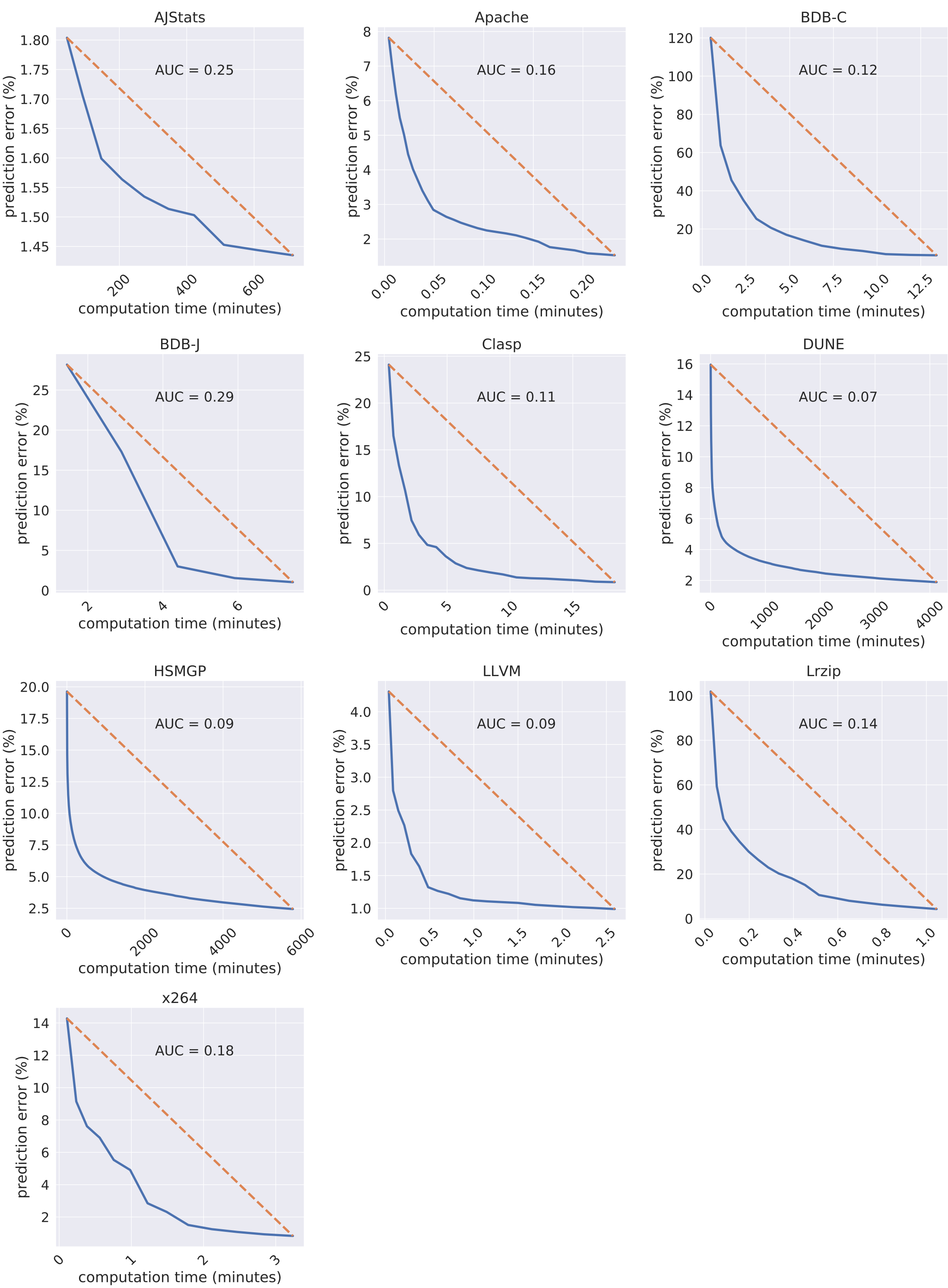


Figure 4.3: Time–error tradeoff.

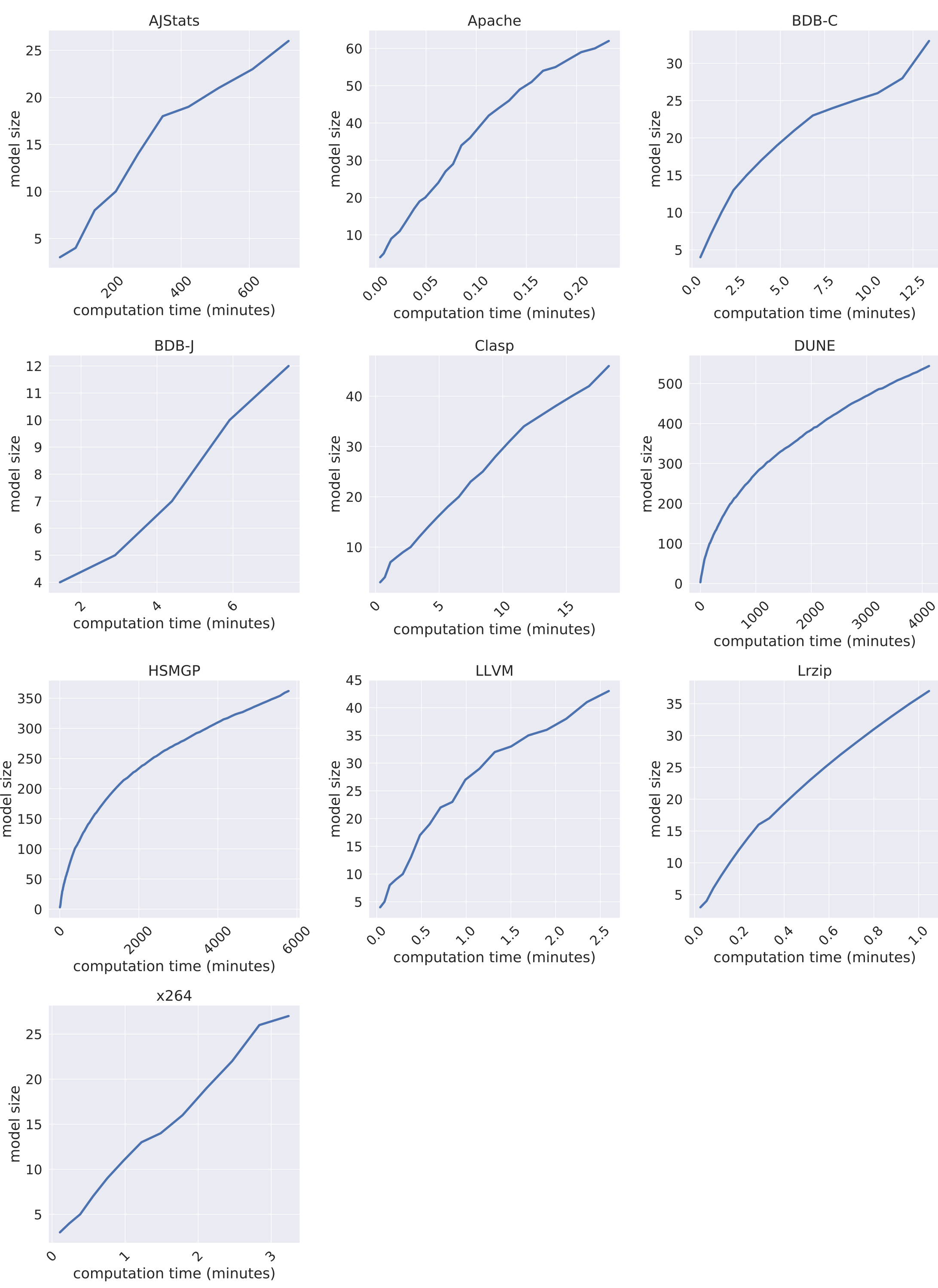


Figure 4.4: Time–size dependency.

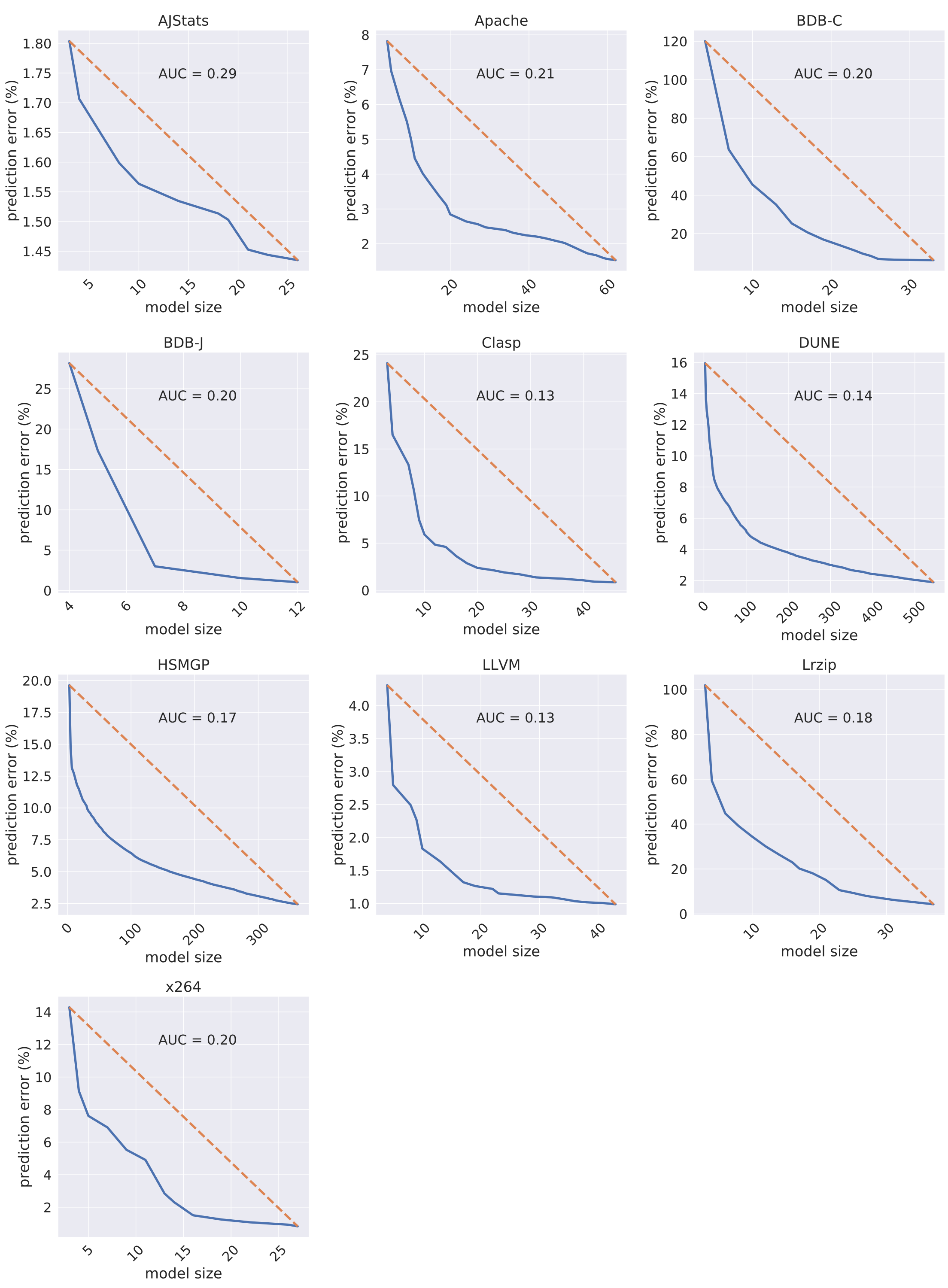


Figure 4.5: Size–error tradeoff.p

Due to the strong positive dependency between computation time and model size, we see also a marginal tradeoff between model size and prediction error, much like we saw it between computation time and prediction error. While, again, we can learn more accurate models that are larger, the increased accuracy benefits are small. The fairly small models, early in the learning process, can characterize the performance of configurable software systems already fairly accurately.

So, with respect to the first research question (RQ1), we conclude that, for learning performance-influence models for configurable software systems, the tradeoffs between computation time and prediction error and between model size and prediction error are marginal; furthermore, model size and computation time have a strong positive dependency. What this means for practice is that learning simple models can be suitable to serve multiple or even all use cases, as identified in our discussions with experts (RQ2). This is an important insight for the research community: Although, the tradeoffs are known in the machine-learning literature, it was previously unclear to what extend they affect learning performance-influence models for software systems, that is, whether a large configuration space can only be accurately described with complex models learned with significant resource investment. Fortunately, we were able to show that this worst-case scenario is not the rule for real-world software systems.

Understanding the Tradeoffs: Influence of Interactions

A followup question that arises from our results is why the tradeoffs are so marginal. That is, why are accurate models also simple and can be learned in feasible time in this domain? To answer this question, we additionally analyzed our experiment's data for the role of interactions. A hypothesis is that the tradeoffs are marginal because performance in software systems can be described with few main effects, whereas many options and most potential interactions do not affect performance much.

Specifically, we analyzed what kinds of terms are learned in each round and how do they contribute to the accuracy of the model. We distinguish between influences of individual options (term size 1), influences of interactions between two options (term size 2), influences of interactions among three options (term size 3), and so forth. We plot our observations in Figure 4.6. Each plot shows how with additional time (left to right) additional terms are learned and how the prediction error is reduced. We specifically distinguish terms of different sizes using different background colors. For example, for AJSTATS, the model with the prediction error of 1.7 % (bottom x-axis) contains four model terms representing the influence of only individual options (i.e., four model terms of

terms size 1 each). Then, during the learning process, a fifth term is added describing an interaction among four options, decreasing the prediction error to 1.6 %. Note that prediction error and computation time share an axis, but the scales are independent: prediction error reduces linearly, but computation time grows superlinearly. So, the final marginal reductions in prediction errors typically require significant investment in computation time.

Figure 4.6 reveals that few mostly small model terms are sufficient to build relatively accurate models. Considering interactions among options is important to achieve accuracy, but high accuracy can be reached without considering a huge number of interactions among many configuration options. For the most of the systems, 10 model terms with size 3 or lower are sufficient to build a model with a prediction error of under 5 %. Adding more interaction terms of larger size later in the learning process results in marginal improvements only. This explains why we did not observe strong tradeoffs among prediction error and computation time earlier. In fact, the substantial increase of the share of larger model terms and the simultaneous growth of the total number of model terms needed for very high accuracy may be an indication for the overfitting effect. These additional model terms may describe measurement noise rather than the actual performance behavior of the system.

Note that the measured computation times should be considered in relation to the corresponding prediction errors and not as absolute values. Consider the Apache case study, which is one of the smallest in terms of configuration options (9) and in terms of configurations (192). To calculate a performance-influence model with 3 % prediction error for this system takes about 3 seconds. But calculating a slightly more inaccurate model with 7 % prediction error is 6 times faster. So the developer can save relatively much time by stopping the learning process at earlier stages. We have a similar picture for one of the largest systems, HSMGP, with 32 configuration options and 3456 configurations. For this system, we can compute a performance-influence model with 7 % prediction error 13 times faster than a model with 3 % prediction error (which needs 3 days to compute). The same pattern applies to all our subject systems irrespective of their size: The time needed to achieve an acceptable prediction accuracy of the model is always multiple times less than the time needed for further marginal increases of the prediction accuracy. These results suggest that the same may apply for very large configuration spaces.

We conclude that the marginality of the tradeoffs can be explained by the fact that interactions among three or more configuration options have only a low influence on the performance of configurable software systems. Note that our result regarding interactions primarily describes a characteristic of performance in configurable software systems, not of machine learning in general.

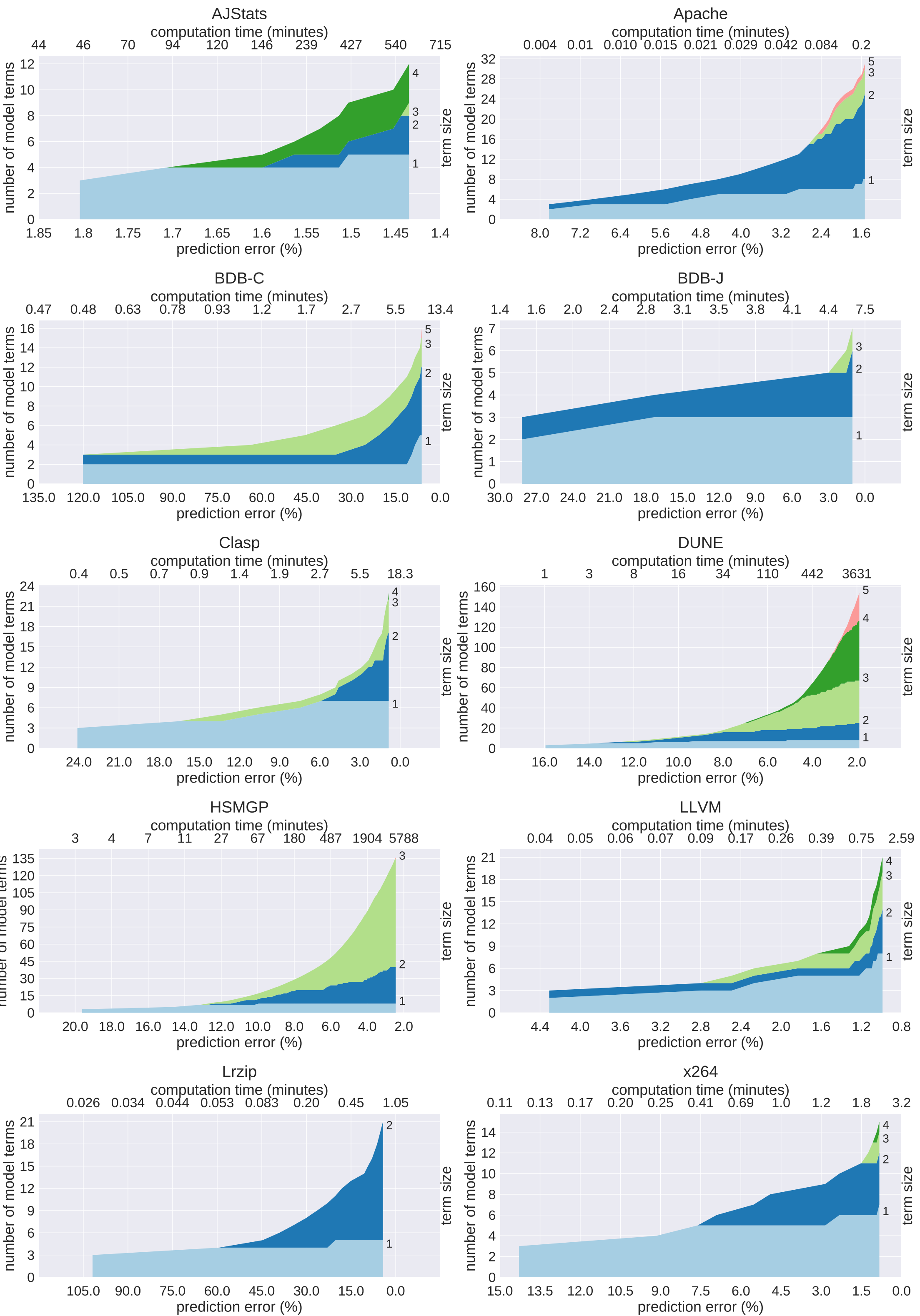


Figure 4.6: Shares of interactions of different size in performance-influence models and their influence on the prediction error.

Finally, the results of our analysis have implications for sampling and learning algorithms. For example, if an algorithm considers interactions among configuration options, it can concentrate on interactions among three configuration options and fewer. Excluding interaction of larger size reduces the search space and may improve the performance of the algorithm without sacrificing accuracy. The reduction of the search space may be substantial if we consider that the number of potential interactions grows exponentially with the number of configuration options. Already for 10 optional configuration options without additional constraints we get 1018 ($2^{10} - 6$) potential interactions. Considering the emergent nature of interactions, we have to admit that any of these potential interactions may actually exist. But, as our results show, for real-world configurable systems, the number of actually relevant (i.e., those that have influence on performance) interactions is much smaller than the number of potential interactions. For example, for the smallest subject system APACHE with 9 configuration options (which, according to APACHE'S documentation, all may have influence on performance), the number of potential interactions is 124 (considering the dependencies among the configuration options). By considering only 3 interactions of 124, we already can learn a relatively accurate performance-influence model with only 4 % prediction error. That is, only 2 % of all potential interactions do actually exist and have relevant influence on performance of the APACHE system.

Analysis of the Influence of Configuration Options and their Interactions

To identify commonalities among the influential configuration options and their interactions across the subject systems, we conducted an exploratory analysis by reading the systems' documentation and the source code to build hypotheses, and talking to the systems' developers in unclear cases. We proceeded iteratively until we were able to explain the influences. Finally, we formulated the commonalities as patterns that explain the influences, such as, *dominant configuration option*, *data pipeline*, and *workload tuning*. A complete overview of the analyzed configuration options and their interactions is given in the appendix on page 134.

Dominant Configuration Option. During our analysis, we found that the most influential configuration option for APACHE is KeepAlive, which has an influence of 876.61, on average. This value denotes that enabling KeepAlive increases the response rate of the Web server by, on average, 876.61 responses per second (cf. performance metric in Table 4.1). That is, enabling this configuration option increases the performance of the Web server. The performance

increases, because this configuration option enables the *persistence connection* functionality of the HTTP 1.1 protocol, which enables sending multiple requests over the same TCP connection. This functionality saves the overhead of establishing a separate connection for each request. Note that the influence of KeepAlive is larger than the sum of absolute influences of the next three most influential options. So, KeepAlive is a *dominant* configuration option, which largely determines the performance of the system and, consequently, the prediction error of the corresponding performance-influence model. We found that such dominant configuration options are also present in other subject systems: S1MiB in BDB-J, heuristicUnit in CLASP, Smoother_GSACBE in HSMGP, etc. We also observed that the dominant configuration options interact with other configuration options in highly influential interactions in all subject systems (except for X264). Some dominant configuration options can be identified based on domain knowledge and documentation. For example, the APACHE documentation states that enabling KeepAlive can result in almost 50 % speedup.[6] As our data suggests, knowing the dominant configuration options from the documentation, we can also assume that they interact with other configuration options.

Data Pipeline. Regarding the most influential interactions, we found that, for CLASP, DUNE, HSMGP, LLVM, and X264, the interactions arise due to the architecture of the systems that prescribes which system modules/algorithms (enabled through the configuration options) supply input data to other system modules/algorithms. That is, the architecture constitutes a *data pipeline*, and the parts of this pipeline are determined by configuration options. For example, in the CLASP solver, the options eq and satPreproYes enable preprocessing steps that can reduce the initial problem, such that the solving algorithm can find a solution for this problem faster. Therefore, the most influential interactions for this system are among the preprocessing options and solver heuristics, for example, eq · heuristicUnit.

We observe a similar picture for DUNE and HSMGP. These systems are built such that the input data are preprocessed before they reach a solver, and the output of the solver is post-processed. The corresponding configuration options (pre*[7] and post* for DUNE; numPre* and numPost* for HSMGP) define the number of these pre- and post-processing steps (e.g., if pre1 is enabled, one preprocessing step is made). Each pre- and post-processing step introduces a computational overhead, which increases the solution time. Therefore, we observe that the most influential interactions of these systems include pre-,

[6] https://httpd.apache.org/docs/2.4/mod/core.html#keepalive

[7] pre* denotes all configuration options starting with “pre”.

post-processing and solver-related (or smoother-related in the case of HSMGP) configuration options.

Data pipelines also explain why larger interactions include partly the same configurations options as smaller interactions. That is, why for a set of interacting options there exist interactions for its respective subsets. The reason is that smaller interactions describe smaller parts of the pipelines and larger interactions include these smaller parts. For example, this is the case in DUNE with cells*, pre*, and post* options, and configuration options for solvers, which build up data pipelines (and, consequently, interactions) of size up to 5.

We assume that, in the case of a data-pipeline architecture, developers can deduce from the system's architecture which configuration options are likely to interact. Therefore, identifying these interactions using performance-influence models can be seen as a sanity check or regression test if parts of the architecture are changed. This use case corresponds to program comprehension and debugging as described in Section 4.1.1.

Workload Tuning. Furthermore, we found configuration options that adjust the workload by tuning the main data processing algorithm of the system. These tuning configuration options often interact with configuration options denoting processing algorithms. Configuration options level* in LRZIP, for example, determine the compression level for the data compression algorithms. With a growing compression level the compression time grows too. Therefore, interactions among configuration options that specify the compression level and the compression algorithm arise.

Domain-Specific Interactions. Other interactions that we analyzed had a more domain-specific nature. For example, the interaction between inline and licm configuration options, which enable code optimizations in the LLVM compiler. The inline optimization inlines code of methods at the call sites and licm moves code out of loops. The peculiarity of LLVM is that these optimizations can be executed in arbitrary order (determined by the order of the corresponding command line parameters). If inlining is performed before loop optimization (which was the case in our experiment) there may be some code that gets inlined into loops. Consequently, the loop optimization has more code to process and, consequently, requires more computation time. As a result, we observe a performance interaction between inline and licm.

Another example of a domain-specific interaction is an interaction in X264 between no_fast_pskip, which disables Fast-P-Skip optimization, and ref_9, which sets the number of reference frames. Both configuration options tune the main encoding algorithm, but Fast-P-Skip optimization is more effective

with more reference frames. This dependency between the two configuration options induces an interaction between them.

The two examples of the interactions show that their domain-specific nature does not allow us to describe them in general terms (like data pipelines) and requires deep understanding of the system's workings to explain their influences.

Summary. Based on domain knowledge, systems' documentation, and information provided to us by systems' developers, we were able to explain the most influential configuration options and interactions of the subject system. Furthermore, we identified several interaction patterns across multiple subject systems providing further insights in the domain of configurable systems. The patterns explain how decisions about a system's architecture may lead to the emergence of performance feature interactions. They can be used as performance anti-patterns [Cor$^+$10] and help to prevent or at least to anticipate the possible presence of performance feature interactions already in the early stages of the configurable systems development when architectural decisions are made.

4.3 Threats to Validity

Internal Validity. To learn a performance-influence model, we rely on benchmark measurements that are susceptible to measurement errors. There is a threat that these measurement errors may bias the results of the learning procedure, such that the resulting model may not properly characterize the actual performance of the system. To investigate the potential influence of measurement errors on the prediction error of a performance-influence model, we conducted a separate experiment. We added random noise (representing measurement errors) to the original measurements for our subject systems[8] and repeated the learning process. Then, we compared the prediction error of the noisy models to the prediction error of the original models to see the potential influence of measurement errors.

A noise value was computed for each original measurement value by randomly sampling a value from a normal distribution. The parameter σ of the normal distribution specified the standard deviation of the noise values: the larger σ, the larger the noise value can be (i.e., the larger the measurement error). We set the initial value of σ to 0.75 (the average standard deviation of the original measurement values of our subject systems). For each subject system, we doubled σ and repeated the learning process five times to simulate the influence of increasing measurement errors.

[8]We excluded HSMGP, because conducting this additional experiment with the system would have taken several months of computation time.

Analyzing the noisy models, we found that most of them had approximately the same prediction error as the original models until σ (i.e., potential magnitude of the simulated measurement errors) reached a value of 6 (i.e., the potential errors were 8 times larger, than the errors of our original measurement). From this result, we can conclude that the learning algorithm that we used is robust against realistic measurement errors.

Our simple model size measure (Section 4.2.2.1) could be further refined to reflect the complexity of the model more accurately, for example, by considering the number of the interaction terms. We decided against this refinement, because we do not have enough empirical evidence to quantify the influence of the interaction terms on the complexity of a model. Still, our interviews (Section 4.1.1) indicate that our simple model size measure quantifies the complexity of the models rather well.

External Validity. Our results are not automatically transferable to larger models or other subject systems. However, to increase the external validity of our study, we collected 10 real-world systems of different sizes, complexities, and from different application domains. Furthermore, the subject systems differ in the number of configuration options and in the number of resulting configurations. They are implemented in different programming languages and support configuration at compile or load time, or both. As we observed differing results regarding the number of interactions, but found a similar picture regarding the model properties, we gain some confidence that our results are general to a certain extent, because our selection of subject systems covers heterogeneous systems of important domains.

The use of a particular machine-learning technique, namely multivariate linear regression with forward feature selection, may limit the generalizability of our results.

4.4 Related Work

Our goal was not to propose a certain machine-learning technique for learning performance-influence models of configurable systems, but to explore the design space of performance-influence models with respect to prediction error, computation time, and model size. Next, we discuss learning techniques, feature-interaction-detection approaches, and model-size definitions related to our study.

Learning. There are a number of machine-learning techniques that can be used to learn performance-influence models. Classification and regression trees repre-

sent a successful method to learn prediction models from a learning set [SC09]. Guo et al. [Guo+13] applied this technique to configurable software systems. They required only a limited computation time and achieved a high prediction accuracy. However, decision trees and the related forests [LW02] have two drawbacks: First, they model variants and not configuration options and their interactions, which hinders comprehension in that the influence of individual configuration options and their interactions on performance is not explicitly denoted; Second, decision trees are unstable in that even small changes in the training set can lead to vastly different models (in contrast, we did not observe the instability problem with linear regression in our experimental setting). Other learning techniques using support vector machines [SC08], Bayesian networks [Ben07], evolutionary algorithms [Sim13], or Fourier transforms [Zha+15] trade off even more comprehensibility of the underlying prediction models in return for prediction accuracy or focus more on finding the fastest configuration or reducing the number of samples instead of quantifying the influence of individual configuration options and their interactions on performance. Hence, there is only a limited choice of techniques that let us explore the tradeoffs among prediction accuracy, computation time, and model size of performance-influence models. Brosig et al. [Bro+15] study stochastic performance models acquired with different model generation approaches, but the main focus lies on the accuracy and efficiency of the generation approaches and the corresponding tradeoffs. Furthermore, the applicability of the models with regard to their complexity is not considered.

Performance Engineering. The field of performance engineering aims at modeling non-functional properties of a system to evaluate if these properties satisfy a given set of requirements [Bal+04; BVK13; Poo00]. We do not focus on a single system, but on a potentially exponential number (in the number of configuration options) of system configurations, which lifts the problem to a higher level of complexity. But what is more important, we are primarily concerned with presenting the influence of the system's configuration options and their interactions on the system's performance to a user in a concise and understandable way without sacrificing the model's prediction accuracy.

Feature Interactions. Interactions among configurations options are the key to learning accurate performance-influence models. Nhlabatsi et al. [NLN08] and others as well as Calder et al. [CM06] surveyed detection mechanisms for feature interactions. Other approaches focus on properties such as semantic correctness [Ape+11; Cla+10] and global system behavior [Pre04] in the presence of feature interactions. Zhang et al. [Zha+16] propose a mathematical model

of performance-relevant feature interactions and describe two algorithms to automatically detect them and quantify their influence. A number of techniques for finding performance interactions using sampling heuristics in combination with linear programming have been proposed in previous work [Sie+12; Sie+13]. Here, we focus not on detecting interactions, but on the effect of interactions on model size, computation time, and prediction error of the learned performance-influence model. We are not aware of any other studies that explore the properties of performance-influence models of configurable systems in this way.

Model Complexity. There is no single accepted definition and measure for model complexity. One approach is to define model complexity through its size. The larger the model the more difficult it is to comprehend and to use. Several measures have been proposed to measure such model size: Schruben et al. [SY93] proposed a measure based on McCabe's software complexity measure, Wallace [Wal87] defined a similar measure. Although the given definition of complexity is similar to ours, the proposed measures cannot be applied to our models, because they have been developed for graph-based model representations. Another approach is to define the size of a model through its susceptibility for overfitting, that is, the more complex a model, the higher its ability to fit random noise in the data [MP04]. Although this definition describes an important property of a model, it does not fit the research questions that we addressed in this study.

4.5 Summary

Performance-influence models help developers and users to better understand performance characteristics of complex configurable software systems. They quantify the influence of individual configuration options and their interactions on systems' performance. An ideal performance-influence model should have low prediction error, short computation time, and small model size. However, there are usually tradeoffs between these properties that do not allow to optimize for all of them at once.

In our discussions with four domain experts, we identified two important practical use cases for performance-influence models: performance prediction and program comprehension. Performance prediction would require a model with the lowest possible prediction error; program comprehension would require a model of small size and short computation time. Since it is unclear to what extent the tradeoffs among prediction error, model size, and computation time affect the applicability of performance-influence models in these two use cases, we conducted an empirical study with the goal of systematically exploring the

properties of the configuration spaces of 10 real-world configurable software systems. Our results show that there are indeed tradeoffs between prediction error and model size and between prediction error and computation time. However, we found that these tradeoffs are rather marginal, such that accurate and also simple performance-influence models can be learned in feasible time, which is surprising and good news.

To further understand why efficient learning is possible, we analyzed the learned performance-influence models regarding the influences they capture. We found that individual configuration options and interactions between, at most, three options explain most of the performance variances. That is, identifying and learning the influence of interactions between more than three options will likely improve the prediction accuracy only by a tiny fraction, but will still increase computation time and model size considerably. This finding have an immediate practical consequence for the techniques that aim at detecting feature interactions or that rely on sampling: using our findings they may focus on feature combinations that include two or three features, because they may highly likely give rise to the influential feature interactions.

To gain further insights into the nature of interactions among configuration options, which is one of the goals of our dissertation, we investigated why the systems' configuration options and their interactions have that particular influence on performance, which we observed in the experiments. We traced the reasons for the observed influences back to the architecture of the systems and interdependencies among system components. Therefrom, we extracted general interaction patterns, such as dominant configuration option and data pipeline. The patterns explain how decisions about a system's architecture may lead to the emergence of performance feature interactions. They can be used as anti-patterns and help prevent or at least to anticipate the possible presence of performance feature interactions already in the early stages of the configurable systems development when architectural decisions are made.

In the next chapter, we rely on performance-influence models to study relations among different types of feature interactions. Therefore, showing practicability and general applicability of performance models is a prerequisite for the techniques described in the next chapter to have the same properties.

CHAPTER 5

On the Relation of External and Internal Feature Interactions

This chapter shares material with the following publication: S. Apel et al. "Exploring Feature Interactions in the Wild: The New Feature-interaction Challenge". In: *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 2013, pp. 1–8 [Ape+13a]

In this chapter we address one of the ultimate questions of our dissertation: Whether we can efficiently predict certain kinds of feature interactions (such as interactions influencing performance of configurable systems) based on the information about other kinds of feature interactions. To this end, we introduce two classes of interactions according to their visibility: (1) *external feature interactions*, which can be identified by observing the external behavior of a system, such as performance; and (2) *internal feature interactions*, which can be identified by analyzing or interpreting the source code of a system, for example, using control-flow analysis. A key hypothesis is that there is a relation between internal and external interactions, and that we can make use of this relation to automatically identify external interactions by identifying internal interactions in a fast and efficient way. For example, multiple function calls from one feature to another (internal feature interactions) can result in a performance overhead. This performance overhead arises only if both—the caller and the callee features—are present in a configuration (external feature interaction). This way, the internal interaction is related to its external counterpart. This relation, if present, would give us hints about the existence of external feature interactions based on the internal ones. In this chapter, we follow up on this

idea and report on an exploratory case study in which we investigated the *control flow* among features and its relation to *performance* feature interactions. We conjecture that by supplying the performance-prediction procedure with hints about which feature combinations are more likely or less likely to exhibit external feature interactions, the procedure can be made more focused on finding actual interactions.

Technically, we use a state-of-the-art machine-learning technique (Section 2.4.3) to learn external (performance) feature interactions among features of two subject systems: MBEDTLS and SQLITE.[1]. Furthermore, we manually inspected the code of the systems and checked whether the learned performance interactions actually exist and whether they are actually caused by the interplay of the corresponding features, and not just misinterpreted artefacts of measurement bias or environment noise. Using a variability-aware control-flow analysis augmented by manual code inspection (Section 5.3.2), we also identify control-flow interactions among the features of MBEDTLS and SQLITE. That is, we identified the code locations where the features pass the control to one another.

Comparing the set of internal (control-flow) interactions with the set of external (performance) interactions revealed that those features that interact internally also interact externally (Section 5.4.3), which is in line with our expectation. Using the identified relation, we were able to substantially shrink the search space of performance feature interactions (Section 5.5). Furthermore, we made first steps towards developing an automated predictor for identifying features that are likely to interact externally based on the set of internal interactions, although, with mostly negative results (Section 5.3.5). To the best of our knowledge, this is the first case study that analyzed both the external and the internal feature interactions for the same system and investigated possible connections between internal and external interactions.

5.1 Visibility of Feature Interactions

Different levels of visibility of feature interactions have been discussed in the literature [Bru05; LA11; WE05; WEL07]. Feature interactions may appear at the level of the externally-visible behavior, which we call henceforth external feature interactions, for short, and at the level of the internal properties of a system, which we call henceforth internal feature interactions, for short.

[1] `https://tls.mbed.org/` `https://www.sqlite.org/`

5.1.1 External Feature Interactions

Feature interactions, as such, have been described first in the domain of telecommunication systems [Bow+89]. There, feature interactions have been described as inadvertent deviations from the expected externally-visible behavior of a system. Basically, the behavior of a system composed of features is more (or less) than the sum of the (well-known and well-defined) behaviors of the individual features involved.

Functional Interactions. Two lines of research on external feature interactions emerged in the recent years. One line of research is concerned mainly with interactions that violate the functional specification (Section 2.2) of a composed system, which includes all kinds of bugs, including segmentation faults, race conditions, and deadlocks. We call these interactions *functional* feature interactions. [2]

Consider Hall's e-mail system with features for message encryption and forwarding as an example [Hal05]: While encryption and forwarding operate individually as expected, their combination gives rise to an undesired feature interaction. The interaction occurs if one host sends an encrypted message to a second host that forwards the message automatically to a third host. If the second host does not have the public key of the third host, it forwards the message in plain text. The reason is that the forwarding feature has been developed independently of the encryption feature, so it does not "know" whether an e-mail is encrypted. This interaction is clearly undesired: it contradicts what we expect from the encryption feature, and it violates the specification of the encryption feature (if there is one), which states that messages that have been encrypted initially must never be sent unencrypted over the network.

Finding feature interactions that violate the functional specification of a composed system boils down to combining analysis techniques, such as testing, static analysis, and model checking, with strategies to reduce the analysis effort in the face of feature combinatorics (e.g., sampling, feature-based and variability-aware analyses). In the e-mail example, one could create (a subset of) feature combinations and analyze whether messages are sent unintentionally in plain text over the network using the following temporal-logic specification [Ape+13b]:

$$\begin{gathered}\mathbf{AG}\ \big(\mathsf{recv}(\mathsf{msg}\ m) \wedge m.\mathsf{isEncrypted}\big) \Rightarrow \\ \big((\mathsf{send}(\mathsf{msg}\ m) \Rightarrow m.\mathsf{isEncrypted})\ \mathbf{R}\ \mathsf{send}(\mathsf{msg}\ m)\big)\end{gathered} \tag{5.1}$$

This specification states essentially that all incoming messages (recv) that were encrypted (isEncrypted) must be encrypted when leaving the system (send).

[2] The concept of *interaction faults* used in the interaction-testing community is very similar [GC11; Joh+12; KWG04].

Non-Functional Interactions. Another line of research is concerned with interactions that influence non-functional properties of a composed system, including performance, memory consumption, energy consumption, etc. We call these interactions *non-functional* feature interactions. Non-functional feature interactions have been discussed in the literature with regard to explicit and implicit specification [RGP12; Sie+12; WE05]. If we have an explicit specification of the desired non-functional properties of a system at hand (e.g., the maximum latency), we can typically decide whether a given feature combination satisfies the specification (e.g., whether it is fast enough).

If we do not have a specification at hand, it is still useful to reason about non-functional feature interactions. An assumption that guides work on the prediction of non-functional properties [Sie+15; Sie+12; Sie+13] is that each feature has an influence on the non-functional properties of a system and that this influence can be quantified. Features are considered not to interact, if their contributions to a given non-functional property can be simply aggregated (e.g., by adding their execution times or taking the maximum peak performance). This statement is actually an implicit specification that serves to detect feature interactions, and to make predictions more accurate [Sie+12]. In this sense, feature are considered to interact, if a non-functional property of the composed system diverges from the aggregation of the individual contributions of the features involved, for example, in that the performance goes substantially down.

For example, many features in a database system can be freely combined to tailor the system to the specific needs of a customer or application scenario, including encryption, compression, and various kinds of index structures and locking strategies [Ros+09]. However, there are subtle feature interactions that lead to performance abnormalities, for example, when a coarse-grained locking strategy hinders query evaluation and optimization [Moh92].

Detecting non-functional feature interactions is, at least, as challenging as detecting functional feature interactions. Typically, various techniques for the measurement, prediction, and modeling of non-functional properties are combined with strategies to reduce the analysis effort in the face of feature combinatorics [Sie+15; Sie+12]. For example, if we measure the performance of a database system with and without encryption and with and without compression, we will notice that these two feature interact: encrypting compressed data is computationally less expense than encrypting uncompressed data (as we discussed in Chapter 1, p. 4).

5.1.2 Internal Feature Interactions

Beside the behavior-centric view, researchers have proposed to take an implementation-centric view, which aims at the internals of a system, to under-

stand the feature-interaction problem [BHK11; Käs+09; LBL06]. Specifications are given usually implicitly, as we will discuss.

Structural Interactions. It is a matter of fact that a feature is typically not an island; it communicates and cooperates with other features and the environment. In the end, the communication and cooperation among features needs to be implemented somewhere. To let features interact, we need corresponding *coordination code* (denoted using the # operator with features that require coordination [BHK11]). For example, if we attempt to coordinate the call-forwarding and call-waiting features of the telephony example (Chapter 1, p. 4), we have to add additional code for this task (e.g., to deactivate one of the two features in favor of the other). If we activate both features in a system, we need to include also the corresponding coordination code:

$$\mathsf{CallForw} \wedge \mathsf{CallWait} \ \Rightarrow \mathsf{CallForw}\#\mathsf{CallWait}$$

Coordination code breaks feature modularity and hinders compositional reasoning [KAO11]. But, there is more to this. Much like with external feature interactions at the behavioral level, in the worst case, the number of pieces of coordination code grows exponentially with the number of features. Although researchers have proposed and discussed a number of solutions, there is no "silver bullet" to this problem [Käs+09]. The problem becomes even more problematic when all interacting features are supposed to be independently selectable or activatable by the user [Käs+09].

The key observation that is important here is that coordination code gives rise to a *structural* feature interaction. Features are considered to interact structurally if some coordination code is necessary that is different from the combination of the code of the individual features involved [BHK11; Käs+09; LBL06].

In many cases, structural feature interactions can be easily identified statically (e.g., based on naming or coding conventions, code-nesting structure, feature-tracing approaches, or dedicated implementation techniques [Käs+09; LBL06; Pre97]). As an example, in practice, the presence of coordination code is often controlled by nested preprocessor directives [Lie+10] or dedicated glue-code modules [Käs+09], such as lifters in feature-oriented programming [Pre97] and connector plugins in ECLIPSE.

Operational Interactions. Apart from just analyzing the code base and searching for coordination code that gives rise to structural interactions, one can collect more detailed information on internal feature interactions by analyzing the execution or operation of a system. Which features refer to which other

features? Which features pass control to which other features? Which features pass data to which other features? This information on *operational* interactions cannot be easily extracted from just looking syntactically at the source code, but requires more sophisticated (static or dynamic) analyses of the control and data flow. These analyses may provide valuable insights. For example, if we find that a contact-management and a messaging feature in an office groupware interact at the level of the control flow, but not at the level of the data flow (i.e., they do not exchange or share any data, even not via other features), we can infer that private contact data will not be send via the messaging feature to an untrusted receiver. This kind of information would help to make analysis techniques smarter and more efficient, as we will discuss in the next section.

Features are considered to interact operationally, if the occurrence of specific control and data flows, diverges from the combination of the flows of the individual features involved [BHK11; Käs+09; LBL06]. For example, two feature interact at the level of the control flow if there are control flows that occur *only* when the two features are combined, and that are not just the addition or union of the control flows induced by the two individual features.

5.2 Examples of Relations among Interactions

To illustrate how internal and external feature interactions can be related, we use a simple example of an audio streaming system with five optional features: Compress compresses the audio stream; Encrypt encrypts data; AddMetadata adds data about the stream quality, description of the audio content, information about its authors, etc., to the stream; LogIP logs IPs of the users receiving the stream; RankContent ranks the audio content according to its popularity. The performance of the system is measured by the maximum number of users that can simultaneously receive an audio stream without the system becoming overloaded.

5.2.1 Control-Flow Interactions (Internal, Operational)

In Figure 5.1a, we illustrate an excerpt of the annotation-based implementation (Section 2.1.3) of the audio streaming system using C and C preprocessor. We denote internal interactions among features with arrows. The boxes on the arrows contain *presence conditions* for the corresponding interactions [vRhe+15], that denote which features must be enabled (or disabled) for the interaction to take place. For example, if both features AddMetadata and Encrypt are enabled, then metadata are encrypted along with the audio data. For this purpose, AddMetadata calls the encryption function of feature Encrypt (denoted by the

(a) Control-flow (solid line) and data-flow (dashed line) interactions interactions in the audio streaming system.

```
#ifdef Encrypt
void encrypt(payload_t *payload) {...}
#endif

#ifdef AddMetadata
void add_metadata(packet_t *packet) {
  ...
#ifdef Encrypt
  encrypt(metadata);
#endif
  ...
}
#endif

#ifdef LogIP
void log(char *ip) {
  ...
#ifdef Encrypt
  encrypt(log_entry);
#endif
  ...
}
#endif

#ifdef Compress
void compress(payload_t *payload) {...}
#endif

#ifdef RankContent
void rank() {...}
#endif
```


(b) A performance-influence model with performance interactions.

$$100 - 15\cdot\mathsf{Compress} - 15\cdot\mathsf{Encrypt} - 5\cdot\mathsf{AddMetadata} - 5\cdot\mathsf{LogIP} - 5\cdot\mathsf{Rank} - 5\cdot\mathsf{AddMetadata}\cdot\mathsf{Encrypt} + 10\cdot\mathsf{Compress}\cdot\mathsf{Encrypt}$$

feature interactions

Figure 5.1: Interactions in the audio streaming system.

solid red arrow). Consequently, there is a control-flow interaction between these two features.

Likewise, there is a control-flow interaction between features LogIP and Encrypt (denoted by the dashed green arrow), since the log entries are encrypted if both features are enabled.

Finally, a internal interaction exists between features Compress and Encrypt (denoted by the dotted blue arrow). This is a data-flow interaction, because both features operate on the same resource (i.e., the audio stream).

5.2.2 Performance Interactions (External, Non-functional)

In Figure 5.1b, we show a performance-influence model (Section 2.4) of the audio streaming system. For a given system configuration, the model can predict the maximum number of users that can simultaneously receive an audio stream without overloading the system.[3] For example, for the configuration with feature Compress enabled and the rest of the features disabled the system can reliably serve $100 - 15 \cdot 1 - 15 \cdot 0 + \cdots + 10 \cdot 0 = 85$ users.

The model reveals two feature interactions (denoted by gray boxes). The first interaction is between AddMetadata and Encrypt. If both of this feature are enabled, the system encrypts not only the audio stream, but also the metadata that are added to the stream. The larger amount of data to be encrypted leads to a computational overhead that reduces the system's performance by 5 users that can be served. The second interaction is between Compress and Encrypt. Each of the two features individually has a negative influence of -15 on the system's performance, but encryption is faster if the data were compressed before. Therefore, the combined influence of both features on performance is less than the sum of their individual influences: $-15 - 15 + 10 = -20$ and not -30. That is, the interaction has a positive influence on performance.

Furthermore, for our example, we assume that encrypting a small string containing an IP address is so fast that this has no measurable effect on the performance of the system. Therefore, there is no a performance interaction between features LogIP and Encrypt. Consequently, there is no a corresponding term in our performance-influence model. Finally, feature RankContent as well as all other possible feature combinations have no measurable influence on performance and, therefore, they are not in the performance-influence model.

[3] Here, we assume that the model is 100 % accurate.

Table 5.1: A lists of interacting features from Figure 5.1. It illustrates which of the features interact internally (control-flow interaction), externally (performance interaction), or both.

Interacting Features	Control flow	Performance
AddMetadata, Encrypt	✓	✓
Compress, Encrypt	✓	✓
LogIp, Encrypt	✓	–

5.2.3 Relating Control-Flow and Performance Interactions

Table 5.1 summarizes the control-flow and the corresponding performance interactions from our example (Fig. 5.1). The feature combinations (AddMetadata, Encrypt) and (Compress, Encrypt) give rise to both control-flow and performance interactions. Based on our knowledge about the implementation, we can explain the causal relation between the control-flow and performance interactions captured by these feature combinations: The call to the computationally expensive encryption functionality (a control-flow interaction) leads to the performance decrease in the configurations containing the features that implement and use the encryption functionality (i.e., a performance interaction between these features occurs). Notice that the related control-flow and performance interaction involve exactly the same features, so we can also relate them based on the features they involve. However, the mere presence of control flow among features does not always indicate the presence of a performance interaction. For example, the control-flow interaction between features LogIp and Encrypt has no corresponding performance interaction. So, it is an open question to what extent a presence of a control-flow interaction can be used as an *indicator* for a potentially existing performance interaction.

Also note that from 26[4] feature combinations possible in the audio streaming system only three combinations give rise to feature interactions. All remaining feature combinations can be ignored by an interaction detection technique, because features in these combinations do not interact.

In what follows, we investigate *to what extent* a relation between control-flow and performance interactions exists in a real-world setting. Furthermore, we define and evaluate a predictor that uses control-flow interactions to predict potential performance interactions. With such a predictor in place, we could make interaction detection more efficient and accurate, which would be a

[4] 10 combinations with 2 features, 10 with 3, 5 with 4, and 1 with 5.

valuable contribution to research fields, such as optimization of non-functional properties, combinatorial testing, and sampling techniques.

5.3 Research Questions and Conceptual Framework

In our study, we address the following research questions:

- **RQ1:** Do control-flow feature interactions and performance feature interactions relate (in terms of the definition of Section 5.3.4)?
- **RQ2:** If a relation exists, can it be effectively leveraged to improve existing techniques for detecting external feature interactions or even to predict external feature interactions based on internal ones?

Before we can answer these questions, we have to decide on methods and tools that we will use in our study and how to combine them in a conceptual framework. Using this conceptual framework, we will then study relations among internal and external interactions and answer the research questions. Particularly, we have to choose a suitable research method, specify how we identify control-flow and performance interactions, define what a relation between these two types of interactions exactly is, and describe how we want to leverage it. Next, we describe this conceptual framework.

5.3.1 Research Method

We use the case study research method to explore a relation between control-flow and performance interactions. Shull et al. [SSS07] defines a case study as an "initial investigation of some phenomena". The relation, which we explore in this chapter, is novel and it is studied for the first time. Therefore, our study qualifies as an initial investigation of a phenomena.

Yin [Yin03] has a broader definition of a case study as "an empirical inquiry that investigates a contemporary phenomenon within its real-life context." It is easy to construct an artificial example with a clearly existing relation between control-flow and performance interactions (cf. Section 5.2.3), but this would say nothing about the existence of this phenomenon in real-world configurable systems. The goal of our study is to investigate whether there is or may be such a relation between control-flow and performance interactions in real-world configurable systems, which matches exactly the definition by Yin.

Finally, Flyvbjerg [Fly06] states that "case studies offer in-depth understanding of how and why certain phenomena occur, and can reveal the mechanisms by which cause–effect relationships occur". In our study, we want to obtain deep

insights into the nature of the relation between control-flow and performance feature interactions and not only report statistics. By focusing on two systems, we aim at increasing the internal validity of the study, because, this way, we can better identify and control confounding effects that may vary from one subject system to another (e.g., architecture, size of features). Moreover, our study involves bleeding-edge techniques for detecting control-flow and performance interactions in configurable systems that are technically challenging and cannot be easily applied to a large number of non-trivial real-world systems. For example, information about the variability in the configurable system that is provided in its documentation is often outdated, so compiling all configurations of the system becomes a tedious try-and-error process.

Based on all these considerations we have chosen the case study as our research method. In our case study, we go through the following steps: We take two real-world configurable systems and identify control-flow and performance interactions in these systems; then, we examine if the identified control-flow and performance interactions can be related based on the features that occur in them; finally, we evaluate predictors for performance interactions based on these relations. Next, we describe these steps in more detail and give an overview of the subject systems.

5.3.2 Identifying Control-Flow Interactions

To identify control-flow interactions, we use a variability-aware call-graph analysis [Fer+15] implemented in TypeChef[5] that identifies function calls among features implemented with preprocessor annotations (Fig. 5.1a). The central idea of a variability-aware analysis is to achieve efficiency by analyzing code parts that are shared by multiple system configurations only once. This is achieved by analyzing the source code of the system that still contains variability (e.g., the code with preprocessor annotations in Figure 5.1a), as opposed to analyzing the source code of individual configurations, which may be exponentially many in the number of features. A variability-aware call-graph analysis provides an efficient way to identify function calls among features of a configurable system and makes the detection of internal interactions feasible.

The underlying data structure for the analysis is *the variable abstract syntax tree*. Similar to an abstract syntax tree (AST), a variable AST provides an abstraction of the source code that can be efficiently analyzed, but it also provides information on which part of the code belongs to which features (in the form of presence conditions). Using this information, a call-graph analysis can identify, for each function call, which feature is the caller and which feature

[5]http://fosd.net/TypeChef/

is the callee. Furthermore, the analysis can identify a presence condition for each call, that is, which features must be enabled (or disabled) for the call to take place at runtime. For example, in Figure 5.1a, the call from feature AddMetadata to feature Encrypt (solid red arrow) occurs only if both features AddMetadata and Encrypt are enabled (denoted by the presence condition in the box under the arrow). Due to the static nature of the technique, the collected information about the calls may be an overapproximation, but this is a problem with any static analysis approach. The current implementation of the analysis also uses pointer analysis to increase the accuracy of the call graph [Fer+15].

5.3.3 Identifying Performance Interactions

For detecting performance feature interactions, we learn performance-influence models (Fig. 5.1b). As discussed in Section 2.4, a performance-influence model captures the influences of individual features and their interactions on performance of a configurable system. We learn performance-influence models using the tool SPL Conqueror,[6] which implements a state-of-the-art machine learning algorithm based on multivariable regression and forward feature selection (Section 2.4.3). The algorithm takes as input a sample of system configurations and corresponding performance measurements. The accuracy of the learned performance-influence model depends, among other factors, on how representative the sampled configurations are for the entire configuration space. To get a performance-influence model of the highest possible accuracy, and, consequently, to detect feature interactions as precise as possible (i.e., to obtain the ground truth), we measured not a sample but all configurations of the subject system and used these measurements as the algorithm input. The performance measurements were done using a standard benchmark.

5.3.4 Relating Control-Flow and Performance Interactions

After we have identified the internal (control-flow) interactions, the question is what we can learn from them regarding external (performance) interactions. To answer this question, we relate the control-flow interactions and performance interactions based on the features involved in them, as we explained it in our example in Section 5.2.3. The goal is to find out if the features involved in performance interactions also occur in one or more internal interactions and vice versa. This is a feasibility check to see if the interactions can be related

[6] `http://fosd.net/SPLConqueror/`

based on the features' occurrence at all. That is, if we find no interactions that can be related in this way, this would mean that it is impossible to define any relation between interactions based on the corresponding feature occurrences in these interactions.

We define a performance interaction i_p and a control-flow interaction i_c as related if $\mathit{features}(i_p) \subseteq \mathit{features}(i_c)$ or if $\mathit{features}(i_p) \supseteq \mathit{features}(i_c)$, where $\mathit{features}(i)$ is the set of features that contribute to the interaction i.

Furthermore, for each related pair of interactions, we determine how similar the interactions are (i.e., if they contain exactly the same features or if they also contain features that are present only in one of them). The similarity of the related interactions can be interpreted as the strength of their relation: the higher the similarity, the higher the strength of the relation. We calculate the similarity of interactions using the Jaccard index J [Jac12]:

$$J(i_p, i_c) = \frac{\mathit{features}(i_p) \cap \mathit{features}(i_c)}{\mathit{features}(i_p) \cup \mathit{features}(i_c)}$$

where $\mathit{features}(i)$ is the set of features involved in the interaction i. The Jaccard index equals 1 if both interactions involve exactly the same features and is less than 1 otherwise.

5.3.5 Predicting Performance Interactions

If we find a relation between control-flow and performance feature interactions as defined in Section 5.3.4, the question is whether we can use this relation to predict performance feature interactions.

One method is to build on our argumentation in Section 5.2.3 and to assume that every control-flow interaction corresponds to an existing performance interactions. Of course, we already know that there may be control-flow interactions without corresponding performance interactions. Nevertheless, it is an open question *how* accurate this simple method can be if applied to a real-world system.

We can also use a more advanced method based on reoccurring feature combinations in control-flow interactions: We argue that, if a set of features occurs in multiple control-flow feature interactions, then this set of features is also likely to give rise to one or more external interactions. The rationale behind this argument is that, if a set of feature is involved in many control-flow feature interactions, then chances are high that it is also involved in performance interactions, because the accumulated influence of the control-flow interactions on performance have a measurable effect.

We use *frequent item set mining* [Bor12] as a method to identify such frequent feature sets. This method was successfully used as a general pattern

mining method [MB07; Qia+13]. In terms of frequent item set mining, we refer to a feature as an *item*. For example, features such as AddMetadata and Encrypt in the example in Fig. 5.1 are items. The set of all items (all features) is the *item base* B (e.g, the item base of the example contains all its features). A subset of the item base $I \subseteq B$ is an *item set* that corresponds to a feature combination. An item set (i.e., a feature combination) that denotes an internal interaction in a system is a *transaction* $t \in T$, where T is a set of transactions. In our example, a set of features {AddMetadata, Encrypt} is an item set and it is also a transaction, because these two features interact at the control-flow level (Fig. 5.1a). Based on these definitions, we define the *support* (a.k.a. absolute frequency) s of an item set I: $s = |\{t : t \in T \wedge I \subseteq t\}|$. In Fig. 5.1a, the item set {Encrypt} has a support value of three, because it is a subset of every transaction (i.e., control-flow feature interaction) in the running example. Item set {AddMetadata, Encrypt} has a support value of 1, because there is only one control-flow feature interaction involving these features. The support value and a threshold $E \in [0, \infty)$ is used to decide which of the item sets are considered frequent: All item sets with the support value $s \geq E$ are *frequent* item sets. Based on our hypothesis, frequent item sets predict external feature interactions. In our analysis, we also ignore item sets with only one item (feature), because a feature interaction requires at least two different features. We use an implementation of the Apriori algorithm from the ORANGE library[7] to calculate the support value.

5.3.6 Subject Systems

The case study was conducted using two real-world configurable software systems: the MBEDTLS library implementing the transport security network protocol TLS/SSL and a SQL database engine SQLITE. The initial use case for the systems was the embedded domain, but now they are also used in non-embedded projects, such as OPENVPN and FIREFOX.

Similar to a large number of other real-world configurable systems, the subject systems are written in C using C-preprocessor directives to implement compile-time variable features. MBEDTLS has 97 and SQLITE has 12 features, which results in 1921 and 1533 configurations respectively. The configurations are obtained using a SAT solver (built-in into SPL CONQUEROR) by computing all feature combinations that satisfy the constraints in the feature models (see the following subsections) of the subject systems. MBEDTLS comprises 50 K and SQLITE 195 K lines of code. Both systems have a highly modular architecture, which is thoroughly documented along with the corresponding preprocessor

[7] http://orange.biolab.si/

macro names allowing relatively easy matching of code to the corresponding modules and submodules.

The manageable number of features and configurations makes these systems especially suitable for an in-depth qualitative case study: For example, it allows us to measure performance of all configurations and use these measurements in turn to identify a baseline of performance feature interactions in reasonable time. The feasible number of resulting feature interactions allows us to verify that every one of them actually exists in the system. Furthermore, the size of the subject systems allows us to manually inspect and understand the structure of the systems and the interplay of their features (Sec. 5.4, 5.5). Altogether, the manageable size of the subject systems is a prerequisite for the internal validity of our qualitative case study.

Features and Feature Model of mbedTLS

At the top level, mbedTLS consists of modules, such as *Cipher*, *Public Key*, *Hashing*. Each module implements the corresponding algorithms and protocols. For example, the *Cipher* module includes submodules that implement cipher algorithms, such as AES, DES, and ARC4. Submodules implement the features of the system. The cipher-algorithm features can be combined with other features, such as hash algorithms and public-key implementations, to provide an encryption protocol. We used the original documentation and manual code inspection to construct a feature model for mbedTLS version 2.2.1.

Features and Feature Model of SQLite

SQLite consists of a *Core* providing a C-language interface and being responsible for executing compiled SQL code, an *SQL Compiler*, and a *Backend* providing the low-level implementation of the database. A user can configure the operation of these modules by enabling or disabling their features through compile-time options. For example, *Core* can be configured to operate safely in a multithreaded environment by enabling the SQLITE_THREADSAFE feature. We studied the documentation and the source code of version 3.16.2 to construct a feature model.

Performance Measurements of mbedTLS

The primary application of mbedTLS is the encryption of data transmitted over a TCP/IP network. Ensuring fast and secure data transfer is commonly considered an important property of communication networks, such as the Internet. So, the time required to encrypt data and transfer them over the network is an important non-functional property of mbedTLS. Measuring

the time required by encryption alone is not representative, because different configurations may produce different amounts of payload (e.g., due to data compression and different amounts of generated metadata) influencing the transmission time. Therefore, we defined the performance measure for a configuration of MBEDTLS as the amount of time (in seconds) required to encrypt and successfully transmit a fixed amount of input data.

To detect performance feature interactions reliably based on performance benchmarks, it must be ensured that every feature included in a configuration is invoked during the benchmark of this configuration. Otherwise, the influence of features and their interactions on performance cannot be deduced from the benchmark results. The original automated test framework of MBEDTLS includes tests that check the library's functionality in a client-server environment and is suitable to serve as a typical benchmark suite. During the tests, the functionality of every feature in the configuration is tested, that is, every feature is actually invoked.

We used 2 GB of random data as input to ensure that the fastest configuration requires, at least, five seconds for transmission and to mitigate the influence of warm-up effects on the result. We repeated the benchmark 30 times to further reduce the influence of measurement bias. To exclude the influence of network latencies, we ran the benchmark locally using the local network interface.

SQLite Performance Measurements

The developers of SQLite provide a performance benchmark that measures time required by the database to execute a set of queries.[8] The original benchmark is not compatible with the latest version of the system that we use, so we used it as guidance to create a new compatible benchmark. While constructing the benchmark we made sure that the features of SQLite are actually invoked during the benchmarking process. Our benchmark measures the execution time in seconds. To reduce the influence of warm-up effects and measurement bias, the benchmark runs, at least, 25 seconds and every run is repeated 30 times.

The benchmarks for both systems were conducted on an Intel i5-4590, 16 GB RAM, 256 GB SSD, Ubuntu 16.04.

[8] http://sqlite.org/speed.html

5.4 Results

In this section, we describe the results of applying of our conceptual framework (Section 5.3) to subject systems. To increase internal validity, we report in Sections 5.4.1 and 5.4.2 in detail how we identified performance and control-flow interactions. Based on these data, we report the identified relation between performance and control-flow interactions (Section 5.4.3), which use to answer RQ1 in Section 5.5, and how this relation can be leveraged (Section 5.4.4), which we use to answer RQ2 in Section 5.5.

5.4.1 Performance Interactions

In this section, we report on performance interactions that we identified in MBEDTLS and SQLITE. Using domain knowledge and manual inspection of the source code, we confirm that the identified interactions actually exist and thereby increasing internal validity of our study.

We used SPL CONQUEROR and the performance benchmark results (cf. Section 5.3.6) as input data to identify performance interactions in MBEDTLS and SQLITE, as described in Section 5.3.3. Table 5.2 lists for both systems the performance interactions and their influences on performance of the systems in seconds. The *negative values* in the influence column denote *positive influences* of the corresponding interactions on performance. That is, they denote how much less time a configuration that includes them would need to execute the benchmark.

The mean standard deviation for the performance measurements of MBEDTLS is 0.42 s. Therefore, we classified all interactions with the absolute influences less than this value as noise and discarded them. From the remaining 16 interactions, 11 are interactions between two features; and five are interactions among three features. The mean standard deviation for the performance measurements of SQLITE is 0.09 s. The influences of the three identified interactions for the system are much higher and, therefore, are unlikely to be noise. Two of the interactions are interactions between two features and one is an interaction between three features.

MBEDTLS

All identified interactions in MBEDTLS are among features implementing different ciphers, block cipher modes of operation (simply “modes”), and cryptographic hash functions. This is plausible, because these three types of algorithms work tightly together to implement an encryption protocol. Ciphers (e.g., AES) are used to encrypt data, modes (e.g., CBC) are used in combination with block

Table 5.2: Performance interactions and their influences on performance of the systems in seconds.

System	ID	Influence (sec)	Performance Interaction (features involved)
mbedTLS	1	10.73	CIPHER_MODE_CBC, SHA256_C
	2	-9.71	AES_C, AESNI_C
	3	8.53	AESNI_C, SSL_CBC_RECORD_SPLITTING
	4	6.93	CIPHER_MODE_STREAM, AESNI_C
	5	6.08	SHA256_C, CIPHER_MODE_STREAM
	6	5.75	AES_C, AESNI_C, GCM_C
	7	3.49	CIPHER_MODE_CBC, SHA256_C, SHA256_SMALLER
	8	3.45	SHA256_C, CIPHER_MODE_STREAM, SHA256_SMALLER
	9	3.44	SHA256_C, AESNI_C, CIPHER_MODE_STREAM
	10	3.14	CIPHER_MODE_CBC, RIPEMD160_C
	11	-2.97	AES_C, GCM_C
	12	-2.84	CIPHER_MODE_STREAM, MD5_C
	13	1.93	AESNI_C, CAMELLIA_C
	14	1.68	CIPHER_MODE_CBC, SHA1_C
	15	1.60	CIPHER_MODE_STREAM, AESNI_C, MD5_C
	16	1.51	RIPEMD160_C, CIPHER_MODE_STREAM
SQLite	1	1.50	DEFAULT_MEMSTATUS, THREADSAFE
	2	1.47	MEMDEBUG, THREADSAFE
	3	1.41	DEFAULT_MEMSTATUS, MEMDEBUG, THREADSAFE

To relate the influences to configuration run times, note that the fastest mbedTLS configuration completed its benchmark in 6.7 seconds and the fastest SQLite configuration completed its benchmark in 26.7 seconds.

ciphers to encrypt amounts of data larger than a block (i.e., a fixed amount of data a block cipher can operate on; 128 bit for AES), and cryptographic hash functions (e.g., SHA) are used with modes to implement authentication and to ensure data integrity.

To confirm that the identified performance interactions actually result from the interplay of the corresponding features, we manually inspected the source code of mbedTLS. Next, we present the results of this code inspection.

Interaction 1 in Table 5.2 arises between a mode (CBC) and a hash function (SHA256). CBC uses hashing extensively to calculate keyed-hash message authentication code (HMAC). SHA256 is computationally more expensive than, for example, MD5; therefore, this combination with the mode has a negative influence of 10.73 seconds on performance. Interactions 5, 7, 8, 10, 12, 13, 14, and 16 have a similar cause and explanation. In addition to a mode and a hash function, interactions 7 and 8 also include the feature SHA256_SMALLER, which denotes that an implementation of SHA256 with smaller binary footprint was used. However, this implementation also has a lower performance, which leads to the negative influence of this interaction on performance. Interaction 12 has a positive influence on performance of using a mode (stream mode, in this case) with a less computationally complex (but also less secure) MD5 hash function. In interaction 13, the AES cipher is used as a hash function in combination with the Camellia cipher.

Interaction 2 arises from the usage of the AES cipher for encryption in combination with a native implementation of the AES algorithm in assembler (AESNI). The native implementation makes encryption faster, so this interaction has a positive influence of 9.71 seconds on performance.

Interaction 3 arises from the usage of the AES cipher for encryption in combination with an implementation of the CBC mode that includes a record splitting algorithm. This algorithm is a countermeasure against the BEAST attack on the SSL algorithm [DR11]. The way record splitting is implemented increases the number of packets to be transmitted (compared to the number of packets without this countermeasure). The increased number of packets results, in turn, in a negative influence on performance.

Interactions 4, 6, 9, 11, and 15 arise from the influence of further combinations of ciphers, modes, and hash functions on performance, similar to the first interaction.

SQLite

All performance interactions in SQLite include the feature THREADSAFE. This is plausible, because THREADSAFE is a crosscutting feature that adds the mutex and thread-safety logic to all unsafe regions in the code. This additional

thread-safety code imposes a runtime overhead and makes the benchmarks for the configurations containing it run longer. We inspected the code of SQLite and confirmed that both features DEFAULT_MEMSTATUS and MEMDEBUG retrieve a mutex (i.e., use THREADSAFE feature) at a certain stage of operation that results in interaction among THREADSAFE and these features.

Summary. Overall, we identified 16 performance interactions in mbedTLS. 11 of them occur between 2 features and 5 among 3 features. In SQLite, we identified 3 performance interactions. 2 interactions between 2 features and 1 among 3 features. Using domain knowledge and manual inspection of the source code, we identified the cause of all interactions and thereby confirmed that they actually exist in the systems and are caused by the interplay of the corresponding features.

5.4.2 Control-Flow Interactions

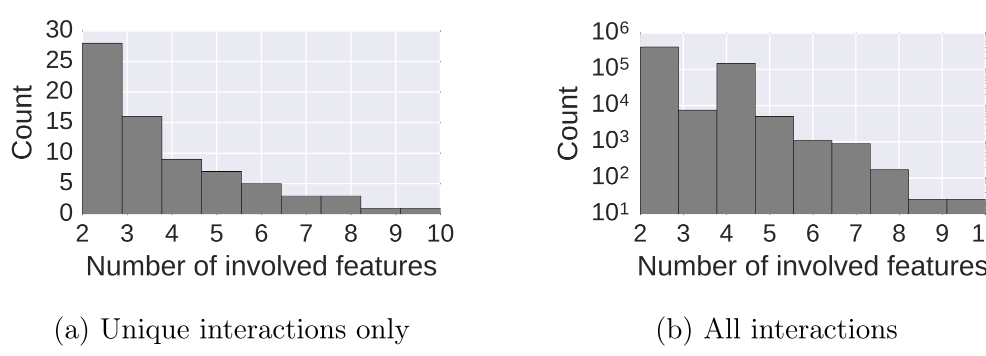
In this section, we report on control-flow interactions that we identified in mbedTLS and SQLite using variability-aware call-graph analysis implemented in TypeChef (Section 5.3.2). Furthermore, we explain the limitations of TypeChef that prevent it from detecting all control-flow interactions (e.g., in cases where one feature uses function pointers to call another). We discuss how we addressed these limitations to increase the internal validity of the study by manually identifying control-flow interactions missed by TypeChef.

mbedTLS

From 761 992 function calls in the system, we detected 575 560 control-flow feature interactions. This number of interactions includes duplicate interactions that appear if the corresponding function call between features occurs in multiple locations in the code. The number of unique control-flow interactions is 73.

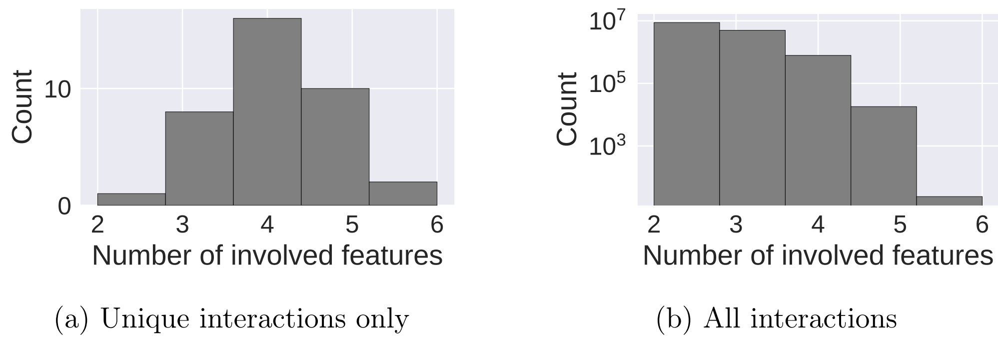
Notably, among the unique control-flow interactions, there are interactions with up to 10 features, but most unique interactions involve only two features (Fig. 5.2a). If we also consider the duplicates (Fig. 5.2b), the overall picture stays largely the same: Only the number of interactions involving four features becomes larger than the number of those involving three features.

While manually exploring the source code of mbedTLS, we found that cipher, mode, and hash algorithms call each other indirectly, using function pointers. This indirection was introduced by the designers of the library to decouple the algorithms and to make their concrete implementations interchangeable. TypeChef would need to be extended with a variability-aware,



(a) Unique interactions only (b) All interactions

Figure 5.2: mbedTLS: counts of features in control-flow interactions.



(a) Unique interactions only (b) All interactions

Figure 5.3: SQLite: counts of features in control-flow interactions.

inter-procedural data-flow analysis to identify which features interact using indirect function calls. Being aware of this technical limitation of TypeChef, we added 11 indirect control-flow interactions that we collected while manually exploring the code to the set of interactions. In our manual exploration of the source code, we relied on our understanding of the subject systems' structure and interplay of the features. For example, based on the knowledge, we knew that cipher, mode, and hash algorithms should closely work together. So, we looked for control flow among all features implementing these algorithms. The total number of the identified unique control-flow interactions is 84 (73 were found using TypeChef and 11 manually). It would be infeasible to find manually all instances of indirect control-flow interactions, so their exact number (including duplicates) is unknown. We discuss the corresponding threats to validity in Section 5.6.

SQLite

From over 14 587 337 function calls in the system, we detected 14 587 335 control-flow feature interactions. That is, all but two function calls involved more than one feature. The number of unique control-flow interactions is 37.

In contrast to MBEDTLS, most unique interactions involve 4 features, and there are interactions with up to 6 features (Fig. 5.3a). Although, if we also consider duplicates (Fig. 5.3b), the picture becomes similar to that in MBEDTLS: Interactions among 2 features prevail and the count of interactions decreases with the increasing number of involved features.

While manually inspecting the code of SQLite, we found that the option SQLITE_DEFAULT_MEMSTATUS (which is used by TypeChef to identify the code belonging to the feature DEFAULT_MEMSTATUS) is used to set a Boolean variable at compile-time. This variable is then used at runtime to check if feature DEFAULT_MEMSTATUS is enabled or disabled. This way, the feature can be enabled or disabled at runtime. Again, TypeChef would need a data-flow analysis to trace the connection the preprocessor macro to the corresponding Boolean variable to detect control-flow interactions in which feature DEFAULT_MEMSTATUS is involved. By further exploring the code, we identified two control-flow interactions of this kind and added them to the set of automatically detected interactions. Therefore, the total number of the identified unique control-flow interactions is 39.

Summary. Overall, we identified 575 571 control-flow interactions in MBEDTLS among which 84 were unique. Some interactions involve up to 10 features, but most interactions are between 2 features. For SQLite, we identified 14 587 335 control-flow interactions, with 39 unique. Due to technical limitations of TypeChef, indirect control-flow interactions in MBEDTLS and interactions induced by runtime variability in SQLite could not be detected by TypeChef. We manually inspected the source code to collect these interactions.

5.4.3 Relating Interactions

In this section, we describe how we identified relations among performance and control-flow interactions that we described in Sections 5.4.1 and 5.4.2

Performance Interactions → Control-Flow Interactions. Using the relation definition $\mathit{features}(i_p) \subseteq \mathit{features}(i_c)$ (Section 5.3.4), for each performance interaction, we identified all unique related control-flow interactions (i.e., all control-flow interactions involving exactly the same features as the performance interaction). Furthermore, for each pair of related interactions, we calculated

Table 5.3: Performance interactions, the number of the control-flow interactions related to them, and the mean value of the corresponding Jaccard indexes.

	ID	Performance Interaction (features involved)	Relations	Jaccard (mean)
MBEDTLS	1	CIPHER_MODE_CBC, SHA256_C	1	1.00
	2	AES_C, AESNI_C	10	0.53
	3	AESNI_C, SSL_CBC_RECORD_SPLITTING	2	0.38
	4	CIPHER_MODE_STREAM, AESNI_C	1	1.00
	5	SHA256_C, CIPHER_MODE_STREAM	1	1.00
	6	AES_C, AESNI_C, GCM_C	4	0.53
	7	CIPHER_MODE_CBC, SHA256_C, SHA256_SMALLER	1	1.00
	8	SHA256_C, CIPHER_MODE_STREAM, SHA256_SMALLER	1	1.00
	9	SHA256_C, AESNI_C, CIPHER_MODE_STREAM	1	1.00
	10	CIPHER_MODE_CBC, RIPEMD160_C	1	1.00
	11	AES_C, GCM_C	13	0.40
	12	CIPHER_MODE_STREAM, MD5_C	1	1.00
	13	AESNI_C, CAMELLIA_C	4	0.35
	14	CIPHER_MODE_CBC, SHA1_C	1	1.00
	15	CIPHER_MODE_STREAM, AESNI_C, MD5_C	1	1.00
	16	RIPEMD160_C, CIPHER_MODE_STREAM	1	1.00
SQLITE	1	DEFAULT_MEMSTATUS, THREADSAFE	1	1.00
	2	MEMDEBUG, THREADSAFE	16	0.45
	3	DEFAULT_MEMSTATUS, MEMDEBUG, THREADSAFE	1	1.00

the Jaccard index (Section 5.3.4), which denotes how similar the interactions are (the index equals 1 if both interactions involve exactly the same features and is less than 1 otherwise).

Table 5.3 summarizes the results. For each performance interaction, it shows the number of the related control-flow interactions and the mean of all Jaccard indexes calculated for these relations. The apostrophe (') denotes the performance interactions that are related to the manually identified indirect control-flow interactions for which we were not able to establish the exact number of occurrences (cf. Section 5.4.2). The numbers show that there is a relation between every performance interaction and, at least, one control-flow interaction. The Jaccard indexes show that the related control-flow interactions that were automatically detected by TypeChef (those are the same as the interactions with the number of relations greater than 1 in Table 5.3), involve, on average, twice as many or even more features than there are in the corresponding performance interactions.

Control-Flow Interactions → Performance Interactions. Using the relation definition $features(i_p) \supseteq features(i_c)$ (Section 5.3.4), for each control-flow interaction, we identified all related performance interactions (i.e., all performance interactions involving exactly the same features as the control-flow interaction).

Table 5.4 summarizes the results. For mbedTLS, among the 84 unique control-flow interactions, we found 15 interactions that have one or more related performance interactions. For SQLite, among the 39 unique control-flow interactions, we found 2 interactions that have one or more related performance interactions. The Jaccard indexes show that the related performance interactions that were automatically detected by TypeChef (interactions 1–4 for mbedTLS) involve mostly the same features as the corresponding control-flow interactions. The manually added control-flow interactions (interactions 5–15 for mbedTLS and all interactions for SQLite) match exactly the related performance interactions.

Summary. We found a relation between every of the 16 identified performance interactions and one or more control-flow interactions. The Jaccard indexes show that the automatically detected control-flow interactions do not generally contain exactly the same features as the related performance interactions, that is, the automatically detected control-flow interactions involve, on average, twice as many features as the corresponding performance interactions. The manually added control-flow interactions involve exactly the same features as the corresponding performance interactions.

Table 5.4: Control-flow interactions, the number of the related performance interactions, and the mean value of the corresponding Jaccard indexes. Control-flow interactions without related performance interactions are not listed.

	ID	Control-Flow Interaction (features involved)	Relations	Jaccard (mean)
mbedTLS	1	AES_C, AESNI_C	2	0.83
	2	GCM_C, AESNI_C	1	0.67
	3	GCM_C, AES_C	2	0.83
	4	GCM_C, AES_C, AESNI_C	1	1.00
	5	CIPHER_MODE_CBC, SHA256_C	1	1.00
	6	CIPHER_MODE_STREAM, AESNI_C	1	1.00
	7	SHA256_C, CIPHER_MODE_STREAM	1	1.00
	8	CIPHER_MODE_CBC, SHA256_C, SHA256_SMALLER	1	1.00
	9	SHA256_C, CIPHER_MODE_STREAM, SHA256_SMALLER	1	1.00
	10	SHA256_C, AESNI_C, CIPHER_MODE_STREAM	1	1.00
	11	CIPHER_MODE_CBC, RIPEMD160_C	1	1.00
	12	CIPHER_MODE_STREAM, MD5_C	1	1.00
	13	CIPHER_MODE_CBC, SHA1_C	1	1.00
	14	CIPHER_MODE_STREAM, AESNI_C, MD5_C	1	1.00
	15	RIPEMD160_C, CIPHER_MODE_STREAM	1	1.00
SQLite	1	DEFAULT_MEMSTATUS, THREADSAFE	1	1.00
	2	DEFAULT_MEMSTATUS, MEMDEBUG, THREADSAFE	1	1.00

5.4.4 Predicting Performance Interactions

MBEDTLS: *Direct Matching.* As we describe in Section 5.3.5, one prediction method is to assume that every control-flow interaction induces a performance interaction that involves exactly the same features. In MBEDTLS, from the 73 automatically identified unique control-flow interactions there are three—interactions 1, 3, and 4 in Table 5.4—that have exactly the same features as the related performance interactions 2, 6, and 11 in Table 5.3. That is, three of the 16 performance interactions could be predicted by the direct matching. Therefore, the precision of the direct matching is 4.11 % and the recall is 18.75 %. If we also incorporate the 11 indirect control-flow interactions, which we identified by manually inspecting the code, the total number of matching control-flow interactions becomes 14. Including indirect control-flow interactions increases the precision and recall to 16.7 % and 51.85 % respectively.

SQLITE: *Direct Matching.* In SQLITE, there are no automatically identified unique control-flow interactions that match exactly any of the performance interactions. Including the manually added control-flow interactions gives the prediction precision of 5.13 % and the recall of 67 %.

MBEDTLS: *Frequent Item Sets.* Using frequent item set analysis (cf. Section 5.3.5) on the set of control-flow interactions for MBEDTLS, we found 44 item sets, of which we calculated the support values. The support values range from 11 % to 34 %, meaning that there are item sets occurring in 11 % to 34 % of all control-flow interactions.

Two of the found item sets match exactly the performance interactions 2 and 11 of Table 5.3. Notice that we ran the frequent item set analysis only on the automatically detected control-flow interactions. We were not able to run it on the indirect control-flow interactions, because then we would have to find every instance of such interaction manually, which is infeasible. Nevertheless, we incorporated the indirect control-flow interactions into further analysis by approximating their support values based on the distribution of support values for similar indirect interactions (see Section 5.6, for threats to validity). Among the 44 detected item sets, there are 33 item sets capturing interactions among ciphers, modes, and hash functions. We assigned support values to the indirect control-flow interactions according to the distribution of the support values of these 33 item sets. That is, 6 % of the interactions were assigned a support value of 11 %, 3 % were assigned a support value of 12 %, and so on.

By varying the threshold E, as described in Section 5.3.5, we are able to decide which of the identified item sets are considered frequent. By setting the threshold to 0, we consider all identified item sets as frequent. When the

threshold is increased the item sets with lower support values are not considered frequent anymore. For example, if we set the threshold to 15 % only 25 % of the identified item sets will have a higher support value and will be considered frequent. Changing the threshold this way allows us to observe its influence on the predictive power (i.e., precision and recall) of the frequent item sets.

To calculate how good the item sets are in predicting performance interactions, we compared how many of them denote the actually identified performance interactions (i.e., contain exactly the same features as the performance interactions). The low precision and recall values for mbedTLS summarized in Table 5.5 show that our predictor based on the frequent item sets has only a low predictive power. Increasing the threshold value decreases the precision and recall of the predictor.

SQLite: *Frequent Item Sets.* Applying the same frequent item set method to the control-flow interactions of SQLite resulted in four frequent item sets with support values ranging from 20 % to 100 %. None of these frequent item sets matched the performance interactions. We could not approximate the distribution of the support values for the manually detected control-flow interactions, because they do not exhibit any commonalities with the calculated frequent item sets as it was the case for mbedTLS.

Summary. We defined two predictors for performance interactions based on their relation with control-flow interactions. The first predictor is based on the assumption that every control-flow interaction induces a performance interaction that involves exactly the same features. The second predictor is based on the assumption that the recurring feature combinations in control-flow interactions capture the related performance interactions. The evaluation showed that both predictors have only low precision and recall values.

5.5 Discussion

Based on our results, we conclude that there is indeed a quantifiable relation between control-flow and performance interactions. We confirmed this by manually inspecting the code and by comparing which features are involved in the detected performance interactions and how these features interact at the control-flow level. We found that features involved in performance interactions work closely together to implement the systems' functionality and thus also interact at the control-flow level. That is, the same features that are involved in performance interactions are also involved in control-flow interactions. There-

Table 5.5: Precision and recall values for the item sets as predictors for the performance interactions in mbedTLS. (*) marks the precision and recall values for the item sets with incorporated indirect control-flow interactions.

Threshold	Precision	Recall	Precision*	Recall*
0	4.5	12.5	23.6	48.1
15	2.3	6.3	5.5	11.1
20	0	0	1.8	3.7

fore, we can positively answer research question RQ1, which asked if control-flow feature interactions and performance feature interactions relate.

The relation we found among control-flow and performance feature interactions has implications for performance prediction techniques for configurable systems. As we discussed in Section 5.4.3, the identified control-flow interactions capture the features that are involved in the performance interactions. Of course, we cannot identify these features precisely, because the same control-flow interactions also involve other features that are not involved in performance interactions (this is also a reason for direct matching prediction having low precision and recall; cf. Sec 5.4.4). Nevertheless, assuming that only the features from the identified control-flow interactions can give rise to a performance interaction considerably reduces the search space of the potential performance feature interactions, because otherwise we have to assume that any (valid) feature combination may give rise to a performance interaction. mbedTLS has 134 057 valid feature combinations of two and three features, but the 84 identified unique control-flow interactions (Section 5.4.2) result in only 452 *potential* performance interactions (among two and three features). Notice that these include all 16 actually existing performance feature interactions that we identified. That is, we are able to shrink the search space of performance feature interactions by almost 300 times (452 instead of 134 057) without losing any of the actually existing performance feature interactions. SQLite has 524 valid feature combinations of two and three features and (based on the 39 identified unique control-flow interactions) only 131 *potential* performance interactions (among two and three features). These potential performance interactions also include all 3 actually existing performance interactions that we identified. That is, the search space shrinks by 4 times. These results have immediate consequences for performance prediction techniques based on machine learning and relying on sampling for building a training dataset: By exploiting our findings they can make sampling more focused on the configurations that potentially include interacting features, which may improve their prediction accuracy.

With respect to RQ2, which asked if relations between control-flow and

performance interactions can be effectively leveraged to improve existing techniques for detecting external feature interactions or to predict external feature interactions based on internal ones, our results are twofold. The shrinkage of the search space of performance feature interactions can help to make performance prediction techniques more focused on potential feature interactions, which is a positive result. As to the predictors based on direct matching and frequent item sets, we obtained only low precision and recall values, which is effectively a negative result. One possible reason for this negative result is that the predictors rely solely on control-flow data, but features can also interact via data flow. For example, they can exchange data through shared data structures. This interplay at the data-flow level can be interpreted as data-flow feature interactions (much like control-flow feature interactions, Figure 5.1a), which may also induce performance interactions. For example, a feature may block other features by locking a shared data structure, which may have a negative influence on the performance of the system. Therefore, enriching the data used by the predictors with the information about data-flow interactions may increase their predictive power. So, a takeaway message here is that predictors should consider the interplay of features not only on the control-flow level, but also at the data-flow level, and other levels. Another reason may be that not all features involved in a control-flow interaction are also involved in a related performance interaction. The Jaccard index values in Table 5.3 show that only about half of the features in a control-flow interaction are also present in the related performance feature interaction. For example, the interaction (AES_C, AESNI_C) has the average Jaccard index of 0.46. This means that, on average, a related control-flow interaction has two other features additionally involved, in addition to features AES_C and AESNI_C. Both predictors for a given control-flow interaction are not able to distinguish among features that are involved in a related performance interaction and those that are not.

Further Observations. A further observation is related to the distribution of the number of features involved in the control-flow and performance interactions. For mbedTLS, in most cases, interactions (both, control-flow and performance) involve two or three features. For SQLite, in most cases, control-flow interactions involve four features, but this is only the case because every single control-flow interaction involves the two crosscutting features THREADSAFE and ENABLE_API_ARMOR. If we ignore these crosscutting features, the pictures becomes similar to mbedTLS. The performance interactions in SQLite involve two or three features as in mbedTLS. From these data, we conclude that the frequency of interactions decreases with the growing number of the involved features. This shows that features tend to interact at the same

rate (two or three features per interaction) independently of the type of the interaction (control-flow or performance). This is another indication for a relation between control-flow and performance interactions.

Finally, for mbedTLS, we found that most of the frequent item sets that we identified in the control-flow interactions contain features from three groups of algorithms: ciphers, modes, and hashes. Even though most of the frequent item sets do not resemble existing performance interactions, they still capture the general pattern of the detected performance interactions, namely, that these interactions involve features from these three groups of algorithms. For SQLite the frequent item sets capture the crosscutting features, such as THREADSAFE and ENABLE_API_ARMOR. The crosscutting feature THREADSAFE was involved in all identified performance interactions.

5.6 Threats to Validity

Internal Validity. Due to technical limitations of TypeChef, we were unable to identify the exact number of indirect function calls between features (i.e., calls made using function pointers) and, consequently, the exact support values for the corresponding item sets (Section 5.4.4). We approximated these support values based on the distribution of the support values for the item sets calculated from direct function calls. Our approximation method may result in an inaccurate calculation of the precision and recall values of the frequent item set predictor. Nevertheless, we expect that improving the approximation would rather improve the precision and recall of the predictor.

Due to the static nature of the call-graph analysis employed by TypeChef, the collected information about the calls may be an approximation, and, as a consequence, a threat to internal validity. To mitigate this threat, we verified all control-flow interactions (identified using call-graph analysis) that are related to performance interactions by manually inspecting the source code of the subject systems and by confirming that these control-flow interactions actually exist.

External Validity. We have chosen a case study as our research method (Section 5.3.1), which suits well the exploratory nature of our study, which aims at the initial investigation of the relation between control-flow and performance feature interactions. The downside of using this research method is that it cannot be efficiently applied to multiple reasonably large configurable systems. In fact, it threatens the external validity of our study, since we focused on analyzing two systems and our results may not hold for other configurable systems. Nevertheless, our study setup has proven successful and can thus serve

as a blueprint for further studies that can rely on our conceptual framework for studying relations among external and internal interactions. We conjecture, that the relation between performance and control-flow interactions that we identified in our subject systems is likely to exist in systems with a larger number of features as well.

5.7 Related Work

In recent years, a number of papers aimed at detecting feature interactions in configurable systems. We summarize and subdivide them according to our classification into those considering internal feature interactions and those considering external feature interactions. Alone the fact that we were able to clearly assign the related work to one of the feature interaction classes shows that previous studies focused on one interaction class at a time and did not consider relations among different classes of feature interactions. To our best knowledge, there is no work that studied these two types of interactions in combination and investigates their relation, as we do it in this case study.

Internal Feature Interactions Detection of internal feature interactions is often used by techniques that aim at minimizing test-suite and test-effort for configurable systems. Reisner et al. [Rei+10], Nguyen et al. [Ngu+16], and Tartler et al. [Tar+12] apply symbolic evaluation, dynamic and static program analysis respectively to infer minimal sets of features responsible for a given code coverage. Kim et al. [KBK11] apply static program analysis to identify features that do not interact with other features with respect to the test-suite. Garvin et al. [GC11] explore a connection between feature interactions and interaction faults. Lillack et al. [LKB14] extend static taint analysis to automatically identify interactions among load-time configuration options.

External Feature Interactions A number of recently proposed performance prediction techniques for configurable systems by Guo et al. [Guo+13], Siegmund et al. [Sie+12], Sarkar et al. [Sar+15], Thereska et al. [The+10], Westermann et al. [Wes+12], and Zhang et al. [Zha+15] use machine-learning techniques, such as, CART, multivariable regression, and Fourier learning, for learning a performance function based on the performance measurements of a configuration sample. These techniques learn performance (external) feature interactions as an integral part of the overall black-box learning process, that is, without considering the internal feature interactions.

5.8 Summary

To answer one of the question of our dissertation, namely, how to use information about internal interactions to predict external interactions, we explored the relation among control-flow (internal) and performance (external) feature interactions that occur in configurable systems. Using the encryption library MBEDTLS and the database engine SQLITE as real-world subject systems, we identified control-flow and performance feature interactions using static program analysis and machine learning. Analyzing the interactions, we found that they can be related based on the involved features. By manually inspecting the code, we confirmed the causal relation between the interplay of features at the control-flow level and the identified performance interactions among the same features. Furthermore, based on the identified relation, we defined two predictors for performance feature interactions and conducted a preliminary evaluation of these predictors. The evaluation showed that the predictors have low precision and recall, presumably, because features also interact at the data-flow level. Future predictors based on the internal feature interactions should consider both control-flow and data-flow interactions to improve their predictive power.

Beside this negative result, using the identified relation among control-flow and performance feature interactions, we are still able to shrink the search space of performance feature interactions (by almost 300 times for MBEDTLS and by 4 times for SQLITE) without losing any of the performance feature interactions actually existing in our subject systems. Performance prediction techniques that rely on sampling can use our results to make their sampling more focused on configurations with potential performance interactions.

Overall, our results suggest that relations among internal and external interactions can be exploited to predict external interactions, which is one of the goals of our dissertation. The conceptual framework that we introduced in this chapter can be used by other researchers to explore relations among different kinds of internal and external interactions.

CHAPTER 6

Conclusion

Configurable software systems are a response of research and industrial practice to the growing number of user requirements and application scenarios that a software must fulfill nowadays. Software vendors build configurable systems with the goal of producing high-quality tailor-made software fast and cost-efficient. One of the major problems of producing high-quality configurable systems is the presence of unanticipated feature interactions that may lead to unexpected or suboptimal behavior of the system. Detecting feature interactions is complicated by potentially exponential explosion of configuration space of configurable systems.

The goal of this dissertation is to contribute to the solution of the feature interaction problem by studying the nature of feature interactions in a systematic and comprehensive way. We achieved this goal by gaining concrete empirical results about properties of feature interactions, their causes, their influence on performance of configurable systems, and about relations among feature interactions of different types.

Specifically, we made the following contributions:

1. We systematically compared family-based, feature-based, and product based analysis strategies using type-checking as a concrete analysis technique. The comparison was made regarding such aspects as the ability to detect different kinds of type errors and the time needed to run the analysis. The comparative evaluation was practically conducted on a set of 12 configurable systems using our own implementation of the techniques in Fuji (an extensible compiler for feature-oriented programming).

The evaluation showed that the family-based strategy is the most efficient. Used as the basis for Variability-aware type-checking based on this strategy is faster than the alternatives, it detects errors that are local to features and that involve several features, it also provides most detailed information about the errors found, which improves the usability of the type-checker. The drawback of this strategy is that it requires adaption of the existing type-checking tools.

This evaluation provides empirical data to the developers of variability-aware analysis techniques and tools—among which are also feature-interaction detection tools—about how different analysis strategies compare with respect to completeness and performance.

2. Using a variability-aware machine-learning technique we systematically studied the tradeoffs among prediction error, model size, and computation time of performance-influence models, which we use in this dissertation to automatically detect performance feature interactions. We found that the tradeoffs are marginal and that accurate and human-comprehensible influence models can be built in feasible time. More interestingly, we found that interactions of size 2 and 3 had the highest influence on performance and that this influence reduced with the growing size of the interactions. This findings about feature interactions have an immediate practical consequence for the techniques that aim at detecting feature interactions or that rely on sampling: using our findings they may focus on feature combinations that include 2 or 3 features, because they may highly likely give rise to the influential feature interactions.

   Furthermore, we investigated the causes for the detected performance feature interactions and found four reoccurring patterns: dominant configuration option, data pipeline, workload tuning, and domain-specific interactions. The patterns explain how decisions about a system's architecture may lead to the emergence of performance feature interactions. This knowledge can help to prevent or at least to anticipate the possible presence of performance feature interactions already in the early stages of the configurable systems development when architectural decisions are made.

3. Finally, to facilitate a more structural approach to studying the nature of feature interactions we classify them in two classes: internal and external interactions. Each of these classes we further subdivide into: structural and operational subclasses, and functional and non-functional subclasses respectively. Based on the empirical evidence, that internal feature interactions are generally easier to detect than the external ones, we conducted

a case study in which we investigated the relation among performance (external, non-functional) interactions and control-flow (internal, operational) interactions. With the goal that, if this relation exists, it can facilitate the prediction of external interactions based on the related internal ones. For the two subject systems that we used in our case study, we were able to confirm that such relation exists and potentially can be used to substantially reduce the search space for external feature interactions.

These results have immediate practical consequences for performance prediction techniques based on machine learning and relying on sampling for building a training dataset: By exploiting our findings they can make sampling more focused on the configurations that potentially include interacting features, which may improve their prediction accuracy. Moreover, other researchers can use the conceptual framework from our study to investigate possible relations among other kinds of interactions.

Avenues of Future Work

Feature interaction patterns. Feature interaction patterns link designer decisions about a system's architecture to possible feature interactions that emerge due to this decisions. With more empirical data on feature interactions and their causes new feature interaction patterns may be discovered. They then can be used as anti-patterns and help prevent feature interactions or as indicators for feature interactions that may emerge due to certain designer decisions.

Relation of external and internal feature interactions. Control-flow feature interactions that we studied in this dissertation is only one type of internal operational feature interactions. A further type is data-flow feature interactions that can also be studied with regard to the relation to performance or, again, other types of external feature interactions. Studying relations among further types of internal and external interactions using the conceptual framework that we proposed in this dissertation is a possible line of future research.

Prediction of feature interactions. With the growing amount of data about relations among internal and external feature interactions there may be a possibility of learning accurate predictors for functional and non-functional feature interactions. Finding and evaluating such predictors is another interesting line of research.

Bibliography

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[AB11] S. Apel and D. Beyer. “Feature Cohesion in Software Product Lines: An Exploratory Study”. In: *Proc. ICSE.* ACM, 2011, pp. 421–430.

[AH10] S. Apel and D. Hutchins. “A Calculus for Uniform Feature Composition”. In: *ACM TOPLAS* 32.5 (2010), 19:1–19:33.

[ALS08] S. Apel, T. Leich, and G. Saake. “Aspectual feature modules”. In: *IEEE Transactions on Software Engineering* 34.2 (2008), pp. 162–180.

[AL08] S. Apel and C. Lengauer. “Superimposition: A language-independent approach to software composition”. In: *International Conference on Software Composition.* Springer. 2008, pp. 20–35.

[Ape+12] S. Apel et al. “Access Control in Feature-Oriented Programming”. In: *Science of Computer Programming* 77.3 (2012), pp. 174–187.

[Ape+11] S. Apel et al. “Detection of Feature Interactions using Feature-Aware Verification”. In: *Proc. ASE.* IEEE, 2011, pp. 372–375.

[Ape+13a] S. Apel et al. “Exploring Feature Interactions in the Wild: The New Feature-interaction Challenge”. In: *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD).* ACM, 2013, pp. 1–8.

[Ape+13b] S. Apel et al. “Feature-Interaction Detection based on Feature-Based Specifications”. In: *Computer Networks* 57.12 (2013), pp. 2399–2409.

[Ape+13c] S. Apel et al. *Feature-Oriented Software Product Lines – Concepts and Implementation*. Springer-Verlag, 2013.

[Ape+13d] S. Apel et al. "Strategies for product-line verification: Case studies and experiments". In: *Proc. ICSE*. IEEE, 2013, pp. 482–491.

[Ape+10] S. Apel et al. "Type Safety for Feature-Oriented Product Lines". In: *Automated Software Engineering* 17.3 (2010), pp. 251–300.

[Bal+04] S. Balsamo et al. "Model-based performance prediction in software development: A survey". In: *IEEE Transactions on Software Engineering* 30.5 (2004), pp. 295–310.

[Bas+06] P. Bastian et al. "The distributed and unified numerics environment (DUNE)". In: *Proceedings of the Symposium on Simulation Technique in Hannover*. 2006, pp. 12–14.

[Bat04] D. Batory. "Feature-oriented programming and the AHEAD tool suite". In: *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*. IEEE. 2004, pp. 702–703.

[BHK11] D. Batory, P. Höfner, and J. Kim. "Feature interactions, products, and composition". In: *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 2011, pp. 13–22.

[BSR04] D. Batory, J. Sarvela, and A. Rauschmayer. "Scaling step-wise refinement". In: *IEEE Transactions on Software Engineering* 30.6 (2004), pp. 355–371.

[BSR10] D. Benavides, S. Segura, and A. Ruiz-Cortés. "Automated analysis of feature models 20 years later: A literature review". In: *Information Systems* 35.6 (2010), pp. 615–636.

[Ben07] I. Ben-Gal. "Bayesian networks". In: Wiley Online Library, 2007.

[BDS13] L. Bettini, F. Damiani, and I. Schaefer. "Compositional Type Checking of Delta-oriented Software Product Lines". In: *Acta Informatica* 50.2 (2013), pp. 77–122.

[Big98] T. Biggerstaff. "A perspective of generative reuse". In: *Annals of Software Engineering* 5.1 (1998), pp. 169–226.

[Bod+13] E. Bodden et al. "SPL$^{\text{LIFT}}$: Statically Analyzing Software Product Lines in Minutes Instead of Years". In: *Proc. PLDI*. ACM, 2013, pp. 355–364.

[Bor12] C. Borgelt. "Frequent item set mining". In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 2.6 (2012), pp. 437–456.

[Bow+89] T. Bowen et al. "The feature interaction problem in telecommunications systems". In: *Proceedings of the International Conference on Software Engineering for Telecommunication Switching Systems (SETSS)*. IEEE, 1989, pp. 59–62.

[Bra+13] C. Brabrand et al. "Intraprocedural Dataflow Analysis for Software Product Lines". In: *Trans. on Aspect-Oriented Software Development* 10 (2013), pp. 73–108.

[Bro+15] F. Brosig et al. "Quantitative evaluation of model-driven performance analysis and simulation of component-based architectures". In: *IEEE Transactions on Software Engineering* 41.2 (2015), pp. 157–175.

[BVK13] A. Brunnert, C. Vögele, and H. Krcmar. "Automatic Performance Model Generation for Java Enterprise Edition (EE) Applications". In: *Computer Performance Engineering: 10th European Workshop, EPEW 2013, Venice, Italy, September 16-17, 2013. Proceedings*. Springer, 2013, pp. 74–88.

[Bru05] G. Bruns. "Foundations for features". In: *Feature Interactions in Telecommunications and Software Systems VIII*. IOS Press, 2005, pp. 3–11.

[CM06] M. Calder and A. Miller. "Feature interaction detection by pairwise analysis of LTL properties: A case study". In: *Formal Methods in System Design* 28.3 (2006), pp. 213–261.

[Cal+03] M. Calder et al. "Feature Interaction: A Critical Review and Considered Forecast". In: *Computer Networks* 41.1 (2003), pp. 115–141.

[CS14] G. Chandrashekar and F. Sahin. "A survey on feature selection methods". In: *Computers & Electrical Engineering* 40.1 (2014), pp. 16–28.

[Cla+10] A. Classen et al. "Model checking lots of systems: Efficient verification of temporal properties in software product lines". In: *Proc. ICSE*. ACM. 2010, pp. 335–344.

[Cor+12] M. Cordy et al. "Simulation-Based Abstractions for Software Product-Line Model Checking". In: *Proc. ICSE*. ACM, 2012, pp. 672–682.

[Cor+10] V. Cortellessa et al. "A process to effectively identify "guilty" performance antipatterns". In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2010, pp. 368–382.

[CE00] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[DCB09] B. Delaware, W. Cook, and D. Batory. "Fitting the Pieces Together: A Machine-Checked Model of Safe Composition". In: *Proc. FSE*. ACM, 2009, pp. 243–252.

[Dom00] P. Domingos. "A unified bias-variance decomposition". In: *Proceedings of International Conference on Machine Learning (ICML)*. Morgan Kaufmann, 2000, pp. 231–238.

[Dom12] A. Dominguez. "Detection of Feature Interactions in Automotive Active Safety Features". PhD thesis. University of Waterloo, 2012.

[DR11] Thai Duong and Juliano Rizzo. "Here come the Ninjas. 2011". In: *Manuscript, https://web.archive.org/web/20150630133214/http://www.hpcc.ecs.soton.ac.uk/~dan/talks/bullrun/Beast.pdf* (2011).

[Fer+15] G. Ferreira et al. "Characterizing complexity of highly-configurable systems with variational call graphs: Analyzing configuration options interactions complexity in function calls". In: *Proc. HotSoS*. ACM, 2015, 17:1–2.

[Fly06] B. Flyvbjerg. "Five misunderstandings about case-study research". In: *Qualitative Inquiry* 12.2 (2006), pp. 219–245.

[GC11] B. Garvin and M. Cohen. "Feature interaction faults revisited: An exploratory study". In: *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2011, pp. 90–99.

[GS03] J. Greenfield and K. Short. "Software factories: assembling applications with patterns, models, frameworks and tools". In: *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM. 2003, pp. 16–27.

[GS06] C. Grelck and S.-B. Scholz. "SaC—A functional array language for efficient multi-threaded execution". In: *International Journal of Parallel Programming* 34.4 (2006), pp. 383–427.

[Guo+13] J. Guo et al. "Variability-aware performance prediction: A statistical learning approach". In: *Proc. ASE*. IEEE, 2013, pp. 301–311.

[Hal05] R. Hall. "Fundamental Nonmodularity in Electronic Mail". In: *Automated Software Engineering* 12.1 (2005), pp. 41–79.

[HTF09] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning 2nd edition*. Springer, 2009.

[Jac12] P. Jaccard. “The distribution of the flora in the alpine zone.” In: *New phytologist* 11.2 (1912), pp. 37–50.

[JZ98] Michael Jackson and Pamela Zave. “Distributed feature composition: A virtual architecture for telecommunications services”. In: *IEEE Transactions on Software Engineering* 24.10 (1998), pp. 831–847.

[Jam+13] G. James et al. *An introduction to statistical learning*. Vol. 112. Springer, 2013.

[Jay+07] P. Jayaraman et al. “Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis”. In: *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. LNCS 4735. Springer, 2007, pp. 151–165.

[JHF12] M. Johansen, Ø. Haugen, and F. Fleurey. “An Algorithm for Generating t-wise Covering Arrays from Large Feature Models”. In: *Proc. SPLC*. ACM, 2012, pp. 46–55.

[Joh+12] M. Johansen et al. “A Technique for Agile and Automatic Interaction Testing for Product Lines”. In: *Testing Software and Systems*. LNCS 7641. Springer, 2012, pp. 39–54.

[Kan+90] K. Kang et al. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. CMU/SEI-90-TR-21, 1990.

[KA09] C. Kästner and S. Apel. “Virtual separation of concerns-a second chance for preprocessors”. In: *Journal of Object Technology* 8.6 (2009), pp. 59–78.

[KAK08] C. Kästner, S. Apel, and M. Kuhlemann. “Granularity in software product lines”. In: *Software Engineering, 2008. ICSE’08. ACM/IEEE 30th International Conference on*. IEEE. 2008, pp. 311–320.

[KAO11] C. Kästner, S. Apel, and K. Ostermann. “The Road to Feature Modularity?” In: *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 2011, 5:1–5:8.

[Kas+09] C. Kastner et al. "FeatureIDE: A tool framework for feature-oriented software development". In: *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE. 2009, pp. 611–614.

[Käs+09] C. Kästner et al. "On the Impact of the Optional Feature Problem: Analysis and Case Studies". In: *Proceedings of the International Software Product Line Conference (SPLC)*. Software Engineering Institute, 2009, pp. 181–190.

[Käs+12] C. Kästner et al. "Type Checking Annotation-Based Product Lines". In: *ACM Trans. on Software Engineering and Methodology* 21.3 (2012), 14:1–14:29.

[Kic+97] G. Kiczales et al. "Aspect-oriented programming". In: *Object-oriented programming (ECOOP)* (1997), pp. 220–242.

[KBK11] C. Kim, D. Batory, and S. Khurshid. "Reducing combinatorics in testing product lines". In: *Proc. AOSD*. ACM, 2011, pp. 57–68.

[KKB08] C. Kim, C. Kästner, and D. Batory. "On the modularity of feature interactions". In: *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 2008, pp. 23–34.

[Kol+13] S. Kolesnikov et al. "A comparison of product-based, feature-based, and family-based type checking". In: *Generative Programming: Concepts and Experiences, (GPCE)*. ACM, 2013, pp. 115–124.

[Kol+18] S. Kolesnikov et al. "Tradeoffs in modeling performance of highly configurable software systems". In: *Software and Systems Modeling (SoSyM)* (Feb. 2018). Online first, pp. 1–19.

[Kuc+13] S. Kuckuk et al. "A Generic Prototype to Benchmark Algorithms and Data Structures for Hierarchical Hybrid Grids". In: *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*. IOS Press. 2013, pp. 813–822.

[KWG04] D. Kuhn, D. Wallace, and A. Gallo, Jr. "Software Fault Interactions and Implications for Software Testing". In: *IEEE Transactions on Software Engineering (TSE)* 30.6 (2004), pp. 418–421.

[LTP09] K. Lauenroth, S. Toehning, and K. Pohl. "Model Checking of Domain Artifacts in Product Line Engineering". In: *Proc. ASE*. IEEE, 2009, pp. 269–280.

[LA11] C. Lengauer and S. Apel. "Feature-Oriented System Design and Engineering". In: *International Journal of Software and Informatics (IJSI)* 5.1–2, Part II (2011). Special Issue on Foundations and Practice of Systems and Software Engineering, Festschrift in Honor of Manfred Broy., pp. 231–244.

[LKF02] H. Li, S. Krishnamurthi, and K. Fisler. "Verifying Cross-cutting Features as Open Systems". In: *Proc. FSE.* ACM, 2002, pp. 89–98.

[LW02] A. Liaw and M. Wiener. "Classification and regression by randomForest". In: *R News* 2.3 (2002), pp. 18–22.

[LKA11] J. Liebig, C. Kästner, and S. 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.* ACM. 2011, pp. 191–202.

[Lie15] Jörg Liebig. "Analysis and Transformation of Configurable Systems." PhD thesis. University of Passau, 2015.

[Lie+10] J. Liebig et al. "An analysis of the variability in forty preprocessor-based software product lines". In: *Software Engineering, 2010 ACM/IEEE 32nd International Conference on.* Vol. 1. IEEE. 2010, pp. 105–114.

[Lie+13] J. Liebig et al. "Scalable Analysis of Variable Software". In: *Proc. ESEC/FSE.* ACM, 2013, pp. 81–91.

[LKB14] M. Lillack, C. Kästner, and E. Bodden. "Tracking load-time configuration options". In: *Proc. ASE.* ACM, 2014, pp. 445–456.

[LBL06] J. Liu, D. Batory, and C. Lengauer. "Feature oriented refactoring of legacy applications". In: *Proceedings of the International Conference on Software Engineering (ICSE).* ACM, 2006, pp. 112–121.

[LBL11] Jing Liu, Samik Basu, and Robyn Lutz. "Compositional model checking of software product lines using variation point obligations". In: *Automated Software Engineering* 18.1 (Mar. 2011), pp. 39–76.

[MB07] O. Maqbool and H. Babri. "Hierarchical clustering for software architecture recovery". In: *IEEE Transactions on Software Engineering* 33.11 (2007), pp. 759–780.

[Mem+12] R. Membarth et al. "Generating Device-specific GPU Code for Local Operators in Medical Imaging". In: *Proceedings of the International Parallel & Distributed Processing Symposium (IPDPS).* IEEE. 2012, pp. 569–581.

[Moh92] C. Mohan. “Interactions Between Query Optimization and Concurrency Control”. In: *Proceedings of the International Workshop on Research Issues on Data Engineering: Transaction and Query Processing (RIDE-TQP)*. IEEE, 1992, pp. 26–35.

[MP04] J. Myung and M. Pitt. “Model comparison methods”. In: *Methods in Enzymology* 383 (2004), pp. 351–366.

[Ngu+16] T. Nguyen et al. “iGen: Dynamic interaction inference for configurable software”. In: *Proc. FSE*. ACM, 2016, pp. 655–665.

[NLN08] A. Nhlabatsi, R. Laney, and B. Nuseibeh. “Feature interaction: The security threat from within software systems”. In: *Progress in Informatics* 5 (2008), pp. 75–89.

[OMR10] S. Oster, F. Markert, and P. Ritter. “Automated Incremental Pairwise Testing of Software Product Lines”. In: *Proc. SPLC*. LNCS 6287. Springer, 2010, pp. 196–210.

[Pas+15] L. Passos et al. “Feature scattering in the large: a longitudinal study of Linux kernel device drivers”. In: *Proceedings of the 14th International Conference on Modularity*. ACM, 2015, pp. 81–92.

[Pie02] B. C. Pierce. *Types and programming languages*. MIT press, 2002.

[PBvD05] K. Pohl, G. Böckle, and F. van Der Linden. *Software product line engineering: foundations, principles and techniques*. Springer, 2005.

[Poo00] R. Pooley. “Software Engineering and Performance: A Roadmap”. In: *Proceedings of the Conference on The Future of Software Engineering*. ACM, 2000, pp. 189–199.

[Pre97] C. Prehofer. “Feature-oriented programming: A fresh look at objects”. In: *Object-Oriented Programming (ECOOP)* (1997), pp. 419–443.

[Pre04] C. Prehofer. “Plug-and-play composition of features and feature interactions with statechart diagrams”. In: *Software and Systems Modeling* 3.3 (2004), pp. 221–234.

[Qia+13] Y. Qiao et al. “Analyzing malware by abstracting the frequent itemsets in API call sequences”. In: *Proc. TrustCom*. IEEE, 2013, pp. 265–270.

[Rei+10] E. Reisner et al. “Using symbolic evaluation to understand behavior in configurable software systems”. In: *Proc. ICSE*. ACM, 2010, pp. 445–454.

[RGP12] T. Repasi, S. Giessl, and C. Prehofer. “Using model-checking for the detection of non-functional feature interactions”. In: *Proceedings of the International Conference on Intelligent Engineering Systems (INES)*. IEEE, 2012, pp. 167–172.

[Ros+09] M. Rosenmüller et al. “Tailor-Made Data Management for Embedded Systems: A Case Study on Berkeley DB”. In: *Data & Knowledge Engineering (DKE)* 68.12 (2009), pp. 1493–1512.

[Sal04] David Salomon. *Data compression: the complete reference*. Springer Science & Business Media, 2004.

[SW11] C. Sammut and G. I Webb. *Encyclopedia of machine learning*. Springer, 2011.

[Sar+15] A. Sarkar et al. “Cost-efficient sampling for performance prediction of configurable systems”. In: *Proc. ASE*. IEEE, 2015, pp. 342–352.

[Say+13] A. Sayyad et al. “Scalable product line configuration: A straw to break the camel’s back”. In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 465–474.

[SY93] L. Schruben and E. Yucesan. “Complexity of Simulation Models a Graph Theoretic Approach”. In: *Proceedings of the Conference on Winter Simulation*. ACM, 1993, pp. 641–649.

[SSS07] F. Shull, J. Singer, and D.I.K. Sjøberg. *Guide to Advanced Empirical Software Engineering*. Springer, 2007.

[Sie12] N. Siegmund. “Measuring and Predicting Non-Functional Properties of Customizable Programs”. PhD thesis. University of Magdeburg, 2012.

[SvRA13] N. Siegmund, A. von Rhein, and S. Apel. “Family-based Performance Measurement”. In: *Proc. GPCE*. ACM, 2013, pp. 95–104.

[Sie+15] N. Siegmund et al. “Performance-influence models for highly configurable systems”. In: *Proc. ESEC/FSE*. ACM, 2015, pp. 284–294.

[Sie+12] N. Siegmund et al. “Predicting Performance via Automated Feature-Interaction Detection”. In: *Proc. ICSE*. IEEE, 2012, pp. 167–177.

[Sie+13] N. Siegmund et al. “Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption”. In: *Information and Software Technology (IST)* 55.3 (2013), pp. 491–507.

[Sim13] D. Simon. *Evolutionary optimization algorithms*. John Wiley & Sons, 2013.

[SGC07] N. Singh, C. Gibbs, and Y. Coady. "C-CLR: a tool for navigating highly configurable system software". In: *Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*. ACM. 2007, p. 9.

[SPL17] SPLC. *Hall of Fame*. 2017. URL: `https://web.archive.org/web/20171231211902/http://splc.net/hall-of-fame/` (visited on 03/20/2018).

[SC09] D. Steinberg and P. Colla. "CART: Classification and regression trees". In: *The top ten algorithms in data mining* 9 (2009), p. 179.

[SC08] I. Steinwart and A. Christmann. *Support Vector Machines*. Springer, 2008.

[TOS02] P. Tarr, H. Ossher, and S. Sutton Jr. "Hyper/J: Multi-Dimensional Separation of Concerns for Java". In: *Proc. ICSE*. ACM, 2002, pp. 689–690.

[Tar13] R. Tartler. "Mastering variability challenges in Linux and related highly-configurable system software". PhD thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2013.

[Tar+12] R. Tartler et al. "Configuration coverage in the analysis of large-scale system software". In: *SIGOPS Operating Systems Review (ACM OSR)* 45.3 (2012), pp. 10–14.

[Tar+11] R. Tartler et al. "Feature Consistency in Compile-time-configurable System Software: Facing the Linux 10,000 Feature Problem". In: *Proc. EuroSys*. ACM, 2011, pp. 47–60.

[Tha+07] S. Thaker et al. "Safe Composition of Product Lines". In: *Proc. GPCE*. ACM, 2007, pp. 95–104.

[The+10] E. Thereska et al. "Practical performance models for complex, popular applications". In: *SIGMETRICS Performance Evaluation Review* 38.1 (2010), pp. 1–12.

[TBK09] T. Thum, D. Batory, and C. Kastner. "Reasoning about edits to feature models". In: *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society. 2009, pp. 254–264.

[Thü+14] T. Thüm et al. "A classification and survey of analysis strategies for software product lines". In: *ACM Computing Surveys (CSUR)* 47.1 (2014), p. 6.

[Thü+12a] T. Thüm et al. *Analysis Strategies for Software Product Lines*. Tech. rep. FIN-004-2012. University of Magdeburg, 2012.

[Thü+12b] T. Thüm et al. "Family-based Deductive Verification of Software Product Lines". In: *Proc. GPCE*. ACM, 2012, pp. 11–20.

[vRhe16] A. von Rhein. "Analysis Strategies for Configurable Systems". PhD thesis. Universität Passau, 2016.

[vRhe+15] A. von Rhein et al. "Presence-condition simplification in highly configurable systems". In: *Proc. ICSE*. Vol. 1. IEEE, 2015, pp. 178–188.

[vRhe+13] A. von Rhein et al. "The PLA Model: On the Combination of Product-Line Analyses". In: *Proc. VaMoS*. ACM, 2013, pp. 73–80.

[Wal87] J. Wallace. "The control and transformation metric: Toward the measurement of simulation model complexity". In: *Proceedings of the Conference on Winter Simulation*. ACM, 1987, pp. 597–603.

[WE05] M. Weiss and B. Esfandiari. "On Feature Interactions Among Web Services." In: *International Journal of Web Services Research (IJWSR)* 2.4 (2005), pp. 22–47.

[WEL07] M. Weiss, B. Esfandiari, and Y. Luo. "Towards a classification of web service feature interactions". In: *Computer Networks* 51.2 (2007), pp. 359–381.

[Wes+12] D. Westermann et al. "Automated inference of goal-oriented performance prediction functions". In: *Proc. ASE*. ACM, 2012, pp. 190–199.

[Wic+95] BA Wichmann et al. "Industrial perspective on static analysis". In: *Software Engineering Journal* 10.2 (1995), pp. 69–75.

[YCP06] C. Yilmaz, M. Cohen, and A. Porter. "Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces". In: *IEEE Transactions on Software Engineering (TSE)* 32.1 (2006), pp. 20–34.

[Yin03] R. Yin. *Case Study Research–Design and Methods*. Sage, 2003.

[Zha+16] Y. Zhang et al. "A mathematical model of performance-relevant feature interactions". In: *Proceedings of the International Systems and Software Product Line Conference*. ACM. 2016, pp. 25–34.

[Zha+15] Y. Zhang et al. "Performance prediction of configurable software systems by fourier learning". In: *Proc. ASE*. IEEE. 2015, pp. 365–373.

Appendix

A.1 Influence of Configuration Options and their Interactions

Table A.1: A list of the most influential configuration options and interactions grouped by subject system. For each configuration option and interaction, we indicate its influence on performance and give a description. We also denote if the configuration option (if enabled) or interaction (if present) increases (↑) or decreases (↓) the performance of the system.

№	Config. Option/ Interaction	Influence	Description
	Apache		
1	KeepAlive	876.61 ↑	Allow multiple requests over the same TCP connection; speeds up transmission
2	HostnameLookups	-233.61 ↓	Perform a DNS lookup for every request; causes communication overhead
3	InMemory	197.48 ↑	Copy specified files into RAM on startup; reduces I/O
4	AccessLog	-116.80 ↓	Log every request in the logfile on disk; causes I/O overhead
5	InMemory · KeepAlive	166.87 ↑	Files cached in RAM are served over the same connection; speeds up transmission

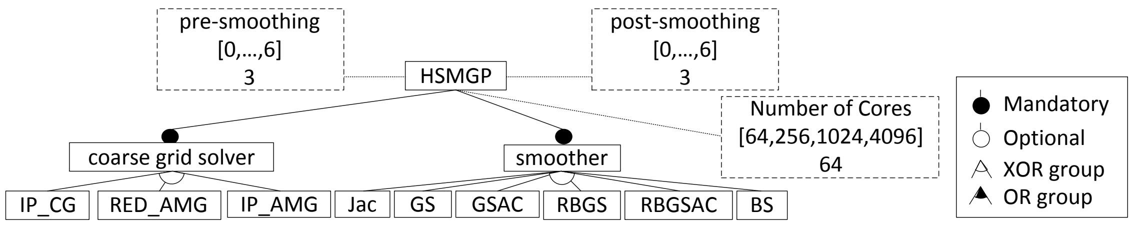
6	AccessLog · KeepAlive	-157.49↓	Disk I/O induced by logging reduces performance even if KeepAlive is enabled
AJSTATS			
1	CodeFormatter	3048.44↓	Preprocess code for parsing; avoids unnecessary parsing
2	Interfaces	-304.99↑	Disable interfaces statistics; speeds up processing
3	ClassMethods	-198.27↑	Disable calls-methods statistics; speeds up processing
4	ClassConstructors · CodeFormatter	-664.49↑	Disable constructor statistics; the effect is increased in the presence of CodeFormatter
BDB-C			
1	PS16K	-1.10↑	Set page size to 16 K; read longer portions of data from the disk and speed up data retrieval
2	PS32K	-1.06↑	Same as 1
3	HAVE_CRYPTO · HAVE_HASH · PS32K	43.29↓	Hash data structure performs poorly if the stored data is encrypted
4	HAVE_CRYPTO · HAVE_HASH · PS16K	16.73↓	Same as 3, but the performance decrease is smaller with smaller page size
BDB-J			
1	S1MiB	44078↓	Sets recovery log size to 1 MB (default is 100 MB); increases I/O overhead
2	Finest · S1MiB	222790↓	Save maximum possible recovery information in multiple small recovery log files; increases I/O overhead
CLASP			
1	heuristicUnit	345493↓	Enable Unit heuristic with an expensive Lookahead operation
2	eq	-92677↑	Enable preprocessing that may reduce the problem and speed up solving
3	heuristic	-35218↑	Enable Berkmin-Heuristic; faster than Unit-heuristic
4	satPreproYes	19959↓	Enable SatElite-like preprocessing that may reduce the problem; preprocessing introduces overhead

5	eq · heuristicUnit	-163900↑	The heuristic works on the reduced problem and can solve it faster
6	heuristicUnit · satPreproYes	-148980↑	The problem is reduced through pre-processing and solved faster by the Unit heuristic
	DUNE		
1	post0	2793.76↓	Disable postprocessing steps; without postprocessing the main algorithm requires more time to calculate a solution, which results in a decreased performance
2	cells50	-1605.82↑	Set the size of the computation domain (workload) to 50 (the smallest workload)
3	CGSolver · post0 · pre1	17946.17↓	The interaction describes a data pipeline including a solver, a post, and a pre-processing step
4	CGSolver · cells55 · post0 · pre1	12810.57↓	The interaction describes a data pipeline as in 3 plus the cells55 option, which sets the highest workload for the pipeline
	HSMGP		
1	Smoother_GSACBE	4669.89↓	Enable the GSACBE smoother, which is an essential part of the multigrid algorithm; GSACBE is one of the slower smoothers
2	Smoother_GS	-513.12↑	Enable the GS smoother, which is faster than GSACBE
3	Smoother_GSACBE · numPost_0 · numPre_1	-4976.29↑	The interaction denotes a data pipeline with a pre, post-processing, and a smoother
4	Smoother_GSACBE · numPost_0 · numPre_2	-3377.14↑	Same as in 3, but increasing the number of pre-processing steps reduces performance compared to 3
	LLVM		
1	gvn	24.27↓	Enable Global Value Numbering optimization; introduces computational overhead
2	instcombine	17.61↓	Enable combining of redundant instructions; introduces computational overhead

3	inline	10.67↓	Enable code inlining; introduces computational overhead
4	inline · licm	31.57↓	More code is inlined in the loops that are processed by licm, introducing more computation overhead and decreasing performance
Lrzip			
1	compressionZpaq	2032161.26↓	Enable ZPAQ compression (slower than the default BZip2)
2	compression	193262.66↓	Enable the default BZip2 compression
3	compressionZpaq · level9	3433850.93↓	The interaction describes the influence of ZPAQ algorithm with the highest compression level 9 on performance
4	compressionZpaq · level8	3415670.93↓	Same as in 3, but a lower compression level results in a slightly increased performance compared to 3
x264			
1	ref_1	-349.61↑	Set the number of reference frames to 1 (the default is 9); less reference frames means less workload and higher performance
2	ref_5	-178.68↑	Same as in Line 1, but the increase in performance is twice as less as in Line 1 because of more reference frames
3	no_fast_pskip · ref_9	110.22↓	Fast-P-Skip can increase encoding speed; it is more effective with more reference frames; disabling Fast-P-Skip with 9 reference frames decreases performance

A.2 Materials Presented to the Interviewees

Interview I (HSMGP)



(round 9, error 18.27)

$$\begin{array}{rl}
+\,310.0 \cdot root & +\,200.0 \cdot Smoother_GSACBE \\
-\,120.0 \cdot Smoother_GS & -\,120.0 \cdot Smoother_JAC \\
-\,160.0 \cdot Smoother_GSAC & -\,13.0 \cdot CGS_IP_AMG \\
+\,4.2 \cdot pre \cdot pre & +\,310.0 \cdot Smoother_GSACBE \cdot post \\
+\,45.0 \cdot Smoother_GSACBE \cdot pre \cdot pre & +\,3.6 \cdot Smoother_GSAC \cdot post \cdot post
\end{array}$$

(round 13, error 7.17)

$$\begin{array}{rl}
+\,150.0 \cdot root & -\,30.0 \cdot Smoother_GSACBE \\
-\,120.0 \cdot Smoother_GS & -\,120.0 \cdot Smoother_JAC \\
-\,85.0 \cdot Smoother_GSAC & -\,13.0 \cdot CGS_IP_AMG \\
+\,28.0 \cdot post & +\,27.0 \cdot pre \\
+\,1.4 \cdot pre \cdot pre & +\,300.0 \cdot Smoother_GSACBE \cdot post \\
+\,310.0 \cdot Smoother_GSACBE \cdot pre & +\,0.0027 \cdot post \cdot numCore \\
-\,1.8 \cdot Smoother_GSACBE \cdot pre \cdot pre & -\,0.90 \cdot Smoother_GSAC \cdot post \cdot post
\end{array}$$

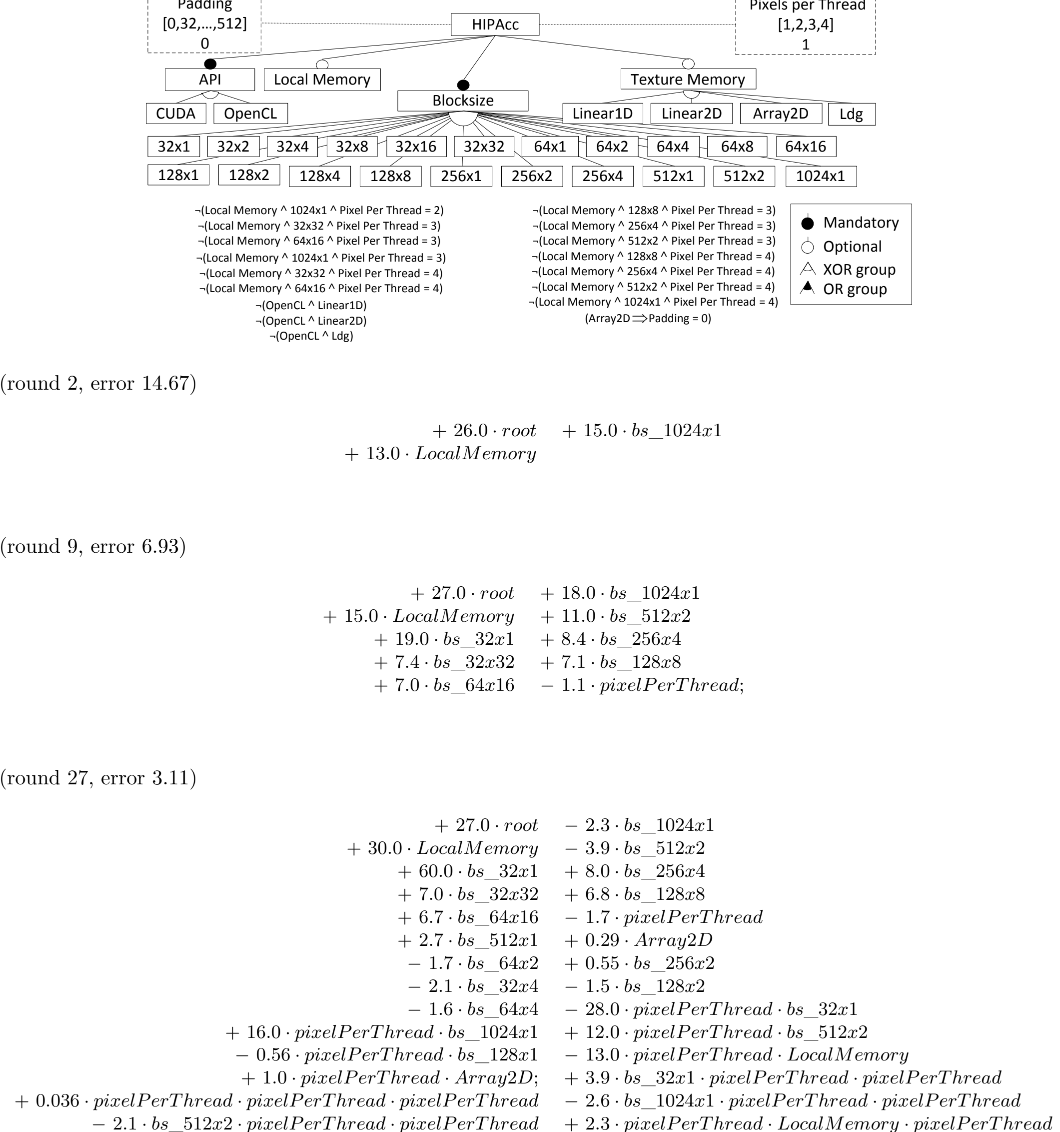
(round 19, error 2.83)

$$\begin{array}{rl}
+\,120.0 \cdot root & +\,13.0 \cdot Smoother_GSACBE \\
-\,31.0 \cdot Smoother_GS & -\,26.0 \cdot Smoother_JAC \\
-\,11.0 \cdot Smoother_GSAC & -\,17.0 \cdot CGS_IP_AMG \\
+\,21.0 \cdot post & +\,18.0 \cdot pre \\
+\,1.8 \cdot pre \cdot pre & +\,290.0 \cdot Smoother_GSACBE \cdot post \\
+\,320.0 \cdot Smoother_GSACBE \cdot pre & +\,0.0027 \cdot post \cdot numCore \\
+\,19.0 \cdot pre \cdot Smoother_GSRBAC & +\,12.0 \cdot pre \cdot Smoother_GSRB \\
+\,2.0 \cdot post \cdot post & -\,1.6 \cdot Smoother_GSACBE \cdot pre \cdot pre \\
-\,2.1 \cdot Smoother_GSAC \cdot post \cdot post & -\,0.41 \cdot CGS_IP_AMG \cdot pre \cdot pre \\
-\,2.5 \cdot Smoother_GS \cdot post \cdot post & -\,2.4 \cdot Smoother_JAC \cdot post \cdot post
\end{array}$$

Complexity

$$\begin{array}{r}
2.0 \cdot Smoother_JAC \\
2.0 \cdot pre \cdot post \\
2.0 \cdot Smoother_JAC \cdot pre \cdot post \\
2.0 \cdot Smoother_JAC \cdot pre \cdot post \cdot numCores
\end{array}$$

Interview II (HIPACC)



(round 2, error 14.67)

$$\begin{aligned} &+ 26.0 \cdot root && + 15.0 \cdot bs_1024x1 \\ &+ 13.0 \cdot LocalMemory && \end{aligned}$$

(round 9, error 6.93)

$$\begin{aligned} &+ 27.0 \cdot root && + 18.0 \cdot bs_1024x1 \\ &+ 15.0 \cdot LocalMemory && + 11.0 \cdot bs_512x2 \\ &+ 19.0 \cdot bs_32x1 && + 8.4 \cdot bs_256x4 \\ &+ 7.4 \cdot bs_32x32 && + 7.1 \cdot bs_128x8 \\ &+ 7.0 \cdot bs_64x16 && - 1.1 \cdot pixelPerThread; \end{aligned}$$

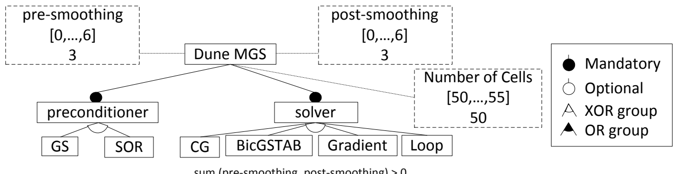
(round 27, error 3.11)

$$\begin{aligned} &+ 27.0 \cdot root && - 2.3 \cdot bs_1024x1 \\ &+ 30.0 \cdot LocalMemory && - 3.9 \cdot bs_512x2 \\ &+ 60.0 \cdot bs_32x1 && + 8.0 \cdot bs_256x4 \\ &+ 7.0 \cdot bs_32x32 && + 6.8 \cdot bs_128x8 \\ &+ 6.7 \cdot bs_64x16 && - 1.7 \cdot pixelPerThread \\ &+ 2.7 \cdot bs_512x1 && + 0.29 \cdot Array2D \\ &- 1.7 \cdot bs_64x2 && + 0.55 \cdot bs_256x2 \\ &- 2.1 \cdot bs_32x4 && - 1.5 \cdot bs_128x2 \\ &- 1.6 \cdot bs_64x4 && - 28.0 \cdot pixelPerThread \cdot bs_32x1 \\ &+ 16.0 \cdot pixelPerThread \cdot bs_1024x1 && + 12.0 \cdot pixelPerThread \cdot bs_512x2 \\ &- 0.56 \cdot pixelPerThread \cdot bs_128x1 && - 13.0 \cdot pixelPerThread \cdot LocalMemory \\ &+ 1.0 \cdot pixelPerThread \cdot Array2D; && + 3.9 \cdot bs_32x1 \cdot pixelPerThread \cdot pixelPerThread \\ &+ 0.036 \cdot pixelPerThread \cdot pixelPerThread \cdot pixelPerThread && - 2.6 \cdot bs_1024x1 \cdot pixelPerThread \cdot pixelPerThread \\ &- 2.1 \cdot bs_512x2 \cdot pixelPerThread \cdot pixelPerThread && + 2.3 \cdot pixelPerThread \cdot LocalMemory \cdot pixelPerThread \end{aligned}$$

Complexity

$$\begin{aligned} &2.0 \cdot Smoother_JAC \\ &2.0 \cdot pre \cdot post \\ &2.0 \cdot Smoother_JAC \cdot pre \cdot post \\ &2.0 \cdot Smoother_JAC \cdot pre \cdot post \cdot numCores \end{aligned}$$

Interview III (DUNE)



(round 1, error 13.74)

$$- 23000.0 \cdot root \quad + 640.0 \cdot cells$$

(round 13, error 7.26)

$$\begin{aligned}
+ 180000.0 \cdot root \quad & - 5300.0 \cdot cells \\
- 1000.0 \cdot post \quad & + 6800.0 \cdot GradientSolver \\
+ 600.0 \cdot BiCGSTABSolver \quad & - 110.0 \cdot cells \cdot GradientSolver \\
+ 2.6 \cdot cells \cdot pre \quad & + 1.9 \cdot cells \cdot post \cdot post \\
+ 530.0 \cdot GradientSolver \cdot pre \cdot pre \quad & + 0.70 \cdot cells \cdot cells \cdot cells \\
+ 380.0 \cdot GradientSolver \cdot post \cdot post \quad & - 38.0 \cdot BiCGSTABSolver \cdot pre \cdot pre \\
- 11.0 \cdot cells \cdot GradientSolver \cdot pre \cdot pre \quad & - 8.3 \cdot cells \cdot GradientSolver \cdot post \cdot post
\end{aligned}$$

(round 25, error 3.54)

$$\begin{aligned}
+ 670000.0 \cdot root \quad & - 19000.0 \cdot cells \\
- 2800.0 \cdot post \quad & - 5000.0 \cdot GradientSolver \\
+ 1200.0 \cdot BiCGSTABSolver \quad & + 11000.0 \cdot pre \\
+ 130.0 \cdot cells \cdot GradientSolver \quad & - 200.0 \cdot cells \cdot pre \\
+ 1700.0 \cdot post \cdot SeqSOR \quad & - 350.0 \cdot post \cdot GradientSolver \\
+ 220.0 \cdot post \cdot BiCGSTABSolver \quad & + 8.3 \cdot cells \cdot post \cdot post \\
+ 12000.0 \cdot GradientSolver \cdot pre \cdot pre \quad & + 2.4 \cdot cells \cdot cells \cdot cells \\
- 130.0 \cdot GradientSolver \cdot post \cdot post \quad & - 16.0 \cdot BiCGSTABSolver \cdot pre \cdot pre \\
- 0.098 \cdot cells \cdot pre \cdot SeqGS \quad & - 4.6 \cdot cells \cdot pre \cdot post \\
- 270.0 \cdot post \cdot SeqSOR \cdot post \quad & - 440.0 \cdot cells \cdot GradientSolver \cdot pre \cdot pre \\
+ 2.7 \cdot cells \cdot GradientSolver \cdot post \cdot post \quad & + 2.2 \cdot cells \cdot pre \cdot post \cdot LoopSolver \\
+ 0.015 \cdot cells \cdot cells \cdot cells \cdot CGSolver \quad & + 9.7 \cdot cells \cdot pre \cdot post \cdot SeqGS \\
- 1.6 \cdot cells \cdot pre \cdot SeqGS \cdot post \cdot post \quad & + 3.8 \cdot GradientSolver \cdot pre \cdot pre \cdot cells \cdot cells
\end{aligned}$$

Complexity

$$\begin{aligned}
2.0 \cdot Smoother_JAC \\
2.0 \cdot pre \cdot post \\
2.0 \cdot Smoother_JAC \cdot pre \cdot post \\
2.0 \cdot Smoother_JAC \cdot pre \cdot post \cdot numCores
\end{aligned}$$