Carnegie Mellon University

Ph.D. Thesis Proposal

White-box Analysis for Modeling and Debugging the Performance of Configurable Systems

Miguel Velez
Carnegie Mellon University

December 14, 2020

Thesis Committee

Christian Kästner (Chair)
Carnegie Mellon University

Claire Le Goues
Carnegie Mellon University

Rohan Padhye
Carnegie Mellon University

Norbert Siegmund
Leipzig University

Abstract

Most software systems today are configurable, providing numerous configuration options, which allow the system to be customized, in terms of functionality and quality attributes, to satisfy specific requirements and needs. The flexibility to customize these systems, however, comes with the cost of increased complexity. The large number of configuration options makes tracking how options and their interactions affect the functionality and quality attributes of systems a difficult task. For this reason, users are often overwhelmed with the large number of options and change options in a trial-and-error fashion without understanding the resulting effects. Likewise, understanding how large number of options interact complicates the process that developers follow to develop, test, and maintain large configuration spaces.

Performance, in terms of execution time, and often directly correlated energy consumption and operational costs, is one of the most important quality attributes for users and developers of configurable systems. Due to the large number of configuration options, users often struggle to configure the systems to run them efficiently, in terms of performance, and debugging surprising performance behaviors is usually challenging for developers.

Several approaches existing to understand how options and their interactions affect the performance of configurable systems. The approaches, however, treat systems as black-boxes, combining different sampling and machine learning techniques, resulting in tradeoffs between measurement effort and accuracy. Additionally, the techniques only analyze the end-to-end performance of the systems, whereas developers debugging unexpected performance behaviors need to understand how options affect the performance in the implementation.

In this thesis, we aim to analyze the performance of configurable systems using white-box techniques. By analyzing the implementation of configurable systems, we efficiently and accurately model the end-to-end performance of the systems, which allows users to make informed tradeoff and configuration decisions to run systems efficiently and helps developers understand how options affect the performance of the systems.

To further help developers debug the performance of their systems in the implementation, we efficiently an accurately model the local performance of regions and aid developers to trace how options affect the performance of those regions. This information helps developers locate where options affect the performance of a system and how options are used in the implementation to affect the performance.

The contributions in this thesis help reduce the energy consumption and operational costs of running configurable systems by helping (1) users to make informed configurations decisions and (2) developers to debug performance behavior issues in their systems.

Contents

Chapter 1

Introduction

Most of today's software systems, such as databases, Web servers, libraries, frameworks, and compilers, provide configuration options to customize the behavior of a system to satisfy a large variety of users' requirements [Apel et al., 2013]. Configuration options allow users to configure a system to satisfy their specific needs, in terms of functionality and quality attributes.

The flexibility provided by configuration options, however, comes at a cost. The large number of configuration options makes tracking how options and their interactions influence the functionality and quality attributes of systems a difficult task. For this reason, users are often overwhelmed with the large number of options and change options in a trial-and-error fashion without understanding the resulting effects [Apel et al., 2013; Hubaux et al., 2012; Xu et al., 2015, 2013] Likewise, a large number of options complicates the process that developers follow to develop, test, and maintain large configuration spaces [Behrang et al., 2015; Halin et al., 2018; Jin et al., 2014; Melo et al., 2016, 2017].

Performance, in terms of execution time, and often directly correlated energy consumption and operational costs, is one of the most important quality attributes for users and developers of configurable systems [Gelenbe and Caseau, 2015; Manotas et al., 2016; Pinto and Castor, 2017]. From the user's perspective, they want to efficiently run systems to reduce energy consumption and operational costs, but at the same time, with the functionality that satisfies their specific needs, which may require making tradeoff decisions between operational costs functionality [Jabbarvand et al., 2015; Kern et al., 2011; Munoz, 2017; Wilke et al., 2013; Zhang et al., 2014]. From the developer's perspective, they want to release efficient configurable systems to provide high quality user experience for attracting new and retaining existing users [Chowdhury and Hindle, 2016; Gui et al., 2016; Hasan et al., 2016; Li et al., 2016; Malik et al., 2015; Pereira et al., 2016]. However understanding how options and their interactions affect the performance of these systems, for users to make informed configuration decisions and for developers to design and implement efficient software, is challenging due to their large configuration spaces.

1.1 Motivating Scenario

We next describe a scenario to showcase the challenge that users and developers face to understand how options and their interactions affect the performance in large in larger configuration spaces. Berkeley DB is an open source embedded database library with over 50 options that affect the functionality of the database, its components, and the quality attributes of the system, including performance.[1] Fig. 1.1 shows a ranking of 2000 randomly selected configurations, in terms of execution time, from the fastest to the slowest configurations when populating a database with $500,000$ entries. In this system, the options not only drastically change the execution time from the fastest to the slowest configurations, but they also produce complex performance behavior (i.e., numerous steps in the figure) that prevents users and developers from easily understanding how options affect the performance of the system.

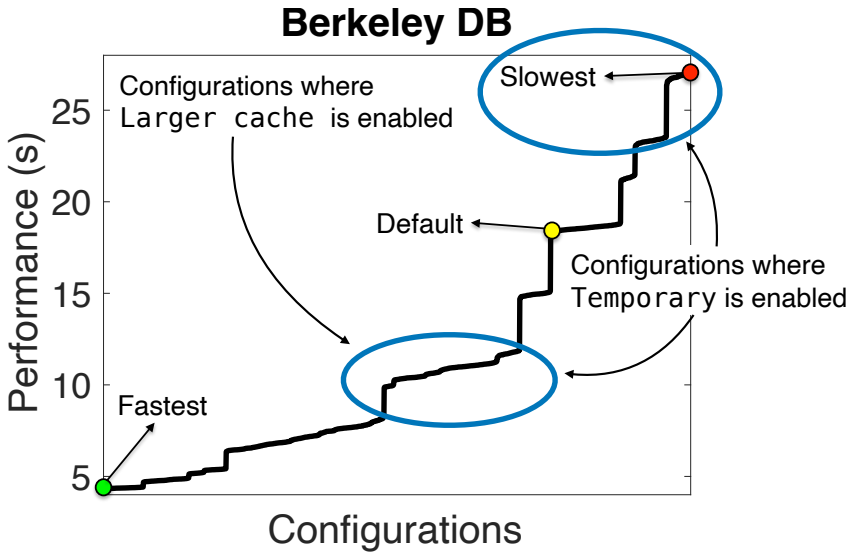

Figure 1.1: Ranked performance behavior, from fastest to slowest execution time, of 2000 randomly selected Berkeley DB configurations when populating a database with $500,000$ entries. Note the influence on the execution time of the interactions between `Temporary` and `Larger cache`. Enabling both options decreases the execution time compared to the default configuration, while only enabling the former increases the execution time.

User's perspective

Users of Berkeley DB want to efficiently run the system for their specific needs, but are often unaware of how options affect the functionality and performance of the system due to the large and complex performance behavior of the configuration space. For instance, users who want to populate a database could enable both `Temporary` and `Large cache` to reduce the execution time if their requirements allow for an in-memory database and their environment has large enough memory. However, identifying those configuration decisions and effects

[1] https://www.oracle.com/database/technologies/related/berkeleydb.html

on the performance of the system is challenging in large configuration spaces with complex performance behavior (e.g., 2^{50} configurations if only considering two values for each option). For these reasons, users often resort to using the default configuration, resulting in executing their systems inefficiently and increase energy consumption and operational costs.

Users concerned with minimizing performance, energy consumption, and operational costs could use a search strategy to optimize the performance of the system [Nair et al., 2017; Oh et al., 2017]. However, such strategies only aim to find the fastest configurations and do not take into account the functionality that users need.

Ideally, users would *understand* how options and their interactions affect the performance of the system to make informed tradeoff and configuration decisions to run the system efficiently and reducing energy consumption and operational costs.

Developer's perspective

When developers of Berkeley DB observe a surprising non-crashing performance behavior, for the same inputs, workloads, environment, but different configurations, they are often unaware which and how options interact to produce the observed performance behavior [Han and Yu, 2016; Nistor et al., 2013a]. For example, the slowest configuration in Fig. 1.1 might appear to be performing unexpectedly, as the option `Temporary` is enabled, which should decrease the execution time as indicated in the documentation. To debug this surprising performance behavior, developers could compare the configuration changes with the default. However, the changes differ in several options in addition to `Temporary`, requiring developers identify how the changed options interact with each other, and with the options with default values, to produce the unexpected performance behavior. Additionally, when developers narrow down the options that are potentially producing the unexpected behavior, they need to debug the implementation to understand where and how options are being used and interacting with other options to produce the unexpected behavior. In this example, a developer would need to identify in the implementation that a temporary database can page to disk if the cache is not large enough to hold the database's contents (i.e., `Large cache` should also be enabled to reduce the execution time).

This debugging process of identifying potential options and analyzing where and how options interact in the implementation to produce the surprising behavior is challenging in large configuration spaces with complex performance behavior (e.g., 2^{50} configurations if only considering two values for each option). Ideally, developers would *understand* how options and their interactions affect the performance of the system *both globally and in the implementation* to *debug* surprising performance behaviors. With this information, developers can determine whether the system was misconfigured and is behaving as expected (such as in the example presented above), or there is a performance bug that they need to fix.

Problem statement

On one hand, users of configurable systems want to make informed configuration decisions to efficiently execute the systems, thus reducing energy consumption and operational costs. On the other hand, developers of configurable systems want to debug surprising performance be-

haviors and configuration-related performance bugs, similarly to reduce energy consumption and operational costs. Both of these activities require *understanding* how options and their interactions affect the performance of a system. However, understanding this information becomes intractable as the configuration spaces of the systems increase and their performance behavior becomes more complex.

1.2 Existing research

Existing research to *understand* how configuration options influence the performance of configurable systems has focused on building performance-influence models, which describe the performance of a system in terms of its configuration options for a specific workload and running in a specific environment. For example, the sparse linear model $17.2 + 9.2 \cdot \texttt{TEMPORARY} - 7.9 \cdot \texttt{LARGER\_CACHE} \cdot \texttt{TEMPORARY} - 7.9 \cdot \texttt{LARGER\_CACHE} + \ldots$ predicts how options and their interactions influence the performance of the system in Fig. 1.1 to predict the performance of arbitrary configurations. This model can be used, for example, by users and developers to determine that both `Temporary` and `Larger cache` need to be enabled to reduce the execution time.

Most approaches to build performance-influence models treat the system as a black-box, measuring the system's execution time for a subset of all configurations and extrapolating a model from those observations [Grebhahn et al., 2019; Ha and Zhang, 2019; Ha and Zhang, 2019; Kaltenecker et al., 2020; Sarkar et al., 2015; Siegmund et al., 2015]. One problem with such approaches is that they typically need to measure a large number of configurations to build accurate models, resulting in a tradeoff between the cost to build the models and their accuracy. For example, fewer configurations are cheaper to measure, but usually lead to less accurate models. This tradeoff affects users, as they might be mislead by an inaccurate model when making configuration decisions, as well as developers, as they might spend an extremely long time building an accurate model. In addition, the models only describe the influence that configuration options have globally (i.e., end-to-end) on the performance of the system, whereas developers debugging the performance the system also want to know where and how, in the implementation, configuration options influence the performance of the system. In this case, developers need to navigate the entire code base to answer those questions.

1.3 Thesis

The main goal of this thesis is to reduce the energy consumption and operational costs of running configurable systems. The work in this thesis contributes towards that goal by using *white-box analyses* to efficiently and accurately *model* the performance of configurable systems to help users and developers *understand* how options and their interactions affect the performance of such systems. With this information, users can make conscious configuration decisions to efficiently run configurable systems. With additional white-box information of *where and how* options interact in the implementation, developers can *debug* the performance of configurable systems to reason and potentially fix unexpected performance behaviors.

Thesis Statement: White-box analysis of how options influence the performance of code-level structures in configurable systems (1) helps to efficiently build accurate and interpretable global and local performance-influence models and (2) guides developers to inspect, understand, and debug configuration-related performance behaviors.

To support the thesis statement, we make the following contributions in the thesis:

Completed Work

- We identify the limitations of current black-box approaches to model and debug the performance of configurable systems. These limitations serve as the motivation to explore white-box techniques.
- Inspired by white-box insights of how options interact and affect the performance of configurable systems, we introduce the insights of *Composition* and *Compression* to locally and globally model the performance of configurable systems, which help users and developers understand how options and their interactions affect the performance of systems.
- We propose and compare two prototypes that operationalize the insights of composition and compression. Our prototypes ConfigCrusher and Comprex are implemented consider different design decisions to model the performance of small-, medium-, and large-scale configurable systems.
- Our evaluation on 13 configurable systems show that our prototypes can efficiently and accurately model the performance of configurable systems, often more efficiently than black-box approaches with comparable accuracy. Additionally, our models are interpretable and can be mapped to specific code regions.
- Our comparison between the two prototypes shows that the design decisions made for Comprex, namely (1) using a dynamic taint analysis to identify the influence of options on regions of systems and (2) measuring the performance of methods as regions with an off-the-shelf sampling profiler, scales our white-box analysis to model the performance of medium- to large-scale configurable systems.

Proposed Work

- Inspired by the limitations of global performance-influence models to help developers debug the performance of configurable systems, we propose to conduct a user study to explore the process that developers follow and the information needs that they have when debugging the performance of configurable systems.
- Guided by the insights of the study above, we propose to develop additional tool support to compare performance profiles and trace how options are propagated through a system to further help developers to debug the performance of configurable systems.

- We propose to conduct a user study to empirically validate the usefulness of the tools that we present in this thesis, namely global and local performance-influence models, comparing performance profiles, and tracing how options are propagated through the system, to help developers inspect, reason, and debug the performance of configurable systems.

The contributions in this thesis enable efficiently and accurately modeling the global performance of configurable systems for users to make informed configuration decisions to run their systems efficiently. Additionally, the contributions help developers understand and debug the global and local performance of configurable systems to improve software quality in general. Overall, the work in this thesis contributes towards the goal of reducing the energy consumption and operational costs of running configurable systems. Finally, we hope that our contributions inspire the configurable systems and performance analysis research communities to use and develop new and scalable white-box techniques to benefit from the information that can be obtained with such analyses.

The remainder of the proposal is structured as follows:

- Chapter 2 introduces existing research on analyzing, modeling, and debugging the performance of configurable systems and the limitations of current approaches.
- Chapter 3 describes the key insights for efficient and accurate performance modeling of configurable systems, the implementation of our two prototypes, and their evaluation.
- Chapter 4 outlines our proposed work to develop additional tool support, guided and validated by user studies, to help developers debug the performance of configurable systems.
- Chapter 5 concludes the thesis proposal with a research plan.

Chapter 2

State of the Art on Performance Analysis of Configurable Systems

This chapter introduces the state of the art of analyzing, modeling, and debugging the performance of configurable systems.

We first present the terminology used in this thesis (Sec. 2.1) and describe configurable systems in general (Sec. 2.2). Subsequently, we describe the state of the art black-box approaches for modeling the performance of configurable systems (Sec. 2.3). Based on the limitations of such approaches, we explore how white-box analysis have been used to analyze the performance and other characteristics of configurable and (non-configurable) software systems (Sec. 2.4). Subsequently, we describe how white-box approaches have been used specifically to debug the performance of configurable systems, but also to debug, more generally, configurable and (non-configurable) software systems (Sec. 2.5). The insights and limitations that we identify in this discussion motivate the work of this thesis.

This chapter is derived in part from our ASE Journal'20 article "ConfigCrusher: Towards White-box Performance Analysis for Configurable Systems" [Velez et al., 2020a] and a conference submission under review at the time of writing – "White-box Analysis over Machine Learning: Modeling Performance of Configurable Systems" [Velez et al., 2020b].

2.1 Terminology

Performance analysis of configurable systems has been explored in the past using different terminologies. We first establish and define the terms that we will use throughout the document.

Option refers to an input that modifies the operation of a system. An option is also known as *variation*, *feature*, or *flag* in the literature. We consider options as a special type of inputs with a small finite domain (e.g., Boolean options), that a user might explore to change functionality or quality attributes. For simplicity, we describe the work in this thesis in terms of Boolean options, but other non-binary option types can be encoded or discretized as Boolean options.

Configuration refers to a complete setting of all options in a system [Apel et al., 2013].

Configuration space refers to all configurations of a system.

Misconfiguration a configuration error in which the performance of the system is behaving correctly, but not as desired.

Control-flow statement refers to a program statement that affects the flow of execution of a system based on a decision. Examples of control-flow statements include `if` and `switch` statements and `for` and `while` loops.

Control-flow decision refers to an actual execution of a control-flow statement in which a decision is made and a specific branch is executed.

Workload refers to an input that is used by a system to perform a task. We consider a workload as a fixed type of input with a large and possible infinite domain (e.g., images to compress), that a developer fixes when modeling and debugging the performance of a configurable system.

Environment refers to the underlying software and hardware in which the system executes. We consider the environment to be fixed when developers model and debug the performance of a configurable system.

Performance refers to the execution time of a system or a region in a system.

Performance-influence model describe the performance behavior of a system, in terms of options and interactions, for a specific workload running on a specific environment.

2.2 Configurable Systems

Most software systems today are configurable, which allow users to customize the functionality and quality attributes of the systems to satisfy their requirements and needs [Apel et al., 2013]. The systems are usually built with some reusable core functionality and implement some deferred design decisions, such as which specific algorithm implementation to use or whether to enable or disable some functionality, as configuration options [Becker et al., 2009; Esfahani et al., 2013], allowing users to choose between multiple alternative and optional implementations.

The benefits of developing and customizing configurable systems, however, come with the cost of increased complexity. The large number of options makes tracking how options and their interactions influence the functionality and quality attributes of systems a difficult task. For this reason, users are often overwhelmed with the large number of options and change options in a trial-and-error fashion without understanding the resulting effects [Apel et al., 2013; Hubaux et al., 2012; Xu et al., 2015, 2013]. Likewise, a large number of options complicates the process that developers follow to develop, test, and maintain large configuration spaces [Behrang et al., 2015; Halin et al., 2018; Jin et al., 2014; Melo et al., 2016, 2017].

Performance in Configurable Systems. Performance, in terms of execution time, and often directly correlated energy consumption and operational costs, is one of the most important quality attributes for users and developers of configurable systems [Gelenbe and Caseau, 2015;

Manotas et al., 2016; Pinto and Castor, 2017]. The options enable or disable functionality, and execute different implementations that affect the performance of the systems. For example, enabling encryption increase the security of a system, but encryption usually also increases the execution time. Likewise, selecting which algorithm to use when compressing a video will affect, among several aspects, the quality of the output video and the time to compress the video. Accordingly, users and developers can benefit from understanding how options affect the performance of these systems; users can make informed tradeoff and configuration decisions to run the system efficiently [Jabbarvand et al., 2015; Kern et al., 2011; Munoz, 2017; Wilke et al., 2013; Zhang et al., 2014] and developers can debug the performance of the systems [Chowdhury and Hindle, 2016; Gui et al., 2016; Hasan et al., 2016; Li et al., 2016; Malik et al., 2015; Pereira et al., 2016].

2.3 Modeling Performance in Configurable Systems

Modeling the performance of configurable systems helps users and developers understand how options influence the performance of these systems [Grebhahn et al., 2019; Kaltenecker et al., 2020; Kolesnikov et al., 2018; Viswanadham and Narahari, 2015].

Traditionally, designers and developers model the performance of a system's architecture (e.g., using Queuing networks, Petri Nets, and Stochastic Process Algebras) and workload in the design stage of a project [Harchol-Balter, 2013; Kounev, 2006; Serazzri et al., 2006]. At this state, design decision are usually modeled as configuration options [Becker et al., 2009; Esfahani et al., 2013], allowing users to choose between multiple alternative and optional implementations.

In the context of configurable systems, *performance-influence models* are built to explain the performance of a system in terms of configuration options and their interactions, and predict the performance of the entire configuration space [Guo et al., 2013; Ha and Zhang, 2019; Ha and Zhang, 2019; Jamshidi et al., 2017a, 2018, 2017b; Kolesnikov et al., 2018; Siegmund et al., 2015; Valov et al., 2017]. For example, the sparse linear model $8 + 15\mathtt{A} + 10\mathtt{C} + 3\mathtt{AB} + 30\mathtt{AC}$ captures the execution time of the system in Fig. 2.1, which predicts the performance of arbitrary configurations, and explains how the options `A`, `B`, and `C` and their interactions influence the system's performance. The models are typically built by executing the system with an specific workload and in a specific environment under different configurations, to learn how configuration options affect the performance of the system. The model are useful for users to make deliberate configuration decisions and for developers to understand and debug how configuration options affect the performance of their systems.

Building performance-influence models

Performance-influence models are typically built by measuring the execution time of a system with a specific workload in a specific environment under different configurations [Siegmund et al., 2015]. Almost all existing approaches are *black-box* in nature: They do not take the system's implementation into account and measure the end-to-end execution time of the system.

```
1  def main(List workload)
2      a = getOpt("A"); b = getOpt("B");
3      c = getOpt("C"); d = getOpt("D");
4      ... // execution: 1s
5      int i = 0;
6      if(a) // variable depends on option A
7          ... // execution: 1s
8          foo(b); // variable depends on option B
9          i = 20;           Region depends on option A
10     else
11         ... // execution: 2s
12         i = 5;
13     while(i > 0)
14         bar(c); // variable depends on option C
15         i--;
16 def foo(boolean x)        Region depends on options A and B
17     if(x) ... // execution: 4s
18     else ... // execution: 1s
19 def bar(boolean x)        Region depends on options A and C
20     if(x) ... // execution: 3s
21     else ... // execution: 1s
```

Figure 2.1: Example configurable system with 4 configuration options and 3 highlighted in which the options influence the performance of the system.

Brute-force. The simplest approach is to observe the execution of *all* configurations in a *brute-force* approach. The approach obviously does not scale, but for the smallest configuration spaces, as the number of configurations grows exponentially with the number of options. In our example system in Fig. 2.1, the approach will measure all configurations, inefficiently exploring interactions with option `D`, which does not affect the performance of the system.

Sampling and Learning. In practice, most current approaches measure executions only for a *sampled* subset of all configurations and extrapolate performance behavior for the rest of the configuration space using machine learning [Grebhahn et al., 2019; Ha and Zhang, 2019; Ha and Zhang, 2019; Kaltenecker et al., 2020; Sarkar et al., 2015; Siegmund et al., 2015], which we collectively refer to as *sampling and learning* approaches. Specific approaches differ in how they sample, learn, and represent models: Common sampling techniques include uniform random, feature-wise, and pair-wise sampling [Medeiros et al., 2016], design of experiments [Montgomery, 2006], and combinatorial sampling [Al-Hajjaji et al., 2016; Halin et al., 2018; Hervieu et al., 2011, 2016; Nie and Leung, 2011]. Common learning techniques include linear regression [Kaltenecker et al., 2019; Siegmund et al., 2015, 2012a,b], regression trees [Grebhahn et al., 2019; Guo et al., 2013, 2017; Sarkar et al., 2015], Fourier Learning [Ha and Zhang, 2019], Gaussian Processes [Jamshidi et al., 2017b], and neural networks [Ha and Zhang, 2019].

Different sampling and learning techniques yield different tradeoffs between measurement effort, prediction accuracy, and interpretability of the learned models [Grebhahn et al., 2019; Kaltenecker et al., 2020; Kolesnikov et al., 2018]. For example, larger samples are more expen-

sive, but usually lead to more accurate models; random forests, with large enough samples, tend to learn more accurate models than those built with linear regressions, but the models are harder to interpret when users want to understand performance or developers want to debug their systems [Grebhahn et al., 2019; Kaltenecker et al., 2020; Molnar, 2019].

Although some sampling strategies rely on a coverage criteria to sample specific interaction degrees, such as t-wise sampling [Medeiros et al., 2016; Nie and Leung, 2011], the strategies might miss important interactions, leading to inaccurate models, or measure interactions that are not relevant for performance. In our example system in Fig. 2.1, a sampling strategy might inefficiently measure interactions between options `B` and `C`, which do not affect the performance of the system, or interactions with option `D`.

Limitation of existing black-box approaches: Current black-box performance modeling approaches do not consider the internals of the system that they analyze. Instead, the approaches rely on sampling strategies to potentially capture performance relevant interactions to learn the performance behavior of a system from incomplete samples. Using sampling strategies results in either under-approximating or over-approximating the number of configurations that are measured, which affects the cost to build models and the accuracy of the models. The approaches are also extremely sensitive to the learning technique that is used, in terms of the accuracy of the models that are built and the interpretability of the models.

Goals of performance-influence modeling

Performance-influence models can be used for different tasks in different scenarios, which benefit from different characteristics of the type of model that is used.

Performance optimization. In the simplest case, a user wants to optimize the performance of a system by selecting the fastest configuration for a specific workload and running the system in a specific environment. Performance-influence models have been used for optimization [Guo et al., 2013; Nair et al., 2017; Oh et al., 2017; Zhu et al., 2017], though metaheuristic search (e.g., hill climbing) is often more effective at pure optimization problems [Hutter et al., 2011; Jamshidi and Casale, 2016; Oh et al., 2017; Olaechea et al., 2014; Zhu et al., 2017], as they do not need to understand the entire configuration space.

Performance prediction. In other scenarios, users want to predict the performance of individual configurations. Scenarios include *automatic reconfiguration* and *runtime adaptation*, where there is no human-in-the-loop and online search is impractical. For example, when dynamically deciding during a robot's mission which options to change to react to low battery levels [Jamshidi et al., 2018, 2017b; Wang et al., 2018; Zhu et al., 2017]. In these scenarios, the model's prediction accuracy over the entire configuration space is important, but understanding the structure of the model is not important. In this context, deep regression trees [Guo et al., 2013, 2017; Sarkar et al., 2015], Fourier Learning [Ha and Zhang, 2019], and neural networks [Ha and Zhang, 2019] are commonly used, which build accurate models, with a large

enough number of sampled configurations, but are not easy to interpret by humans [Grebhahn et al., 2019; Kaltenecker et al., 2020; Kolesnikov et al., 2018; Molnar, 2019; Siegmund et al., 2015].

Performance understanding. When users want to make deliberate configuration decisions [Grebhahn et al., 2019; Kaltenecker et al., 2020; Kolesnikov et al., 2018; Siegmund et al., 2015; Wang et al., 2018; Xu et al., 2013] (e.g., whether to accept the performance overhead of encryption), *interpretability* regarding how options and interactions influence performance becomes paramount. In these settings, researchers usually suggest sparse linear models, such as $8 + 15 \cdot \mathtt{A} + 10 \cdot \mathtt{C} + 3 \cdot \mathtt{A} \cdot \mathtt{B} + 30 \cdot \mathtt{A} \cdot \mathtt{C}$, typically learned with stepwise linear regression or similar variations [Kaltenecker et al., 2019; Siegmund et al., 2015, 2012a,b]. Such models are generally accepted as *inherently interpretable* [Molnar, 2019], as the information of how configuration options and their interactions influence the performance of a system is easy to inspect and interpret by users [Kaltenecker et al., 2020; Kolesnikov et al., 2018; Molnar, 2019]. By contrast, opaque machine-learned models (e.g., random forests and neural networks) are not considered inherently interpretable [Molnar, 2019]. While there are many approaches to provide *post-hoc explanations* [Lundberg and Lee, 2017; Molnar, 2019; Ribeiro et al., 2016; Štrumbelj and Kononenko, 2014], such approaches are not necessarily faithful and may provide misleading and limited explanations [Rudin, 2019].

In addition to users who configure a system, developers who maintain the system can also benefit from performance-influence models to understand and debug the performance behavior of their systems. For example, when presenting performance-influence models to developers in high-performance computation, Kolesnikov et al. [2018] reported that a developer "was surprised to see that [an option] had only a small influence on system performance," indicating a potential bug. In such setting, understanding how individual options and interactions influence performance is again paramount, favoring interpretable models.

Thesis goal: We aim to build interpretable performance-influence models to help users and developers understand how options and their interactions affect the performance of configurable systems.

2.4 Analyzing Performance in Configurable Systems

In this thesis, we seek to overcome the limitations of black-box approaches to analyze and help debug the performance of configurable systems. In this section, we first explore white-box program analyses broadly, focusing on how the analyses have been used to inspect configurable and (non-configurable) software systems, in terms of performance and other program characteristics. Subsequently, we explore how these program analyses are used to specifically debug the performance configurable software systems (see Sec. 2.5).

2.4.1 Analysis of Software Systems

White-box program analysis seeks to automatically inspect the source code to study the behavior of software systems regarding some property, such as correctness, safety, complexity, and performance [Arzt et al., 2014; Bell and Kaiser, 2014; King, 1976; Nielson et al., 2010; Schwartz et al., 2010; Vallée-Rai et al., 1999; Weiser, 1981]. While the realm of program analysis is to broad to cover in a single chapter (e.g., data-flow analyes [Nielson et al., 2010; Vallée-Rai et al., 1999], Hoare logic [Hoare, 1969], Satisfiability Modulo Theories [Barrett and Tinelli, 2018], program synthesis [Gulwani et al., 2017], concolic execution [Sen, 2007], model checking [Clarke Jr et al., 2018]), here we describe the program analyses techniques that have been used in the context of analyzing and debugging configurable systems.

Static and dynamic program analyses have been used extensively to analyze and debug configurable systems.

Symbolic execution is an approach to execute a system abstractly to cover the execution of multiple inputs [King, 1976; Schwartz et al., 2010]. During the execution of a system, symbolic values, in terms of inputs and variables in the system, are propagated to analyze the behavior of the system, such as which inputs cause each part of the program to execute. Relevant to the topics in this thesis, the technique has been used for performance debugging [Bornholt and Torlak, 2018] and studying some characteristics of configurable systems [Reisner et al., 2010].

In contrast to symbolic execution, variational execution is an approach to dynamically analyze the effects of multiple inputs by tracking concrete values [Meinicke et al., 2016; Wong et al., 2018]. In other communities (e.g., security), this technique is called faceted execution [Austin and Flanagan, 2012]. Relevant to the topics in this thesis, the technique has been used to debug configurable systems [Meinicke et al., 2018], exhaustively test configurable systems [Wong et al., 2018], as well as, similarly to symbolic execution, to study some characteristics of configurable systems [Meinicke et al., 2016].

Taint analysis, also known as information flow analysis, is a static [Arzt et al., 2014] or dynamic [Austin and Flanagan, 2009; Bell and Kaiser, 2014] data-flow analysis typically used in security research to detect, for example, information leaks and code injection attacks [Newsome and Song, 2005; Schwartz et al., 2010]. A value is initially marked as tainted, and all values derived (directly or indirectly) from the initial value are tainted as well, which is used to identify if the values are used in locations where they should not (e.g., sent over the network). Relevant to the topics in this thesis, the technique has been used to track how options are used [Lillack et al., 2018] and modify data [Toman and Grossman, 2016a,b] in configurable systems.

In contrast to taint analysis, program slicing is an approach to compute the relevant fragments of a system based on a criteria either statically [Weiser, 1981] and dynamically [Agrawal and Horgan, 1990; Korel and Laski, 1988]. While the approach has not been used in the context of configurable systems, the approach has been successful in helping developers debug software systems [Ko and Myers, 2004; LaToza and Myers, 2011; Xu et al., 2005], making it a promising technique to be used for debugging configurable systems. For example, the technique has been used for narrowing the parts of the program that developers need to analyze when debugging software systems [Agrawal and Horgan, 1990; Korel and Laski, 1988; Weiser, 1981], as well as in the backend of visualization tools, which help developers navigate [LaToza

and Myers, 2011] and debug software systems [Ko and Myers, 2004].

2.4.2 Performance Analysis of Software Systems

Static and dynamic program analyses have been used extensively to analyze the performance of software systems. Examples include traditional off-the-shelf performance profiles (either sampling- or instrumentation-based), such as JProfiler [JPR, 2019], Valgrind [Nethercote and Seward, 2007], and VisualVM [VVM, 2020], but also more targeted program analyses that identify inefficient code structures [Bornholt and Torlak, 2018; Grechanik et al., 2012; Han et al., 2012; Jin et al., 2012; Jovic et al., 2011; Liu et al., 2014; Nistor et al., 2015, 2013b; Song and Lu, 2014, 2017] and synchronization bottlenecks [Alam et al., 2017; Curtsinger and Berger, 2016; Yu and Pradel, 2016, 2018]. This line of work aims at helping developers debug the performance of software systems, which we discuss in more detail in Sec. 2.5.

2.4.3 Analysis of Configurable Systems

Several researchers have leveraged some kind of static and dynamic program analyses to track and characterize options in configurable systems [Dong et al., 2016; Hoffmann et al., 2011; Lillack et al., 2018; Meinicke et al., 2016; Nguyen et al., 2016; Rabkin and Katz, 2011; Reisner et al., 2010; Souto and d'Amorim, 2018; Toman and Grossman, 2016a,b; Wang et al., 2013; Xu et al., 2016]. Thüm et al. [2014] presented a comprehensive survey of analyses for software product lines also applicable to configurable systems.

Taint analysis has been used to track how options are propagated and used in configurable systems [Hoffmann et al., 2011; Lillack et al., 2018; Toman and Grossman, 2016a,b]. For example, Lotrack [Lillack et al., 2018] used a static analysis to identify under which configurations code fragments may be executed. Likewise, Staccato [Toman and Grossman, 2016b] used dynamic taint analysis to identify the use of stale configuration data.

Other program analysis techniques have been used to analyze the behavior of interactions in configurable systems [Meinicke et al., 2016; Nguyen et al., 2016; Reisner et al., 2010; Wong et al., 2018]. Reisner et al. [2010] and Meinicke et al. [2016] used symbolic execution and variational execution, respectively, to identify that (1) not all options tend to affect the execution of a system on a given workload, (2) not all options tend to interact with each other in a system, and (3) options tend to interact only with a few other options in specific parts of a system.

Insights of how options interact in configurable systems: Not all options tend to affect the execution of a system on a given workload.Not all options tend to interact with each other in a system.Options tend to interact only with a few other options in specific parts of a system.

2.4.4 Performance Analysis of Configurable Systems

Static and dynamic approaches have been scarcely used to analyze the performance of configurable systems [Li et al., 2020; Siegmund et al., 2013].

For several years, the only existing white-box approach to analyze the performance of configurable systems was Family-Based Performance Measurement [Siegmund et al., 2013]. The approach builds performance-influence models to help users and developers understand how options interact and affect the performance of a system; one of the goals of performance-influence modeling described in Sec. 2.3.

Specifically, the approach uses a static mapping between options to code regions and instruments the system to measure the execution time spent in the regions. Subsequently, it executes the system once with all options enabled, tracking how much each option contributes to the execution time. The approach works well when all options are directly used in control-flow statements and only contribute extra behavior. That is, an option would not switch between two implementations, but only activate additional code. Current implementations, however, derive the static map from compile-time variability mechanisms (preprocessor directives) and do not handle systems with load-time variability (i.e., loading and processing options in variables at runtime). Furthermore, the static map only covers direct control-flow interactions from nested preprocessor directives, and can lead to inaccurate models when indirect data-flow interactions occur. In our example system in Fig. 2.1, data-flow analysis is needed to detect that the while loop in line 13 indirectly depends on the option `A`, with implicit data-flow through the variable `i`, leading to an inaccurate performance-influence model otherwise.

Limitation of the existing white-box approach: The only existing white-box approach for performance modeling imposes strict constraints on the structure of the system and the performance behavior of the options of the system.

More recently, LearnConf [Li et al., 2020] used static taint analysis with intraprocedural control-flow analysis to identify usage patterns of individual options in the source code for predicting performance properties based on the patterns (e.g., linear relationship). The approach, however, is orthogonal to performance-influence modeling, as the approach can only predict the performance property of an option (e.g., selecting an option would increase the performance by a constant amount), not the actual performance effect, in terms of time, which is relevant information that users and developers need to make informed tradeoff decisions and debug the performance of a system, respectively. Additionally, the approach does not consider how interacting options might affect the performance of a system.

2.5 Debugging Performance in Configurable Systems

In this thesis, we seek to use white-box program analyses to help developers analyze and debug the performance of configurable systems. In Sec. 2.4, we discussed, relevant to this thesis, how white-box program analyses have been used to inspect the performance of configurable systems, specifically in terms of performance-influence modeling (see Sec. 2.3). In this section, we explore how white-box program analyses techniques have been used, primarily, to debug the performance of configurable systems, but also to debug, more generally, configurable and (non-configurable) software systems.

2.5.1 Debugging in Software Systems

Debugging software systems is the process of understanding the behavior of a system when an unexpected behavior occurs [Zeller, 2009]. Regardless of the root cause of the unexpected behavior (e.g., software bugs or misconfigurations), systems often misbehave with similar symptoms, such as crashes, missing functionality, incorrect results [Andrzejewski et al., 2007; Attariyan and Flinn, 2010; Medeiros et al., 2016; Meinicke et al., 2018; Zeller, 1999, 2009], and, in terms of performance, long execution times and increased energy consumption [Han and Yu, 2016; He et al., 2020; Jin et al., 2012; Li et al., 2016; Song and Lu, 2017; Wilke et al., 2013]. When a system misbehaves, users usually report the problem to developers, who often spend a long time diagnosing the system to localize and fix a bug or determine that the system was misconfigured [Breu et al., 2010; Chaparro et al., 2017; Han and Yu, 2016; Jovic et al., 2011; Park et al., 2012; Parnin and Orso, 2011; Zeller, 2009].

In the simplest case, developers could manually debug the system. This approach, however, is limited to small systems and simple bugs, as even expert developers of large systems need automatic techniques or tool support to understand, locate, and diagnose unexpected behaviors [Burg et al., 2013; Ko and Myers, 2008; Ko et al., 2006; LaToza and Myers, 2010; Lawrance et al., 2013; Scaffidi et al., 2011; Zeller, 2009].

Traditionally, developers rely on some kind of technique or tool support to narrow down and debug the set of potential causes for unexpected behaviors [Burg et al., 2013; Ko et al., 2006; Lawrance et al., 2013; Parnin and Orso, 2011; Scaffidi et al., 2011; Zeller, 2009]. While some techniques can automatically fix bugs [Le Goues et al., 2012], we consider debugging and repairing as two different processes, in which the latter is focused on automatically finding a patch for a buggy program, whereas the former is the process that a developer follows to understand unexpected behaviors in a system.

There are several prominent program analysis techniques that facilitate debugging of software systems [Agrawal and Horgan, 1990; Andrzejewski et al., 2007; King, 1976; Korel and Laski, 1988; Weiser, 1981; Zeller, 1999]. Delta debugging [Zeller, 1999] has helped developers debug unexpected behaviors by automatically and systematically narrowing down the inputs that are relevant for causing a fault. Likewise, program slicing [Agrawal and Horgan, 1990; Korel and Laski, 1988; Weiser, 1981] also facilitates debugging by providing developers with a slice of the relevant fragments of a system based on a criteria. These and similar techniques have been useful for developers, since the techniques narrow down and isolate relevant inputs and parts of a system where developers should focus their debugging efforts.

Researchers have also integrated program analyses in the backend of tools to help developers debug software systems [Burg et al., 2013; Ko and Myers, 2004; LaToza and Myers, 2011; Pothier et al., 2007]. For instance, the Whyline [Ko and Myers, 2004] combines static and dynamic slicing to allow developers to ask "why did " and "why did not" questions directly about a system's output. The tool then presents relevant code to answers those questions. Similarly, Reacher [LaToza and Myers, 2011] uses static data-flow analysis to help developers answer reachability questions as they navigate call graphs. These and similar tools have been useful for developers, since the tools provide relevant and useful information to guide developers through the implementation to understand and debug the behavior of the system.

2.5.2 Performance Debugging in Software Systems

Off-the-shelf performance profilers [JPR, 2019; VVM, 2020; Nethercote and Seward, 2007] (either sampling- or instrumentation-based) are typically the first tool that developers use to analyze the performance of a system. Profilers are typically used to identify the places in a system that are taking the most time to execute, commonly known as hotspots [Castro et al., 2015; Cito et al., 2018; Curtsinger and Berger, 2016; Gregg, 2016; Yu and Pradel, 2018]. After identifying hotspots, developers locate them in the code for analysis and potentially optimize and performance issues.

In addition to performance profilers, researchers have developed targeted techniques to help developers identify inefficient code structures [Bornholt and Torlak, 2018; Grechanik et al., 2012; Han et al., 2012; Jin et al., 2012; Jovic et al., 2011; Liu et al., 2014; Nistor et al., 2015, 2013b; Song and Lu, 2014, 2017] and synchronization bottlenecks [Alam et al., 2017; Curtsinger and Berger, 2016; Yu and Pradel, 2016, 2018]. For instance, statistical debugging has been used to identify performance anti-patterns by analyzing program predicates in regular and slow executions and using statistical models [Song and Lu, 2014]. Likewise, Toddler [Nistor et al., 2013b] detects performance bugs by identifying repetitive memory read sequences across loop iterations. Furthermore, Coz [Curtsinger and Berger, 2016] introduced causal profiling to help developers identify which components in their concurrent system they should optimize to improve performance.

2.5.3 Debugging in Configurable Systems

In the context of configurable systems, there are specific techniques and tools that facilitate debugging in the presence of a large number of options [Attariyan and Flinn, 2010; Dong et al., 2016; Hoffmann et al., 2011; Lillack et al., 2018; Meinicke et al., 2018; Rabkin and Katz, 2011; Toman and Grossman, 2016a,b; Wang et al., 2013; Wong et al., 2018; Xu et al., 2016; Zhang and Ernst, 2013, 2014, 2015]. The techniques aim to identify the option or interaction that cause the unexpected behavior, as well as to guide developers to debug relevant parts of a system.

In general, the techniques track how options are propagated through the system to identify how they cause an unexpected behavior. For instance, ConfAid [Attariyan and Flinn, 2010] is a misconfiguration troubleshooting tool based on causal analysis. The tool instruments the program to record information flow during the execution and attributes undesired behavior to configuration options. Likewise, Varviz [Meinicke et al., 2018] generates variational traces to help developers understand how data- and control-flow influence executions, and thus, how different configurations cause a fault. Furthermore, Staccato [Toman and Grossman, 2016b] used dynamic taint analysis to identify the use of stale or inconsistent configuration data. Indicating where and how options interact in the implementation is paramount to help developers debug unexpected behaviors.

Related to debugging, researchers have also used techniques for testing large configuration spaces [Kim et al., 2013; Souto and d'Amorim, 2018; Souto et al., 2017]. SPLat [Kim et al., 2013] instruments a program to dynamically track the configurations that produce distinct execution paths. It reexecutes the system until all configurations with distinct paths are explored.

2.5.4 Performance Debugging in Configurable Systems

In recent years, researchers have explored more closely the area of performance debugging of configurable systems, in terms of empirically studying the characteristics and prevalence of configuration-related performance bugs [Han and Yu, 2016; Han et al., 2018] and providing developers tool support to debug unexpected performance behaviors in configurable systems [He et al., 2020; Li et al., 2020; Siegmund et al., 2015].

Similar to functional bugs [Park et al., 2012; Yin et al., 2011] and performance bugs in software systems [Jin et al., 2012; Nistor et al., 2013a], configuration-related performance bugs are prevalent in software systems today [Han and Yu, 2016; Han et al., 2018]. Options affect the performance of configurable systems at control-flow statements, depending on which branch is executed and how many times the branches are executed [Han and Yu, 2016; Han et al., 2018; Siegmund et al., 2013], similarly to the pattern observed in (non-configurable) software systems [Jin et al., 2012; Nistor et al., 2015, 2013a]. Most configuration-related performance bugs are caused by a single option ($\sim 72\%$), but a non-trivial amount of performance bugs are caused by an interaction of two or more options ($\sim 28\%$). Additionally, configuration-related performance bugs are usually more complex to debug than general performance bugs.

Insight of how options affect the performance of configurable systems: The performance of configurable systems tends to change at control-flow statements, depending on which branch is executed and how many times the branches are executed.

Based on the above insights, researchers have developed techniques to help developers analyze and debug the performance of configurable systems [He et al., 2020; Li et al., 2020]. As discussed in Sec. 2.4, LearnConf [Li et al., 2020] used intraprocedural control-flow analysis to identify usage patterns of individual options in the source code for predicting performance properties, not actual effects, based on the patterns. Without using program analysis techniques, yet relevant to debug the performance of configurable systems, He et al. [2020] suggested using developers' expected performance behavior of individual and pairs of options as a testing oracle for identifying the incorrect implementation of configurations. These approaches work well when developers already *know* the options that caused unexpected performance behaviors. However, developers usually first need to understand how options affect the performance of the entire configuration space of a system to, then, identify the option or interaction that are causing an unexpected performance behavior.

Assumptions of existing work on debugging the performance of configurable systems: Current techniques to help developers debug the performance of configurable systems assume that developers already know the options or interactions that cause the unexpected performance behavior. Instead, developers first need to understand the performance behavior of the entire configuration space to identify the options that are potentially causing an unexpected performance behavior.

As discussed in Sec. 2.3, sparse linear performance-influence models can be helpful for developer to debug the performance of configurable systems, as the models are easy to inspect

and interpret how options and their interactions affect the performance of a system [Kaltenecker et al., 2020; Kolesnikov et al., 2018; Molnar, 2019]. For example, the sparse linear model $8 + 15\texttt{A} + 10\texttt{C} + 3\texttt{AB} + 30\texttt{AC}$ for the system in Fig. 2.1 can be used to determine whether the increase of 30 seconds in the execution time when both `A` and `C` are selected complies with the system's requirements. Ideally though, the models would also indicate *where* the influence of options occurs in the implementation and *how* options influence, in the implementation, the performance of those locations in the system. However, current performance-influence models, before the work on this thesis, only provide information of how options influence the performance of the system globally, often requiring developers to navigate the entire code base to debug the performance of the systems.

Limitation of existing performance-influence models for performance debugging: Current performance-influence models only describe how options influence the performance of the entire system. The models do not indicate where the influence occurs and how options are used in the implementation to influence the performance of the system, which is useful for developers to debug the performance of configurable systems.

2.6 Summary

In this chapter, we discussed the state of the art of analyzing, modeling, and debugging the performance of configurable systems. We also explored how white-box approaches have been used to analyze and debug the performance and other characteristics of (non-configurable) software systems. The insights and limitations that we identified through this discussion motivate the work of this thesis.

Chapter 3

White-box Performance Modeling of Configurable Systems

OUTLINE?

In Chapter 2, we discussed the limitations of existing approaches for modeling the performance of configurable systems and the insights from existing white-box analyses of how options influence the performance of configurable system. Inspired by those limitations and insights, we propose, in this chapter, to use a white-box analysis to efficiently and accurately model the *global and local* performance of configurable systems.

The work in this chapter is an excerpt of our ASE Journal'20 article "ConfigCrusher: Towards White-box Performance Analysis for Configurable Systems" [Velez et al., 2020a] and of a conference submission under review at the time of writing – "White-box Analysis over Machine Learning: Modeling Performance of Configurable Systems" [Velez et al., 2020b].

3.1 Key Insights for Efficient and Accurate Performance Analysis of Configurable Systems

Inspired by the limitations of existing performance modeling approaches for configurable systems and the insights from existing white-box analyses of how options influence the performance of configurable systems (Chapter 2), we present the key insights to efficiently and accurately analyze and model the performance of configurable systems.

We seek to develop a *white-box analysis* to efficiently and accurately analyze the performance of configurable systems, *without the use of machine learning* to avoid inaccuracies of extrapolating from incomplete measurements. The analysis contributes to the thesis goal of reducing the energy consumption and operational costs of running configurable systems since the analysis helps model the global performance of configurable systems for (1) users to make informed configuration decisions and (2) developers to debug the performance of their systems.

To analyze the performance of configurable systems, we measure the execution time of multiple configurations, similar to existing approaches, but we guide the exploration with a

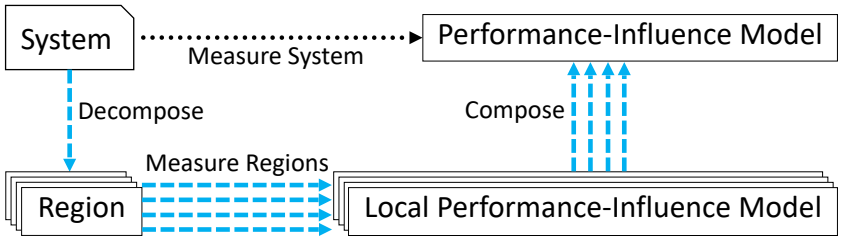

Figure 3.1: Building performance-influence models is compositional: Instead of building a single model for the entire system (dotted black arrow), we can simultaneously build a local model per region and compose those models (dashed blue arrows).

white-box analysis of the internals of the system. For a given set of inputs, a configurable system with a set of Boolean options O can exhibit up to $2^{|O|}$ distinct execution paths, one per configuration.[1] If we measure the execution time of each distinct path, we can map performance differences to options and their interactions, without any approximation through machine learning.

Our approach to efficiently and accurately analyze the performance of configurable systems relies on two insights: (1) Performance-influence models can be built *compositionally*, composing models built independently for *smaller regions* of the code than the entire system (cf. Fig. 3.1). (2) Multiple performance-influence models for *smaller regions* can be built simultaneously by measuring the execution of a system often with only a few configurations, which we refer to as *compression*.

Compositionality. Building performance-influence models is compositional: We can measure the performance of smaller regions in a system (e.g., considering each method as a region) and build a performance-influence model per region separately, which describes the performance behavior of each region in terms of options. Subsequently, we can compose the local models to describe the performance of the entire system; computed as the sum of the individual influences in each model (e.g., composing $5 + 4\mathsf{A}$ and $1 - 1\mathsf{A} + 2\mathsf{B}$ to $6 + 3\mathsf{A} + 2\mathsf{B}$).

Compositionality helps reduce the cost to model the performance of configurable systems, as many smaller regions of a system are often influenced only by a subset of all options. Hence, the number of distinct paths to observe in a region is usually much smaller than the number of distinct paths in the entire system. If we have an analysis to find the subset of options that directly and indirectly influence smaller regions (see Sec. 3.2.1), we can build a local performance-influence model by observing all distinct paths in a region often with only

[1]For simplicity, we describe the work in this thesis in terms of Boolean options, but other finite option types can be encoded or discretized as Boolean options. The distinction between inputs and options is subjective and domain specific. We consider options as a special type of inputs with a small finite domain (e.g., Boolean options), that a user might explore to change functionality or quality attributes. We consider fixed values for other inputs. Note that a user might fix some configuration options as inputs and consider alternative values for inputs as options (e.g., use an option for different workloads). We analyze the performance influence of options with finite domains, assuming all other inputs are fixed at specific values, thus resulting in a finite, but typically very large configuration space.

```
1 if(a) // variable depends on option A
2     ... // execution: 1s
3 if(b) // variable depends on option B
4     ... // execution: 2s
5 if(c) // variable depends on option C
6     ... // execution: 3s
```

Figure 3.2: Three independent regions influenced by different options.

a few configurations.

Insight: Performance-influence models can be built by composing models built independently for smaller regions of the code.

Compression. Compression makes our approach scale without relying on machine learning approximations: When executing a single configuration, we can *simultaneously* measure the execution time of multiple regions. If the regions are influenced by different options, a common case confirmed by prior empirical research described in Chapter 2, we can measure the performance of all regions with a few configurations, instead of exploring all combinations of all options. For example, the three independent regions in Fig. 3.2 influenced by options `A`, `B`, and `C`, respectively, each have two distinct paths. Instead of exploring all 8 combinations of the three options, we can explore all distinct paths in each region with only 2 configurations, as long as each option is enabled in one configuration and disabled in the other configuration.

Insight: Compression allows us to simultaneously explore paths in multiple independent regions with a few configurations.

We combine compositionality and compression to efficiently build accurate performance-influence models, without traditional sampling or machine-learning techniques. To help users and developers understand the influnece of options on the performance of systems, the resulting models can be presented in an interpretable format (e.g., sparse linear models) and even be mapped to individual code regions. Key to our approach is the property that not all options interact in the same region, instead influencing different parts of the system independently; a pattern observed empirically in configurable systems (Chapter 2).

To operationalize compositionality and compression for efficiently building accurate and interpretable performance-influence models, we need three technical components, shown in Fig. 3.3: First, we identify which regions are influenced by which options to select configurations to explore all paths per region and map measured execution time to options and their interactions (Sec. 3.2.1). Second, we execute the system to measure the performance of all paths of all regions (Sec. 3.2.2). Third, we build local performance-influence models per region and compose them into one global model for the system (Sec. 3.2.3).

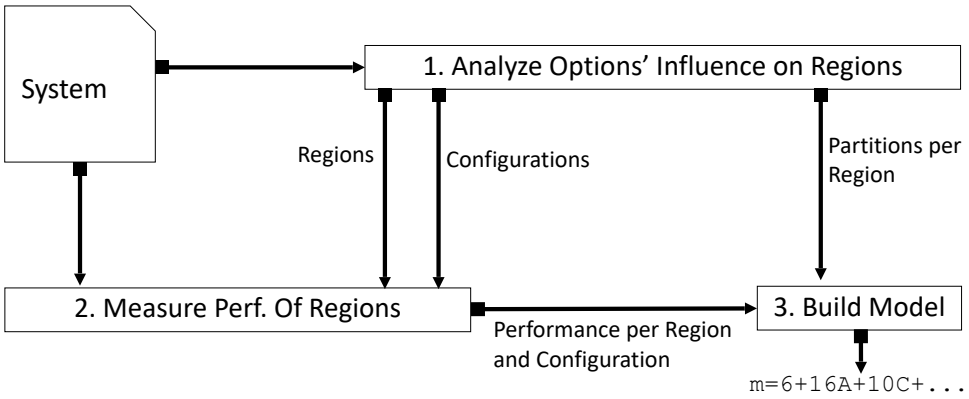

Figure 3.3: Overview of components to efficiently building accurate and interpretable performance-influence models.

3.2 Components for Modeling the Performance of Configurable Systems

3.2.1 Analyze Options' Influence on Regions

As a first step, we identify which options (directly or indirectly) influence control-flow statements in which regions, which we use to select configurations to explore all paths per region and map measured performance differences to options and their interactions (Sec. 3.2.2).[2] To this end, we track *information flow* from options (sources) to *control-flow statements* (sinks) in each region. If an options flows, directly or indirectly (including implicit flows), into a control-flow statement in a region, it implies that selecting or deselecting the option may lead to different execution paths within the region. Thus, we should observe at least one execution with a configuration in which the options is selected and another execution in which the option is not selected.

More specifically, we conservatively partition the configuration space per region into subspaces, such that every configuration in each subspace takes the same path through the control-flow statements within a region, and that all distinct paths are explored when taking one configuration from each subspace. A *partition* of the configuration space is a grouping of configurations into nonempty subsets, which we call *subspaces*, such that each configuration is part of exactly one subspace. For notational convenience, we describe subspaces using propositional formulas over options. For example, $[\![\mathtt{A} \wedge \neg \mathtt{B}]\!]$ describes the subspace of all configurations in which option `A` is selected and option `B` is deselected.

To track information flow between options and control-flow statements in regions, we

[2] As previously discussed in Chapter 2, we focus on configuration changes in control-flow statements, as a system's execution time changes in those statements, depending on which branch is executed and how many times it is executed, confirmed by empirical research [Han and Yu, 2016; Jin et al., 2012; Nistor et al., 2015, 2013a; Siegmund et al., 2013]. Execution differences caused by nondeterminism are orthogonal and must be handled in conventional ways (e.g., averaging multiple observations or controlling the environment).

Table 3.1: Performance map per region and configuration for our running example in Fig. 2.1.

Configurations				Regions			
A	B	C	D	Base	$R_1 \equiv \{A\}$	$R_2 \equiv \{A, B\}$	$R_3 \equiv \{A, C\}$
F	F	F	F	1s	2s	0s	5s
F	T	T	F	1s	2s	0s	15s
T	F	F	F	1s	1s	1s	20s
T	T	T	F	1s	1s	4s	60s

F: False; T: True. We show the set of options that influence each region.

use a *taint analysis*. During the analysis, we track how API calls load configuration options (sources) and propagate them along data-flow and control-flow dependencies, including implicit flows, to the decisions of control-flow statements (sinks). By tracking how options flow through the system, we can identify, for each control-flow statement, the set of options that reach the statement, potentially leading to different execution paths in a region. Subsequently, we conservatively partition the configuration space of a region into subspaces based on the set of options that reach the statement.

Example: The options in our running example in Fig. 2.1 (Lines 2–3) are the fields `A` – `D`. Lines 4–5 are not influenced by any options. Lines 6–12 and Lines 13–15 are influenced by the set of options $\{A\}$, which leads to the partition $\{[\![A]\!], [\![\neg A]\!]\}$. Lines 17–18 by $\{A, B\}$, which leads to the partition $[\![\neg A \wedge \neg B]\!]$, $[\![\neg A \wedge B]\!]$, $[\![A \wedge \neg B]\!]$, $[\![A \wedge B]\!]$. Lines 20– 21 by $\{A, C\}$, which leads to the partition $[\![\neg A \wedge \neg C]\!]$, $[\![\neg A \wedge C]\!]$, $[\![A \wedge \neg C]\!]$, $[\![A \wedge C]\!]$.

3.2.2 Measure Performance of Regions

We measure the time the system spends in each region when executing a configuration, resulting in performance measurements for each pair of configuration and region. We measure *self-time* per region to track the time spent in the region itself, which excludes the time of calls to execute code from other regions.

Ideally, we want to find a minimal set of configurations, such that we explore at least one configuration per subspace for each region's partition. Since finding the optimal solution is NP-complete[3] and existing heuristics from combinatorial interaction testing [Al-Hajjaji et al., 2016; Hervieu et al., 2011, 2016; Kuhn et al., 2013] are expensive, we developed our own simple greedy algorithm: Incrementally intersecting subspaces that overlap in at least one configuration, until no further such intersections are possible. Then, we simply pick one configuration from each subspace.

Example: In our running example in Fig. 2.1, four configurations cover all subspaces, for instance, $\{\{\}, \{A\}, \{B, C\}, \{A,B,C\}\}$, where each set represent options that are selected in the

[3]The problem can be reduced to the set cover problem, in which the union of a collection of subsets (all subspaces) equals a set of elements called "the universe" (the union of all subspaces). The goal is to identify the smallest sub-collection whose union equals the universe.

configuration. Table 3.1 presents a performance map per region and executed configuration of our running example.

3.2.3 Building the Performance-Influence Model

In the final step, we build performance-influence models for each region based on the (1) the partitions identified per region and (2) the performance map per region and configuration. We then compose the local models into a performance-influence model for the entire system.

Since we collect at least one measurement per distinct path through a region, building models is straightforward, without the need of using machine learning to extrapolate from incomplete samples. For a region with a partition and a set of configurations with corresponding performance measurements, we associate each measurement with the subpsace of the partition to which the configuration belongs. If multiple measured configurations belong to the same subspace, we expect the same performance behavior for that region (modulo measurement noise) and average the measured results. As a result, we can map each subspace of a region's partition to a performance measurement. For instance, for the region in `foo` in our running example in Fig. 2.1, all configurations in which A is deselected take 0 seconds, all configurations in which A is selected and B is deselected take 1 second, and all configurations in which A and B are selected take 4 seconds.

For interpretability, to highlight the influence of options and avoid negated terms, we write linear models in terms of options and interactions, for example $\mathtt{m}_{\mathtt{foo}} = 1\mathtt{A} + 3\mathtt{AB}$.

The global performance-influence model is obtained simply by aggregating all local models; we add the individual influences of each model. Note the local models can be useful for understanding and debugging individual regions, as they describe the performance behavior of each region (see Chapter. 4).

Example: With the performance map per region and configuration in Table 3.1 for our running example in Fig. 2.1, we build the local models $\mathtt{m}_{\mathtt{base}} = 1$, $\mathtt{m}_{\mathtt{R1}} = 2 - 1\mathtt{A}$, $\mathtt{m}_{\mathtt{R2}} = 1\mathtt{A} + 3\mathtt{AB}$, and $\mathtt{m}_{\mathtt{R3}} = 5 + 15\mathtt{A} + 10\mathtt{C} + 30\mathtt{AC}$, which can be composed into the global performance-influence model $\mathtt{m} = 8 + 15\mathtt{A} + 10\mathtt{C} + 3\mathtt{AB} + 30\mathtt{AC}$.

3.3 Design Decisions for Modeling the Performance of Configurable Systems

3.3.1 Analyze Option's Influence on Regions

Our approach to analyze the (direct and indirect) influence of options on the decisions of control-flow statements with a taint analysis can be performed statically or dynamically with different tradeoffs.

The main benefit of a static taint analysis is that it covers all execution paths in a single analysis of the system [Arzt et al., 2014]. However, the analysis might cover parts of the program that are never executed, which can increase the time of the analysis and threaten its scalability in large-scale systems. Additionally, the analysis only indicates the options or

interactions that *might* affect the decisions in control-flow statements (i.e., there might be false positives), which might unnecessarily increase the number of configurations that we measure.

The main benefit of a dynamic taint analysis is that it executes the system tracking how options actually influence the decisions in control-flow statements (i.e., no false positives) [Bell and Kaiser, 2014]. However, dynamic analyses are, by definition, unsound; we cannot not know how options influence the decisions in control-flow statements in the parts of the program that are not executed. Accordingly, we would need to execute the analysis multiple times with different configurations, which might threaten its scalability in systems with large configuration spaces.

3.3.2 Granularity of Regions, Compression, and Measuring Performance

Our approach to model the performance of configurable systems can be applied to different levels of granularity, but with different tradeoffs.

On one extreme, we could consider the entire system as a single region (as black-box approaches do), but would not benefit from compression. At the other extreme, we could consider each control-flow statement as the start of its own region, ending with its immediate post-dominator, which results in maximum compression, but in excessive measurement cost; this fine-grained granularity is analogous to using an *instrumentation profiler*, but instead of focusing on a few locations of interest, as usually recommended [Lange, 2011], we would add instrumentation throughout the entire system at control-flow statements.

We can also consider methods as regions. In this case, we may lose some compression potential compared to more fine-grained regions, if multiple control-flow statements within a method are influence by distinct options. On the other hand, we can use off-the-shelf *sampling profilers* that accurately capture performance with low overhead, and simply map the performance of methods to the closest regions on the calling stack.

3.3.3 Implementing Two Prototypes

We implemented and evaluated two prototypes to model the performance of configurable systems considering the tradeoffs of the taint analyses and granularity of regions discussed above. We implemented ConfigCrusher [Velez et al., 2020a], which uses a static taint analysis considering control-flow statements as regions. We also implemented Comprex [Velez et al., 2020b], which uses a dynamic taint analysis considering methods as regions. In the following sections, we describe the implementation of each prototype.

3.4 ConfigCrusher

3.4.1 Analyze Options' Influence on Regions

We used the state of the art object-, field-, context-, and flow-sensitive static taint analysis engine FlowDroid for Java systems [Arzt et al., 2014]. We tracked control-flow and and data-

flow dependencies (including implicit flows) as described in Sec. 3.2.1 considering control-flow statements as regions.

3.4.2 Measure Performance of Regions

To measure the performance of control-flow statements as regions, we need to instrument the regions. We developed an algorithm to identify and instrument the start and end of regions (more details in [Velez et al., 2020a]). One important task of the algorithm is to find the end of a region where all the paths originating from a control-flow statement meet again (i.e., the immediate post-dominator).

We instrument the start and end of regions with statements to log their execution time. We also instrument the entry point of the system (e.g., the main method in a Java system) to measure the performance of code not influenced by any options. The result of executing an instrumented system is the total time spent in each region.

Optimization. When we executed our instrumented systems, we observed excessive execution overhead even in small systems. We found that the overhead arose from redundant, nested regions (i.e., regions with the same set of influencing options), and regions executed repeatedly in loops. Consequently, we identified optimizations to reduce measurement overhead through instrumenting regions differently without altering the measurements that we collect.

Specifically, we developed two algorithms to propagate the options that influence statements up and down a control-flow graph (i.e., intraprocedually), as well as across graphs (i.e., interprocedually), to combine regions and pull out nested regions. The algorithms never create new interactions nor do they alter the performance measurements that we collect, but significantly reduce the overhead of measuring the instrumented system (more details of the algorithms can be found in [Velez et al., 2020a]).

3.5 Comprex

3.5.1 Analyze Options' Influence on Regions

We used Phosphor, the state of the art tool for dynamic taint analysis in Java [Bell and Kaiser, 2014]. We tracked control-flow and and data-flow dependencies (including implicit flows) as described in Sec. 3.2.1 considering methods as regions. However, to partition the configuration space per region, we iteratively execute the dynamic taint analysis with different configurations until we have explored all distinct paths in each region.

Incrementally partitioning the configuration space. We developed an algorithm to partition the configuration space per region, based on incremental updates from our dynamic taint analysis (more details in [Velez et al., 2020b]). Intuitively, we execute the system in a configuration and observe when data-flow and control-flow taints from options reach each control-flow decision in each region, and subsequently update each region's partition: Whenever we reach

		{A,D}	{A,B,C}	{}	{C}
1	`def main(List workload)`				
2	`  a = getOpt("A"); b = getOpt("B");`				
3	`  c = getOpt("C"); d = getOpt("D");`				
4	`  ...`				
5	`  int i = 0;`				
6	`  if(a)`				
7	`  ...`				
8	`  foo(b);`	[[A]], [[¬A]]	[[A]], [[¬A]]	[[A]], [[¬A]]	[[A]], [[¬A]]
9	`  i = 20;`				
10	`  else`				
11	`  ...`				
12	`  i = 5;`				
13	`  while(i > 0)`				
14	`  bar(c);`				
15	`  i--;`				
16	`def foo(boolean x)`	[[A∧B]], [[A∧¬B]], [[¬A]]	[[A∧B]], [[A∧¬B]], [[¬A]]	[[A∧B]], [[A∧¬B]], [[¬A]]	[[A∧B]], [[A∧¬B]], [[¬A]]
17	`  if(x) ...`				
18	`  else ...`				
19	`def bar(boolean x)`	[[A∧C]], [[A∧¬C]], [[¬A]]	[[A∧C]], [[A∧¬C]], [[¬A]]	[[A∧C]], [[A∧¬C]], [[¬A∧C]], [[¬A∧¬C]]	[[A∧C]], [[A∧¬C]], [[¬A∧C]], [[¬A∧¬C]]
20	`  if(x) ...`				
21	`  else ...`				

Figure 3.4: Example of executing the iterative analysis on our running example in Fig. 2.1. Four configurations explore all subspaces for the three regions in the system, where each set represents the options selected in the configuration. For each configuration, we show the subspaces generated for each region. Subspaces in red still need to be explored, whereas subspaces in green have been explored in previous configurations. Note how we explore the nested `if` statement in method `foo` with 3 instead of 4 subspaces by separately tracking data-flow and control-flow taints. Also note how we update the $[\![\neg A]\!]$ subspace in method `bar` after the third configuration to explore the region with both values of `C` when `A` is deselected.

a control-flow statement during execution, we identify, based on taints that reach the condition of the statement, the sets of configurations that would possibly make different decisions, thus updating the partition that represents different paths for this region. Since a dynamic taint analysis can only track information flow in the current execution, but not for alternative executions (i.e., for paths not taken), we repeat the process with new configurations, selected from the partitions identified in prior executions, updating partitions until we have explored one configuration from each subspace of each partition; that is, until we have observed each distinct path in each region at least once. Note that some subspaces in the region might make the same control-flow decision as other subspaces, but we do not know which subspace will make which decision until we actually execute those configurations.

Distinguishing data-flow taints from control-flow taints allows us to perform an additional optimization to more efficiently explore nested decisions (e.g., `if(a){ if(b) ... }`). Control-flow taints specify which options (directly or indirectly) influenced outer control-flow decisions, which indicate that different assignments to options in the control-flow taints *may* lead to paths where the current decision is not reached in the first place. Hence, we do not necessarily need to explore all interactions of options affecting outer and inner decisions. Instead of exploring combinations for all options of data-flow and control-flow taints, as we did when using a static taint analysis, we first split the configuration space into those configurations for which we know that they will reach the current decision, because they share the assignments

of options in control-flow taints, and the remaining configurations which may not reach the current decision. Then, we only create subspaces for interactions of options in data-flow taints for configurations that reach the current decision, and consider the entire set of configurations that may not reach the decision as a single subspace. The iterative nature of our analysis ensures that at least one of the configurations which may not reach the current decision will be explored, and, if the configuration also reaches the same decision, the region's partition will be further divided.

The iterative analysis executes the system in different configurations until one configuration from each subspace of each partition in each region has been explored. That is, we start by executing any configuration (e.g., the default configuration), which reveals the subspaces per regions that could make different decisions. The algorithm then selects the next configuration to explore unseen subspaces in the regions, which may further update the regions' partitions. To select the next configuration, we use a greedy algorithm to pick a configuration that explores the most unseen subspaces across all regions.[4]

Example: Fig. 3.4 presents an example of executing the iterative analysis on our running example in Fig. 2.1.

Dynamic Taint Analysis Overhead. We observed that tracking control-flow dependencies imposes significant overhead in the system's execution. For instance, one execution of our subject system Berkeley DB takes $\sim$ 1 hour with the dynamic taint analysis, whereas ~ 300 configurations can be executed in the same time! In general, we observe $26\times$ to $300\times$ overhead from taint tracking, which varies widely between systems. In fact, the iterative analysis did no finish executing after 24 hours in all subject systems, except for Apache Lucene, which executed in 11 hours. To reduce cost, we execute the iterative analysis with a *drastically* reduced workload size.

This optimization is feasible when the workload is *repetitive* and repetitions of operations are affected similarly by options, which we conjecture to be common in practice. Many performance benchmarks execute many operations, which are similarly affected by configuration options. For instance, Berkeley DB's `MeasureDiskOrderedScan` benchmark populates a database, which can be scaled by a parameter that controls the number of entries to insert, but does not affect which *operations* are performed. In our evaluation, we show that we can generate accurate performance-influence models using a significantly smaller workload in the iterative analysis.

3.5.2 Measure Performance of Regions

To measure the performance of methods as regions, we use JProfiler, an off-the-shelf sampling profiler that accurately captures performance of methods with low overhead [JPR, 2019].

[4]To avoid enumerating an exponential number of configurations, we use a greedy algorithm that picks a random subspace and incrementally intersects it with other non-disjoint subspaces, which seems sufficiently effective in practice. The problem can also be encoded as a MAXSAT problem, representing subspaces as propositional formulas, to find the configuration that satisfies the formula with the most subspaces.

Table 3.2: State of the art approaches compared to ConfigCrusher in [Velez et al., 2020a] and Comprex in [Velez et al., 2020b].

Approach	Compared to ConfigCrusher	Compared to Comprex
Feature-wise & SL	✗	✓
Feature-wise & LL	✗	✓
Feature-wise & SLR	✓	✓
Feature-wise & EN	✗	✓
Feature-wise & SDT	✗	✓
Feature-wise & DT	✗	✓
Feature-wise & RF	✗	✓
Feature-wise & NN	✗	✓
Pair-wise & SL	✗	✓
Pair-wise & LL	✗	✓
Pair-wise & SLR	✓	✓
Pair-wise & EN	✗	✓
Pair-wise & SDT	✗	✓
Pair-wise & DT	✗	✓
Pair-wise & RF	✗	✓
Pair-wise & NN	✗	✓
50 random configurations & SL	✗	✓
50 random configurations & LL	✗	✓
50 random configurations & SLR	✗	✓
50 random configurations & EN	✗	✓
50 random configurations & SDT	✗	✓
50 random configurations & DT	✗	✓
50 random configurations & RF	✗	✓
50 random configurations & NN	✗	✓
200 random configurations & SL	✗	✓
200 random configurations & LL	✗	✓
200 random configurations & SLR	✗	✓
200 random configurations & EN	✗	✓
200 random configurations & SDT	✗	✓
200 random configurations & DT	✗	✓
200 random configurations & RF	✗	✓
200 random configurations & NN	✗	✓
Family-based[1]	✓	Not Applicable

SL: Simple linear regression; LL: Lasso linear regression; SLR: Stepwise linear regression; EN: Elastic net linear regression; SDT: Shallow decision tree (max depth=3); DT: Decision tree; RF: Random forest; NN: Multi-layer preceptron;

[1] Originally evaluated in [Velez et al., 2020a], as it was the current white-box state of the art approach. We do not propose to evaluate the approach with Comprex, due to the former's known limitations (Chapter 2).

3.6 Evaluation

We originally evaluated ConfigCrusher [Velez et al., 2020a] and Comprex [Velez et al., 2020b], separately, using different subject systems and against different state of the art approaches

Table 3.3: Subject systems evaluated with ConfigCrusher in [Velez et al., 2020a] and Comprex in [Velez et al., 2020b].

System	Domain	#SLOC	#Opt.	#Conf.	Prototype used for evaluation
Pngtastic Counter	Image processor	1250	5	32	ConfigCrusher
Pngtastic Optimizer	Image optimizer	2553	5	32	ConfigCrusher
Elevator	SPL benchmarkr	575	6	20	ConfigCrusher
Grep	Utility	2152	7	128	ConfigCrusher
Kanzi	Compressor	20K	7	128	ConfigCrusher
Email	SPL benchmark	696	9	40	ConfigCrusher
Prevayler	Database	1328	9	512	ConfigCrusher
Sort	Utility	2163	12	4096	ConfigCrusher
H2	Database	142K	16	65K	Comprex
Berkeley DB	Database	164K	16	65K	Comprex
Apache Lucene	Index/Search	396K	17	131K	Comprex
Density Converter V1	Image processor	1359	22	4.9M	ConfigCrusher
Density Converter V2	Image processor	49K[1]	22	4.9M	Comprex

Opt: Options; Conf: Configurations; [1]: The system is an interface to several libraries for processing images. We included and analyzed all Java dependencies in this version of the system.

to build performance-influence models, for a specific workload, input size, and underlying hardware. Table 3.2 shows an overview of the state of the art approaches we evaluated.

Proposed work: We propose to evaluate ConfigCrusher and Comprex together against the same state of the art approaches.

Subject Systems

We selected 13 configurable widely-used open-source Java systems that satisfy the following criteria (common in our domain): (a) systems from a variety of domains to increase external validity, (b) systems with binary and non-binary options, and (c) systems with fairly stable execution time (we observed execution times within usual measurement noise for repeated execution of the same configuration). Table 3.3 provides an overview of all subject systems.

ConfigCrusher limitation. We observed that the scalability of ConfigCrusher is limited by the used static taint analysis. Specifically, the analysis is challenged by the size of the call graph, which restricts the size of the systems that our implementation can analyze [Arzt et al., 2014; Avdiienko et al., 2015; Bodden, 2018; Do et al., 2017; Lerch et al., 2015; Pauck et al., 2018; Qiu et al., 2018; Wang et al., 2016]; he largest subject system for which the static taint analysis terminated was Kanzi, which has over 20K SLOC. Accordingly, we only evaluated ConfigCrusher using the 9 subject systems with under 20K SLOC.

Comprex. We originally evaluated Comprex on medium- to large-scale systems in [Velez et al., 2020b] to demonstrate that a white-box analysis can model the performance of large

System	Brute-force	Feature-wise	Pair-wise	Family-Based[1]	ConfigCrusher[2]
Pngtastic Counter	32 [2.9m]	5 [27.2s]	16 [1.5m]	N/A	4 [21.9s, 7.8s]
Pngtastic Optimizer	32 [42.2m]	5 [1.6m]	16 [10.0m]	N/A	10 [10.7m, 30.6s]
Elevator	20 [10.8m]	3 [50.0s]	9 [3.3m]	1 [49.5s]	64 [—]
Grep	128 [10.6m]	7 [22.1s]	29 [1.9m]	N/A	64 [5.1m, 10.2s]
Kanzi	128 [1.2h]	7 [1.5m]	29 [8.8m]	N/A	64 [35.4m, 12.6s]
Email	40 [16.9m]	4 [23.5s]	11 [1.7m]	1 [1.1m]	8 [1.5m, 12.8s]
Prevayler	512 [3.7h]	9 [2.7m]	46 [16.0m]	N/A	32 [14.5m, 12.6s]
Sort	1298 [18.4h]	12 [13.1m]	79 [1.4h]	N/A	256 [3.7h, 21.6s]
Density Converter V1	1414 [14.7h]	22 [21.3m]	254 [4.1h]	N/A	256 [2.1h, 42.1s]

[1] Not applicable to systems without static map derived from compile-time variability.
[2] Time includes the overhead of the static taint analysis.

(a) Cost of building performance-influence models.

System	Feature-wise	Pair-wise	Family-Based[1]	ConfigCrusher
Pngtastic Counter	**0.8**	2.0	N/A	**1.1**
Pngtastic Optimizer	19.7	**0.9**	N/A	**1.1**
Elevator	51.1	**1.5**	**2.7**	∅
Grep	32.1	114.7	N/A	**3.6**
Kanzi	**1.9**	**1.3**	N/A	**2.7**
Email	100	44.2	**2.3**	23.0
Prevayler	111.2	29.2	N/A	**9.2**
Sort	90.0	653.0	N/A	**1.6**
Density Converter V1	635.2	218.9	N/A	**4.3**

Bolded values in cells indicate indistinguishable lowest errors. ∅ approach sampled all configurations, thus no performance to predict.

[1] Not applicable to systems without static map derived from compile-time variability.

(b) Mean Absolute Percentage Error (MAPE) comparison (lower is better).

Table 3.4: Cost and accuracy comparison.

systems in terms of SLOC. We propose to evaluate all subject systems with Comprex in the final dissertation document.

Proposed work: We propose to evaluate all subject systems with Comprex.

3.6.1 ConfigCrusher

We compared ConfigCrusher to the Family-Based approach [Siegmund et al., 2013], and two combinations of sampling approaches with stepwise linear regression [Siegmund et al., 2015, 2012a,b]: feature-wise sampling (i.e., enable one option at a time) and pair-wise sampling (i.e., cover all combinations of all pairs of options) [Medeiros et al., 2016]. Specifically, we measured the accuracy of the models and the cost to build the models, in terms of the number of configurations measured, the time to measured those configurations, and for ConfigCrusher,

the time to run the static taint analysis.

Table 3.4 summarizes the cost and error results. ConfigCrusher's prediction error is statistically indistinguishable or lower than other approaches. Furthermore, ConfigCrusher's high accuracy is usually achieved with lower cost compared to the other accurate approaches. The efficiency originates from ConfigCrusher's white-box analysis to identify a small number of relevant configurations to capture the performance-relevant interactions.

Though feature-wise and pair-wise tended to measure fewer configurations than ConfigCrusher, when their errors are taken into account, we can conclude that more configurations had to be measured to make more accurate predictions. By comparison, for those systems, ConfigCrusher sampled more configurations, but attained significantly lower errors.

3.6.2 Comprex

We compared Comprex to numerous combinations of sampling and learning approaches (see Table 3.2). For learners, we evaluate variations of linear regressions [Siegmund et al., 2015, 2012a,b], decision trees and random forest [Grebhahn et al., 2019; Guo et al., 2013, 2017; Sarkar et al., 2015], and a neural network. For sampling, we evaluate uniform random sampling with 50 and 200 configurations, feature-wise sampling (i.e., enable one option at a time), and pair-wise sampling (i.e., cover all combinations of all pairs of options) [Medeiros et al., 2016]. We selected 50 and 200 random configurations to use more configurations than other sampling strategies and use sampling sets comparable to ones used in related research. In this proposal, we highlight the results for stepwise linear regression and random forest. Similar conclusions can be drawn when comparing Comprex to the other black-box approaches. Specifically, we measured the accuracy of the models and the cost to build the models, in terms of the number of configurations measured, the time to measured those configurations, and for Comprex, the time to run the iterative taint analysis. Since we compare Comprex to non-linear models, we also discuss the interpretability of the models generated by each approach.

Interpretability. We intend the models generated with our approach to be used in performance understanding and debugging tasks. Hence, is beneficial for these tasks if the models are easy to interpret by users and developers (Chapter 2).

Despite much research on the intrepretability of models, there is no generally agreed measure or even definition for interpretability [Doshi-Velez and Kim, 2017; Molnar, 2019]. Nevertheless, interpretability typically captures the ability of humans to make predictions, understand predictions, or understand the decisions of the model. Researchers typically distinguish between (1) *inherently interpretable* models, which humans can inspect and directly reason about the model structure and parameters, and (2) *post-hoc explanations*, where tools provide explanations [Lundberg and Lee, 2017; Molnar, 2019; Ribeiro et al., 2016; Štrumbelj and Kononenko, 2014] while model internals are not directly shown [Molnar, 2019]. Our discussion focuses on the former, since post-hoc explanations may be unreliable or misleading [Rudin, 2019].

Sparse linear models, with dozens of individual and interacting terms are generally accepted as inherently interpretable [Molnar, 2019]. Humans can inspect them, reason about factors, and make and understand predictions For instance, machine learning researchers recommend

these models in high-stakes decisions, when auditing the model is paramount [Rudin, 2019]. Likewise, interviews have shown that developers understand linear performance models with a few dozen terms [Kolesnikov et al., 2018]. Hence, we argue that the kind of models we build are interpretable.

Decision trees are also often considered as inherently interpretable [Rudin, 2019] when understanding the decisions behind a single prediction, as following a specific path and all involved decisions in a model is easy. However, identifying influences of factors globally is more challenging, as a factor may occur in many places in a tree and one has to reason about many or all paths (e.g., how much interacting options slow down the systems). When decision trees get deep, the models becomes more tedious to understand.

In contrast to decision trees, *random forests* are not considered inherently interpretable, because they are an ensemble of numerous (e.g. 100) decision trees. Understanding random forests would require to understand the *average* effect of options around all trees, which are usually fairly deep.

Results. Table 3.5 summarizes the cost and error results. Overall, Comprex builds models that are similarly accurate to those learned by the most accurate and expensive black-box approach (random forests with 200 samples), but our models are interpretable and usually built more efficiently, despite the cost of the iterative analysis. Comprex outperforms other approaches that build linear models by a wide margin.

While random forest with 200 samples produced slightly more accurate models than Comprex, Comprex was usually more efficient, in some cases building models in half the time, while also generating local models and interpretable models. The efficiency originates from Comprex's white-box analysis to identify a small number of relevant configurations to capture the performance-relevant interactions. By contrast, as our results show, black-box approaches perform significantly worse on such small samples (e.g., compare R50 and R200 results).

Thesis contribution: Our white-box prototypes can efficiently and accurately model the performance configurable systems. The accurate models are often built more efficiently than approaches with comparable accuracy. Furthermore, the models are interpretable and can be mapped to specific code regions.

3.7 Discussion

3.7.1 Proposed Work: Static vs. Dynamic Taint Analysis

The different types of running a taint analysis resulted in different tradeoffs when modeling the performance of configurable systems. On one hand, we executed the static taint analysis once in a few seconds, which reduces the cost to build our models. However, the static taint analysis was limited to relatively small systems. On the other hand, we executed multiple configurations to run the iterative taint analysis, but we were able to analyze medium- to large-scale systems. These results indicate that using a dynamic taint analysis should be used to model the performance of medium- to large-scale systems.

Sample	Apache Lucene	H2	Berkeley DB	Density Converter V2
BF	2^{17} [~48.4d]	2^{16} [~16.0d]	2^{16} [~8.6d]	2^{22} [~3.6y]
R50	50 [26.7m]	50 [16.4m]	50 [9.1m]	50 [10.1m]
R200	200 [1.8h]	200 [1.1h]	200 [36.4m]	200 [40.4m]
FW	17 [8.6m]	16 [2.4m]	16 [4.8m]	22 [8.2m]
PW	154 [1.3h]	137 [21.1m]	137 [39.4m]	254 [1.8h]
Comprex	26 [14.9m]	64 [22.6m]	144 [30.2m]	88 [16.6m]

The time to measure configurations for BF is extrapolated from 2000 randomly selected configurations.

(a) Cost of sampling configurations.

Approach	Apache Lucene	H2	Berkeley DB	Density Converter V2
R50 & LR	8.9s	6.6s	5.7s	14.9s
R200 & LR	6.8m	4.6m	4.9m	1.6m
FW & LR	9.4s	4.3s	7.7s	19.8s
PW & LR	1.7m	44.8s	3.6m	5.5m
* & RF	$\leq$0.2s	$\leq$0.2s	$\leq$ 0.3s	$\leq$0.2s
Comprex	28.9m	9.3m	11.2m	8.5m

(b) Learning/Analysis time.

Approach	Apache Lucene	H2	Berkeley DB	Density Converter V2
R50 & LR	**4.5**	124.1	19.7	1037.2
R200 & LR	**2.9**	93.9	14.9	434.5
FW & LR	**7.9**	129.3	768.7	1596.0
PW & LR	**4.7**	113.3	34.2	1596.0
R50 & RF	**0.8**	**6.5**	14.1	268.7
R200 & RF	**0.3**	**0.7**	**1.1**	**5.5**
FW & RF	**8.7**	119.0	106.1	1185.9
PW & RF	**4.0**	124.6	53.5	403.6
Comprex	**3.2**	**2.9**	**5.0**	**9.4**

LR: Stepwise linear regression; RF: Random forest; R50: 50 random configurations; R200: 200 random configurations; FW: Feature-wise; PW: Pair-wise; **Bolded** values in cells indicate similarly low errors.

(c) Mean Absolute Percentage Error (MAPE) comparison (lower is better).

Table 3.5: Cost and accuracy comparison.

Originally, we did not evaluate Comprex on the relatively small configurable systems for which ConfigCrusher efficiently generate accurated performance-influence models. We propose to evaluate ConfigCrusher and Comprex more closely by evaluating the relatively small configurable systems with Comprex to compare the cost to build the models, including the analyses times, and their accuracy. We expect the comparison to yield similar results to evaluating the two versions of Density Converter with ConfigCrusher in Table 3.4 and Comprex in Table 3.5; the approaches generated similarly accurate models (4.3 vs. 9.4 in terms of MAPE),

Table 3.6: Number of regions and configurations to measure with compression at different region granularities.

	Control-flow		Method		Program	
System	#Reg.	#Conf.	#Reg.	#Conf.	#Reg.	#Conf.
Lucene	1654	**26**	551	**26**	1	16384
H2	2483	**64**	932	**64**	1	256
Berkeley DB	2152	**144**	718	**144**	1	2048
Density Converter	190	**88**	62	**88**	1	4608

#Reg: Number of regions; #Conf: Number of configurations. **Bolded** values in cells indicate the fewest number of configurations to cover all partitions' subspaces.

but Comprex measured far fewer configurations (256 vs 88), and overall took less time to build the model (2.1 hours vs. 25.1 minutes), thanks to (1) actually tracking, with a dynamic analysis, how options influence the decisions of control-flow statements and (2) separately tracking control-flow and data-flow taints to identify that some options are only relevant in certain executions of outer control-flow decisions.

3.7.2 Granularity of Regions, Compression, and Measuring Performance

Considering different granularities of regions yielded different tradeoffs between compression potential and measuring the performance of regions. One one hand, considering control-flow statements as regions in ConfigCrusher resulted in maximum compression, but caused excessive measurement overhead, as we instrument numerous locations in the program. On the other hand, we considered methods as regions in Comprex, which allowed us to use an off-the-shelf sampling profiler to accurately measure the performance of methods with low overhead, but potentially lost some compression opportunities.

We explored the impact of choosing regions at different granularities on the *number of configurations* to measure. Specifically, we executed Comprex's iterative analysis considering each method as a region. We additionally tracked partitions for control-flow statements and derived partitions for the entire program by combining the partitions of all methods. We performed the analysis using the 4 subject systems that we originally evaluated in [Velez et al., 2020b], but conjecture that we will obtain similar results in the other systems, as both all systems have similar characteristics of how options are used and interact in configurable systems.

Results. Table 3.6 reports the size of the minimum set of configurations needed to cover each subspace of each region's partition for each granularity. When considering the entire program as a region, significantly more configurations need to be explored, as we do not benefit from compression. Interestingly though, while there are, as expected, fewer regions at the method level than at the control-flow statement level, the number of configurations needed is the same. These results show that compression at finer-grained levels than the method level

do not yield additional benefits in our subject systems.

We found that the control-flow statement regions combined within a method are usually partitioned in the same way. Only in 3 out of the 2263 method level regions, the method's partition had more subspaces than the corresponding control-flow statement regions (e.g., two `if` statements depending on different options). However, in all three cases, the additional subspaces were already explored in other parts of the program. Hence, no additional configurations needed to be explored.

We conclude that fined-grained compression is highly effective, but that control-flow granularity does not seem to offer significant compression benefits over method granularity. Accordingly, we conclude that considering methods as regions is highly efficient to reduce (1) the number of configurations to measure and (2) the overhead to measure the performance of regions.

3.8 Conclusion

This chapter presented *Compositionality* and *Compression*, the key insights for efficiently and accurately modeling the performance of configurable systems. Based on different alternatives to implemented our approach, in terms which type of taint analysis to use and how to measure, we presented two prototypes, ConfigCrusher and Comprex. Our evaluation of ConfigCrusher and Comprex demonstrated that a white-box analysis can be used to efficiently build accurate and interpretable performance-influence models. However, using a dynamic taint analysis and measuring the performance of methods as regions, which is how Comprexis implemented, we can scale the analysis to medium- to large-scale configurable systems.

Chapter 4

White-box Performance Debugging in Configurable Systems

In Chapter 2, we discussed existing research indicating that global performance-influence models provide useful, yet limited, information for debugging the performance of configurable systems. In Chapter 3, we presented local performance-influence models and argued that the models can further help debugging, as they indicate where options affect the performance in configurable systems. In this chapter, we discuss additional tool support to provide additional information to further help developers debug the performance of configurable systems.

We first explore the usefulness of local performance-influence models to understand how options affect the performance of a system in the implementation (Sec. 4.1). Based on the information and lack thereof provided by local models, we set out to understand, through a user study, the process that developers follow and the information that they need to debug the performance of configurable systems (Sec. 4.2). Based on preliminary results, we outline potential tool designs to provide that information (Sec. 4.3). Finally, we propose to conduct a user study to validate the usefulness of the information that we provide to debug the performance of configurable systems. (Sec. 4.4).

The work in this chapter is derived in part from our ASE Journal'20 article "ConfigCrusher: Towards White-box Performance Analysis for Configurable Systems" [Velez et al., 2020a].

4.1 Exploratory Analysis of Local Models

The global performance-influence models that we generate with our white-box technique can help developers understand the influence of options on the end-to-end performance of a system. Ideally though, the models would also indicate *where* the influence of options occurs in the implementation and *how* options influence, in the implementation, the performance of those locations in the system. Currently, however, developers would typically need to navigate the entire code base to answer those questions.

In Chapter 3, we presented local performance-influence models and argued that the models can further help developers debug the performance of configurable systems, as the models indicate *where* the influence of options occurs in the implementation.

Table 4.1: Analysis of options in local performance-influence models.

	PNIO		Performance influenced by options			
	NEG	Non-NEG				
System	Regions	Regions	Regions	Min ID	Max ID	Structure
Pngtastic Counter	13	2	0	N/A	N/A	Loop, I/O
Pngtastic Optimizer	3	1	3	3	3	Loop, I/O
Grep	0	0	1	6	6	Loop
Kanzi	19	2	2	6	6	Loop, I/O
Email	7	0	4	2	8	Loop, Sleep
Prevayler	22	1	5	2	5	Loop, I/O
Sort	0	0	1	8	8	Loop
Density Converter V1	8	1	1	8	8	Loop, I/O

PNIO: Performance not influenced by options; NEG: Negligible execution time (region which contribute $< 5\%$ of the execution time of the system); ID: Interaction degree.

To investigate the usefulness of local performance-influence models to understand how options affect the performance behavior of a system *in the implementation*, we conducted an exploratory study of the local models of the smaller systems we evaluated in Chapter 3. Specifically, we examined local performance-influence models and analyzed their corresponding regions in the code. We classified regions according to how options affect the regions' performance, determined how many options influence the performance of the regions, and identified the code structures that cause the performance changes in the regions.

Table 4.1 summarizes the analysis of the local performance-influence models and their corresponding regions. The local models helped us identify that the influence of options on performance can be localized to a few regions in a system, where only subsets of all options interact. Additionally, we easily located these regions in the source code to further analyze their performance behavior.

The performance behavior of the regions was caused by options influencing a loop or a control-flow statement within a loop, which either manipulated data structures, performed I/O operations, or caused threads to sleep. The structures that caused the performance behavior in a region were sometimes located in the same method where the region was instrumented (e.g., a loop performing I/O operations). In other cases, though, we found, by navigating through the code, that the structures were located in other methods called by the region (e.g., a loop that calls methods that perform I/O operations).

In the vast majority of regions, the control-flow statements were *indirectly* influenced by configuration options; options were not directly used those statements. Rather, the objects, collections, or variables used in those statements were manipulated by options, *often outside of the region.* That is, the local models helped us localize *where* options affect the performance behavior of a system (symptom), but the reasons for *how* options manipulate objects, collections, or variables that are propagated through the system to regions are, in the vast majority of cases, located in other parts of the system (the causes of the symptom). Fig. 4.2 shows an example of the difference between *where* and *how* options affect performance. Nevertheless, locating the corresponding regions of local models helped us to navigate the code to find and understand how options affect the performance behavior of the system, instead of potentially

analyzing the entire system if we did not have that white-box information.

The analysis of the source code also helped us debug surprising performance behaviors between the documentation and actual implementation of options. For instance, we discovered that two options of Pngtastic Counter did not affect performance as we expected based on the documentation. The valid range of one option was $0.0 - 1.0$, and we conjectured that the system would behave differently when different values are selected. However, a control-flow statement where this option was used always executed the same branch if the value was > 0. Locating *where and how* the option is used, allowed us to debug these inconsistencies, which are common in configurable systems [Cashman et al., 2018; Han and Yu, 2016; He et al., 2020; Rabkin and Katz, 2011; Xu et al., 2013].

Discussion

While local performance-influence models helped us to further understand how options affect the performance of configurable systems, by locating *where* options influence performance, we had to manually navigate the code to identify *how* options affect the objects, collections, or variables used in those regions.

To further help developers understand how options affect the performance of configurable systems in the implementation, we propose to develop new tool support to provide developers with relevant information for inspecting, understanding, and debugging the performance of configurable systems. The information that we provide, from global and local performance-influence models, and additional tool support, contributes to the thesis goal of reducing the energy consumption and operational costs of running configurable systems since the information will help developers debug configuration-related performance behaviors.

4.2 Proposed Work: Explore Information Needs

To ground the information that we will provide developers to debug the performance of configurable systems, we propose to conduct two users studies to *explore the process that developers follow and the information needs that they have* when (1) they begin debugging the performance of a configurable system and (2) they debug, later in the debugging process, after identifying potential options that affect the performance of certain regions, how options influence the performance of specific hotspot regions in the system. The insights from the first study will help us identify how global and local performance-influence models can help developers in the process of debugging the performance of configurable systems. The insights from the second study will guide our design of new tool support to provide relevant information to debug the performance of configurable systems. Overall, the studies will help us understand the process that participants follow and the information they need to understanding *which, where, and how* options affect the performance of a system.

Since both studies focus on understanding how developers debug the performance of configurable systems at different stages of the process (at the beginning and once developers have more information about the performance behavior of the system), we propose to conduct the studies together as two parts of a performance debugging study. The study asks participants

to help a user understand why a system is taking some specific time to execute based on a configuration.

Participant recruitment. We propose to conduct the study with developers and researchers with various levels of experience in performance analysis and debugging, and working with configurable systems. We have already conducted 14 studies with researches/developers working in academia and expect to recruit at least 5 developers working in industry.

Analysis. We propose to analyze the studies using quantitative and standard qualitative research methods [Saldaña, 2015; Schreier, 2012] used in related research [LaToza et al., 2007; LaToza and Myers, 2010; Lawrance et al., 2013; Scaffidi et al., 2011]. We propose to transcribe and code the studies to identify topics and trends in the process that developers follow and the information that they need.

Proposed work: We propose to conduct and analyze two users studies to explore the process that developers follow and the information needs that they have when (1) they begin debugging the performance of a configurable system (Study 1.1) and (2) they debug, later in the debugging process, how options influence the performance of specific hotspot regions in the system (Study 1.2).

4.2.1 Study 1.1: Process of Debugging Performance in Configurable Systems

In the first study, we propose to explore the process developers follow and the information needs that they have when they begin debugging the performance of a configurable system. The insights that we collect from this study will help us identify how global and local performance-influence models can help developers in the process of debugging the performance of configurable system. Specifically, we seek to answer the following research questions:

RQ1.1.1: What activities do developers perform to understand the performance behavior of configurable systems?

RQ1.1.2: What information do developers need to understand the performance behavior of configurable systems?

RQ1.1.3: How do developers obtain this information?

RQ1.1.4: What are the challenging actions to understand the performance behavior of of configurable systems?

Preliminary Results

Based on a simple analysis, we show the process that participants followed and information that they collected when they began debugging the performance of a configurable system in Fig. 4.1. All participants started debugging the system by reading the options' documentation, with the goal of identifying potential options that affect the performance of the system. Once

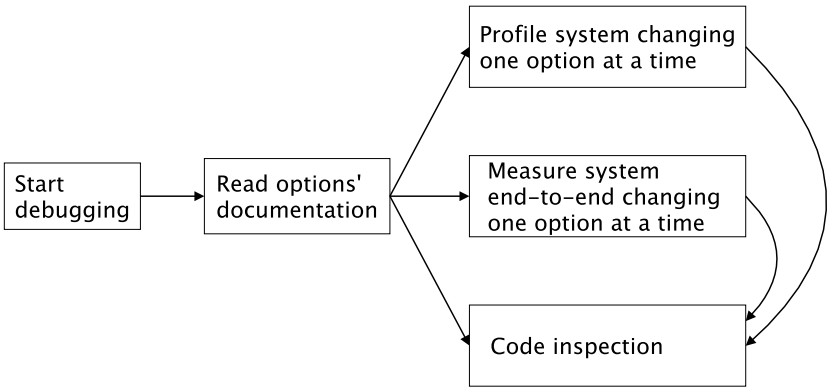

Figure 4.1: Preliminary results of the process that developers follow and information that they collect to debug the performance of a configurable system.

they identified potential options of interest, a small number of participants inspected how those options were used in the program, with the goal of understanding how the options might affect the performance of the system. The majority of participants either profiled the system or measured the end-to-end performance of the system changing one option at a time, with the goal of identifying which options affect the execution time of the system. The participants that profiled the system, however, also wanted to identify the methods where the system spends the most time executing (i.e., hotspots). Once participants identified which options affected the performance of the system, they inspected the code to understand how those options were used in the program. The participants that profiled the system usually started analyzing how the options were used in the hotspots of the system and some traced how the options were propagated through the system to the hotspots.

Based on this simple analysis, we have some evidence that when most developers start debugging the performance of a configurable system, they want to first determine how options and their interactions affect the *global* performance of the system to, subsequently, analyze *how* the options with the biggest performance impact are used in the system. Hence, for the moment, we hypothesize that global and local performance-influence models are useful for these tasks, as the global models indicate *which options* have the biggest performance impact on the system and the local models indicate *where* those options affect the performance of the system.

Proposed work: We propose to finish conducting this study with developers working in industry and analyze all studies using quantitative and standard qualitative research methods to understand the process that developers follow and the information needs that they have when they begin debugging the performance of a configurable system.

4.2.2 Study 1.2: Debugging How Options Influence the Performance of Hotspot Regions

In the second study, we propose to explore the process that developers follow and the information needs that they have when they debug, later in the debugging process, after identifying potential options that affect the performance of certain regions, how options influence the performance of specific hotspot regions in the system. The insights that we collect from this study will guide our design of new tool support to provide relevant information to debug the performance of configurable systems. Specifically, we seek to answer the following research questions:

RQ1.2.1: What activities do developers perform to understand how options affect the performance of hotspots?

RQ1.2.2: What information do developers need to understand how options affect the performance of hotspots?

RQ1.2.3: How do developers obtain this information?

RQ1.2.4: What are the challenging actions to understand how options affect the performance of hotspots?

Preliminary Results

Based on a simple analysis, we observed that most participants were surprised that options which affected the performance of hotspots were not directly used in the hotspots regions. That is, most participants assumed that the hotspots regions had the following code structure `if(option == true) { expensiveCall() }`. Rather, as we discussed in Sec. 4.1, the hotspots were *indirectly* influenced by the option, which manipulated objects, collections, or variables used in the hotspot. That is, in the vast majority of cases, a hotspot indicates the location of the symptom (i.e., the performance behavior), but the causes of the symptom are usually located somewhere else in the code. After analyzing the hotspot, participants traced how the option affected objects, collections, or variables, *in other parts of the code*, to influence the performance of hotspots.

Participants, either manually or with their IDE's debugger, analyzed how options were propagated through the system, manipulating objects, collections, or variables. Several participants also compared performance profiles for different configurations, which provides the stack traces of the hotspots. The stack traces helped participants identify whether different calls to the hotspots were made depending on which configuration was executed.

All participants struggled to completely trace how options were propagated through the system and eventually reached the hotspots to influence the hotspots' performance. Several participants mentioned that, while comparing performance profiles was useful, identifying how profiles differed was difficult, as they had to manually compare long stack traces with long method names and signatures and different execution times. Participants indicated that they would like to see the differences in the execution time and stack traces highlighted to easily spot them. Additionally, several participants mentioned that the manual tracing was tedious and error-prone, and that they would like a tool guide them through this process.

Based on this simple analysis, we hypothesize, for the moment, that developers who de-

bug how options influence the performance of specific hotspot regions in the system want to (a) trace how options reach hotspots and (b) compare performance profiles.

Proposed work: We propose to finish conducting this study with developers working in industry and analyze all studies using quantitative and standard qualitative research methods to understand the process that developers follow and the information needs that they have when they debug how options influence the performance of specific hotspot regions in the system.

4.3 Proposed Work: Tool Support to Provide Information for Debugging Performance in Configurable Systems

Based on the hypothesized process that developers follow and the information needs that they have, we propose to develop tool support to address those needs to further help developers in the process of debugging the performance of configurable systems. Based on the preliminary results, we envision (1) comparing performance profiles between configurations and (2) tracing how options are propagated through the system to influence the performance of hotspots.

4.3.1 Comparing Performance Profiles

Based on the preliminary results, we anticipate that we will compare the hotspot view of performance profiles between configurations. We could highlight the differences in stack traces in terms of the execution time and the methods that are called.

4.3.2 Tracing Options Through the System

Based on the preliminary results, we envision that we will trace how options are directly and indirectly propagated from the entry point of the system to hotspots. We could indicate the methods that options traverse to reach hotpots, as well as the objects, collections, or variables that are manipulated within each method and are passed around to influence the performance of the hotpost. Fig. 4.2 shows an example of the information that we could potentially provide developers to help them trace the options though the system.

4.3.3 Proposed Tool Support

Proposed work: We propose to develop new tool support, based on the information that we collect from our users studies, that will provide developers with relevant information to debug the performance of configurable systems in the implementation.

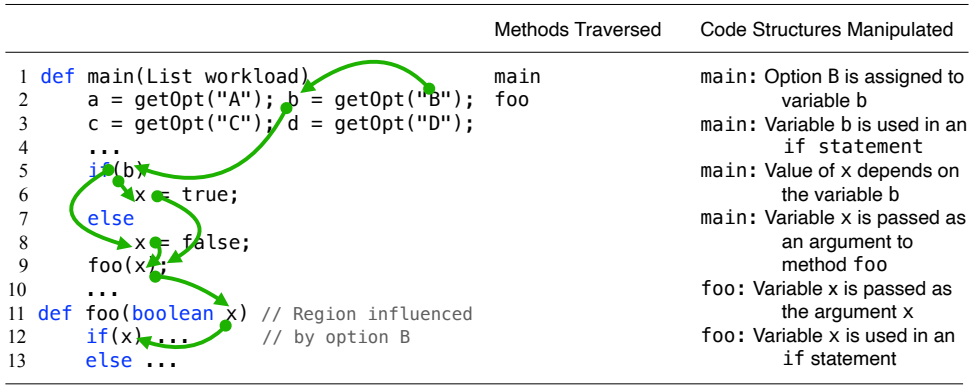

Figure 4.2: Example of the information that we could potentially provide developers to help them trace how options are propagated through the system. We might indicate the methods that options traverse, as well as the code structures manipulated in each method. We use option `B` and hostpot `foo` as an example.

4.4 Proposed Work: Study 2: Validate Tool Support

We propose to conduct a user study to validate the usefulness of the tools presented in this thesis to help developers debug the performance of configurable systems. Specifically, we propose to validate the usefulness of the existing tool support that we provide: (1) global and (2) local performance-influence models, and (3) the additional tools that we develop. The study will help us validate how useful the information provided by the tools is for developers to understand *which, where, and how* options affect the performance of a system. Specifically, we seek to answer the following research questions:

RQ2.1: To what extent do global performance-influence models help developers understand the influence of configuration options on the performance of a system?

RQ2.2: To what extent do local performance-influence models help developers locate and understand the influence of configuration options on the performance of methods?

RQ2.3: To what extent does additional tool support help developers understand how options influence the performance of hotspots?

Study design. We propose to conduct the same performance debugging study that we conduct in our exploratory studies in Sec. 4.2. However, we will provide participants with global and local performance-influence models and any additional tool support, and encourage them to use the tools to debug the performance of the system.

We propose to *not* have a control group in this study. Despite some time to collect the information with our tools, the tools automatically provide developers with information to debug the performance of configurable systems. By contrast, developers without tools *interactively* collect that information (e.g., running and profiling the system with multiple configurations), which our exploratory studies have shown that take a significant amount of time. Additionally, all participants struggled to understand how options affect the performance of hotspots in the system, even after we told them the options that affect the performance and the hotspots that were affected. Hence our exploratory studies have shown that without tools and infor-

mation, debugging takes a long time, while in this study, we seek to show that participants who have tools and information available to them, up front, can solve the task.

Participant recruitment. Similarly to our exploratory studies, we propose to conduct the study with developers and researchers with various levels of experience in performance analysis and debugging, and working with configurable systems. We expect to recruit at least 10 researchers/developers working in academia.

Analysis. Similarly to our exploratory studies, we propose to analyze the studies using quantitative and standard qualitative research methods [Saldaña, 2015; Schreier, 2012] used in related research [LaToza et al., 2007; LaToza and Myers, 2010; Lawrance et al., 2013; Scaffidi et al., 2011]. We propose to transcribe and code the studies to identify topics and trends in the process that developers follow and the information that they need.

Proposed work: We propose to conduct and analyze a user study to validate the usefulness of the tools that we present in this thesis to help developers debug the performance of configurable systems, namely (1) global and (2) local performance-influence models, and (3) the additional tools that we develop.

Thesis contribution: We propose to demonstrate that global and local performance-influence models, and additional tool support that we propose to develop are useful for developers in the process of debugging the performance of configurable systems.

4.4.1 Proposed Optional Work: Study 3: Validate Tool Support in the Field

To further validate the usefulness of the tools that we present in the thesis to help developers debug real bug reports and real systems, we propose, *depending on availability and opportunity*, to evaluate our tools in the field with practitioners working in industry or open-source communities. There are several study designs that we could conduct.

Ideally, practitioners will independently use our tools to debug the performance of their own configurable systems. In this design, we will instrument the tools to collect information of how developers use the tools to determine how useful the tools are. We could compliment the study with surveys or interviews to further understand how developers used the tools when debugging the performance of their systems. This study design would take a significant time to perform and we would lose control of how practitioners use our tools, but the design would help us demonstrate that our tools can be independently used to debug the performance of real configurable systems. However, conducting such a study would require a significant amount of engineering effort for developers to independently use the tools, making it not the best viable option.

Alternatively, we could work with a few practitioners to help them debug a bug report in their own configurable systems. In this design, we would help practitioners to set up and

use our tools while we observe them debugging their own systems. Afterwards, we would interview the practitioners to get their perspectives on the usefulness of the tools. This study design, however, would be biased, as we will be interacting with practitioners as they use our tools, but the design would help us demonstrate that our tools are useful in real systems and real bug reports.

Another option is to conduct case studies, where we use the tools to debug open-source configurable systems. In this design, we would select bug reports from mailing lists or issue trackers and debug the systems ourselves. Subsequently, we will respond to the bug reports indicating any findings, misconfigurations, or bug fixes. While this study design does not involve practitioners using our tools, the design would help us demonstrate that developers who are unfamiliar with a system can use our tools to debug the performance of configurable systems.

Depending on availability and opportunity, we propose to conduct one of the study design options outlined.

Proposed work: Depending on availability and opportunity, we propose to conduct a user study to validate the the usefulness of the tools that we present in this thesis with practitioners working in industry or open-source communities.

4.5 Summary

In this chapter, we explored the usefulness, and limitations, of local performance-influence models to understand how options affect the performance of a system in the implementation. We then proposed to conduct two user studies to explore the process that developers follow and the information needs that they have when debugging the performance of configurable systems at different stages of the debugging process. Finally, we proposed to develop and validate additional tool support to help developers in the process of debugging the performance of configurable systems.

Chapter 5

Research Plan

In this chapter, we summarize the remaining steps and estimate the time to complete the thesis.

At the time of writing, the work presented in Chapter 3 about ConfigCrusher is completed and published [Velez et al., 2020a]. Additionally, the work presented in Chapter 3 about Comprex is completed [Velez et al., 2020b], but still under review. We estimate the time to polish the writing for the conference submission to be **2 weeks**. To polish the evaluation of ConfigCrusher and Comprex, we propose to (1) evaluate both prototypes against the same state of the art approaches and (2) evaluate Comprexwith all subject systems. We estimate time to set up and run the experiments to be **2 weeks**.

The major missing component of the proposed thesis is providing and validating additional tool support to debug the performance of configurable systems, discussed in Chapter 4. Th remaining steps are:

- Conduct a user study to explore the process that developers follow and the information needs that they have when debugging the performance of configurable systems at different stages of the debugging process (Study 1.1 and Study 1.2). We have already conducted 14 studies with researches/developers working in academia and expect to recruit at least 5 developers working in industry. We estimate the time to recruit and conduct the study with the remaining participants to be **1 month**.

- Quantitatively and Qualitatively analyze the studies to identify the process and information needs of developers when debugging the performance of configurable systems. We estimate the time to complete this analysis to be **1 month**.

- Design and develop new tool support, based on the findings of our user studies, to help developers understand how options affect, in the implementation, the performance of configurable systems. We estimate the time to implement our tool to be **6 months**.

- Conduct a user study to validate the tools presented in this thesis to help developers debug the performance of configurable systems (Study 2); (1) global and (2) local performance-influence models, and (3) the additional tools that we develop. We estimate the time to recruit and conduct the study to be **2 months**.

- Finally, we reserve another **1 month** to write and submit the work to a major conference.

Finally, we reserve **2 months** for writing and defending the proposed thesis. In summary, we expect to finish the proposed thesis in **13 months**.

5.1 Risks

Some potential risks of the remaining work of this thesis are:

- **We cannot recruit experienced developers in industry for Study 1.1 and Study 1.2.** There is a possibility that we might not be able to recruit developers in industry who have experience debugging performance on configurable systems. To mitigate this risk, we plan to contact people in our professional network to direct us to potential participants. After conducting the studies with those participants, we will ask them to point us to additional people to gather more participants. As a last resort, we could interview additional researchers in academia to potentially obtain additional perspectives.
- **We cannot validate the tools presented in this thesis in the field (Proposed optional Study 3).** There is a possibility that we cannot deploy our tools in the field to validate their usefulness to debug real programs and real bug reports. We discussed a few options in Chapter 4 to mitigate this risk, such as helping developers to use our tools to debug their systems or performing case studies where we debug open-source systems.

5.2 Final dissertation outline

We structured this proposal document to highlight the work that we have already completed and the work that we propose to conduct to finish the proposed thesis. We propose the outline of the final dissertation document to be:

- Introduction
- Information needs for Understanding and Debugging the Performance of Configurable Systems
  - Exploratory study of the process that developers follow and information needs that they have when debugging the performance of configurable systems (Study 1.1 and Study 1.2).
- State of the Art on Performance Analysis of Configurable Systems
- Modeling Performance in Configurable Systems
  - Key insights for efficient and accurate performance analysis of configurable systems

  - Components for modeling the performance of configurable systems
  - ConfigCrusher
  - Comprex
  - Evaluation of Prototypes
  - Validation study of the usefulness of global and local performance-influence models for debugging the performance in configurable systems (part of Study 2).

- Debugging Performance in Configurable Systems
  - Comparing performance profiles between two configurations.
  - Tracing how options are propagated through a system.
  - Validation study of the usefulness of the above tool support for debugging the performance in configurable systems (part of Study 2).

- Conclusion and Future Work

Bibliography

2019. *JProfiler 10.* Retrieved December 10, 2019 from https://www.ej-technologies.com/products/jprofiler/overview.html Page 14, Page 17, Page 30

2020. *VisualVM.* Retrieved November 24, 2020 from https://visualvm.github.io/ Page 14, Page 17

Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic program slicing. *ACM SIGPlan Notices* 25, 6 (1990), 246–256. Page 13, Page 16

Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. 2016. IncLing: Efficient Product-line Testing Using Incremental Pairwise Sampling. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)* (Amsterdam, Netherlands). ACM, New York, NY, USA, 144–155. Page 10, Page 25

Mohammad Mejbah ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. 2017. SyncPerf: Categorizing, Detecting, and Diagnosing Synchronization Performance Bugs. In *Proc. European Conference on Computer Systems (EuroSys)* (Belgrade, Serbia). ACM, New York, NY, USA, 298–313. Page 14, Page 17

David Andrzejewski, Anne Mulhern, Ben Liblit, and Xiaojin Zhu. 2007. Statistical Debugging Using Latent Topic Models. In *Proc. European Conf. Machine Learning* (Warsaw, Poland). Springer-Verlag, Berlin, Heidelberg, 6–17. Page 16

Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer-Verlag, Berlin/Heidelberg, Germany. Page 1, Page 7, Page 8

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. Conf. Programming Language Design and Implementation (PLDI)* (Edinburgh, UK). ACM, New York, NY, USA, 259–269. Page 13, Page 26, Page 27, Page 32

Mona Attariyan and Jason Flinn. 2010. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proc. Conf. Operating Systems Design and Implementation (OSDI)* (Vancouver, BC, Canada). USENIX Association, Berkeley, CA, USA, 237–250. Page 16, Page 17

Thomas H. Austin and Cormac Flanagan. 2009. Efficient Purely-dynamic Information Flow Analysis. In *Proc. Workshop Programming Languages and Analysis for Security (PLAS)* (Dublin, Ireland). ACM, New York, NY, USA, 113–124. Page 13

Thomas H. Austin and Cormac Flanagan. 2012. Multiple Facets for Dynamic Information Flow. In *Proc. Symp. Principles of Programming Languages (POPL)* (Philadelphia, PA, USA). ACM, New York, NY, USA, 165–178. Page 13

Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining Apps for Abnormal Usage of Sensitive Data. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Florence, Italy). IEEE Press, Piscataway, NJ, USA, 426–436. Page 32

Clark Barrett and Cesare Tinelli. 2018. Satisfiability modulo theories. In *Handbook of Model Checking*. Springer, 305–343. Page 13

Steffen Becker, Heiko Koziolek, and Ralf Reussner. 2009. The Palladio Component Model for Model-driven Performance Prediction. *J. Syst. Softw.* 82, 1 (Jan. 2009), 3–22. Page 8, Page 9

Farnaz Behrang, Myra B. Cohen, and Alessandro Orso. 2015. Users Beware: Preference Inconsistencies Ahead. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)* (Bergamo, Italy). ACM, New York, NY, USA, 295–306. Page 1, Page 8

Jonathan Bell and Gail Kaiser. 2014. Phosphor: Illuminating Dynamic Data Flow in Commodity Jvms. *SIGPLAN Notices* 49, 10 (Oct. 2014), 83–101. Page 13, Page 27, Page 28

Eric Bodden. 2018. Self-adaptive Static Analysis. In *Proc. Int'l Conf. Software Engineering (ICSE): New Ideas and Emerging Results* (Gothenburg, Sweden). ACM, New York, NY, USA, 45–48. Page 32

James Bornholt and Emina Torlak. 2018. Finding Code That Explodes Under Symbolic Evaluation. *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* 2, Article 149 (Oct. 2018), 26 pages. https://doi.org/10.1145/3276519 Page 13, Page 14, Page 17

Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. 2010. Information Needs in Bug Reports: Improving Cooperation between Developers and Users. In *Proc. Conf. Computer Supported Cooperative Work (CSCW)* (Savannah, GA, USA). ACM, New York, NY, USA, 301–310. Page 16

Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. 2013. Interactive Record/Replay for Web Application Debugging. In *Proc. Symposium User Interface Software and Technology (UIST)* (St. Andrews, Scotland, United Kingdom). ACM, New York, NY, USA, 473–484. Page 16

Mikaela Cashman, Myra B. Cohen, Priya Ranjan, and Robert W. Cottingham. 2018. Navigating the Maze: The Impact of Configurability in Bioinformatics Software. In *Proc. Int'l Conf. Automated Software Engineering (ASE)* (Montpellier, France). ACM, New York, NY, USA, 757–767. Page 41

Pablo De Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. 2015. CERE: LLVM-Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization. *ACM Trans. Archit. Code Optim. (TACO)* 12, 1, Article 6 (April 2015), 24 pages. Page 17

Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. 2017. Detecting Missing Information in Bug Descriptions. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)* (Paderborn, Germany). ACM, New York, NY, USA, 396–407. Page 16

Shaiful Alam Chowdhury and Abram Hindle. 2016. GreenOracle: Estimating Software Energy Consumption with Energy Measurement Corpora. In *Proc. Int'l Conf. Mining Software Repositories* (Austin, TX, USA). ACM, New York, NY, USA, 49–60. Page 1, Page 9

Jürgen Cito, Philipp Leitner, Christian Bosshard, Markus Knecht, Genc Mazlami, and Harald C. Gall. 2018. PerformanceHat: Augmenting Source Code with Runtime Performance Traces in the IDE. In *Proc. Int'l Conf. Software Engineering: Companion Proceeedings* (Gothenburg, Sweden). ACM, New York, NY, USA, 41–44. Page 17

Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. 2018. *Model Checking*. MIT press. Page 13

Charlie Curtsinger and Emery D. Berger. 2016. COZ: Finding Code that Counts with Causal Profiling. In *USENIX Annual Technical Conference (ATC)*. USENIX Association, Denver, CO, USA. Page 14, Page 17

Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. 2017. Just-in-time Static Analysis. In *Proc. Int'l Symp. Software Testing and Analysis (ISSTA)* (Santa Barbara, CA, USA). ACM, New York, NY, USA, 307–317. Page 32

Z. Dong, A. Andrzejak, D. Lo, and D. Costa. 2016. ORPLocator: Identifying Read Points of Configuration Options via Static Analysis. In *Proc. Int'l Symposium Software Reliability Engineering (ISSRE)*. 185–195. Page 14, Page 17

Finale Doshi-Velez and Been Kim. 2017. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608* (2017). Page 34

N. Esfahani, A. Elkhodary, and S. Malek. 2013. A Learning-Based Framework for Engineering Feature-Oriented Self-Adaptive Software Systems. *IEEE Transactions on Software Engineering* 39, 11 (Nov 2013), 1467–1493. Page 8, Page 9

Erol Gelenbe and Yves Caseau. 2015. The Impact of Information Technology on Energy Consumption and Carbon Emissions. *Ubiquity* 2015, June, Article 1 (June 2015), 15 pages. Page 1, Page 8

Alexander Grebhahn, Norbert Siegmund, and Sven Apel. 2019. Predicting Performance of Software Configurations: There is no Silver Bullet. arXiv:1911.12643 [cs.SE] Page 4, Page 9, Page 10, Page 11, Page 12, Page 34

Mark Grechanik, Chen Fu, and Qing Xie. 2012. Automatically Finding Performance Problems with Feedback-Directed Learning Software Testing. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Zurich, Switzerland). IEEE Press, 156–166. Page 14, Page 17

Brendan Gregg. 2016. The Flame Graph. *Commun. ACM* 59, 6 (May 2016), 48–57. Page 17

Jiaping Gui, Ding Li, Mian Wan, and William G. J. Halfond. 2016. Lightweight Measurement and Estimation of Mobile Ad Energy Consumption. In *Proc. Int'l Workshop Green and Sustainable Software (GREENS)* (Austin, TX, USA). ACM, New York, NY, USA, 1–7. Page 1, Page 9

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119. Page 13

Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wąsowski. 2013. Variability-aware performance prediction: A statistical learning approach. In *Proc. Int'l Conf. Automated Software Engineering (ASE)* (Silicon Valley, CA, USA). IEEE Computer Society, ACM, New York, NY, USA, 301–311. Page 9, Page 10, Page 11, Page 34

Jianmei Guo, Dingyu Yang, N. Siegmund, S. Apel, Atrisha Sarkar, Pavel Valov, K. Czarnecki, A. Wasowski, and H. Yu. 2017. Data-efficient performance learning for configurable systems. *Empirical Software Engineering* 23 (2017), 1826–1867. Page 10, Page 11, Page 34

Huong Ha and Hongyu Zhang. 2019. DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Montreal, Quebec, Canada). IEEE Press, 1095–1106. Page 4, Page 9, Page 10, Page 11

H. Ha and H. Zhang. 2019. Performance-Influence Model for Highly Configurable Software with Fourier Learning and Lasso Regression. In *Int'l Conf. Software Maintenance and Evolution (ICSME)*. 470–480. Page 4, Page 9, Page 10, Page 11

Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. 2018. Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. *Empirical Software Engineering* (July 2018). Page 1, Page 8, Page 10

Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. 2012. Performance Debugging in the Large via Mining Millions of Stack Traces. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Zurich, Switzerland). IEEE Press, Piscataway, NJ, USA, 145–155. Page 14, Page 17

Xue Han and Tingting Yu. 2016. An Empirical Study on Performance Bugs for Highly Configurable Software Systems. In *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)* (Ciudad Real, Spain). ACM, New York, NY, USA, Article 23, 10 pages. Page 3, Page 16, Page 18, Page 24, Page 41

Xue Han, Tingting Yu, and David Lo. 2018. PerfLearner: Learning from Bug Reports to Understand and Generate Performance Test Frames. In *Proc. Int'l Conf. Automated Software Engineering (ASE)* (Montpellier, France). ACM, New York, NY, USA, 17–28. Page 18

Mor Harchol-Balter. 2013. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action* (1st ed.). Cambridge University Press, New York, NY, USA. Page 9

Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. 2016. Energy Profiles of Java Collections Classes. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Austin, TX, USA). ACM, New York, NY, USA. Page 1, Page 9

Haochen He, Zhouyang Jia, Shanshan Li, Erci Xu, Tingting Yu, Yue Yu, Ji Wang, and Xiangke Liao. 2020. CP-Detector: Using Configuration-related PerformanceProperties to Expose Performance Bugs. In *Proc. Int'l Conf. Automated Software Engineering (ASE).* Page 16, Page 18, Page 41

A. Hervieu, B. Baudry, and A. Gotlieb. 2011. PACOGEN: Automatic Generation of Pairwise Test Configurations from Feature Models. In *Int'l Symposium Software Reliability Engineering*. 120–129. Page 10, Page 25

Aymeric Hervieu, Dusica Marijan, Arnaud Gotlieb, and Benoit Baudry. 2016. Optimal Minimisation of Pairwise-covering Test Configurations Using Constraint Programming. *Information and Software Technology* 71 (March 2016), 129 – 146. Page 10, Page 25

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. Page 13

Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. 2011. Dynamic Knobs for Responsive Power-aware Computing. In *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Newport Beach, CA, USA). ACM, New York, NY, USA, 199–212. Page 14, Page 17

Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. 2012. A User Survey of Configuration Challenges in Linux and eCos. In *Proc. Workshop Variability Modeling of Software-Intensive Systems (VAMOS)* (Leipzig, Germany). ACM, New York, NY, USA, 149–155. Page 1, Page 8

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-based Optimization for General Algorithm Configuration. In *Proc. Int'l Conf. Learning and Intelligent Optimization* (Rome, Italy). Springer-Verlag, Berlin/Heidelberg, Germany, 507–523. Page 11

Reyhaneh Jabbarvand, Alireza Sadeghi, Joshua Garcia, Sam Malek, and Paul Ammann. 2015. EcoDroid: An Approach for Energy-based Ranking of Android Apps. In *Proc. Int'l Workshop Green and Sustainable Software (GREENS)* (Florence, Italy). IEEE Press, Piscataway, NJ, USA, 8–14. Page 1, Page 9

Pooyan Jamshidi and Giuliano Casale. 2016. An Uncertainty-Aware Approach to Optimal Configuration of Stream Processing Systems. In *Int'l Symp.Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)* (London, UK). 39–48. Page 11

Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. 2017a. Transfer Learning for Performance Modeling of Configurable Systems: An Exploratory Analysis. In *Proc. Int'l Conf. Automated Software Engineering (ASE)* (Urbana-Champaign, IL, USA). ACM, New York, NY, USA, 13. Page 9

Pooyan Jamshidi, Miguel Velez, Christian Kästner, and Norbert Siegmund. 2018. Learning to Sample: Exploiting Similarities Across Environments to Learn Performance Models for Configurable Systems. In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)* (Lake Buena Vista, FL, USA). ACM, New York, NY, USA, 71–82. Page 9, Page 11

Pooyan Jamshidi, Miguel Velez, Christian Kästner, Norbert Siegmund, and Prasad Kawthekar. 2017b. Transfer Learning for Improving Model Predictions in Highly Configurable Software. In *Proc. Int'l Symp. Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* (Buenos Aires, Argentina). IEEE Computer Society, Los Alamitos, CA, USA, 31–41. Page 9, Page 10, Page 11

Dongpu Jin, Xiao Qu, Myra B. Cohen, and Brian Robinson. 2014. Configurations Everywhere: Implications for Testing and Debugging in Practice. In *Companion Proc. Int'l Conf. Software Engineering* (Hyderabad, India). ACM, New York, NY, USA, 215–224. Page 1, Page 8

Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-world Performance Bugs. In *Proc. Conf. Programming Language Design and Implementation (PLDI)* (Beijing, China). ACM, New York, NY, USA, 77–88. Page 14, Page 16, Page 17, Page 18, Page 24

Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. 2011. Catch Me If You Can: Performance Bug Detection in the Wild. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* (Portland, Oregon, USA). ACM, New York, NY, USA, 155–170. Page 14, Page 16, Page 17

C. Kaltenecker, A. Grebhahn, N. Siegmund, and S. Apel. 2020. The Interplay of Sampling and Machine Learning for Software Performance Prediction. *IEEE Software* (2020). Page 4, Page 9, Page 10, Page 11, Page 12, Page 19

Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. 2019. Distance-Based Sampling of Software Configuration Spaces. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press. Page 10, Page 12

Eva Kern, Markus Dick, Timo Johann, and Stefan Naumann. 2011. *Green Software and Green IT: An End Users Perspective.* Springer Berlin Heidelberg, Berlin, Heidelberg, 199–211. Page 1, Page 9

Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don Batory, Sabrina Souto, Paulo Barros, and Marcelo d'Amorim. 2013. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)* (Saint Petersburg, Russia). ACM, New York, NY, USA, 257–267. Page 17

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. Page 13, Page 16

Andrew J. Ko and Brad A. Myers. 2004. Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior. In *Proc. Conf Human Factors in Computing Systems (CHI)* (Vienna, Austria). ACM, New York, NY, USA, 8. Page 13, Page 14, Page 16

Andrew J. Ko and Brad A. Myers. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions About Program Behavior. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Leipzig, Germany). ACM, New York, NY, USA, 301–310. Page 16

Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Trans. Softw. Eng.* 32, 12 (Dec. 2006), 971–987. Page 16

Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, Alexander Grebhahn, and Sven Apel. 2018. Tradeoffs in modeling performance of highly configurable software systems. *Software and System Modeling (SoSyM)* (08 Feb. 2018). Page 9, Page 10, Page 12, Page 19, Page 35

Bogdan Korel and Janusz Laski. 1988. Dynamic program slicing. *Information processing letters* 29, 3 (1988), 155–163. Page 13, Page 16

Samuel Kounev. 2006. Performance modeling and evaluation of distributed component-based systems using queueing petri nets. *IEEE Transactions on Software Engineering* 32, 7 (2006), 486–502. Page 9

D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. 2013. *Introduction to Combinatorial Testing* (1st ed.). Chapman & Hall/CRC. Page 25

Fabian Lange. 2011. *Measure Java Performance – Sampling or Instrumentation?* Page 27

Thomas D. LaToza, David Garlan, James D. Herbsleb, and Brad A. Myers. 2007. Program Comprehension as Fact Finding. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)* (Dubrovnik, Croatia). ACM, New York, NY, USA, 361–370. Page 42, Page 47

Thomas D. LaToza and Brad A. Myers. 2010. Hard-to-Answer Questions about Code. In *Evaluation and Usability of Programming Languages and Tools* (Reno, NV, USA). ACM, New York, NY, USA, Article 8, 6 pages. Page 16, Page 42, Page 47

T. D. LaToza and B. A. Myers. 2011. Visualizing call graphs. In *Symposium Visual Languages and Human-Centric Computing (VL/HCC)*. 117–124. Page 13, Page 16

Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D. Fleming. 2013. How Programmers Debug, Revisited: An Information Foraging Theory Perspective. *IEEE Trans. Softw. Eng. (TSE)* 39, 2 (Feb. 2013), 197–215. Page 16, Page 42, Page 47

Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Zurich, Switzerland). IEEE Press, Piscataway, NJ, USA, 3–13. Page 16

J. Lerch, J. Späth, E. Bodden, and M. Mezini. 2015. Access-Path Abstraction: Scaling Field-Sensitive Data-Flow Analysis with Unbounded Access Paths (T). In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. IEEE Computer Society, Washington, DC, USA, 619–629. Page 32

Chi Li, Shu Wang, Henry Hoffmann, and Shan Lu. 2020. Statically Inferring Performance Properties of Software Configurations. In *Proc. European Conf. Computer Systems (EuroSys)* (Heraklion, Greece). ACM, New York, NY, USA, Article 10, 16 pages. Page 14, Page 15, Page 18

Ding Li, Yingjun Lyu, Jiaping Gui, and William G.J. Halfond. 2016. Automated Energy Optimization of HTTP Requests for Mobile Applications. ACM, New York, NY, USA. Page 1, Page 9, Page 16

Max Lillack, Christian Kästner, and Eric Bodden. 2018. Tracking Load-time Configuration Options. *IEEE Transactions on Software Engineering* 44, 12 (12 2018), 1269–1291. Page 13, Page 14, Page 17

Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 1013–1024. Page 14, Page 17

Scott M Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 4765–4774. http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf Page 12, Page 34

Haroon Malik, Peng Zhao, and Michael Godfrey. 2015. Going Green: An Exploratory Analysis of Energy-Related Questions. In *Proc. Int'l Conf. Mining Software Repositories* (Florence, Italy). IEEE Press, 418–421. Page 1, Page 9

Irene Manotas, Christian Bird, Rui Zhang, David Shepherd, Ciera Jaspan, Caitlin Sadowski, Lori Pollock, and James Clause. 2016. An Empirical Study of Practitioners' Perspectives on Green Software Engineering. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Austin, TX, USA). ACM, New York, NY, USA, 237–248. Page 1, Page 9

Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Austin, TX, USA). ACM, New York, NY, USA, 643–654. Page 10, Page 11, Page 16, Page 33, Page 34

Jens Meinicke, Chu-Pan Wong, Christian Kästner, and Gunter Saake. 2018. Understanding differences among executions with variational traces. *arXiv preprint arXiv:1807.03837* (2018). Page 13, Page 16, Page 17

Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On Essential Configuration Complexity: Measuring Interactions in Highly-configurable Systems. In *Proc. Int'l Conf. Automated Software Engineering (ASE)* (Singapore, Singapore). ACM, New York, NY, USA, 483–494. Page 13, Page 14

Jean Melo, Claus Brabrand, and Andrzej Wąsowski. 2016. How Does the Degree of Variability Affect Bug Finding?. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Austin, TX, USA). ACM, New York, NY, USA, 679–690. Page 1, Page 8

Jean Melo, Fabricio Batista Narcizo, Dan Witzner Hansen, Claus Brabrand, and Andrzej Wasowski. 2017. Variability through the Eyes of the Programmer. In *Proc. Int'l Conference Program Comprehension (ICPC)* (Buenos Aires, Argentina). IEEE Press, 34–44. Page 1, Page 8

Christoph Molnar. 2019. *Interpretable Machine Learning*. https://christophm.github.io/interpretable-ml-book/. Page 11, Page 12, Page 19, Page 34

Douglas C. Montgomery. 2006. *Design and Analysis of Experiments*. John Wiley & Sons. Page 10

Daniel-Jesus Munoz. 2017. Achieving energy efficiency using a software product line approach. In *Proc. Int'l Conf. Systems and Software Product Line*. 131–138. Page 1, Page 9

Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2017. Using Bad Learners to Find Good Configurations. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 257–267. Page 3, Page 11

Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. Conf. Programming Language Design and Implementation (PLDI)* (San Diego, CA, USA). ACM, New York, NY, USA, 89–100. Page 14, Page 17

James Newsome and Dawn Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. Page 13

Thanhvu Nguyen, Thanhvu Koc, Javran Cheng, Jeffrey S. Foster, and Adam A. Porter. 2016. iGen Dynamic Interaction Inference for Configurable Software. In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)* (Seattle, WA, USA). IEEE Computer Society, Los Alamitos, CA, USA. Page 14

Changhai Nie and Hareton Leung. 2011. A Survey of Combinatorial Testing. *ACM Comput. Surv. (CSUR)* 43, 2, Article 11 (Feb. 2011), 29 pages. Page 10, Page 11

Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2010. *Principles of Program Analysis.* Springer Publishing Company, Incorporated. Page 13

Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and Fixing Performance Problems That Have Non-intrusive Fixes. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Florence, Italy). IEEE Press, Piscataway, NJ, USA, 902–912. Page 14, Page 17, Page 18, Page 24

Adrian Nistor, Tian Jiang, and Lin Tan. 2013a. Discovering, Reporting, and Fixing Performance Bugs. In *Proc. Int'l Conf. Mining Software Repositories* (San Francisco, CA, USA). IEEE Press, 237–246. Page 3, Page 18, Page 24

Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013b. Toddler: Detecting Performance Problems via Similar Memory-access Patterns. In *Proc. Int'l Conf. Software Engineering (ICSE)* (San Francisco, CA, USA). IEEE Press, Piscataway, NJ, USA, 562–571. Page 14, Page 17

Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding Near-optimal Configurations in Product Lines by Random Sampling. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)* (Paderborn, Germany). ACM, New York, NY, USA, 61–71. Page 3, Page 11

Rafael Olaechea, Derek Rayside, Jianmei Guo, and Krzysztof Czarnecki. 2014. Comparison of Exact and Approximate Multi-objective Optimization for Software Product Lines. In *Proc. Int'l Software Product Line Conference (SPLC)* (Florence, Italy). ACM, New York, NY, USA, 92–101. Page 11

J. Park, M. Kim, B. Ray, and D. Bae. 2012. An empirical study of supplementary bug fixes. In *Proc. Int'l Conf. Mining Software Repositories* (Zurich, Switzerland). 40–49. Page 16, Page 18

Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *Proc. Int'l Symp. Software Testing and Analysis (ISSTA)* (Toronto, Canada). ACM, New York, NY, USA, 199–209. Page 16

Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do Android Taint Analysis Tools Keep Their Promises?. In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)* (Lake

Buena Vista, FL, USA). ACM, New York, NY, USA, 331–341. https://doi.org/10.1145/3236024.3236029 Page 32

Rui Pereira, Marco Couto, João Saraiva, Jácome Cunha, and João Paulo Fernandes. 2016. The Influence of the Java Collection Framework on Overall Energy Consumption. In *Proc. Int'l Workshop Green and Sustainable Software (GREENS)* (Austin, TX, USA). ACM, 15–21. Page 1, Page 9

Gustavo Pinto and Fernando Castor. 2017. Energy Efficiency: A New Concern for Application Software Developers. *Commun. ACM* 60, 12 (Nov. 2017), 68–75. Page 1, Page 9

Guillaume Pothier, Éric Tanter, and José Piquer. 2007. Scalable Omniscient Debugging. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* (Montreal, Quebec, Canada). ACM, New York, NY, USA, 535–552. Page 16

Lina Qiu, Yingying Wang, and Julia Rubin. 2018. Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe. In *Proc. Int'l Symp. Software Testing and Analysis (ISSTA)* (Amsterdam, Netherlands). ACM, New York, NY, USA, 176–186. Page 32

Ariel Rabkin and Randy Katz. 2011. Static Extraction of Program Configuration Options. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Waikiki, Honolulu, HI, USA). ACM, New York, NY, USA, 131–140. Page 14, Page 17, Page 41

Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. 2010. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, New York, NY, USA, 445–454. Page 13, Page 14

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proc. Int'l Conf. Knowledge Discovery and Data Mining (KDD)* (San Francisco, CA, USA). ACM, New York, NY, USA, 1135–1144. Page 12, Page 34

Cynthia Rudin. 2019. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence* 5 (2019), 206–215. Page 12, Page 34, Page 35

Johnny Saldaña. 2015. *The coding manual for qualitative researchers*. Sage. Page 42, Page 47

Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. 2015. Cost-Efficient Sampling for Performance Prediction of Configurable Systems. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. IEEE Computer Society, Washington, DC, USA, 342–352. Page 4, Page 10, Page 11, Page 34

Chris Scaffidi, Scott Fleming, David Piorkowski, Margaret Burnett, Rachel Bellamy, and Joseph Lawrance. 2011. Unifying Software Engineering Methods and Tools: Principles and Patterns from Information Foraging. (05 2011). Page 16, Page 42, Page 47

Margrit Schreier. 2012. *Qualitative content analysis in practice*. Sage publications. Page 42, Page 47

Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proc. Symp. Security and Privacy (SP)* (Oakland, CA, USA). IEEE Computer Society, Washington, DC, USA, 317–331. Page 13

Koushik Sen. 2007. Concolic testing. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. 571–572. Page 13

G. Serazzri, G. Casale, M. Bertoli, G. Serazzri, G. Casale, and M. Bertoli. 2006. Java Modelling Tools: an Open Source Suite for Queueing Network Modelling andWorkload Analysis. In *Third International Conference on the Quantitative Evaluation of Systems - (QEST'06)*. 119–120. Page 9

Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-influence Models for Highly Configurable Systems. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)* (Bergamo, Italy). ACM, New York, NY, USA, 284–294. Page 4, Page 9, Page 10, Page 12, Page 18, Page 33, Page 34

Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. 2012a. Predicting Performance via Automated Feature-interaction Detection. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Zurich, Switzerland). IEEE Press, Piscataway, NJ, USA, 167–177. Page 10, Page 12, Page 33, Page 34

Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. 2012b. SPL Conqueror: Toward Optimization of Non-functional Properties in Software Product Lines. *Software Quality Journal* 20, 3-4 (Sept. 2012), 487–517. Page 10, Page 12, Page 33, Page 34

Norbert Siegmund, Alexander von Rhein, and Sven Apel. 2013. Family-Based Performance Measurement. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)* (Indianapolis, IN, USA). ACM, New York, NY, USA, 95–104. Page 14, Page 15, Page 18, Page 24, Page 33

Linhai Song and Shan Lu. 2014. Statistical Debugging for Real-World Performance Problems. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* (Portland, OR, USA). ACM, New York, NY, USA, 561–578. Page 14, Page 17

Linhai Song and Shan Lu. 2017. Performance Diagnosis for Inefficient Loops. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Buenos Aires, Argentina). IEEE Press, Piscataway, NJ, USA, 370–380. Page 14, Page 16, Page 17

S. Souto and M. d'Amorim. 2018. Time-space efficient regression testing for configurable systems. *Journal of Systems and Software* (2018). Page 14, Page 17

Sabrina Souto, Marcelo d'Amorim, and Rohit Gheyi. 2017. Balancing Soundness and Efficiency for Practical Testing of Configurable Systems. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Buenos Aires, Argentina). IEEE Press, Piscataway, NJ, USA, 632–642. Page 17

Erik Štrumbelj and Igor Kononenko. 2014. Explaining prediction models and individual predictions with feature contributions. *Knowledge and information systems* 41, 3 (2014), 647–665. Page 12, Page 34

Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv. (CSUR)* 47, 1, Article 6 (June 2014), 45 pages. Page 14

John Toman and Dan Grossman. 2016a. Legato: An At-Most-Once Analysis with Applications to Dynamic Configuration Updates. In *European Conf. Object-Oriented Programming (ECOOP)*. Amsterdam, Netherlands. Page 13, Page 14, Page 17

John Toman and Dan Grossman. 2016b. Staccato: A Bug Finder for Dynamic Configuration Updates. In *European Conf. Object-Oriented Programming (ECOOP)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 24:1–24:25. Page 13, Page 14, Page 17

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proc. Conf. Centre for Advanced Studies on Collaborative Research (CASCON)* (Mississauga, Ontario, Canada). IBM Press, 13–. Page 13

Pavel Valov, Jean-Christophe Petkovich, Jianmei Guo, Sebastian Fischmeister, and Krzysztof Czarnecki. 2017. Transferring Performance Prediction Models Across Different Hardware Platforms. In *Proc. Int'l Conf. on Performance Engineering (ICPE)* (L'Aquila, Italy). ACM, New York, NY, USA, 39–50. Page 9

Miguel Velez, Pooyan Jamshidi, Florian Sattler, Norbert Siegmund, Sven Apel, and Christian Kästner. 2020a. ConfigCrusher: Towards White-Box Performance Analysis for Configurable Systems. *Autom Softw Eng* (2020). Page 7, Page 21, Page 27, Page 28, Page 31, Page 32, Page 39, Page 49

Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. 2020b. White-box Analysis over Machine Learning: Modeling Performance of Configurable Systems. In *Under review*. Page 7, Page 21, Page 27, Page 28, Page 31, Page 32, Page 37, Page 49

Nukala Viswanadham and Yadati Narahari. 2015. *Performance modeling of automated systems*. PHI Learning Pvt. Ltd. Page 9

Bo Wang, Leonardo Passos, Yingfei Xiong, Krzysztof Czarnecki, Haiyan Zhao, and Wei Zhang. 2013. SmartFixer: Fixing Software Configurations Based on Dynamic Priorities. In *Proc. Int'l Software Product Line Conference (SPLC)* (Tokyo, Japan). ACM, New York, NY, USA, 82–90. https://doi.org/10.1145/2491627.2491640 Page 14, Page 17

Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. 2018. Understanding and Auto-Adjusting Performance-Sensitive Configurations. In *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Williamsburg, VA, USA). ACM, New York, NY, USA, 154–168. Page 11, Page 12

Yan Wang, Hailong Zhang, and Atanas Rountev. 2016. On the Unsoundness of Static Analysis for Android GUIs. In *Proc. Int'l Workshop State Of the Art in Program Analysis (SOAP)* (Santa Barbara, CA, USA). ACM, New York, NY, USA, 18–23. Page 32

Mark Weiser. 1981. Program Slicing. In *Proc. Int'l Conf. Software Engineering (ICSE)* (San Diego, CA, USA). IEEE Press, Piscataway, NJ, USA, 439–449. Page 13, Page 16

Claas Wilke, Sebastian Richly, Sebastian Götz, Christian Piechnick, and Uwe Amann. 2013. Energy Consumption and Efficiency in Mobile Applications: A User Feedback Study. In *Proc. Int'l Conf. Green Computing and Communications*. IEEE Computer Society, 134–141. Page 1, Page 9, Page 16

Chu-Pan Wong, Jens Meinicke, Lukas Lazarek, and Christian Kästner. 2018. Faster Variational Execution with Transparent Bytecode Transformation. *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* 2, Article 117 (Oct. 2018), 30 pages. Page 13, Page 14, Page 17

Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes* 30, 2 (2005), 1–36. Page 13

Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, You Have Given Me Too Many Knobs!: Understanding and Dealing with Over-designed Configuration in System Software. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)* (Bergamo, Italy). ACM, New York, NY, USA, 307–319. Page 1, Page 8

Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proc. Conf.Operating Systems Design and Implementation (OSDI)* (Savannah, GA, USA). USENIX Association, Berkeley, CA, USA, 619–634. Page 14, Page 17

Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do Not Blame Users for Misconfigurations. In *Proc. Symp. Operating Systems Principles* (Farminton, PA, USA). ACM, New York, NY, USA, 244–259. Page 1, Page 8, Page 12, Page 41

Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How Do Fixes Become Bugs?. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)* (Szeged, Hungary). ACM, New York, NY, USA, 26–36. Page 18

Tingting Yu and Michael Pradel. 2016. SyncProf: Detecting, Localizing, and Optimizing Synchronization Bottlenecks. In *Proc. Int'l Symp. Software Testing and Analysis (ISSTA)* (Saarbrücken, Germany). ACM, New York, NY, USA, 389–400. Page 14, Page 17

Tingting Yu and Michael Pradel. 2018. Pinpointing and Repairing Performance Bottlenecks in Concurrent Programs. *Empirical Softw. Eng.* 23, 5 (Oct. 2018), 3034–3071. Page 14, Page 17

Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why? *SIGSOFT Softw. Eng. Notes* 24, 6 (Oct. 1999), 253–267. Page 16

Andreas Zeller. 2009. *Why programs fail: a guide to systematic debugging*. Elsevier. Page 16

C. Zhang, A. Hindle, and D. M. German. 2014. The Impact of User Choice on Energy Consumption. *IEEE Software* 31, 3 (2014), 69–75. https://doi.org/10.1109/MS.2014.27 Page 1, Page 9

Sai Zhang and Michael D. Ernst. 2013. Automated Diagnosis of Software Configuration Errors. In *Proc. Int'l Conf. Software Engineering (ICSE)* (San Francisco, CA, USA). IEEE Press, Piscataway, NJ, USA, 312–321. Page 17

Sai Zhang and Michael D. Ernst. 2014. Which Configuration Option Should I Change?. In *Proc. Int'l Conf. Software Engineering (ICSE)* (Hyderabad, India). ACM, New York, NY, USA, 152–163. Page 17

Sai Zhang and Michael D. Ernst. 2015. Proactive Detection of Inadequate Diagnostic Messages for Software Configuration Errors. In *Proc. Int'l Symp. Software Testing and Analysis (ISSTA)* (Baltimore, MD, USA). ACM, New York, NY, USA, 12–23. Page 17

Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *Proc. Symposium Cloud Computing (SoCC)* (Santa Clara, CA, USA). ACM, New York, NY, USA, 338–350. Page 11