Debugging the Performance of Configurable Software Systems: A Human-Centered White-box Approach

Miguel Velez

CMU-ISR-21-112

October 2021

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Christian Kästner (Chair)
Claire Le Goues
Rohan Padhye
Norbert Siegmund (Leipzig University)

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Software Engineering.*

This research was supported in part by the NSF (awards 1318808, 1552944, 1717022, and 2007202), AFRL, DARPA (FA8750-16-2-0042), NASA (RASPERRY-SI 80NSSC20K1720), the Software Engineering Institute, the German Federal Ministry of Education and Research (BMBF, 01IS19059A and 01IS18026B), and the German Research Foundation (DFG, AP 206/7-2, AP 206/11-1, SI 2171/2, SI 2171/3-1).

Keywords: Static Analysis, Dynamic Analysis, Performance Debugging, Performance-Influence Modeling, Configurable Software System, White-box Analysis

To the few that gave me a chance when most did not.

Abstract

Most of today's software systems are configurable. The flexibility to customize these systems, however, comes with the cost of increased complexity. Understanding how configuration options and their interactions affect performance, in terms of execution time, and often directly correlated energy consumption and operational costs, is challenging, due to the large configuration spaces of these systems. For this reason, developers often struggle to debug and maintain their systems when surprising performance behaviors occur.

While there are numerous performance and program debugging techniques that developers could use to debug their systems, there is limited empirical evidence of how useful the techniques are to help developers debug the performance of configurable software systems; the techniques typically solve a specific technical challenge that is usually evaluated in terms of accuracy, not usability. Hence, we could only, at best, speculate which techniques might support developers' needs to debug unexpected performance behaviors in configurable software systems.

In this dissertation, we take a human-centered approach to identify solutions to support developers' actual needs in the process of debugging the performance of configurable software systems. Specifically, we identify white-box analyses and techniques that can be tailored to provide relevant performance-behavior information for developers to understand how configuration options and their interactions cause performance issues.

In our human-centered research design, we first conduct an exploratory user study to identify the information needs that developers have when debugging the performance of configurable software systems. Afterwards, we identify the program analyses and techniques that can be tailored to support those needs. In this process, we note the limitations of existing performance-influence modeling techniques, and present and evaluate a white-box approach that overcomes those limitations. Afterwards, we describe how we design and implement information providers, tailoring the white-box analyses that we identified, to support developers' needs; namely, global and local performance-influence models, CPU profiling, and program slicing. Finally, we conduct two users studies to validate and confirm that our designed information providers support the needs that developers have and speed up the process of debugging the performance of complex configurable software systems.

The contributions in this dissertation help reduce the energy consumption and operational costs of running configurable software systems by providing developers with tool support to help them debug and maintain their systems.

Acknowledgments

This dissertation is a major milestone in a long journey, in which I have been blessed to have received the help and guidance of several individuals.

First of all, I would like to thank God (*Dios*) for helping and guiding me through all of the obstacles that I encountered throughout my Ph.D.

I would not have been able to conduct my research without the help, guidance, and mentorship from my advisor **Christian Kästner**. From Christian, I learned invaluable skills to become an independent researcher, critical thinker, technical writer, public speaker, and careful reviewer. These skills will be valuable for the rest of my career. I was fortunate to have met Christian at the right time. In fact, I never applied to the Software Engineering program at CMU. I only clicked a box in the CS application that said "Click here if you want other departments to review your application." I can honestly say that I am extremly happy to have clicked that box.

I want to thank my collaborator and committee member **Norbert Siegmund**, who has been helping me in all aspects of my research from the very beginning. From him, I learned a lot about analyzing and modeling the performance of configurable software systems. Specially, I want to thank Norbert for his work on "Family-Based Performance Measurement". This was the first white-box approach that analyzed the influence of configuration options on the performance of software systems. This paper motivated all of the work that I did during my Ph.D.

I would like to thank my collaborator **Sven Apel**. He was also an author of "Family-Based Performance Measurement", and we meet weekly for several years to help me in my research. He also reviewed my papers countless times, and gave me actionable feedback for my papers to be, in his own words, "perfect".

I would like to thank my collaborator **Pooyan Jamshidi**. It was fortunate that we both started at CMU at the same time (Pooyan as a Postdoc), and we both focused on analyzing the performance of configurable software systems. Although, Pooyan used machine learning techniques and I focused on program analyses, we always had thoughtful discussions on the techniques that we used in each other's work.

I want to thank my committee members **Claire Le Goues** and **Rohan Padhye** for their time and feedback. Their input on this dissertation is invaluable. Specially, I want to thank Claire for her feedback on our ICSE'21 paper, and Rohan for his feedback on our ICSE'21 paper and the design of our confirmatory user study.

I would like to thank **Chu-Pan Wong** and **Jens Meinicke** for the thoughtful discussions on how to analyze variabilty and performance in software systems. After this dissertation, I think I can finally answer the question "What is a performance bug?" I also enjoyed our arguments on who gave the best silly name to our research tools. There is no question that "ConfigCrusher", "Comprex", and "GLIMPS" are much cooler and original names than "VarexJ", "VarexC", and, especially, "Varviz".

I would like to thank my performance analysis collaborators **Johannes Dorn**, **Stefan Mühlbauer**, **Florian Sattler**, and **Max Weber**. I cannot wait to see the awesome research that you will contribute to our community.

I would like to thank the **FOSD 2017**, **2018**, **2019**, and **2021 meeting partici-**

pants for their feedback and suggestions. I enjoyed being part of this community.

I am extremely grateful for the help of **Katherine Hough** and **Jonathan Bell**, for maintaining Phosphor, **Max Lillack** for explaining how Lotrack works, and **Steven Artz** for fixing bugs in FlowDroid. Without their contributions and invaluable help, I would not have been able to complete some of the work in this dissertation.

I would like to thank **James Herbsleb**, for explaining how to conduct a Wizard of Oz experiment, **Thomas LaToza**, for describing how to analyze and code sessions for identifying information needs, and **Janet Siegmund** and **Bogdan Vasilescu**, for helping me on the design our confirmatory user study.

I would also like to thank the 31 anonymous participants of our user studies, who helped us in our research, despite being in the middle of a global pandemic.

I would like to thank my **research group**, **Gabriel Ferreira**, **Courtney Miller**, **Nadia Nahar**, **Chenyang Yang**, and **Shurui Zhou**, for all the knowledge, feedback, and learning process that we went through together.

I would like to thank my **officemates** and **colleages**, who made all of these years at CMU more enjoyable: **Maria Casimiro**, **Leo Chen**, **Kattiana Constantino**, **Tobias Dürschmid**, **Eduardo Figueiredo**, **Darya Melicher**, **Jean Melo**, **Larissa Rocha Soares**, **Ivan Ruchkin**, **Mauricio Soto**, **Ștefan Stănciulescu**, **Peter Story**, **Roykrong Sukkerd**, **Marat Valiev**, and **Jenna Wise**.

Many thanks to **Connie Herold**, **Jennifer Cooper**, **Helen Higgins**, **Jamie Lou Hagerty**, **Emanuel Bowes**, and **Ryan Johnson** for all the prompt assistance.

I would like to specially thank two of my professors at St. Thomas: **Patrick Jarvis**, my undergraduate advisor, for the invaluable and entertaining feedback on how to become a millionaire, and **Jason Sawin**, who introduced me to Software Engineering research and encouraged me to pursue my Ph.D.

I would like to specially thank **Armando Solar-Lezama**, my advisor when I was a research intern at MIT. I have been been told that his recommendation got me accepted to CMU, and I am sure that he helped me to get into other Ph.D. programs.

I would like to thank my managers at Google, **James Worcester**, **Yuan Zhang**, **Edwin Elia**, **Ali Anwar**, **Mark Wakabayashi**, **Jeremy Chua**, **Justin Bishop**, and **Tim Ford**, for helping me to take a break from doing research and mentoring me on how to become a better software engineer. After my internships, I always applied what I learned when implementing the technical components in this dissertation.

On a personal note, I would like to thank my wife, **Laura Nagel**, who has always loved me and has made countless sacrifices (e.g., moving from Minneapolis to Pittsburgh) for me to have an emotionally and financially stable environment to pursue my Ph.D. I would like to thank my parents, **Miguel Velez** and **Jenny Cevallos** for educating me and cultivating in me the passion to seek knowledge, perfection, and the truth throughout my entire life. I would like to thank the **ISR faculty** and **staff**, my **friends**, my **family**, and my **colleagues** for an incredible journey so far.

Finally, I would like to leave you with a quote that I read on a door at CMU that has motivated me throughout my Ph.D.: *"I never want to reach the point in my life where I've already done the most epic thing I will ever do" -Anonymous*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

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

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

Performance, in terms of execution time, and often directly correlated *energy consumption* and *operational costs*, is one of the most important quality attributes for developers, as well as users, of configurable software systems [Gelenbe and Caseau, 2015, Manotas et al., 2016, Pinto and Castor, 2017]. Developers want to design, implement, release, and maintain efficient configurable software systems that provide flexibility and high-quality user experience to attract new and retain 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]. Similarly, users want to run systems efficiently to reduce energy consumption and operational costs, but at the same time, with the functionality that satisfies their specific needs. Hence, users may need to make tradeoff decisions between functionality and operational costs [Jabbarvand et al., 2015, Kern et al., 2011, Munoz, 2017, Wilke et al., 2013, Zhang et al., 2014]. For developers, as well as for users, however, understanding how configuration options and their interactions affect the performance of software systems is challenging due to the large configuration spaces of these systems.

In this dissertation, we aim to help *developers* understand the impact that configuration options and their interactions have on the performance of configurable software systems. Our goal is to provide developers with relevant *configuration-related performance information* to help them *debug* performance issues in their systems.

Understanding how configuration options and their interactions affect the performance of software systems can also help users make conscious configuration decisions to run systems

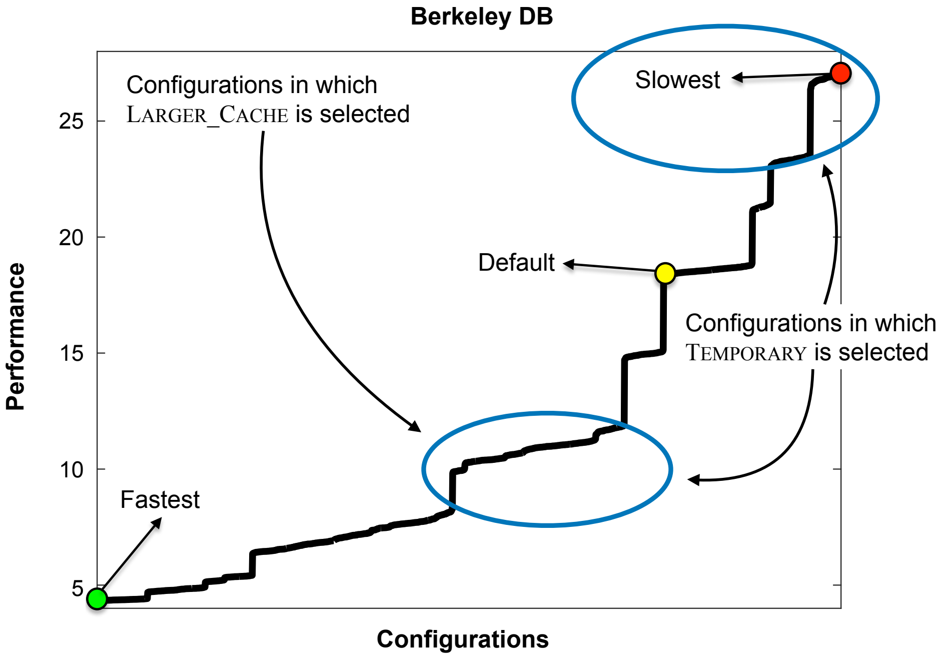


Figure 1.1: Ranked performance behavior, from fastest to slowest execution time, of 2000 randomly selected configurations in *Berkeley DB* when populating a database with 500K entries. Note the influence on the execution time of the interaction between the configuration option TEMPORARY and LARGER_CACHE. Selecting (i.e., setting to `true`) both configuration options decreases the execution time compared to the default configuration, while only selecting TEMPORARY increases the execution time.

efficiently.

1.1 Performance in Configurable Software Systems

To illustrate the challenge that developers, as well as users, have to understand how configuration options and their interactions affect performance in large configuration spaces, we use a real scenario.

1.1.1 Example Scenario

Berkeley DB is an open-source embeddable database with hundreds of configuration options that affect the functionality of the database, its components, and the quality attributes of the system, including performance.[1] Figure 1.1 shows the ranked performance, from fastest to slowest execution time, of 2000 randomly selected configurations when populating a database with 500K entries. In this system, the configuration options not only drastically change the execution time from the fastest to the slowest configurations for the *same scenario* and *workload* (from about 5

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

seconds to about 27 seconds), but the configuration options also interact and produce complex performance behaviors (i.e., numerous steps in the figure), which prevents developers, as well as users, from easily understanding how configuration options and their interactions affect the performance of the system.

When a surprising *non-crashing* performance behavior occurs in configurable software systems, such as *Berkeley DB*, developers often spend a substantial amount of time diagnosing the system to either localize and fix a *performance bug*, determine that the system was *misconfigured*, or conclude that system is behaving as expected, but the user had a *different expectation* about what the performance of the system should be under a specific configuration [Breu et al., 2010, Chaparro et al., 2017, Han and Yu, 2016, Jin et al., 2012, Jovic et al., 2011, Krishna et al., 2020, Nistor et al., 2013a, Park et al., 2012, Parnin and Orso, 2011, Zeller, 2009]. For example, the slowest configuration in Figure 1.1 might appear to be performing unexpectedly; the configuration option TEMPORARY is selected (i.e., set to `true`), which should decrease the execution time as indicated in the documentation. To debug this potential performance issue, developers might compare changes in the slowest configuration with the default configuration. However, *Berkeley DB* has hundreds of configuration options, and the changes differ in several of them (in addition to TEMPORARY), requiring developers to identify how the changed configuration options interact with each other, and with the unchanged configuration options, to produce the surprising performance behavior. Additionally, if developers narrow down the configuration options and interactions that are potentially producing the performance issue, they most likely need to *debug the implementation* to understand where and how the configuration options are being used and interact 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 possible debugging process of identifying potentially problematic configuration options and analyzing where and how they are used in the implementation to produce a surprising performance behavior is challenging; particularly, in large configuration spaces with complex performance behaviors (e.g., if we consider 50 configuration options for *Berkely DB*, the system would have 2^{50} configurations if we only consider two values for each configuration option. This value is similar to $56\times$ the average number of red blood cells in the human body!). Ideally, developers would *understand* how performance issues are related to configuration options and their interactions, to debug the issues and maintain efficient software. 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 to reduce energy consumption and operational costs.

Users also struggle to understand how configuration options and their interactions affect the performance of software systems with large configuration spaces [Apel et al., 2013, Hubaux et al., 2012, Siegmund et al., 2015, Xu et al., 2013, 2015]. For instance, users typically want to run configurable software systems, such as *Berkeley DB*, efficiently for their specific needs. To this end, users need to make informed tradeoff and configuration decisions between performance, functionality, and other quality attributes. However, users are often unaware of how configuration options affect the functionality and performance of a system. For this reason, users often resort to using the default configuration, resulting in executing their systems inefficiently, and increasing energy consumption and operational costs.

While 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], such strategies only aim to find the fastest configurations, and do not take into account the functionality that users need.

1.1.2 Problem Statement

When configuration-related performance issues occur, developers need to debug their systems to maintain efficient software, and to reduce the energy consumption and operation costs of running their systems. Debugging the performance of configurable software systems requires *understanding* how configuration options and their interactions *cause* performance issues. However, understanding this information becomes intractable as the configuration spaces of the systems increase and their performance behavior becomes more complex.

1.1.3 Dissertation Scope

The goal of this dissertation is to help *developers debug* the performance of configurable software systems. We use a human-centered approach [Farooqui et al., 2019, Myers et al., 2016] to identify and evaluate *solutions* for providing relevant performance-behavior information for developers to *understand* how configuration options *cause* performance issues.

Understanding how configuration options and their interactions affect the performance of software systems is also a concern of many areas of research. In what follows, we scope the concerns that we address and evaluate in this dissertation.

One area of research seeks to help *users* make informed 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]. While some of our solutions contribute to this research area, an empirical evaluation of the usefulness of our solutions for users is beyond the scope of this dissertation.

Another area of research concerns with accurately predicting the performance of individual configurations. For example, in scenarios when dynamically deciding during a robot's mission which configuration options to change to react to low-battery levels [Jamshidi et al., 2017, 2018, Wang et al., 2018, Zhu et al., 2017]. In this dissertation, we contribute to this research area by developing and evaluating solutions to predict the performance of configurable software systems.

Techniques to understand how configuration options affect performance have also been used to optimize the performance of configurable software systems [Guo et al., 2013, Nair et al., 2017, Oh et al., 2017, Zhu et al., 2017]. While some of our solutions could be used for this task, there are more targeted techniques for optimizing performance [Hutter et al., 2011, Jamshidi and Casale, 2016, Oh et al., 2017, Olaechea et al., 2014, Zhu et al., 2017]. Hence, we do not evaluate our solutions in terms of this optimization goal.

Table 1.1 summarizes the concerns that users and developers have in terms of performance in configurable software systems, as well as whether we make major or minor contributions that address those concerns in this dissertation.

Now that we have scoped this dissertation to help developers debug the performance of configurable software systems, we discuss this concern in more detail.

Table 1.1: Scope of the contributions in this dissertation for the concerns that users and developers have in terms of performance in configurable software systems.

	Developers	Users
Debugging	●	—
Prediction	●	●
Making informed configuration decisions	—	○
Optimization	—	○

●: Major contribution evaluated empirically.

○: Minor contribution without an empirical evaluation.

—: Stakeholder does not typically have this concern.

1.2 Debugging Performance in Configurable Software Systems

Developers often spend a substantial amount of time diagnosing a configurable software system to localize and fix a performance bug, or to determine that the system was misconfigured [Breu et al., 2010, Chaparro et al., 2017, Han and Yu, 2016, Jin et al., 2012, Jovic et al., 2011, Krishna et al., 2020, Nistor et al., 2013a, Park et al., 2012, Parnin and Orso, 2011, Zeller, 2009]. This struggle is quite common when maintaining configurable software systems. Some empirical studies find that 59% of performance issues are related to configuration errors, 88% of these issues require fixing the code [Han and Yu, 2016, Han et al., 2018], of which 61% take an average of 5 weeks to fix [Krishna et al., 2020], and that 50% of patches in open-source cloud systems and 30% of questions in forums are related to configurations [Wang et al., 2018]. Regardless of how developers find the root cause of the issue or misconfiguration, performance issues impair user experience, which often result in long execution times and increased energy consumption and operational costs [Han and Yu, 2016, He et al., 2020, Jin et al., 2012, Krishna et al., 2020, Li et al., 2016, Song and Lu, 2017, Wilke et al., 2013].

When performance issues occur, developers typically use profilers to identify the locations of performance bottlenecks [Castro et al., 2015, Cito et al., 2018, Curtsinger and Berger, 2016, Gregg, 2016, Yu and Pradel, 2018]. Unfortunately, locations where a system spends the most time executing are not necessarily the sign of a performance issue. Additionally, traditional profilers only indicate the locations of the *effect* of performance issues (i.e., where a system spends the most time executing) for one configuration at a time. Developers are left to inspect the code to analyze the *root cause* of the performance issues and to determine how the issues relate to configurations.

With an example scenario in Figure 1.2, we illustrate the kind of performance challenge that developers may face in configurable software systems and that we seek to support: A user executes a configuration in this system with 50 configuration options, which results in an *unexpected* $20\times$ slowdown. The *only visible effect* is the excessive execution time. While, in some situations, developers might be able to change some configuration options to work around the problem, users might not know which configuration options cause the problem, and may want to select certain configuration options to satisfy specific needs (e.g., enable encryption, use a specific transformation algorithm, or set a specific cache size). In these situations, developers need to determine whether the system has a potential *bug*, is *misconfigured*, or *works correctly*, but the user has a *different expectation* about what the performance of the system should be

```
class Main
  boolean commit; boolean sync;
  def main()
    trans = getOpt("TRANSACTIONS");
    dups = getOpt("DUPLICATES");
    Database db = new Database(trans, dups);
    init(db);
    new Cursor().put(this.commit, this.sync);
  def init(Database db) {
    ....
    this.commit = db.trans ? true : false;
    this.sync = db.dups ? true : false;
    ....


class Database
  boolean trans; boolean dups;
  def Database(boolean trans, boolean dups)
    this.trans = trans;
    this.dups = dups;


class Cursor {
  def put(boolean commit, boolean sync)
    if(commit)
      if(sync)
        synchronized(...)
          ....
```

What is the issue? (Effect)

Executing one configuration results in a 20x slowdown in this system with 50 configuration options

Why this issue occurs? (Cause)

Setting the configuration options `Transactions` and `Duplicates` to `true`, drastically increases the execution time of the method `Cursor.put`. `Transactions` sets the value of `commit` and `Duplicates` sets the value of `sync`. When the variables are `true`, the system synchronizes, which is *not* required when inserting duplicate data using transactions. The variables are initialized in `Main.init`, using the configuration options. The configuration options are passed through the `Database` object created in `Main.main`.

Figure 1.2: Artificial example contrasting effects and causes when debugging the performance of configurable software systems.

under the configuration that they execute. To determine the cause of the potential problematic performance behavior, developers would need to debug the system and, most likely, the *implementation* to identify which configuration options or interactions in this configuration are the *root cause* of the *unexpected performance behavior* (e.g., the system works as expected, setting a specific CACHE_SIZE results in a misconfiguration, or setting the options TRANSACTIONS and DUPLICATES to `true` results in a bug.).

When performance issues as in our example occur, there are numerous techniques that developers could use to determine whether there is a performance bug or the system was misconfigured. In addition to off-the-shelf profilers [JPR, 2019, VVM, 2020, Nethercote and Seward, 2007], developers could use more targeted profiling techniques [Alam et al., 2017, Castro et al., 2015, Curtsinger and Berger, 2016, Yu and Pradel, 2016, 2018], visualize performance behavior [Adamoli and Hauswirth, 2010, Bezemer et al., 2015, Cito et al., 2018, Gregg, 2016, Sandoval Alcocer et al., 2019, Trümper et al., 2013], search for inefficient coding patterns [Bornholt and Torlak, 2018, Liu et al., 2014, Nistor et al., 2013b, 2015, Song and Lu, 2017], use information-flow analyses [Li et al., 2020, Lillack et al., 2018, Meinicke et al., 2016, Toman and Grossman, 2016, Wong et al., 2018, Xu et al., 2016, Zhang and Ernst, 2014], or model the performance of the systems in terms of its configuration options and interactions [Ha and Zhang, 2019, Kolesnikov et al., 2018, Siegmund et al., 2015, Weber et al., 2021]. Likewise, developers could use established program debugging techniques, such as delta debugging [Zeller, 1999], program slicing [Agrawal and Horgan, 1990, Korel and Laski, 1988, Weiser, 1981], and statistical debugging [Andrzejewski et al., 2007, Song and Lu, 2014] for some part of the debugging process.

One reason why we cannot reliably suggest developers to use any of the above techniques is that there is *limited* empirical evidence of how useful the techniques are to help developers debug the performance of configurable software systems; the techniques typically solve a specific technical challenge that is usually evaluated in terms of accuracy, not usability [Parnin and Orso, 2011]. Hence, we could only, at best, speculate which techniques might support developers' needs to debug unexpected performance behaviors in configurable software systems.

1.3 Thesis

The main goal of this dissertation is to *reduce* the *energy consumption* and *operational costs* of running configurable software systems. The work in this dissertation contributes towards this goal by taking a human-centered approach [Farooqui et al., 2019, Myers et al., 2016] to *identify*, *design*, *implement*, and *evaluate white-box* solutions to support the needs that developers have in the process of debugging the performance of configurable software systems; particularly, in situations such as our example in Figure 1.2. Based on an *exploratory user study*, we identify that developers struggle to find relevant information to *understand* how configuration options and their interactions are used *in the implementation* to affect the performance of configurable software systems. Consequently, we identify, *tailor*, and evaluate *white-box* analyses to efficiently and accurately track this information; specifically, by *modeling* the performance of configurable software systems and *tracing* how configuration options affect the performance of software systems *in the implementation*.

> ***Thesis Statement****: Tailoring specific white-box analyses to track how configuration options influence the performance of code-level structures in configurable software systems helps developers to (1) efficiently build accurate and interpretable global and local performance-influence models and (2) more easily inspect, trace, understand, and debug configuration-related performance issues.*

Our human-centered research design consists of three steps, summarized in Figure 1.3: We first conduct an *exploratory user study* to *identify* the *information needs* that developers have when debugging the performance of configurable software systems. Afterwards, we *design* and *implement information providers*, adapting and *tailoring* ingredients (i.e., techniques and information sources), to support developers' needs. In the process of identifying relevant ingredients, we note the *limitations* of existing *performance-influence modeling* techniques that we need to overcome to provide relevant information to developers. Consequently, we present and evaluate a *white-box* performance-influence modeling approach that overcomes the limitations. Finally, we conduct two user studies to *validate* and *confirm* that the designed information providers are useful to developers when debugging the performance of complex configurable software systems, in terms of supporting their information needs and speeding up the debugging process.

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

- We identify the information needs that developers have when debugging the performance of configurable software systems. Developers struggle to find relevant information to

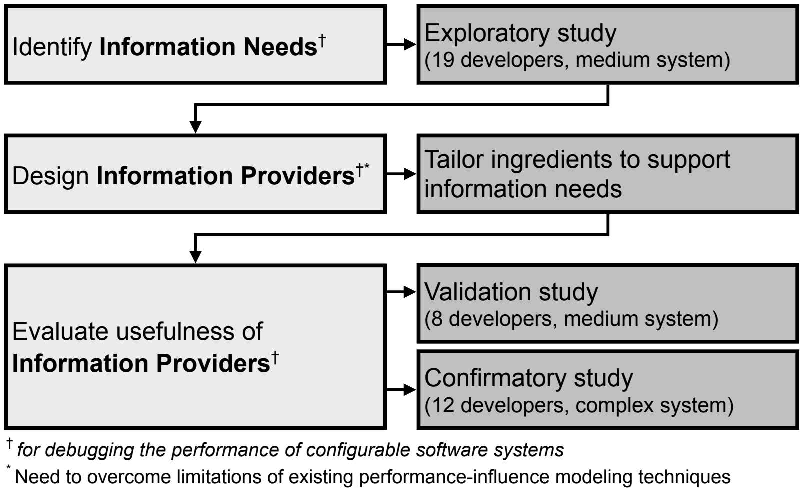

Identify **Information Needs**†
Exploratory study
(19 developers, medium system)
Design **Information Providers**†*
Tailor ingredients to support information needs
Evaluate usefulness of **Information Providers**†
Validation study
(8 developers, medium system)
Confirmatory study
(12 developers, complex system)
† *for debugging the performance of configurable software systems*
* Need to overcome limitations of existing performance-influence modeling techniques

Figure 1.3: Overview of our human-centered approach to support the needs that developers have when debugging the performance of configurable software systems.

(a) identify influencing options; the configuration options or interactions causing an unexpected performance behavior, (b) locate option hotspots; the methods where configuration options affect the performance of the system, and (c) trace the cause-effect chain; how influencing options are used in the implementation to directly and indirectly affect the performance of option hotspots (Chapter 2).

- We identify the ingredients (i.e., techniques and information sources) that can be tailored to support above needs. Interpretable global and local performance-influence models can help developers identify influencing options and locate option hotspots, and CPU profiling and program slicing can help developer trace the cause-effect chain (Chapter 2).
- We note the limitations of existing performance-influence modeling approaches to efficiently build accurate and interpretable models (Chapter 2).
- We introduce the insights of *compositionality* and *compression* of our white-box approach to accurately and efficiently model the local and global performance of configurable software systems (Chapter 3).
- Our white-box approach tailors a taint analysis to identify how configuration options and their interactions influence the performance of independent code regions (Chapter 3).
- We discuss different design decisions to operationalize our insights and tailored taint analysis, in terms of the type of analysis to use and the granularity of code regions. We implement two prototypes, **ConfigCrusher** and **Comprex**, considering different design decisions (Chapter 3).
- Our empirical evaluation on 13 widely-used open-source configurable software systems, comparing **ConfigCrusher** and **Comprex** to numerous state-of-the-art approaches, demonstrate that our prototypes can efficiently and accurately model the performance of configurable software systems, often more efficiently than black-box approaches that generate

models with comparable accuracy. Additionally, our prototypes build local and global models that are interpretable and can be mapped to specific code regions (Chapter 3).

- We describe how we design and implement information providers to support developers' needs. Specifically, we describe how we tailor **G**lobal and **L**ocal performance-**I**nfluence **M**odels, CPU **P**rofiling, and program **S**licing, and integrate them in a cohesive tool called **GLIMPS** (Chapter 4).
- We empirically validate and confirm that our designed information providers support the needs that developers have and speed up the process of debugging the performance of complex configurable software systems (Chapter 4).

1.4 Outline

The remainder of this dissertation is structured as follows:

- Chapter 2 presents a user study, in which we identified the information needs that developers have when debugging the performance of configurable software systems. Based on the findings, we identify the ingredients that can be tailored to support those needs. We also note the limitations of the ingredients to build global and local performance-influence models.
- Chapter 3 introduces the key insights and how we tailor a taint analysis for efficient, accurate, and interpretable performance-influence modeling of configurable software systems. We discuss the design decisions to implement two prototypes, and evaluate the prototypes.
- Chapter 4 details how we design and implement information providers, tailoring the ingredients that we identified, to help developers debug. Two user studies provide evidence that our designed information providers support the needs that developers have when debugging the performance of configurable software systems.
- Chapter 5 concludes the dissertation, summarizes potential impact, and briefly outlines future work.

Chapter 2

Debugging Performance in Configurable Software Systems: Information Needs

In Section 1.2, we discussed the struggle that developers experience when debugging the performance of configurable software systems, and the limited empirical evidence of the usefulness of existing performance and program debugging techniques to help developers in this process.

In this dissertation, we take a human-centered approach [Farooqui et al., 2019, Myers et al., 2016] to *identify* solutions to support developers' actual needs in the process of debugging the performance of configurable software systems. In this chapter, we first conduct an *exploratory user study* to *identify* the *information needs* that developers have when debugging the performance of configurable software systems. Our exploratory study reveals that developers *struggle* to find relevant information to (a) identify influencing options; the configuration options or interactions causing an unexpected performance behavior, (b) locate option hotspots; the methods where configuration options affect the performance of the system, and (c) trace the cause-effect chain; how influencing options are used in the implementation to directly and indirectly affect the performance of option hotspots. Afterwards, we discuss and *identify ingredients* (i.e., techniques and information sources) that can be *tailored* to support the above needs. Specifically, we discuss how *interpretable global* and *local performance-influence modeling* can be tailored to help developers identify influencing options and locate option hotspots, and how *CPU profiling* and *program slicing* can be tailored to help developers trace the cause-effect chain. While the latter two techniques can be tailored to support developers' needs without major modifications, there are some limitations with existing *performance-influence modeling techniques* that we need to overcome to provide relevant information to developers.

In summary, we make the following contributions:

- The information needs – influencing options, option hotspots, and cause-effect chain– that developers have when debugging the performance of configurable software systems.
- The ingredients (i.e., techniques and information sources) – interpretable global and local performance-influence modeling, CPU profiling, and program slicing – that can be tailored to support above needs.

The rest of this chapter is organized as follows: We first explore existing performance and program debugging techniques, and note the *limited empirical evidence* of *how useful* the techniques are to support developers' needs when debugging the performance of configurable software systems (Section 2.1). Consequently, we conduct an *exploratory user study* to *identify* the *information needs* that developers have during this process (Section 2.2). Finally, we *identify* the *ingredients* (i.e., techniques and information sources) that can be *tailored* to support developers' needs, while also discussing the limitations of certain techniques and that we need to overcome (Section 2.3).

This chapter shares material with a conference submission under review at the time of writing: "On Debugging the Performance of Configurable Software Systems: Developer Needs and Tailored Tool Support" [Velez et al., 2021a].

2.1 Background

There is substantial literature on debugging the performance of software systems [e.g., Curtsinger and Berger, 2016, Han et al., 2012, He et al., 2020, LaToza and Myers, 2011, Liu et al., 2014, Meinicke et al., 2018, Nistor et al., 2013b, Song and Lu, 2014]. Our goal in this dissertation is to support developers in the process of debugging the performance of configurable software systems; in particular, when developers do not even know which configuration options or interactions in their current configuration cause an unexpected performance behavior.

Debugging performance in software systems. When performance issues occur in software systems, developers need to identify relevant information to debug the unexpected performance behavior [Breu et al., 2010, Chaparro et al., 2017, Han et al., 2018, Nistor et al., 2013a]. For this task, in addition to using traditional off-the-shelf profilers, such as JPRofiler [JPR, 2019], Valgrind [Nethercote and Seward, 2007], and VisualVM [VVM, 2020], some researchers suggest using more targeted profiling techniques [Castro et al., 2015, Cito et al., 2018, Curtsinger and Berger, 2016, Gregg, 2016, Yu and Pradel, 2018] and different visualizations [Adamoli and Hauswirth, 2010, Bezemer et al., 2015, Cito et al., 2018, Gregg, 2016, Sandoval Alcocer et al., 2019, Trümper et al., 2013] to identify and analyze the locations of performance bottlenecks. For instance, Coz [Curtsinger and Berger, 2016] introduced causal profiling to help developers identify which components in their concurrent system they should optimize to improve performance. Likewise, flame graphs [Gregg, 2016] are compact visualizations of call stacks that allow developers understand the execution of a system. Alternatively, some researchers suggest using techniques to search for inefficient coding patterns [Bornholt and Torlak, 2018, Curtsinger and Berger, 2016, Liu et al., 2014, Nistor et al., 2013b, 2015, Song and Lu, 2017] For instance, Toddler [Nistor et al., 2013b] detects performance bugs by identifying repetitive memory read sequences across loop iterations. While these techniques are quite useful, there is limited evidence of their usefulness when debugging the performance of configurable software systems; particularly, to determine how performance issues are related to configuration options and their interactions.

In addition to performance debugging techniques, there are several established *program debugging techniques* that can help developers narrow down and isolate relevant parts of a system to focus their debugging efforts [Agrawal and Horgan, 1990, Andrzejewski et al., 2007, King, 1976, Korel and Laski, 1988, Weiser, 1981, Zeller, 1999]. For instance, 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] provides developers with a slice of the relevant fragments of a system based on a criterion. For these reasons, the techniques have been implemented in the backend of tools to help developers debug [Burg et al., 2013, Ko and Myers, 2004, Pothier et al., 2007]. For instance, Whyline [Ko and Myers, 2004] combines static and dynamic program slicing to allow developers to ask "why did" and "why did not" questions about a system's output. While these tools have been evaluated in terms of their technical accuracy and usability with numerous user studies, there is limited evidence of which and how these *program debugging techniques* can be used or adapted for debugging the performance of configurable software systems.

Performance issues in configurable software systems. Performance issues are often caused by misconfigurations or software bugs, both of which impair user experience. Misconfigurations are errors in which the system and the input are correct, but the system does not behave as desired, because the selected configuration is inconsistent or does not match the intended behavior [Xu et al., 2013, Zhang et al., 2014, Zhang and Ernst, 2015]. By contrast, a software bug is a programming error that degrades a system's behavior or functionality [Andrzejewski et al., 2007, Zeller, 1999, 2009]. Research has repeatedly found that configuration-related performance issues are common and complex to fix in software systems [Han and Yu, 2016, Han et al., 2018, He et al., 2020, Krishna et al., 2020, Wang et al., 2018]. Regardless of the root cause of the unexpected behavior (misconfigurations or software bugs), systems often misbehave with similar symptoms, such as crashes, incorrect results [Andrzejewski et al., 2007, Meinicke et al., 2018, Zeller, 1999, 2009], and, in terms of performance, long execution times or increased energy consumption [Han and Yu, 2016, He et al., 2020, Jin et al., 2012, Krishna et al., 2020, Li et al., 2016, Song and Lu, 2017, Wilke et al., 2013].

Debugging performance in configurable software systems. Similar to debugging performance in general, identifying relevant information is key when debugging unexpected performance behaviors in configurable software systems. Ideally, developers would have relevant information to debug how performance issues are related to specific configuration options and their interactions. Unfortunately, there are situations in which developers only know the *effect* of an unexpected performance behavior (e.g., an unexpected slowdown when a user executes a configuration in Figure 1.2). In these situations, developers need to debug the system to determine whether the system has a potential *bug*, is *misconfigured*, or *works correctly*, but the user has a *different expectation* about the performance behavior of the system. For these reasons, our goal in this dissertation is to support developers in *finding relevant information* to debug unexpected performance behaviors in configurable software systems.

There are some research areas that use information-flow analyses to help developers understand how configuration options and their interactions affect the behavior of configurable software systems [Dong et al., 2016, Hoffmann et al., 2011, Li et al., 2020, Lillack et al., 2018, Meinicke et al., 2016, Nguyen et al., 2016, Rabkin and Katz, 2011, Reisner et al., 2010, Toman and Grossman, 2016, 2018, Wang et al., 2013, Wong et al., 2018, Xu et al., 2016, Zhang and

Ernst, 2014]. Thüm et al. [2014] presented a comprehensive survey of analyses for software product lines also applicable to configurable software systems.

Some researchers have used taint analyses to track how configuration options are used and propagated in configurable software systems [Hoffmann et al., 2011, Lillack et al., 2018, Toman and Grossman, 2016, 2018]. A taint analysis is a static [Arzt et al., 2014] or dynamic [Austin and Flanagan, 2009, Bell and Kaiser, 2014] information-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]. In taint analysis, a value is initially marked as tainted, and all values derived (directly or indirectly) from the initial value are also tainted, which is then used to identify if the values are used in locations where they should not (e.g., sent over the network). In the context of configurable software systems, Lotrack [Lillack et al., 2018] used a static taint analysis to identify under which configurations particular code fragments are executed. Likewise, Staccato [Toman and Grossman, 2016] used a dynamic taint analysis to identify the use of stale configuration data. While these techniques can solve specific technical challenges, there is limited empirical evidence of the usefulness of these techniques, particularly, for debugging the performance of configurable software systems.

Other researchers have used symbolic execution and variational execution to analyze the behavior of configuration option and interactions [Meinicke et al., 2016, 2018, Nguyen et al., 2016, Reisner et al., 2010, Wong et al., 2018]. 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 system to execute. By contrast, 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]. In the context of configurable software systems, Reisner et al. [2010] and Meinicke et al. [2016] used symbolic execution and variational execution, respectively, to identify how configuration options interact in software systems. While these techniques are accurate at solving specific technical challenges, there is limited evidence of how these techniques can be used or adapted for debugging the performance of configurable software systems.

In terms of understanding performance, some researchers suggest that performance-influence models can help developers debug unexpected performance behaviors [Ha and Zhang, 2019, Kolesnikov et al., 2018, Siegmund et al., 2015, Weber et al., 2021], as the models describe the performance of a system in terms of its configuration options and their interactions. For example, the model $\mathtt{m} = 2 + 8 \cdot \mathrm{A} \cdot \mathrm{B} + 5 \cdot \mathrm{C}$ explains the influence of the configuration options A, B, and C, and their interactions on the performance of a system; for instance, selecting (i.e., setting to `true`) C increases the execution time of the system by 5 seconds, and selecting A and B, *together*, further increase the execution time by 8 seconds. Some approaches also model the performance of individual methods [Han et al., 2021, Weber et al., 2021], which can be useful to locate *where* configuration options affect the performance of a system. However, while performance-influence models have been evaluated in terms of accuracy [Kaltenecker et al., 2019, Siegmund et al., 2012a,b, 2015] and optimizing performance [Guo et al., 2013, Nair et al., 2017, Oh et al., 2017, Zhu et al., 2017], the models have not been evaluated in terms of usability; in particular, to support developers' needs when debugging the performance of configurable software systems.

Discussion. Despite the numerous performance and program debugging techniques available to developers, there is limited evidence of how useful the techniques are to help developers debug the performance of configurable software systems. Hence, we could only, at best, speculate which techniques might support developers in this process. Consequently, we take a human-centered approach [Farooqui et al., 2019, Myers et al., 2016] to identify the *information needs* that developers have when debugging the performance of configurable software systems. Afterwards, we identify the *ingredients* (i.e., techniques and information sources) that can be tailored to support those needs.

2.2 Exploring Information Needs

We investigate the *information needs* that developers have and the process that they follow to debug the performance of configurable software systems. Specifically, we answer the following research questions:

RQ1: What information do developers look for when debugging the performance of configurable software systems?

RQ2: What is the process that developers follow and the activities that they perform to obtain this information?

RQ3: What barriers do developers face during this process?

2.2.1 Method

We conducted an *exploratory user study* to *identify* the *information needs* that developers have when debugging the performance of configurable software systems. Using Zeller's terminology [Zeller, 2009], we want to understand how developers *find possible infection origins*: where configuration options affect performance, and *analyze the infection chain*: what are the causes of an unexpected performance behavior, when debugging the performance of configurable software systems.

Study design. We conducted the exploratory study, combining a think-aloud protocol [Jääskeläinen, 2010] and a Wizard of Oz approach [Dahlbäck et al., 1993], to observe how participants debug a performance issue for 50 minutes: We encourage participants to verbalize what they are doing (or trying to do), while the experimenter plays the role of some tool that can provide performance behavior information (e.g., performance profiles and execution time of specific configurations) on demand, thus avoiding overhead from finding or learning specific tools.

We decided to provide additional information to participants halfway through the study, after we found, in a pilot study with 4 graduate students from our personal network, that participants spend an extremely long time (about 60 minutes in a relatively small system) just identifying relevant configuration options and methods. To additionally explore how participants search for the cause of performance issues once they have identified this information, we told participants, after 25 minutes, which configuration options cause the performance issue and the methods where the configuration options influence performance. In this way, we can both observe how participants start addressing the problem and analyze how configuration options affect the performance in the implementation.

After the task, we conducted a brief semi-structured interview to discuss the participants' experience in debugging the system, as well as the information that they found useful and would like to have when debugging the performance of configurable software systems.

Due to the COVID-19 pandemic, we conducted the study remotely over Zoom. We asked participants to download and import the source code of the subject system to their favorite IDE, to avoid struggles with using an unfamiliar environment. We also asked participants to share their screen. With the participants' permission, we recorded audio and video of the sessions for subsequent analysis.

Task and subject system. Based on past studies [Meinicke et al., 2018, Melo et al., 2016, 2017, Parnin and Orso, 2011] that have shown how time-consuming debugging even small configurable software systems is, we prepared one performance debugging task for one configurable software system of moderate size and complexity. We selected *Density Converter (Complete)*[1] as the subject system, which transforms images to different dimensions and formats. We selected this Java system because it is medium-sized, yet non-trivial (over 49K SLOC and 22 binary and non-binary configuration options), and has many configuration options that influence its performance behavior (execution time on the same workload ranged from a few seconds to a couple of minutes, depending on the configuration). The task involved a user-defined configuration that spends an excessive amount of time executing. We introduced a bug caused by the incorrect implementation of one configuration option, representative of bugs reported in past research [Abal et al., 2018, Han and Yu, 2016, Jin et al., 2012] (the system was spending a long time to transform and output a JPEG image). Participants were asked to identify and explain which and how configuration options caused the unexpected performance behavior.

Participants. We recruited 14 graduate students and 5 professional software engineers with extensive experience analyzing the performance of configurable Java systems. We stopped recruiting when we observed similar information needs and patterns in the debugging process. We used our professional network and LinkedIn for recruiting. The graduate students had a median of 6.5 years of programming experience, a median of 5 years in Java, a median of 3 years analyzing performance, and a median of 4.5 years working with configurable software systems. The software engineers had a median of 13 years of programming experience, a median of 13 years in Java, a median of 5 years analyzing performance, and a median of 5 years working with configurable software systems.

Analysis. We analyzed transcripts of the audio and video recordings of the debugging task and interviews using standard qualitative research methods [Saldaña, 2015]. The author who

[1] We refer to this version of the system as *"Complete"* since we included all of its Java dependencies to increase the size and complexity of the system. Otherwise, the system has about 1K SLOC and acts as an interface to call several image processing libraries.

Table 2.1: Information needs that developers have and the activities that they perform to debug the performance of configurable software systems.

Information Need	Description	Frequency	Activities	Frequency
Influencing Options	Which configuration options influence the performance of the system?	19/19	Read configuration options' documentation	19/19
			Measure performance of numerous configurations	19/19
User Hotspots	What are the hotspots under the problematic configuration?	9/19	Profile the system under the problematic configuration	9/19
Option Hotspots	Where do configuration options influence the performance of the system?	10/19	Manually trace configuration options in the implementation	4/10
			Analyze the user hotspots' source code	6/10
			Inspect the user hotspots' call stacks	6/10
Cause-Effect Chain	How are influencing options used in the implementation to directly and indirectly influence the performance of option hotspots?	19/19	Analyze the option hotspots's source code	19/19
			Inspect the option hotspots's call stacks	12/19
			Use a debugger to analyze how the influencing options affect the values of the variables used in the option hotspots	10/19
			Manually trace how influencing options are used in the implementation to directly and indirectly affect the performance of option hotspots	19/19

conducted the study coded the sessions using open and descriptive coding, summarizing observations and discussions. All authors met weekly to discuss the codes and observations. When codes were updated, previously analyzed sessions were reanalyzed to update the sessions' coding.

Threats to Validity and Credibility. We observe how developers debug the performance of a system that they had not used before. Developers who are familiar with a system might have different needs or follow different processes. While readers should be careful when generalizing our findings, the needs help us identify the information that, at the very least, developers want to find when debugging the performance of unfamiliar configurable software systems.

Conducting a study with one system in which one configuration option causes a performance issue has the potential to overfit the findings to this scenario, even though the scenario mirrors common problems in practice [Han and Yu, 2016]. While we intentionally vary some aspects of the design in a subsequent study (Chapter 4) to observe whether our solutions generalize to other tasks, generalizations about our results should be done with care.

2.2.2 Results

We observed that participants struggle for a long time looking for relevant information to debug the performance of the configurable subject system. In fact, no participant was able to finish debugging the system within 50 minutes! In what follows, we present the information needs that participants had, the process that they followed, and the barriers that they faced during the debugging process.

RQ1: Information Needs. Table 2.1 lists the four information needs that we identified and the number of participants that had each need. We refer to the information needs as influencing options, option hotspots, cause-effect chain, and user hotspots. The participants referred to these needs using varying terms.

When participants faced a non-trivial configuration space, they all tried to identify the influencing options – the configuration option or interaction causing the unexpected performance behavior. More specifically, the participants tried to identify *which configuration options* in the *problematic configuration* caused the unexpected performance behavior.

Some participants tried locating option hotspots – the methods where configuration options affect the performance of the system. More specifically, the participants tried to *locate* where the *effect* of the problematic configuration could be observed; the methods whose execution time increased under the problematic configuration.

When we told participants which configuration options cause the unexpected performance behavior (i.e., the influencing options) and the methods where these configuration options influence performance (i.e., the option hotspots), all participants tried tracing the cause-effect chain – how influencing options are used in the implementation to directly and indirectly affect the performance of option hotspots. More specifically, as the participants knew which configuration options were *causing* an unexpected performance behavior and had observed the *effect* of those configuration options on the system's performance, the participants tried to find the *root cause* of the unexpected performance behavior.

Some participants also looked for user hotspots – the methods that spend a long time executing under the user-defined problematic configuration. However, as we will discuss in RQ2, these participants looked for this information trying to locate option hotspots.

Summary RQ1: Developers look for information to (1) identify influencing options, (2) locate option hotspots, and (3) trace the cause-effect chain of how configuration options influence performance in the implementation.

RQ2: Process and Activities. Table 2.1 lists the activities that participants performed when looking for relevant information and the number of participants that performed each activity. Overall, all participants *compared* the problematic configuration to the default configuration, to understand the causes of the unexpected performance behavior. In particular, the participants compared the values selected for each configuration option and analyzed how the changes were affecting the performance of the system in the implementation.

When looking for the influencing options, the participants mainly *read the documentation* and *executed* the system under *multiple configurations*, primarily *comparing* execution times. With these approaches, the participants tried to identify *which configuration options* in the *problematic configuration* were causing the unexpected behavior.

When looking for option hotspots, the participants mainly *profiled* the system under the problematic configuration, and *analyzed* the *call stacks* and *source code* of hotspots, trying to *locate the methods* where *configuration options* might be *affecting* the performance of hotspots.

When looking for the cause-effect chain, some participants *analyzed* the option hotspots' *source code*, whereas others used a *debugger*; trying to understand *how* the influencing options are *used* in the implementation to affect the performance of option hotspots. Several participants also *compared* the hotspots' *call stacks* under the problematic and default configurations, trying to understand how the influencing options affected how the option hotspots were called. Ultimately, all participants tried to *manually trace how* the influencing options were being *used* in the implementation to directly and indirectly affect the performance of the option hotspots.

While identifying the influencing options and locating the option hotspots is needed to trace the cause-effect chain, the order in which the first two pieces of information was acquired did not affect the debugging process. For instance, 9 participants started looking for influencing options, but gave up trying after a while. Then, the participants looked for and were able to identify user hotspots. Six of these participants subsequently started looking for option hotspots.

Summary RQ2: Overall, developers compare the problematic configuration to a (baseline) non-problematic configuration to understand the causes of an unexpected performance behavior. Initially, developers compare execution times to identify influencing options, and analyze call stacks and source code to locate option hotspots. These two pieces of information are necessary to trace the cause-effect chain of how influencing options are used in the implementation to directly and indirectly influence the performance of option hotspots.

RQ3: Barriers. Our participants struggled for a long time trying to find relevant information to debug how configuration options influence the performance of the system in the implementation. Most participants discussed the *"tedious and manual"* process of executing multiple configurations when looking for influencing options. For instance, only 10 out of the 19 participants identified the influencing options. While we told participants the system's execution time under any configuration that they wanted, several participants mentioned that finding the problematic configuration option would have *"taken me hours."*

Most participants also mentioned the struggle to locate option hotspots. In fact, no participant found any option hotspot! Several participants mentioned that *locating* these methods is challenging since configuration options are not typically directly used in expensive methods.

The participants struggled the most when trying to trace the cause-effect chain. In fact, no participant could establish the cause-effect chain, even when our task consisted of tracing *a single* influencing option and we explicitly told participants the influencing option and the option hotspots they needed to analyze. Most participants mentioned that manually tracing even one configuration option through a relatively small system is *"error-prone."* Additionally, some participants discussed that *identifying differences* in the option hotspots' *call stacks* was difficult for determining whether the influencing options were affecting how the option hotspots were called. As mentioned by several participants: variables used in option hotspots are often a *"result of several computations"* involving influencing options. Since the influencing options are used in various parts of the system, *"tracing which paths to follow is very challenging."*

Summary RQ3: Developers struggle for a substantial amount of time looking for relevant information to identify influencing options, locate option hotspots, and trace the cause-effect chain.

Contribution - Information needs: Developers want to (1) identify influencing options, (2) locate option hotspots, and (3) trace the cause-effect chain when debugging the performance of configurable software systems.

Thesis contribution: We identified empirically the information needs – influencing options, option hotspots, and cause-effect chain – that developers have when debugging the performance of configurable software systems. With these findings, we can determine the techniques that can be tailored to support these needs to help developers maintain their systems to reduce energy consumption and operational costs.

2.3 Ingredients for Supporting Information Needs

We aim to support developers in identifying influencing options, locating option hotspots, and tracing the cause-effect chain. To this end, we discuss, in more detail, which and how the performance and program debugging techniques presented in Section 2.1 can be used and adapted to support the above needs. In Chapter 4, we describe how we tailor and integrate the techniques into a cohesive tool, and evaluate the extent that the information that we provide support the needs that developers have.

2.3.1 Identifying Influencing Options: Performance-Influence Modeling

Developers want to identify the influencing options – the configuration option or interaction causing an unexpected performance behavior.

To understand how configuration options and their interactions affect the performance of a system, we suggest using *global performance-influence models* [Ha and Zhang, 2019, Kolesnikov et al., 2018, Siegmund et al., 2015, Weber et al., 2021], as the models describe the performance of a system in terms of its configuration options and their interactions. For example, the model $\mathtt{m} = 2 + 8 \cdot \mathrm{A} \cdot \mathrm{B} + 5 \cdot \mathrm{C}$ explains the influence of the configuration options A, B, and C, and their interactions on the performance of a system. With these models, developers can determine *which* configuration options and interactions are causing an unexpected performance behavior.

Goals of performance-influence modeling

Global performance-influence models have been used for different tasks in different scenarios, which benefit from different characteristics of the type of model that is used. In the context of helping developers debug the performance of configurable software systems, we suggest using a specific type of model: *Interpretable* models. In what follows, we describe the different scenarios and model characteristics in more detail.

- **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. Global 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.
- **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 configuration options to change to react to low-battery levels [Jamshidi et al., 2017, 2018, 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 irrelevant. 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].
- **Configuring software systems.** 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 configuration options and interactions influence performance becomes paramount. In these situations, researchers usually suggest sparse linear models, such as $\mathtt{m} = 2 + 8 \cdot \mathrm{A} \cdot \mathrm{B} + 5 \cdot \mathrm{C}$, typically learned with stepwise linear regression or similar variations [Kaltenecker et al., 2019, Siegmund et al., 2012a,b, 2015]. 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].
- **Debugging.** In addition to users who configure a system, developers who maintain the sys-

tem 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 computing, Kolesnikov et al. [2018] reported that a developer “was surprised to see that [a configuration option] had only a small influence on system performance”, indicating a potential bug. In such situations, *understanding* how individual configuration options and interactions influence performance is again paramount, favoring *interpretable models*.

Summary: We suggest using interpretable global performance-influence models, such as those generated with linear regressions, as humans can inspect the models, reason about factors, and make and understand predictions.

Building global performance-influence models

Global 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. In what follows, we describe the different approaches to builds performance-influence models in more detail.

- **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 configuration options.
- **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, Weber et al., 2021], 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., 2012a,b, 2015], regression trees [Grebhahn et al., 2019, Guo et al., 2013, 2017, Sarkar et al., 2015, Weber et al., 2021], Fourier Learning [Ha and Zhang, 2019], Gaussian Processes [Jamshidi et al., 2017], 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 expensive, 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.

– **Family-Based Performance Measurement.** In contrast to *black-box* approaches, Family-Based Performance Measurement [Siegmund et al., 2013] is a *white-box* approach to build performance-influence models. The approach uses a static mapping between configuration options to code regions and instruments the system to measure the execution time spent in the regions. Subsequently, the approach executes the system once with all configuration options selected, tracking how much each configuration option contributes to the execution time. The approach works well when all configuration options are directly used in control-flow statements and only contribute extra behavior. That is, a configuration option would not switch between two implementations, but only activate additional code. Current implementations, however, derive the static map from compile-time variability mechanisms (e.g., preprocessor directives) and do not handle systems with load-time variability (i.e., loading and processing configuration 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.

Summary: Most current performance-influence modeling approaches are black box, which rely on sampling strategies to potentially capture performance-relevant interactions to learn the performance behavior of a system from incomplete samples. The sampling strategies affect the cost to build the 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 and the interpretability of the models. The only existing white-box approach imposes strict constraints on the structure of the configurable software systems that it can analyze.

Discussion

We suggest using *interpretable global performance-influence models* to identify influencing options, as the models are represented to indicate *how* configuration options and their interactions affect the performance of a system. However, we do not suggest simply showing linear models to developers. Rather, based on our findings that developers *compare* a problematic configuration to a non-problematic configuration when debugging the performance of configurable software systems (Section 2.2.2), we *adapt* and *tailor* these models to show developers relevant and targeted information. In Chapter 4, we describe how we tailor these models in more detail.

We present a new performance-influence modeling technique to overcome the limitations of existing modeling techniques, in terms of tradeoffs among the cost to build models, and the accuracy and interpretability of the models. Our technique analyzes the performance of configurable software systems using a *white-box* approach to avoid relying on machine learning to extrapolate from incomplete samples. We describe our white-box performance-influence modeling approach in more detail in Chapter 3.

Contribution - Ingredient to support developers' needs: We suggest using and tailoring interpretable global performance-influence models to help developers identify influencing options. However, we develop a white-box performance-influence modeling technique to overcome the limitations of existing modeling techniques.

2.3.2 Locating Option Hotspots: Performance-Influence Modeling

Developers want to locate option hotspots – the methods where configuration options affect the performance of the system.

To locate where configuration options and their interactions affect the performance of a system, we suggest using *local performance-influence models* [Weber et al., 2021]. Analogous to how global performance-influence models describe the influence of options and interactions on the performance of a *system* (Section 2.3.1), local models describe the influence of configuration options and interactions of *individual methods*. Hence, local models indicate *where* options affect the performance in the implementation [Weber et al., 2021]. For instance the local model $\mathbb{m}_{\text{FOO}} = 2 + 8 \cdot \text{A} \cdot \text{B}$ explains the influence of A and B on the performance of the method FOO.

Building local performance-influence models

Existing performance-influence modeling techniques have mostly modeled the global performance of configurable software systems. The same black-box techniques could be used to model the performance of methods: A sampling strategy could be used to measure the performance of methods, and then a learning algorithm could be used to build a model for each method. This strategy, however, has seldom been discussed in the performance-influence modeling literature.

Recently, Weber et al. [2021] presented an approach to build performance-influence models for methods and the entire system. The approach, however, suffers from most of the limitations described in Section 2.3.1 about black-box approaches: The approach uses a sampling strategy to profile the performance of methods under different configurations, and generates performance-influence models using machine learning. Afterwards, the approach identifies inaccurate local models, and repeats the same sampling and learning process again, using a more accurate, but expensive, profiling technique. While the local and global performance-influence models are accurate, the approach makes tradeoffs by using sampling and learning techniques, and, most importantly, does not generate interpretable models, which developers need for debugging.

Summary: The only performance-influence modeling technique that builds local models makes tradeoffs in terms of the cost to build models, and the accuracy and interpretability of the models. Most importantly, the technique does not build interpretable models, which we need to help developers debug.

Discussion

We suggest using *interpretable local performance-influence models* for locating option hotspots, as the models are represented to indicate *how* configuration options and their interactions affect the performance of specific *methods* of a system. Similar to our suggestion of using global performance-influence models, we do not suggest simply showing linear models to developers. Rather, we *adapt* and *tailor* these models to show developers relevant and targeted information. In Chapter 4, we describe how we tailor these models in more detail.

We design our white-box performance-influence modeling approach, which overcomes the limitations of existing techniques, to model the *local* and *global* performance of configurable software systems. In Chapter 3, we describe our white-box performance-influence modeling approach in more detail.

Contribution - Ingredient to support developers' needs: We suggest using and tailoring interpretable local performance-influence models to help developers locate option hotspots. However, we develop a white-box performance-influence modeling technique to model local performance of configurable software systems.

2.3.3 Tracing the Cause-Effect Chain: CPU Profiling and Program Slicing

Developers want to trace the cause-effect chain – how influencing options are used in the implementation to directly and indirectly affect the performance of option hotspots.

To help developers understand how configuration options and their interactions are used in the implementation to directly and indirectly affect the performance of a system, we suggest using *CPU profiling* and *program slicing*.

CPU Profiling

We suggest using traditional off-the-shelf *CPU profiling* to help developers trace the cause-effect chain; specifically, to analyze whether influencing options *affect* how option hotspots are called. However, simply profiling the performance of a system under different configuration is not sufficient to debug the performance of configurable software systems (see results in Section 2.2.2). Instead, we show developers a *comparison* of the *hotspot view* between a problematic configuration and a non-problematic configuration to help developers understand whether the influencing options influence how option hotspots are called. The hotspot view is the inverse of a call tree: A list of all methods sorted by their total execution time, cumulated from all different call stack, and with back traces that show how the methods are called.

Since we suggest using off-the-shelf profiling, we do not envision the need to develop new profiling techniques. In Chapter 4, we describe how we tailor CPU profiling in more detail.

Program Slicing

We suggest using *program slicing* to help developers trace the cause-effect chain. Several debugging tools have been implemented on top of program slicers [Fu et al., 2020, Ko and Myers, 2004, Xu et al., 2005] to help developers narrow down and isolate relevant inputs and parts of a system where developers should focus their debugging efforts. We suggest using program slicing to *track* how influencing options are *used in the implementation* to directly and indirectly *influence* the performance of option hotspots. More specifically, we suggest slicing the system from the point when influencing options are first loaded in to the system (e.g., the `main` method in Java system) to option hotspots.

We plan to use standard program slicing techniques – our suggestion is on *how* to slice the system. Hence, we do not envision the need to improve existing techniques. In Chapter 4, we describe how we tailor program slicing in more detail.

Contribution - Ingredients to support developers' needs: We suggest using and tailoring CPU profiling and program slicing to help developers trace the cause-effect chain.

Thesis contribution: We identified the ingredients (i.e., techniques and information sources) – interpretable global and local performance-influence modeling, CPU profiling, and program slicing – that can be tailored to support developers' needs when debugging the performance of configurable software systems. Providing relevant information from these ingredients can help developers maintain their systems to reduce energy consumption and operational costs.

2.4 Summary

In this chapter, we took a *human-centered* approach to identify solutions to support developers actual needs in the process of debugging the performance of configurable software systems. We conducted an *exploratory user study*, with 19 developers, in which we identified that developers struggle to find relevant information to identify influencing options; the configuration options or interactions causing an unexpected performance behavior, locate option hotspots; the methods where configuration options affect the performance of the system, and trace the cause-effect chain; how influencing options are used in the implementation to directly and indirectly affect the performance of option hotspots. Based on these findings, we suggested that *interpretable global* and *local performance-influence modeling* can be used to help developers identify influencing options and locate option hotspots, and that *CPU profiling* and *program slicing* can be used to help developers trace the cause-effect chain.

Identifying the information needs and ingredients that can support those needs contribute to our thesis goal of reducing the energy consumption and operational costs of running configurable software systems, since we can design tailored tool support to help developers debug and maintain their systems.

In Chapter 4, we describe how we tailor and integrate the ingredients into a cohesive tool, and evaluate the extent that the information that we provide support the needs that developers have. However, before discussing how we tailor the ingredients in more detail, we need to overcome the limitations of existing performance-influence modeling techniques. Hence, we first present and evalute our white-box approach in Chapter 3.

Chapter 3

White-box Performance-Influence Modeling

In Chapter 2, we suggested using interpretable global and local performance-influence models to help developers identify influencing options and locate option hotspots. However, most existing performance-influence modeling approaches have limitations, in terms of tradeoffs among the cost to build models, and the accuracy and interpretability of the models. Hence, we present a new modeling technique that overcomes these limitations by analyzing the performance of configurable software systems using a *white-box* approach to avoid relying on machine learning.

Our white-box approach to model the performance of configurable software systems analyzes and instruments the source code to accurately capture configuration-specific performance behavior, without using machine learning to extrapolate incomplete samples. We reduce measurement cost by *simultaneously analyzing* and *measuring* multiple regions of the system, building a local *linear* performance-influence model per region with a few configurations (an insight that we call *compression*). Subsequently, we *compose* the local models into a global *linear* model for the entire system. We *tailor* a *taint analysis* to identify *where* and *how* load-time configuration options influence control-flow statements in the system, through control- and data-flow dependencies.

Our empirical evaluation on several widely-used open-source configurable software systems demonstrates that our white-box approach *efficiently* builds *accurate* performance-influence models. Additionally, the models are *interpretable*, which not only predict performance of configurations, but also quantify the influence on performance of *individual configuration options* and *interactions*. Furthermore, our white-box approach generates *local* models that map the influence of configuration options and interactions to specific code regions.

In Chapter 4, we show how we tailor interpretable global and local models to support developers' needs to debug the performance of configurable software systems.

In summary, we make the following contributions:

- The insights of *compositionality* and *compression* to accurately infer the influence of configuration options and their interactions on the performance of numerous independent regions of a system with a few configurations.
- The tailoring of a taint analysis to identify how configuration options and their interactions influence the performance of independent code regions.

- A comparison of the design decisions to operationalize our white-box approach, and the implementation of two prototypes: **ConfigCrusher** and **Comprex**.
- An empirical evaluation on 13 open-source configurable software systems, comparing the two prototypes to numerous state-of-the-art approaches, demonstrating that our white-box approach efficiently builds interpretable and accurate performance-influence models.

The rest of this chapter is organized as follows: We first present our insights of *compositionality* and *compression* for efficiently and accurately modeling the performance of configurable software systems (Section 3.1). Afterwards, we describe the three technical components of our white-box approach to operationalize the insights, including how we tailor a taint analysis to identify how configuration options influence performance (Section 3.2). Next, we discuss design decisions to implement our approach (Section 3.3) and describe two prototypes, **ConfigCrusher** and **Comprex** (Section 3.4 and Section 3.5). Finally, we evaluate the two prototypes against numerous state-of-the-art approaches to model the performance of 13 configurable software systems (Section 3.6), and discuss the impact of the design decisions made in each prototype (Section 3.7).

This chapter shares materials with our ASE Journal'20 article "ConfigCrusher: Towards White-box Performance Analysis for Configurable Systems" [Velez et al., 2020] and ICSE'21 paper "White-box Analysis over Machine Learning: Modeling Performance of Configurable Systems" [Velez et al., 2021b].

3.1 Insights for Efficient and Accurate Performance-Influence Modeling

We present the insights for a *white-box* approach to efficiently and accurately analyze and model the *global* and *local* performance of configurable software systems, *without* the use of machine learning to avoid inaccuracies of extrapolating from incomplete measurements. The approach not only provides relevant information for developers, but the approach also contributes to the thesis goal of reducing the energy consumption and operational costs of running configurable software systems, as the models that we generate can help users make informed configuration decisions. In Chapter 4, we show how we tailor global and local models to help developers debug the performance of their configurable software systems.

Similar to existing approaches, we build performance-influence models by observing the execution of a system under different configurations, but we guide the exploration with a white-box analysis of the internals of the system. For a given set of inputs, a configurable software system with a set of binary configuration 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 per-

[1] For simplicity, we describe our approach in terms of binary configuration options, but other finite configuration option types can be encoded or discretized as binary configuration options. The distinction between inputs and configuration options is subjective and domain specific. We consider configuration options as a special type of inputs with a small finite domain (e.g., Boolean configuration options), that a developer or user might explore to change functionality or quality attributes. We consider fixed values for other inputs. Note that a developer or user might fix some configuration options as inputs and consider alternative values for inputs as configuration options (e.g., use a configuration option for different workloads). We analyze the influence on performance of configuration

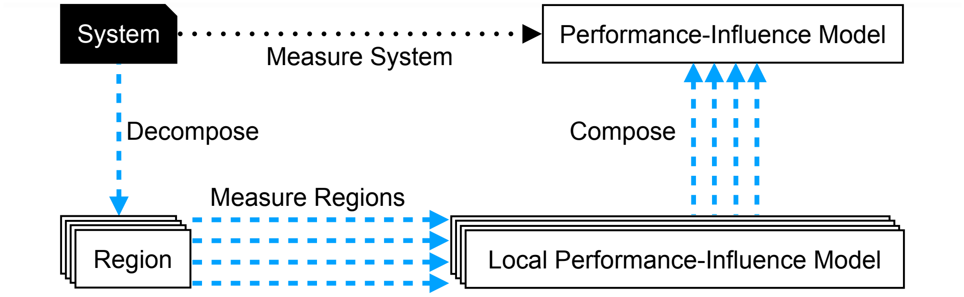

System
Measure System
Performance-Influence Model
Decompose
Compose
Measure Regions
Region
Local Performance-Influence Model

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).

formance differences to configuration options and their interactions, without any approximation through machine learning.

Our approach to efficiently and accurately analyze the performance of configurable software systems relies on two insights inspired by prior work [Meinicke et al., 2016, Nguyen et al., 2016, Reisner et al., 2010, Wong et al., 2018], that we identified by analyzing how configuration options influence the performance of software systems: (1) Performance-influence models can be built *compositionally*, composing models built independently for *smaller regions* of the code than the entire system (cf. Figure 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*.

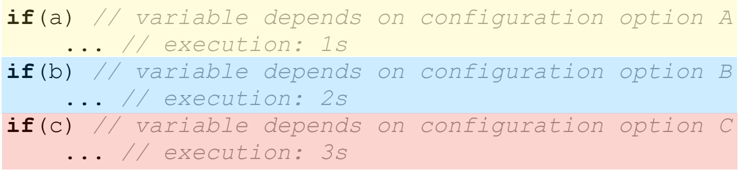
3.1.1 Compositionality

Building performance-influence models is compositional: We can measure the time that smaller regions in a system spend executing in the CPU and build a performance-influence model per region (e.g., considering each method as a region), which describes the performance behavior of each region in terms of configuration options.[2] 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 $\mathtt{m}_1 = 5 + 4 \cdot \mathrm{A}$ and $\mathtt{m}_2 = 1 - 1 \cdot \mathrm{A} + 2 \cdot \mathrm{B}$ into $\mathtt{m} = 6 + 3 \cdot \mathrm{A} + 2 \cdot \mathrm{B}$).

Compositionality helps reduce the cost to model the performance of configurable software systems, as many smaller regions of a system are often influenced only by a subset of all configuration options, a common case confirmed by prior empirical research [Meinicke et al., 2016, Nguyen et al., 2016, Reisner et al., 2010, Wong et al., 2018]. Hence, the number of distinct paths in a region is usually much smaller than the number of distinct paths in the entire system. If

options with finite domains, assuming all other inputs are fixed at specific values, thus resulting in a finite, but typically large configuration space.

[2] Note that we measure performance as the time that regions spend executing in the CPU, similar to the measurement conducted by performance profilers, which measure the execution time of methods, or the `time` util, which tracks the execution time of threads. This time is commonly referred to as "user-time"; the time the CPU spends in "user-mode". By contrast, wall-clock time is the actual time taken from the start of execution to the end.



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

Figure 3.2: Three independent regions influenced by different configuration options.

we have an analysis to find the subset of configuration options that directly and indirectly influence smaller regions (see Section 3.2.1), we can build a local performance-influence model by observing all distinct paths in a region often with only a few configurations.

> **Contribution - Our insight for white-box performance-influence modeling:** Performance-influence models can be built by composing models built independently for smaller regions of the code.

3.1.2 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 configuration options, a common case confirmed by prior empirical research [Meinicke et al., 2016, Nguyen et al., 2016, Reisner et al., 2010, Wong et al., 2018], we can measure the performance of all regions with a few configurations, instead of exploring all combinations of all configuration options. For example, the three independent regions in Figure 3.2 influenced by configuration options A, B, and C, respectively, each have two distinct paths. Instead of exploring all 8 combinations of the three configuration options, we can explore all distinct paths in each region with only 2 configurations, as long as each configuration option is selected (i.e., set to `true`) in one configuration and not selected (i.e., set to `false`) in the other configuration.

> **Contribution - Our insight for white-box performance-influence modeling:** Compression allows us to simultaneously explore paths in multiple independent regions with a few configurations.

3.1.3 Combining Compositionality and Compression

Our white-box approach combines compositionality and compression to efficiently build accurate performance-influence models, without traditional sampling or machine-learning techniques. To help developers, as well as users, understand the influence of configuration options on the performance of software 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 configuration options interact in the same region, instead influencing different parts of the system independently, a pattern observed empirically in config-

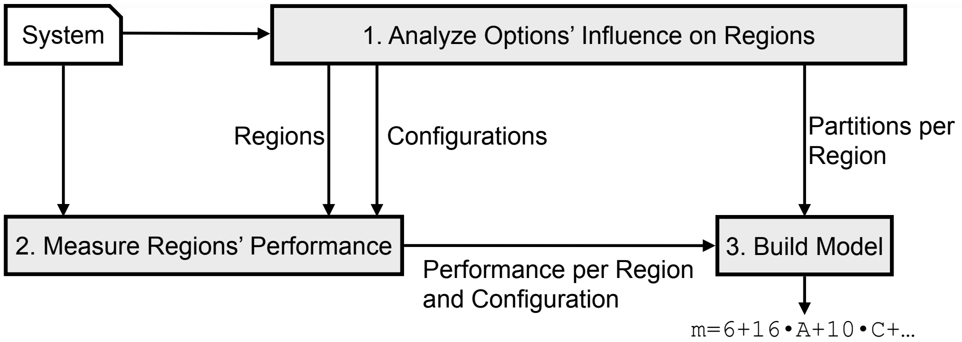

System
1. Analyze Options' Influence on Regions
Regions
Configurations
Partitions per Region
2. Measure Regions' Performance
Performance per Region and Configuration
3. Build Model
m=6+16•A+10•C+…

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

urable software systems [Meinicke et al., 2016, Nguyen et al., 2016, Reisner et al., 2010, Wong et al., 2018].

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

3.2 Components for Modeling Performance

We describe the three technical components of our white-box approach to operationalize our insights of compositionality and compression for efficiently building accurate and interpretable performance-influence models.

3.2.1 Analyze Configuration Options' Influence on Regions

As a first step, we identify which configuration options directly and 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 configuration options and their interactions (Section 3.2.2).[3] To this end, we track *information flow* from configuration options (sources) to *control-flow statements* (sinks) in each region. If a configuration option flows directly and indirectly (including implicit flows) into a control-flow statement in a region, this flow implies that selecting or deselecting the configuration option may lead to different execution paths within

[3] We focus on different execution paths caused by configuration changes, fixing all other inputs. 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., 2013a, 2015, 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).

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

Figure 3.4: Running example of a software system with 4 configuration options and 3 highlighted regions as methods, in which the configuration options influence the performance of the system.

the region. Thus, we should observe at least one execution with a configuration in which the configuration options is selected and another execution in which the configuration 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 configuration options. For example, $[\![\mathrm{A} \wedge \neg \mathrm{B}]\!]$ describes the subspace of all configurations in which configuration option A is selected and configuration option B is not selected.

To track information flow between configuration options and control-flow statements in regions, we tailor 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 configuration options flow through the system, we can identify, for each control-flow statement, the configuration 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 configuration options that reach the statement.

Example: The configuration options in our running example in Figure 3.4 are the fields A – D (Lines 2–5). Lines 6–7 and Line 18 are not influenced by any configuration options. Lines

8–14 and Lines 15–17 are influenced by the configuration option A, which leads to the partition $\llbracket \text{A} \rrbracket, \llbracket \neg\text{A} \rrbracket$. Lines 20–21 are influenced by the configuration options A and B, which leads to the partition $\llbracket \neg\text{A} \wedge \neg\text{B} \rrbracket$, $\llbracket \neg\text{A} \wedge \text{B} \rrbracket$, $\llbracket \text{A} \wedge \neg\text{B} \rrbracket$, $\llbracket \text{A} \wedge \text{B} \rrbracket$. Lines 23–24 are influenced by the configuration options A and C, which leads to the partition $\llbracket \neg\text{A} \wedge \neg\text{C} \rrbracket$, $\llbracket \neg\text{A} \wedge \text{C} \rrbracket$, $\llbracket \text{A} \wedge \neg\text{C} \rrbracket$, $\llbracket \text{A} \wedge \text{C} \rrbracket$.

Discussion. Note how using a taint analysis helps identify the configuration options and interactions that do not influence regions in the system. For instance, we now know that we do not need to explore the interaction between B and C, and consider configurations specifically for D.

Contribution - Tailor information-flow analysis for performance-influence modeling: We tailor a taint analysis to identify how configuration options and their interactions influence the performance of code regions.

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[4], 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 Figure 3.4, the subspaces that we need to cover are $\llbracket \text{A} \rrbracket, \llbracket \neg\text{A} \rrbracket, \llbracket \neg\text{A} \wedge \neg\text{B} \rrbracket, \llbracket \neg\text{A} \wedge \text{B} \rrbracket, \llbracket \text{A} \wedge \neg\text{B} \rrbracket, \llbracket \text{A} \wedge \text{B} \rrbracket, \llbracket \neg\text{A} \wedge \neg\text{C} \rrbracket, \llbracket \neg\text{A} \wedge \text{C} \rrbracket, \llbracket \text{A} \wedge \neg\text{C} \rrbracket$, and $\llbracket \text{A} \wedge \text{C} \rrbracket$. Four configurations cover all subspaces, for instance, $\big\{\{\}, \{\text{A}\}, \{\text{B, C}\}, \{\text{A, B, C}\}\big\}$, where each set represent the configuration options that are selected in the configuration. For instance, we picked the configuration $\{\text{A, B, C}\}$ by intersecting the subspaces $\llbracket \text{A} \rrbracket, \llbracket \text{A} \wedge \text{B} \rrbracket$ and $\llbracket \text{A} \wedge \text{C} \rrbracket$. Table 3.1 shows the performance measured per region and executed configuration.

3.2.3 Building the Performance-Influence Model

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

[4] 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.

Table 3.1: Measured performance per region and configuration for our running example in Figure 3.4.

Configurations				Regions		
A	B	C	D	$\texttt{main} \equiv \text{A}$	$\texttt{foo} \equiv \text{A, B}$	$\texttt{bar} \equiv \text{A, C}$
`false`	`false`	`false`	`false`	3s	0s	5s
`false`	`true`	`true`	`false`	3s	0s	15s
`true`	`false`	`false`	`false`	2s	1s	20s
`true`	`true`	`true`	`false`	2s	4s	60s

We show the configuration options that influence each region.

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 method `foo` in our running example in Figure 3.4, all configurations in which A is not selected take 0 seconds, all configurations in which A is selected and B is not selected take 1 second, and all configurations in which A and B are selected take 4 seconds.

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

The global performance-influence model is obtained simply by aggregating all local models; we add the individual influences of configuration options and their interactions in each model. In Chapter 4, we show how we tailor the global and local linear models to support developers' needs to debug the performance of configurable software systems.

Example: With the measured performance per region and configuration in Table 3.1 for our running example in Figure 3.4, we build the local models $\texttt{m}_{\texttt{main}} = 3 - 1 \cdot \text{A}$, $\texttt{m}_{\texttt{foo}} = 1 \cdot \text{A} + 3 \cdot \text{A} \cdot \text{B}$, and $\texttt{m}_{\texttt{bar}} = 5 + 15 \cdot \text{A} + 10 \cdot \text{C} + 30 \cdot \text{A} \cdot \text{C}$, which can be composed into the global performance-influence model $\texttt{m} = 8 + 15 \cdot \text{A} + 10 \cdot \text{C} + 3 \cdot \text{A} \cdot \text{B} + 30 \cdot \text{A} \cdot \text{C}$.

3.3 Design Decisions and Tradeoffs

We discuss the tradeoffs of our white-box approach to analyze and measure the influence of configuration options on regions, and how different decisions impact our approach for modeling the performance of configurable software systems.

3.3.1 Analysis of Configuration Option's Influence on Regions

In the first step of our approach, we tailor a taint analysis to inspect the direct and indirect influence of configuration options on the decisions of control-flow statements (Section 3.2.1). The analysis can be performance *statically* or *dynamically* with different tradeoffs.

The main benefit of using a *static* taint analysis is covering all execution paths in a single analysis of the configurable software system [Arzt et al., 2014]. However, the analysis might cover parts of the system that are never executed, which can increase the time of the analysis and threaten its scalability in large-scale configurable software systems. Additionally, the analysis only indicates the configuration 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 need to measure.

The main benefit of using a *dynamic* taint analysis is tracking how configuration 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 know how configuration options influence the decisions in control-flow statements in the parts of the system 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.

The above considerations can affect the accuracy of the models that we generate and the scale of the systems that our white-box approach can analyze.

3.3.2 Granularity of Regions, Compression, and Measuring Performance

We can consider regions at different granularities, which impact how much compression we obtain and the effort to measure the performance of the regions (Section 3.2.2).

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 influenced by distinct configuration options. On the other hand, we can use off-the-shelf *sampling profilers* that accurately measure performance with low overhead, and simply map the performance of methods to the closest regions on the calling stack.

The above considerations can affect the cost to generate our models, in terms of the number of configurations to measure and the effort to measure regions.

3.3.3 Implementing Designed Decisions in Two Prototypes

To empirically evaluate the tradeoffs of the taint analyses and granularity of regions discussed above, we implemented two prototypes for modeling the performance of configurable software system. We implemented an "idealistic" approach, **ConfigCrusher** [Velez et al., 2020], which uses a *static* taint analysis (i.e., one execution of the taint analysis) considering *control-flow statements* as regions (i.e., maximum compression), and a "conservative" approach, **Comprex** [Velez et al., 2021b], which uses a *dynamic* taint analysis (i.e., multiple executions of the taint analysis) considering *methods* as regions (i.e., conservative compression). In the following sections, we describe the implementation of each prototype.

3.4 ConfigCrusher

We describe the implementation of our approach to model the performance of configurable software systems using a *static* taint analysis and considering *control-flow statements* as regions. Using a static taint analysis allows us to execute the analysis *once* (Section 3.3.1), and considering control-flow statements as regions results in *maxium compression* (Section 3.3.2).

3.4.1 Analyze Configuration 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 data-flow dependencies (including implicit flows) as described in Section 3.2.1 considering control-flow statements as regions.

Static Taint Analysis Limitation

We observed that the static taint analysis did not scale to our larger subject systems. For all systems with over 100K SLOC, the analysis did not finish executing within 24 hours! This issue is caused by the size of the call graph, which restricts the size of the systems that we 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]; the largest subject system for which the static taint analysis terminated was Kanzi, which has about 20K SLOC. Accordingly, we could not generate performance-influence models for 4 of our 13 subject systems.

3.4.2 Measure Performance of Regions

To measure the performance of control-flow statements as regions, we need to instrument the regions.

Instrumenting Regions

We identify a region by a set of control-flow edges that start the region and another set of edges that end the region. Algorithm 1 describes how we identify the regions and their start (Line 3)

Algorithm 1: Instrument control-flow statements as regions

```
   Input: control-flow graph $CFG$, partition for each region $partitions : R \to \mathcal{P}(\mathcal{P}(C))$
   Output: instrumented control-flow statements as regions $R \to \mathcal{P}(E) \times \mathcal{P}(E)$
1  Function instrument_control_flow_statements($CFG$, $partitions$)
2      for each $stmt \in$ statements($CFG$)
3          $idom$ := idom($stmt, CFG$) // Get immediate dominator
4          $partition_{stmt}$ := subspaces($stmt, partitions$)
5          if $partition_{stmt} \neq \varnothing \wedge partition_{stmt} \neq$ subspaces($idom, partitions$) then
6              $r$ := new Region()
               // Omit incoming edges from loops
7              for each $edge \in$ in($stmt, CFG$)
8                  start($r, edge$) // Map $r \to edge$
9              end
10             $pdom$ := ipdom($stmt, CFG$) // Get immediate post-dominator
11             while $partition_{stmt}$ = subspaces($pdom, partitions$) do
12                 $pdom$ := ipdom($pdom, CFG$)
13             end
14             for each $edge \in$ in($pdom$)
15                 end($r, edge$) // Map $r \to edge$
16             end
17         end
18     end
19 end
```

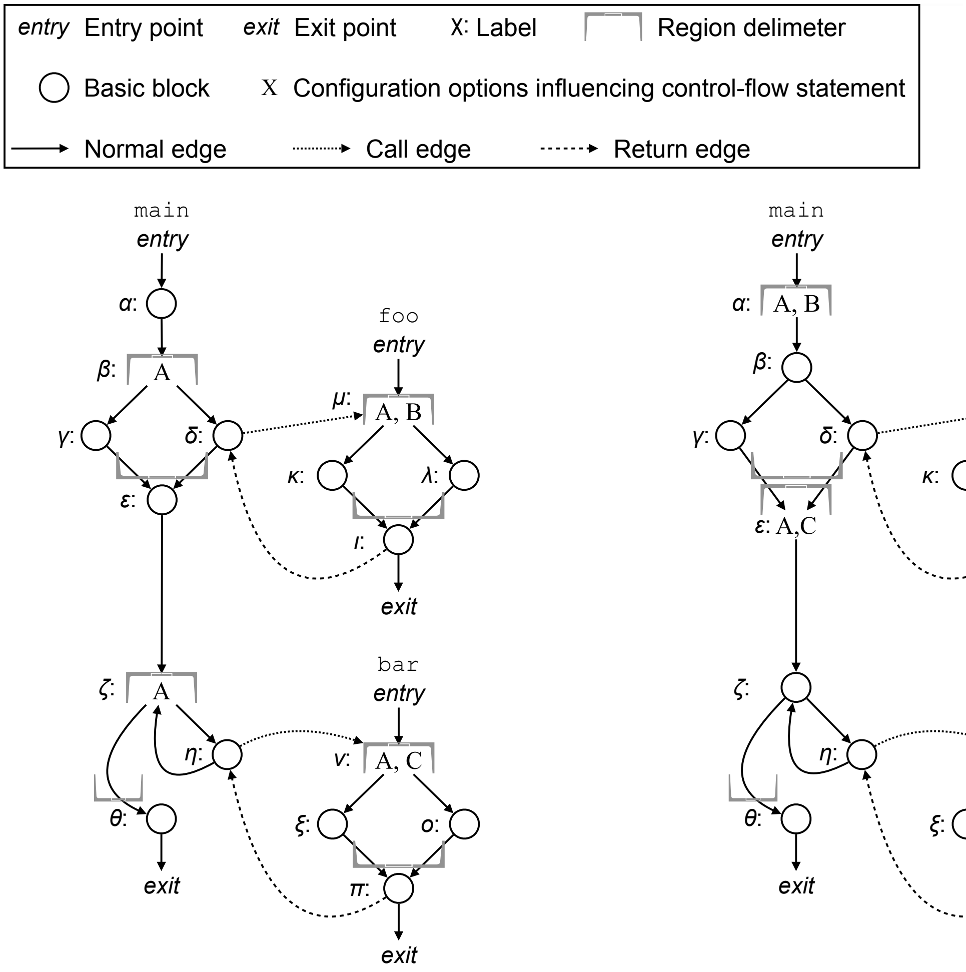
and end edges (Lines 4–17) in a method. One task of the algorithm is to find the end of a region where all the paths originating from a control-flow statement meet again (i.e., the immediate post-dominator) (Lines 10–13). After identifying all regions, we instrument the start and end edges of these regions with statements to measure the execution time. We also instrument the entry point of the system (e.g., the `main` method in a Java program) to measure the performance of code not influenced by any configuration options. The result of executing an instrumented system is the total time spent in each region.

Example: Figure 3.5a shows the four instrumented control-flow statements of our running example in Figure 3.4.

Instrumentation Overhead

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 influencing configuration options), and regions executed repeatedly in loops. Consequently, we identified optimizations to reduce measurement overhead through instrumenting regions differently without altering the performance-influence models that we produce. Specifically, we perform optimizations that preserve the following two invariants:

Invariant 1 (Expanding regions): *Statements not influenced by configuration options can be added to a region without altering the performance-influence model that is generated and without*



(a) Unoptimized **(b) Optimized**

Figure 3.5: Unoptimized and optimized instrumented control-flow statements of our running example in Figure 3.4. For simplicity, we show the basic blocks related to control-flow statements and the configuration options that influence the control-flow statements. However, recall that we partition the configuration space per region into subspaces. For instance, we partition the control-flow statements influenced by A as $[\![\mathbf{A}]\!], [\![\neg\mathbf{A}]\!]$.

increasing measurement effort. Statements not influenced by configuration options contribute the same execution time to all configurations. Therefore, including these statements in a region increases the execution time of the region equally for all configurations, which does not affect the performance difference among configurations for building performance-influence models.

Example: Consider the statement in Line 6 in our running example in Figure 3.4, which takes 1 second to execute under all configurations. Consider also the region at the control-flow statement in Line 8, which takes 1 second to execute when A is selected and 2 seconds when A is not selected. Based on this information, we can generate the partial performance-influence model $\mathtt{m}_{\mathtt{temp}} = 3 - 1 \cdot \mathrm{A}$. Since the statement in Line 6 is not influenced by any configuration options, we can include it in the region at the control-flow statement in Line 8. In this new region, we now observe 2- or 3-second executions, depending on whether A is selected, preserving the same 1 second difference, and resulting in the same model.

Invariant 2 (Merging regions): *$GP : \mathcal{P}(\mathcal{P}(\mathcal{P}(C)))$ is the set of all partitions in the system. Two consecutive regions or an outer and an inner region with partitions $p_1 \in GP$ and $p_2 \in GP$ can be merged if $p_1 \times p_2 \in GP$ without altering the performance-influence model that is generated and without increasing measurement effort.* Merging two consecutive regions or an outer and an inner region forms an interaction between the configuration options that influence both regions. Therefore, we have to combine, with the cross-product operation ($\times$), the partitions of each region to execute all combinations of the interaction to obtain their influence on the new region. The cross product reflects that, to explore all paths among multiple control-flow statements in a region, we need to explore all combinations of the individual paths in the region. If the new partition is already present in the system, we already execute all these configurations anyway. Therefore, we can merge these regions into one that is influenced by the interaction of the two regions. As stated in invariant 1, merging does not affect the absolute performance difference for building the performance-influence model. By merging regions, especially *nested regions* within loops, we *significantly* reduce the number of regions that are executed, which *significantly* reduces the overhead of measuring the instrumented system.

Example: Consider the regions at the control-flow statements in Line 8 and Line 20 in our running example in Figure 3.4. The first region is influenced by A, which leads to the partition $[\![\mathrm{A}]\!], [\![\neg\mathrm{A}]\!]$. We need to measure 2 configurations to determine that the first region takes 2 seconds to execute when A is not selected and 1 second when A is selected. Based on this information, we can generate the partial performance-influence model $\mathtt{m}_{\mathtt{L8}} = 2 - 1 \cdot \mathrm{A}$. The second region is influenced by A and B, which leads to the partition $[\![\neg\mathrm{A} \wedge \neg\mathrm{B}]\!], [\![\neg\mathrm{A} \wedge \mathrm{B}]\!], [\![\mathrm{A} \wedge \neg\mathrm{B}]\!], [\![\mathrm{A} \wedge \mathrm{B}]\!]$. We need to measure 4 configurations to determine that the second region takes 0 seconds to execute when A is not selected, 1 second when A is selected and B is not selected, and 4 seconds when A and B are selected. Based on this information, we can generate the partial performance-influence model $\mathtt{m}_{\mathtt{L20}} = 1 \cdot \mathrm{A} + 3 \cdot \mathrm{A} \cdot \mathrm{B}$. Based on this information, we can generate the partial performance-influence model $\mathtt{m}_{\mathtt{temp}} = 2 + 3 \cdot \mathrm{A} \cdot \mathrm{B}$. Since we already have to measure all configurations for the interaction of A and B in the second region, and the cross product of the partitions of the two regions equals the partition of the second region, $p_{L8} \times p_{L20} = p_{L20}$, a new interaction is not created. Therefore, we can merge both regions into one that is influenced by interaction of A and B without having to sample more configurations. In this case, the merged region would take 5 seconds to execute when A and B are selected and 2 seconds under all other configurations, resulting in the same performance-influence model when we calculate the actual influence of selecting A (i.e., $+0$ seconds) and both A and B (i.e., $+3$ seconds). With the same reasoning, we can also merge the regions at the control-flow statements in Line 16 and Line 23 in our running example.

Propagating partitions. Algorithm 2 describes how we propagate partitions up and down a control-flow graph (i.e., intraprocedually), as well as across graphs (i.e., interprocedually), to expand, merge, and pull out regions. We merge consecutive regions and expand where regions

Algorithm 2: Propagate partitions

```
Input: statement $stmt$, sontrol-flow graph $CFG$, partition for each region $partitions : R \to \mathcal{P}(\mathcal{P}(C))$
Output: optimized partition for each region $R \to \mathcal{P}(\mathcal{P}(C))'$
Function propagate_partitions_down($stmt$, $CFG$, $partitions$)
    $idom$ := ipdom($stmt, CFG$) // Get immediate post-dominator
    $partition_{idom}$ := subspaces($idom, partitions$)
    $partition_{stmt}$ := subspaces($stmt, partitions$)
    if $partition_{idom} \times partition_{stmt} = partition_{stmt}$ then
        update_partition($partitions, idom, stmt$)
    end
end

Input: statement $stmt$, control-flow graph $CFG$, partition for each region $partitions : R \to \mathcal{P}(\mathcal{P}(C))$,
       Set of all partitions in the system $GP : \mathcal{P}(\mathcal{P}(\mathcal{P}(C)))$
Output: optimized partition for each region $R \to \mathcal{P}(\mathcal{P}(C))'$
Function propagate_partitions_up($stmt$, $CFG$, $partitions$, $GP$)
    $partition_{stmt}$ := subspaces($stmt, partitions$)
    for each $pred \in$ preds($stmt, CFG$)
        $partition_{pred}$ := subspaces($pred, partitions$)
        $partition_{new} := partition_{pred} \times partition_{stmt}$
        if $partition_{new} \in GP \wedge partition_{new} \neq partition_{pred}$ then
            update_partition($partitions, pred, stmt$)
        end
    end
end
```

end, as well as pull out nested regions and expand where regions start. Obeying our invariants, we never create new interactions, partitions, or alter the performance-influence models that we generate, but significantly reduce the overhead of measuring the instrumented system. After propagation, we identify and instrument the regions as before (Algorithm 1).

The propagation algorithm is non-deterministic (i.e., different results can be obtained depending on the order in which regions are merged). In fact, different orderings can be used to optimize for different goals. Assuming that most of the overhead occurs in nested regions, especially those inside loops, we prioritize pulling regions out of loops.

Example: Figure 3.5b presents an optimized instrumentation, in which we prioritized pulling out regions in the callees. For instance, we followed invariant 2 to first merge, at the caller, the regions in the control-flow statement in method `foo`, label μ, and the control-flow statement in method `main`, label β. Then, we followed invariant 1 to expand where the new region starts by adding the statement at label α.

Executing the Instrumented System

After instrumentation, we execute the system and track execution times for each region. At the start and end of every region, we record the current time and log the difference as the execution time of the region. Since regions might be nested during execution, we also keep a stack of regions at runtime and subtract the time of nested regions from the time of outer regions. This additional step can become a source of overhead for deeply nested regions, which is what we observed in the unoptimized instrumented systems. We tried building a trace of regions and processing the execution times after the system finished executing. However, due to the large number of regions that were executed, the systems ran out of memory. Our evaluation shows that the dynamic processing incurs low overhead.

3.5 Comprex

We describe the implementation of our approach to model the performance of configurable software systems using a *dynamic* taint analysis and considering *methods* as regions. Using a dynamic taint analysis allows us to track the *actual* influence of configuration options (Section 3.3.1), and considering methods as regions allows us to use off-the-self *sampling profilers* to accurately measure performance with low overhead (Section 3.3.2).

3.5.1 Analyze Configuration 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 Section 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. Algorithm 3 describes how we partition the configuration space per region, based on incremental updates from our dynamic taint analysis. Intuitively, we execute the system in a configuration and observe when data-flow and control-flow taints from configuration options reach each control-flow decision in each region, and subsequently update each region's partition: Whenever we reach 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 (Line 7). 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 (main loop, Line 3); 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.

Updating partitions works as follows: When we reach a control-flow statement in a region with data-flow taints t_d, this information indicates that the configuration options in t_d affect the control-flow decision, *but other configuration options do not*. Thus, we know that all configurations that share the same selection for all configuration options in t_d will result in the same control-flow decision, while configurations with different selections of these configuration options may result in different decisions. Since the taint analysis tells us only that the configuration

Algorithm 3: Iterative Dynamic Taint Analysis

```
   Input: configurable software system $p$
   Output: partition for each region $R \to \mathcal{P}(\mathcal{P}(C))$
1  Function partion_all_regions($p$)
2      $partitions$ := $R \to \{C\}$; $executed\_configs$ := $\emptyset$
3      until explored_all_subspaces($executed\_configs, partitions$) do
4          $cc$ := get_next_config($executed\_configs, partitions$)
5          $executed\_configs$ := $executed\_configs \cup cc$
6          during execute_taint_analysis($p, cc$) when reaching a control-flow decision with
             taints $t_d$ and $t_c$ in region $r$:
7              $partitions$[$r$] := $partitions$[$r$] $\times$ get_partition($t_d, t_c, cc$)
8          end
9      end
10     return $partitions$
11 end

   Input: executed configurations $ec : \mathcal{P}(C)$, $partitions : R \to \mathcal{P}(\mathcal{P}(C))$
   Output: true or false
12 Function explored_all_subspaces($ec$, $partitions$)
13     $all\_subspaces$ := $\bigcup$ image($partitions$)
14     return $\forall s \in all\_subspaces.\ \exists c \in ec.\ c \in s$
15 end

   Input: data-flow taints $t_d$, control-flow taints $t_c$, current configuration $cc$
   Output: partition $p : \mathcal{P}(\mathcal{P}(C))$ for the current decision
16 Function get_partition($t_d, t_c, cc$)
17     $s_{reach} := \{c \in C \mid \forall o \in t_c.\ o \in c \Leftrightarrow o \in cc\}$
18     $p := \{C \setminus s_{reach}\}$ // subspace of configurations that might not reach decision
       // add one subspace for every combination of configuration options in data-flow taints
19     for $a \in \mathcal{P}(t_d)$ do
20         $s := \{c \in C \mid \forall o \in t_d.\ o \in c \Leftrightarrow o \in a\}$
21         $p := p \cup \{s \cap s_{reach}\}$
22     end
23     return $p \setminus \emptyset$
24 end
```

options in t_d may somehow (directly or indirectly) affect the decision's condition, but not how, we will need to explore at least one configuration for every possible assignment to these configuration options, even though multiple or even all may end up taking the same branch. Therefore, we partition the configuration space *at this decision* corresponding to all combinations of the configuration options in t_d (Lines 19–20). For example, for a decision influenced by A and B, we partition the configuration space into four subspaces: all configurations in which A and B are selected together, all configurations in which A is selected but not B, all configurations in which B is selected but not A, and all configurations in which neither A and B are selected. Finally, we update the *region's* partition with the partition derived for the decision by computing their cross product ($\times$, Line 7). This operation reflects that, to explore all paths among multiple control-flow decisions in a region, including multiple executions of the same control-flow statement, we need to explore all combinations of the individual paths in the region.

Distinguishing data-flow taints from control-flow taints allows us to optimize the exploration of nested decisions (e.g., `if(a){ if(b) ... }`). Control-flow taints specify which configuration options (directly or indirectly) influenced outer control-flow decisions, which indicates that different assignments to configuration 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 configuration options affecting outer and inner decisions. Instead of exploring combinations for all configuration options of data-flow and control-flow taints, we first split the configuration space into (1) those configurations for which we know that they will reach the current decision, as they share the assignments of configuration options in control-flow taints (s_{reach}, Line 17), and (2) the remaining configurations which may not reach the current decision ($C \setminus s_{reach}$, Line 18). Then, we only create subspaces for interactions of configuration options in data-flow taints within s_{reach} (Lines 19–21) and consider the entire set of configurations outside s_{reach} as a single subspace (Line 18). The iterative nature of our analysis ensures that at least one of the configurations outside s_{reach} 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 (Line 4), 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.[5]

Example: Figure 3.6 presents an example of executing the iterative analysis on our running example in Figure 3.4. If we execute the configuration $\{$A, D$\}$, in which the configuration options A and D are selected and the other configuration options are not selected, the taint analysis will indicate that the value of the variable `a` is tainted by option A (from Line 2) when reaching the `if` statement in Line 8 (region `main`). Thus, all configurations in which A is selected result in the same control-flow decision and all configurations in which A is not selected result, potentially, in the same or a different decision. Hence, we derive the initial partition $[\![\text{A}]\!], [\![\neg\text{A}]\!]$ for this method.

Continuing the execution, we next reach the `if` statement in Line 20 (region `foo`), where the value of the variable `x` is tainted with control-flow taint A (from Line 8) and data-flow taint B (from variable `b`). Thus, all configurations in which A and B are selected result in the same control-flow decision, all configurations in which A is selected and B is not selected may result in a different control-flow decision, and all configurations in which A is not selected may not reach this decision. Hence, we derive the partition $[\![\text{A} \wedge \neg\text{B}]\!], [\![\neg\text{A}]\!], [\![\text{A} \wedge \text{B}]\!]$ for this region. Note how we explore this nested `if` statement with 3 instead of 4 subspaces by separately tracking data-flow and control-flow taints.

[5] 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.

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

Figure 3.6: Example of iteratively executing the taint analysis on our running example in Figure 3.4. Four configurations explore all subspaces for the three regions (methods) in the system, where each set represents the configuration 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 blue 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 ⟦¬A⟧ subspace in method `bar` after the third configuration to explore the region with both values of C when A is not selected.

Further in the execution, the decision in the `while` statement (Line 15) depends on the tainted value of the variable `i` (implicit flow), in each loop iteration, resulting in ⟦A⟧, ⟦¬A⟧, which is consistent with `main`'s existing partition. Hence, the cross product does not change the partition. Similarly, the decision in Line 23 (region `bar`) repeatedly depends on data-flow taint C and control-flow taint A, resulting in the partition ⟦A ∧ ¬C⟧, ⟦¬A⟧, ⟦A ∧ C⟧.

After this first execution, we identified six distinct subspaces among the partitions of the three regions ⟦A⟧, ⟦¬A⟧, ⟦A ∧ B⟧, ⟦A ∧ ¬B⟧, ⟦A ∧ C⟧, and ⟦A ∧ ¬C⟧, of which ⟦A⟧, ⟦A ∧ ¬B⟧, and ⟦A ∧ ¬C⟧ were explored with the initial configuration. In the next iteration, we select a new configuration, for example {A,B,C}, to explore unseen subspaces in the regions and update partitions. In this case, however, no new partitions are found. We continue executing new configurations to explore unseen subspaces, possibly updating the regions' partitions, until we have explored all subspaces in the regions. After executing only 4 out of 16 configurations, for example {A, D}, {A, B, C}, {}, and {C}, we have explored at least one configuration from each subspace of each partition in each region, and the iterative analysis terminates. The subspaces derived for the three regions's partitions are $[\![A]\!]$, $[\![\neg A]\!]$, $[\![A \wedge \neg B]\!]$, $[\![A \wedge B]\!]$, $[\![\neg A \wedge \neg C]\!]$, $[\![\neg A \wedge C]\!]$, $[\![A \wedge \neg C]\!]$, $[\![A \wedge C]\!]$.

Discussion. Note how the iterative analysis explores regions independently and does not explore paths for configuration options that do not influence a region (e.g., we do not explore the interaction between B and C and never explore configurations specifically for D). Also, note how the taint analysis tracks both direct and indirect dependencies interprocedurally.

The iterative analysis is guaranteed to terminate, as it explores new configurations during each iteration. In the worst case, all configurations in the system (finite set) will be executed, but in practice often much fewer executions are needed.

Our algorithm will produce the same partitions independent of the order in which configurations are executed. All subspaces that are derived during any execution of the taint analysis will be derived at some point, because (1) we eventually explore all paths in each region and (2) we update the partition of each region with the commutative cross-product operation.

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 about 1 hour with the dynamic taint analysis, whereas around 300 configurations can be executed in the same time! In general, we observe 26× to 300× 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 configuration 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, where options determine, for example, whether duplicates are allowed and the durability characteristics of a transaction. The benchmark 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

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

3.6 Evaluation

To evaluate the efficiency and effectiveness of our white-box approach, as well as the design decisions made in **ConfigCrusher** and **Comprex**, we compare the two prototypes to each other

Table 3.2: Subject systems evaluated with ConfigCrusher and Comprex.

System	Domain	#SLOC	#Opt.	#Conf.
Pngtastic Counter	Image processor	1250	5	32
Pngtastic Optimizer	Image optimizer	2553	5	32
Elevator	SPL benchmark	575	6	20
Grep	Utility	2152	7	128
Kanzi	Compressor	20K	7	128
Email	SPL benchmark	696	9	40
Prevayler	Database	1328	9	512
Sort	Utility	2163	12	4096
H2	Database	142K	16	65K
Berkeley DB	Database	164K	16	65K
Apache Lucene	Index/Search	396K	17	131K
Density Converter[1] *(Interface)*	Image processor	1359	22	4.9M
Density Converter[1] *(Complete)*[2]	Image processor	49K	22	4.9M

Opt: configuration options; Conf: Configurations.

[1] The system is an interface to several libraries for processing images.

[2] We included and analyzed all Java dependencies in this version of the system.

and to numerous state-of-the-art performance-influence modeling approaches for configurable software systems. We evaluate the different approaches in terms of the cost to generate the models and the accuracy of the models, and discuss their interpretability. Specifically, we address the following research question:

RQ1: How does ConfigCrusher and Comprex compare to each other and to state-of-the-art performance-influence modeling approaches in terms of cost, accuracy, and interpretability?

3.6.1 Experimental Setup

Subject systems. We selected 13 configurable widely-used open-source Java systems that satisfy the following criteria: (a) systems from a variety of domains, (b) systems with binary and non-binary configuration 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 3.2 provides an overview of all subject systems.

We focus on a large subset of all configuration options that are potentially relevant for performance. We considered configuration options for which the systems' documentation indicated that they would affect performance, but excluded configuration options that might not influence performance, (e.g., `--help`). This selection is representative of common use cases where developers and users are interested in the performance behavior of many, but not all configuration options. Following the evaluation of state of the art approaches [Guo et al., 2017, Halin et al., 2018, He et al., 2020, Kaltenecker et al., 2019, Lillack et al., 2018, Medeiros et al., 2016, Sarkar et al., 2015, Siegmund et al., 2012a,b, 2013, 2015, Weber et al., 2021], we selected, for non-binary options, two different values and encoded the values as a binary option.

We executed a long-running benchmark either shipped with each system or a representative scenario of a developer or user analyzing the system's performance under different configurations. Appendix A describes each system in more detail, including the configuration options, scenario, and workload considered for the measuring performance.

Dynamic Taint Analysis. We changed the workload to run the the iterative dynamic taint analysis in **Comprex** with a smaller workload by factors ranging from 20 to 5000, depending on the system. Appendix A describes how we reduced the workload for each system in more detail.

Hardware. The performance measurements, and the static and dynamic taint analyses were executed on an Ubuntu 18.04 LTS desktop, with a 3.4 GHz 8-core Intel Core i7 processor, 16 GB of RAM, and Java HotSpot^{TM} 64-bit Server VM (v1.8.0_202).

Performance Measurement. To quantify accuracy, we measured the entire configuration space of all subject systems except for *Sort*, *H*2, *Berkeley DB*, and *Density Converter*, due to their intractably large configuration spaces. For these systems, we measured the performance of 2000 randomly selected configurations. When measuring performance, we executed each configuration five times and used the median to reduce the effects of measurement noise. We initiated one VM invocation per configuration, thus all measures include startup time [Georges et al., 2007]. For **Comprex**, we profiled each system with JProfiler's default sampling rate of 5ms.

State-of-the-art approaches. We compare **ConfigCrusher** and **Comprex** to numerous state-of-the-art performance-influence modeling approaches for configurable software systems. More specifically, we compared our prototypes to the Family-Based approach [Siegmund et al., 2013] and to combinations of 5 sampling and 10 learning approaches. For learners, we evaluate variations of linear regressions [Siegmund et al., 2012a,b, 2015], 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 10, 50, and 200 configurations, feature-wise sampling (i.e., enable one configuration option at a time), and pair-wise sampling (i.e., cover all combinations of all pairs of configuration options) [Medeiros et al., 2016]. We selected 10 random configurations to use random sampling in the systems with small configuration spaces. 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. The following list describes the Family-Based approach and the learners that we used, including the hyperparameters that we changed from the default values:

- Family-Based Performance Measurement: This approach is the only other white-box technique for analyzing the performance of configurable software systems. The approach uses a static mapping between configuration options to code regions, and instruments the system to measure the execution time spent in the regions. Subsequently, the approach executes the system once with all options selected, tracking how much each option contributes to the execution time.
- Simple linear regression: We used Scikit learn's `linear_model.LinearRegression` function. The function implements an ordinary least squares linear regression.

- Pair-wise simple linear regression: We used the same simple linear regression function above, but set `interaction_only=True` and `degree=2`.
- Lasso linear regression: We used Scikit learn's `linear_model.Lasso` function. The function implements a linear model trained with L1 prior as the regularizer, also known as the Lasso.
- Pair-wise Lasso linear regression: We used the same Lasso linear regression function above, but set `interaction_only=True` and `degree=2`.
- Stepwise linear regression: We used Matlab's `stepwiselm` function. The function implements a stepwise regression. We set `modelspec='linear'`.
- Elastic net linear regression: We used Scikit learn's `linear_model.ElasticNet` function. The function implements a linear regression with combined L1 and L2 priors as regularizers.
- Decision tree: We used Scikit learn's `tree.DecisionTreeRegressor` function. The function implements a decision tree regressor.
- Shallow decision tree: We used the same decission tree regressor function above, but set `max_depth=6` to have a maximum of 63 decisions.
- Random forest: We used Scikit learn's `ensemble.RandomForestRegressor` function. The function implements a random forest regressor.
- Multi-layer perceptron: We used Scikit learn's `neural_network.MLPRegressor` function. The function implements a multi-layer Perceptron regressor.

Note that we do not compare against approaches for selecting the fastest configuration [Kaltenecker et al., 2019, Oh et al., 2017, Zhu et al., 2017], as those approaches solve a pure optimization problem where modeling the entire configuration space is not necessary.

Cost Metric. We report the number of configurations executed to generate a model and time to measure configurations. For the learning approaches, we report the learning time. For **ConfigCrusher** and **Comprex**, we report the time to execute the taint analyses.

Accuracy Metric. We report the Mean Absolute Percentage Error (MAPE), which measures the mean difference between the values predicted by a model and the values actually observed (i.e., the baseline). Lower is better.

Interpretability. We intend the models generated with our approach to be used in performance understanding and debugging tasks. Hence, developers, as well as users, would benefit if the models are easy to interpret. Unfortunately, measuring interpretability of models is nontrivial and controversial. In the machine learning community, interpretability is an open research problem with an active community, but without a generally agreed measure or even definition for interpretability [Doshi-Velez and Kim, 2017, Lipton, 2018, Molnar, 2019].

Generally, interpretability captures the ability of humans to make predictions, understand predictions, or understand the decisions of a model [Molnar, 2019]. When the model is complex (e.g., the model includes numerous decisions), humans have more difficulty understanding the model directly. Some simpler forms of models are usually considered *inherently interpretable*, because humans can inspect and understand the models directly. For example, scientists have decades of experience using and interpreting *linear models* (e.g., much of empirical software engineering research relies on interpreting linear model coefficients). Models with more complex structures and very large numbers of decisions (e.g., deep neural networks with millions of weights or random forests with hundreds of trees), exceed human capacity for directly understanding the model. With these more complex models, the trend is to develop *post-hoc explanations*, where tools provide explanations for specific aspects of the model (e.g., the reason for a given prediction) without having to understand the model's internals [Lipton, 2018, Lundberg and Lee, 2017, Molnar, 2019, Ribeiro et al., 2016, Štrumbelj and Kononenko, 2014]. The use of post-hoc explanations is, however, controversial, as the explanations usually are only approximations that may be unreliable or even misleading [Rudin, 2019].

We do not attempt to quantify interpretability. Instead, we generally consider *sparse linear models*, with dozens of individual and interacting terms, 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 that 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 configuration 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, as they are an ensemble of numerous (e.g. 100) decision trees. Understanding random forests would require understanding the *average* effect of configuration options around all fairly deep trees.

Similar to random forests, *neural networks* are also not considered inherently interpretable. In our evaluation, we do not analyze the reliability or usefulness of post-hoc explanations.

To assure that the linear models that we produce are *sparse* and, hence, likely interpretable by humans, we report the number of terms they contain. As our algorithm to build these models does not include any machine learning and regularization, the algorithm detects and reports even minuscule amounts of measurement noise. Hence, we exclude all trivial terms that do not make meaningful contributions to the systems' performance. We report the number of terms (configuration options and interactions) that contribute, at least, 0.3 seconds, which is approximately 1% of the execution time of the default configurations of our subject systems.

Threats of Validity. Measurement noise cannot be excluded and may affect *all* results. We reduced this threat by repeating measurements on a dedicated machine and using the median.

The primary threat to external validity is the selection of subject systems. While we selected widely-used open-source Java configurable systems from different domains and sizes, in terms of SLOC and configuration space size, readers should be careful when generalizing results. For instance, most subject systems are multi-threaded, but all systems have mostly deterministic performance behavior. Additionally, we analyzed a single configurable software system, whereas systems composed of numerous configurable software systems, deployed in distributed environments, and implemented in different languages are beyond the scope of this dissertation.

The selected subset of configuration options might not affect performance at all, making modeling a trivial task. We selected options for which the systems' documentation or the configuration options' functionality indicated that they would affect performance, and we observed a wide range of execution times for the configurations that we measured.

Table 3.3: Cost of sampling configurations.

(a) Number sampled configurations.

SA	*PC*	*PO*	*EL*[1]	*GR*	*KA*	*EM*	*PR*	*SO*	*H2*	*BD*	*AL*	*DL*	*DC*
BF	32	32	20	128	128	40	512	2^{12}	2^{16}	2^{16}	2^{17}	2^{22}	2^{22}
FW	5	5	6	7	7	9	9	12	16	16	17	22	22
PW	16	16	9	29	29	11	46	79	137	137	154	254	254
R10	10	10	10	10	10	10	10	10	10	10	10	10	10
R50[2]	—	—	—	50	50	—	50	50	50	50	50	50	50
R200[3]	—	—	—	—	—	—	200	200	200	200	200	200	200
FB[4]	—	—	1	—	—	1	—	—	—	—	—	—	—
CC[5]	4	10	64	64	64	8	32	256	—	—	—	256	—
CP	4	8	64	36	20	8	26	70	64	144	26	88	88

(b) Time to sample configurations.

SA	*PC*	*PO*	*EL*	*GR*	*KA*	*EM*	*PR*	*SO*	*H2*	*BD*	*AL*	*DL*	*DC*
BF	~3m	~42m	~11m	~11m	~1h	~17m	~4h	~1d	~16d	~8d	~48d	~3y	~3y
FW	~27s	~2m	~50s	~22s	~2m	~24s	~3m	~13m	~2m	~5m	~9m	~8m	~8m
PW	~2m	~10m	~3m	~2m	~9m	~2m	~16m	~1h	~21m	~39m	~1h	~2h	~2h
R10	~1m	~5m	~3m	~1m	~5m	~2m	~4m	~11m	~2m	~2m	~8m	~2m	~2m
R50[2]	—	—	—	~4m	~29m	—	~18m	~53m	~16m	~9m	~27m	~10m	~10m
R200[3]	—	—	—	—	—	—	~2h	~4h	~1h	~36m	~2h	~41m	~41m
FB[4]	—	—	~50s	—	—	~1m	—	—	—	—	—	—	—
CC[5]	~22s	~11m	—[6]	~5m	~35m	~2m	~14m	~4h	—	—	—	~1h	—
CP	~22s	~9m	—[6]	~3m	~11m	~2m	~11m	~1h	~22m	~30m	~15m	~16m	~16m

The time to measure configurations for BF is extrapolated from 200 randomly selected configurations.

SA: Sampling; *PC*: *Pngtastic Counter*; *PO*: *Pngtastic Optimizer*; *EL*: *Elevator*; *GR*: *Grep*; *KA*: *Kanzi*; *EM*: *Email*; *PR*: *Prevayler*; *SO*: *Sort*; *BD*: *Berkeley DB*; *AL*: *Apache Lucene*; *DL*: *Density Converter (Interface)*; *DC*: *Density Converter (Complete)*; BF: Brute Force; FW: Feature wise; PW: Pair wise; R10: 10 random configurations; R50: 50 random configurations; R200: 200 random configurations; FB: Family-Based; **CC**: **ConfigCrusher**; **CP**: **Comprex**; s: seconds; m: minutes; h: hours; d: days; y: years;

[1] System has a feature model [Apel et al., 2013] that describes the valid configurations.

[2] Not applicable to systems with configuration spaces with fewer than 50 configurations.

[3] Not applicable to systems with configuration spaces with fewer than 200 configurations.

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

[5] The static taint analysis did finish executing after 24 hours.

[6] The taint analysis found that all configuration options interact in the system.

Table 3.4: **Cost of learning models or analyzing subject systems.**

AP	*PC*	*PO*	*EL*	*GR*	*KA*	*EM*	*PR*	*SO*	*H2*	*BD*	*AL*	*DL*	*DC*
* & SL	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s
* & PSL	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s
* & LL	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s
* & PLL	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s
FW & SLR	<1.0s	<1.0s	<1.0s	<1.0s	~1s	<1.0s	~1s	~2s	~4s	~8s	~9s	~20s	~20s
PW & SLR	~1s	~1s	<1.0s	~2s	~2s	~2s	~4s	~6s	~45s	~4m	~2m	~6m	~6m
R10 & SLR	<1.0s	<1.0s	<1.0s	<1.0s	~1s	<1.0s	~1s	~2s	~3s	~5s	~7s	~13s	~13s
R50 & SLR	—	—	—	~5s	~6s	—	~6s	~7s	~7s	~6s	~9s	~15s	~15s
R200 & SLR	—	—	—	—	—	—	~2m	~3m	~5m	~5m	~7m	~2m	~2m
* & ENL	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s
* & DT	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s
* & SDT	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s
* & RF	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s
* & NN	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s	≤0.3s
CC[1]	~8s	~30s	~12s	~10s	~12s	~13s	~12s	~21s	—	—	—	~42s	—
CP	~32s	~2m	~11m	~5m	~9m	~2m	~3m	~17m	~9m	~11m	~29m	~6m	~8m

AP: Approach; *PC*: *Pngtastic Counter*; *PO*: *Pngtastic Optimizer*; *EL*: *Elevator*; *GR*: *Grep*; *KA*: *Kanzi*; *EM*: *Email*; *PR*: *Prevayler*; *SO*: *Sort*; *BD*: *Berkeley DB*; *AL*: *Apache Lucene*; *DL*: *Density Converter (Interface)*; *DC*: *Density Converter (Complete)*; FW: Feature wise; PW: Pair wise; R10: 10 random configurations; R50: 50 random configurations; R200: 200 random configurations; SL: Simple linear regression; PSL: Pair-wise simple linear regression; LL: Lasso linear regression; PLL: Pair-wise Lasso linear regression; SLR: Stepwise linear regression; ENL: Elastic net linear regression; DT: Decision tree; SDT: Shallow decision tree; RF: Random Forest; NN: Multi-layer perceptron; **CC**: **ConfigCrusher**; **CP**: **Comprex**; s: seconds; m: minutes.

* Any sampling approach.

[1] The static taint analysis did finish executing after 24 hours.

Table 3.5: **MAPE comparison among Family-Based, ConfigCrusher, and Comprex (lower is better).**

AP	*PC*	*PO*	*EL*	*GR*	*KA*	*EM*	*PR*	*SO*	*H2*	*BD*	*AL*	*DL*	*DC*
FB[1]	—	—	**2.7**	—	—	**2.3**	—	—	—	—	—	—	—
CC[2]	**1.1**	**1.1**	—[3]	**3.6**	**2.7**	23.0	**9.2**	**1.6**	—	—	—	**4.3**	—
CP	**1.2**	**1.6**	—[3]	**7.9**	**2.8**	39.2	**9.8**	**6.8**	**2.9**	**5.0**	**3.2**	**8.9**	**9.4**

AP: Approach; *PC*: *Pngtastic Counter*; *PO*: *Pngtastic Optimizer*; *EL*: *Elevator*; *GR*: *Grep*; *KA*: *Kanzi*; *EM*: *Email*; *PR*: *Prevayler*; *SO*: *Sort*; *BD*: *Berkeley DB*; *AL*: *Apache Lucene*; *DL*: *Density Converter (Interface)*; *DC*: *Density Converter (Complete)*; FW: Feature wise; PW: Pair wise; R10: 10 random configurations; R50: 50 random configurations; R200: 200 random configurations; FB: Family-Based; **CC**: **ConfigCrusher**; **CP**: **Comprex**. **Cells** indicate similarly low errors.

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

[2] The static taint analysis did finish executing after 24 hours.

[3] The taint analysis found that all configuration options interact in the system.

Table 3.6: MAPE comparison among linear regressors, ConfigCrusher, and Comprex (lower is better).

AP	*PC*	*PO*	*EL*	*GR*	*KA*	*EM*	*PR*	*SO*	*H2*	*BD*	*AL*	*DL*	*DC*
FW & SL	**0.4**	14.9	41.1	32.8	**3.0**	105.9	111.2	94.4	107.5	104.3	**7.7**	1417.6	1466.7
PW & SL	**1.3**	**7.2**	**1.4**	107.1	**2.7**	43.0	30.5	663.2	129.5	48.9	**7.0**	1478.5	1430.0
R10 & SL	**1.7**	**7.4**	**1.4**	30.3	**2.6**	44.2	108.6	87.0	193.2	47.2	**7.2**	1347.2	1298.3
R50 & SL[1]	—	—	—	22.0	**3.2**	—	95.0	85.7	180.1	45.6	**5.2**	578.7	623.1
R200 & SL[2]	—	—	—	—	—	—	87.4	84.0	186.6	32.7	**4.3**	487.2	424.8
FW & PSL	**2.6**	19.1	56.5	34.9	**2.7**	104.0	110.1	91.3	107.5	550.7	**7.7**	1427.6	1466.7
PW & PSL	**2.0**	**5.2**	**1.8**	90.0	**3.4**	50.1	29.5	654.6	109.3	135.0	11.7	1318.6	1354.2
R10 & PSL	**1.9**	**8.5**	**1.7**	32.8	**4.6**	53.4	110.7	86.2	134.3	329.4	**7.1**	437.2	449.3
R50 & PSL[1]	—	—	—	22.2	**2.9**	—	93.6	84.7	133.4	39.7	**6.3**	615.7	567.1
R200 & PSL[2]	—	—	—	—	—	—	90.3	84.1	131.9	18.3	**4.5**	1137.2	1030.3
FW & LL	**1.5**	15.7	44.2	25.4	**2.9**	104.6	105.6	97.5	106.5	117.6	**9.1**	1578.2	1651.5
PW & LL	**2.0**	**8.4**	**2.9**	94.8	**3.8**	52.6	34.5	660.7	109.0	111.2	**9.2**	1587.2	1451.6
R10 & LL	**0.9**	**8.7**	**8.5**	25.1	**3.1**	58.6	106.5	88.1	103.4	43.8	**6.5**	534.9	587.4
R50 & LL[1]	—	—	—	25.7	**5.0**	—	99.4	87.0	106.5	39.0	**6.3**	498.3	451.8
R200 & LL[2]	—	—	—	—	—	—	94.3	84.5	80.6	46.3	**7.1**	284.2	243.5
FW & PLL	**0.4**	19.7	58.0	34.0	**3.2**	104.6	108.0	93.3	106.5	117.6	**9.1**	1581.9	1651.5
PW & PLL	**2.5**	**5.0**	**3.0**	100.1	**2.3**	45.5	31.8	625.0	109.0	111.2	**9.2**	1451.6	1424.3
R10 & PLL	**1.9**	**7.4**	**7.4**	31.8	**4.6**	49.3	107.5	88.5	103.3	49.3	**8.3**	541.3	548.0
R50 & PLL[1]	—	—	—	26.2	**3.3**	—	93.3	87.2	106.5	44.7	**6.3**	483.1	451.8
R200 & PLL[2]	—	—	—	—	—	—	92.0	83.2	80.5	46.3	**7.1**	273.4	243.5
FW & SLR	**0.8**	19.7	51.1	32.1	**1.9**	100.0	111.2	90.0	129.3	768.7	**7.9**	1648.2	1596.0
PW & SLR	**2.0**	**0.9**	**1.5**	114.7	**1.3**	44.2	29.2	653.0	113.3	34.2	**4.7**	1524.8	1596.0
R10 & SLR	**2.7**	**0.8**	**1.5**	31.8	**2.6**	48.3	113.6	84.7	124.3	28.3	**5.9**	1248.3	1289.2
R50 & SLR[1]	—	—	—	26.3	**1.4**	—	95.2	84.5	124.1	19.7	**4.5**	1087.3	1037.2
R200 & SLR[2]	—	—	—	—	—	—	93.1	80.3	93.9	14.9	**2.9**	397.5	434.5
FW & ENL	**0.8**	16.9	55.3	30.8	**4.9**	109.2	117.9	97.3	106.5	118.1	**9.1**	617.3	599.1
PW & ENL	**0.5**	**4.3**	**2.9**	92.7	**3.4**	53.2	31.8	646.6	109.0	108.5	**9.2**	615.7	660.8
R10 & ENL	**0.4**	**4.4**	**4.3**	29.7	**3.1**	54.1	117.4	88.7	158.3	52.1	**6.8**	489.3	478.2
R50 & ENL[1]	—	—	—	23.8	**4.8**	—	99.3	84.5	145.3	47.4	**6.5**	457.4	411.3
R200 & ENL[2]	—	—	—	—	—	—	86.0	81.3	176.0	48.2	**6.9**	314.2	293.7
CC[3]	**1.1**	**1.1**	—[4]	**3.6**	**2.7**	23.0	**9.2**	**1.6**	—	—	—	**4.3**	—
CP	**1.2**	**1.6**	—[4]	**7.9**	**2.8**	39.2	**9.8**	**6.8**	**2.9**	**5.0**	**3.2**	**8.9**	**9.4**

AP: Approach; *PC*: *Pngtastic Counter*; *PO*: *Pngtastic Optimizer*; *EL*: *Elevator*; *GR*: *Grep*; *KA*: *Kanzi*; *EM*: *Email*; *PR*: *Prevayler*; *SO*: *Sort*; *BD*: *Berkeley DB*; *AL*: *Apache Lucene*; *DL*: *Density Converter (Interface)*; *DC*: *Density Converter (Complete)*; FW: Feature wise; PW: Pair wise; R10: 10 random configurations; R50: 50 random configurations; R200: 200 random configurations; SL: Simple linear regression; PSL: Pair-wise simple linear regression; LL: Lasso linear regression; PLL: Pair-wise Lasso linear regression; SLR: Stepwise linear regression; ENL: Elastic net linear regression; **CC**: **ConfigCrusher**; **CP**: **Comprex**. **Cells** indicate similarly low errors.

[1] Not applicable to systems with configuration spaces with fewer than 50 configurations.

[2] Not applicable to systems with configuration spaces with fewer than 200 configurations.

[3] The static taint analysis did finish executing after 24 hours.

[4] The taint analysis found that all configuration options interact in the system.

Table 3.7: MAPE comparison among tree regressors, ConfigCrusher, and Comprex (lower is better).

AP	*PC*	*PO*	*EL*	*GR*	*KA*	*EM*	*PR*	*SO*	*H2*	*BD*	*AL*	*DL*	*DC*
FW & DT	**0.4**	18.7	50.5	28.3	**4.3**	94.3	98.3	68.7	125.1	143.1	11.5	1724.3	1768.7
PW & DT	**2.0**	**8.3**	**2.8**	105.1	**2.9**	35.5	28.9	103.8	127.7	36.9	**1.0**	101.4	109.2
R10 & DT	**2.1**	**4.4**	**2.8**	27.9	**4.6**	35.8	100.1	42.1	12.7	28.7	**9.8**	872.6	848.6
R50 & DT[1]	—	—	—	25.8	**2.3**	—	64.3	46.7	**3.0**	**6.8**	**2.4**	56.3	57.7
R200 & DT[1]	—	—	—	—	—	—	38.1	26.5	**1.0**	16.0	**0.5**	19.6	18.2
FW & SDT	**2.9**	19.7	57.0	30.8	**4.8**	96.2	101.3	75.2	103.5	123.1	**8.7**	415.7	406.3
PW & SDT	**2.9**	**6.1**	**2.4**	108.4	**3.8**	37.4	29.8	136. 6	123.8	43.9	11.5	1821.8	1779.1
R10 & SDT	**1.9**	**5.8**	**2.4**	28.9	**5.0**	39.2	102.8	48.9	16.3	37.3	**3.2**	242.4	237.3
R50 & SDT[1]	—	—	—	27.6	**3.1**	—	76.2	53.4	**2.8**	21.8	**1.6**	148.7	151.5
R200 & SDT[2]	—	—	—	—	—	—	33.7	29.5	**1.1**	16.1	**0.6**	21.6	18.8
FW & RF	**2.1**	**9.0**	40.9	29.4	**4.8**	92.6	95.7	55.3	119.0	106.1	**8.7**	1239.7	1185.9
PW & RF	**1.1**	**4.2**	**2.9**	99.7	**4.3**	32.3	27.3	98.3	124.6	46.9	**4.0**	28.4	27.3
R10 & RF	**1.7**	**7.6**	**2.8**	29.0	**4.6**	33.1	97.3	28.1	329.3	24.8	**5.3**	72.4	81.3
R50 & RF[1]	—	—	—	21.8	**4.1**	—	41.2	23.9	**1.1**	16.4	**0.4**	62.1	59.9
R200 & RF[2]	—	—	—	—	—	—	**8.3**	**4.8**	**0.7**	**1.1**	**0.3**	**5.8**	**5.5**
FW & NN	**0.5**	18.5	61.2	34.5	**4.6**	107.8	117.7	109.3	440.9	433.2	56.3	1218.7	1265.4
CC[3]	**1.1**	**1.1**	—[4]	**3.6**	**2.7**	23.0	**9.2**	**1.6**	—	—	—	**4.3**	—
CP	**1.2**	**1.6**	—[4]	**7.9**	**2.8**	39.2	**9.8**	**6.8**	**2.9**	**5.0**	**3.2**	**8.9**	**9.4**

AP: Approach; *PC*: *Pngtastic Counter*; *PO*: *Pngtastic Optimizer*; *EL*: *Elevator*; *GR*: *Grep*; *KA*: *Kanzi*; *EM*: *Email*; *PR*: *Prevayler*; *SO*: *Sort*; *BD*: *Berkeley DB*; *AL*: *Apache Lucene*; *DL*: *Density Converter (Interface)*; *DC*: *Density Converter (Complete)*; FW: Feature wise; PW: Pair wise; R10: 10 random configurations; R50: 50 random configurations; R200: 200 random configurations; DT: Decision tree; SDT: Shallow decision tree; RF: Random Forest; **CC**: **ConfigCrusher**; **CP**: **Comprex**. **Cells** indicate similarly low errors.

[1] Not applicable to systems with configuration spaces with fewer than 50 configurations.

[2] Not applicable to systems with configuration spaces with fewer than 200 configurations.

[3] The static taint analysis did finish executing after 24 hours.

[4] The taint analysis found that all configuration options interact in the system.

Table 3.8: MAPE comparison among a neural network, ConfigCrusher, and Comprex (lower is better).

AP	*PC*	*PO*	*EL*	*GR*	*KA*	*EM*	*PR*	*SO*	*H2*	*BD*	*AL*	*DL*	*DC*
FW & NN	**0.5**	18.5	61.2	34.5	**4.6**	107.8	117.7	109.3	440.9	433.2	56.3	1218.7	1265.4
PW & NN	**2.4**	**7.6**	**3.0**	119.9	**3.8**	58.2	33.6	689.1	406.1	464.2	65.8	1727.3	1848.1
R10 & NN	**2.8**	**8.7**	**3.0**	32.4	**5.1**	62.3	115.1	91.7	237.2	85.7	46.3	489.1	487.3
R50 & NN[1]	—	—	—	29.8	**3.2**	—	98.7	87.3	152.5	72.4	31.2	396.7	415.7
R200 & NN[2]	—	—	—	—	—	—	73.3	84.9	212.0	80.5	31.3	304.2	289.0
CC[3]	**1.1**	**1.1**	—[4]	**3.6**	**2.7**	23.0	**9.2**	**1.6**	—	—	—	**4.3**	—
CP	**1.2**	**1.6**	—[4]	**7.9**	**2.8**	39.2	**9.8**	**6.8**	**2.9**	**5.0**	**3.2**	**8.9**	**9.4**

AP: Approach; *PC*: *Pngtastic Counter*; *PO*: *Pngtastic Optimizer*; *EL*: *Elevator*; *GR*: *Grep*; *KA*: *Kanzi*; *EM*: *Email*; *PR*: *Prevayler*; *SO*: *Sort*; *BD*: *Berkeley DB*; *AL*: *Apache Lucene*; *DL*: *Density Converter (Interface)*; *DC*: *Density Converter (Complete)*; FW: Feature wise; PW: Pair wise; R10: 10 random configurations; R50: 50 random configurations; R200: 200 random configurations; NN: Multi-layer perceptron; **CC**: **ConfigCrusher**; **CP**: **Comprex**. **Cells** indicate similarly low errors.

[1] Not applicable to systems with configuration spaces with fewer than 50 configurations.

[2] Not applicable to systems with configuration spaces with fewer than 200 configurations.

[3] The static taint analysis did finish executing after 24 hours.

[4] The taint analysis found that all configuration options interact in the system.

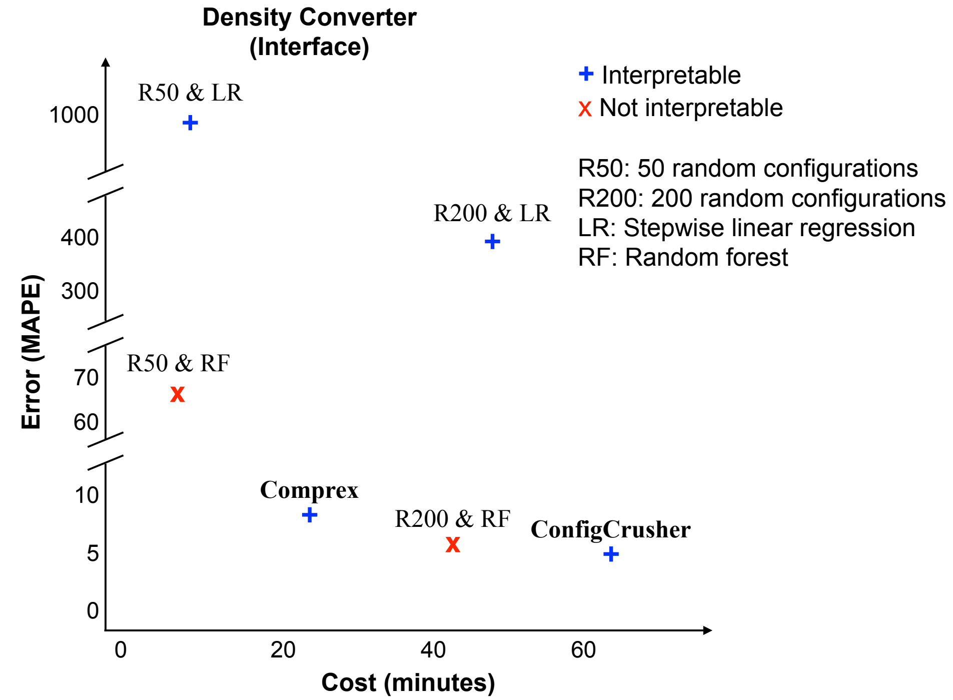


Figure 3.7: Overview of the results of our white-box performance-influence modeling approaches. ConfigCrusher and Comprex build models with similar accuracy (low error), but Comprex is typically more efficient (low cost), in terms of the number of configurations that need to be measured, despite running an iterative dynamic taint analysis instead of a single execution of a static taint analysis. The models generated with our approaches are similarly accurate to the most accurate and expensive sampling and learning approaches (with large enough samples), but our models are interpretable and can often be built more efficiently with Comprex. Additionally, our approaches generate models that are typically more accurate that other sampling and learning approaches that build interpretable linear models.

3.6.2 Results

We report the results in Table 3.3 (sampling cost), Table 3.4 (learning/analysis cost), and Table 3.5, Table 3.6, Table 3.7, and Table 3.8 (accuracy), and present an overview of the results in Figure 3.7. Overall, **ConfigCrusher** and **Comprex** build models with similar accuracy, but **Comprex** is typically more efficient in terms of the number of configurations that need to be measured, despite running an iterative dynamic taint analysis compared to a single execution of a static taint analysis. The models generated with our approaches are similarly accurate to the most accurate and expensive sampling and learning approaches (with large enough samples), but our models are interpretable and can often be build more efficiently with **Comprex**, despite the cost of the taint analysis. Additionally, both **ConfigCrusher** and **Comprex** usually outperform other approaches that build linear models.

Although, **ConfigCrusher** and **Comprex** generated models with similar accuracy, **ConfigCrusher** was not able to model the performance of 4 subject systems. These systems have over 100K SLOC, and FlowDroid's static taint analysis could not analyze the massive call graphs within 24 hours. By contrast, we were able to run **Comprex**'s dynamic taint analysis on all subject systems efficiently, and accurately model their performance.

In the subject systems that our approaches were able to analyze, **Comprex** was typically more efficient than **ConfigCrusher**, in terms of the number of configurations to measure. The efficiency originates from **Comprex**'s *dynamic* taint analysis and *separately* tracking control- and data-flow taints, which helped us explore nested decisions more efficiently when some interactions only occur in certain paths. By contrast, **ConfigCrusher**'s static taint analysis only indicated the configuration options and interactions that *might* affect the decision in control-flow statements. Hence, we needed to explore all combinations of configurations options that reached control-flow statements.

While random forest with 200 samples often generated more accurate models than **ConfigCrusher** and **Comprex**, our approaches generate local and interpretable models, and **Comprex** was usually more efficient than the black-box approach; in some cases, building models in half the time. The efficiency originates from our white-box analysis, in which we identify a small number of relevant configurations to capture the performance-relevant interactions. By contrast, black-box approaches perform worse on small samples (e.g., compare the results of using 10, 50, 200 samples).

The linear models produced with **ConfigCrusher** and **Comprex** are moderate in size, with 10 to 72 performance-relevant terms (configuration options or interactions) depending on the subject system. Our models are similar in size to models learned with variations of linear regression approaches using random samples (e.g., 6 to 30 terms using stepwise linear regression with 200 samples). At this size, we argue that manual inspection of the models is still plausible. More importantly, the performance influences can be mapped to a small number of specific regions (e.g., 1 to 24 regions depending on the system), which we later use to help developer debug the performance of configurable software systems (see Chapter 4).

For Pngtastic Counter, Pngtastic Optimizer, Kanzi, and Apache Lucene, most black-box approaches produced accurate models with low cost. Upon inspection of the results, we discovered that the execution time of these systems were clustered in a few groups. For example, the performance of Kanzi under all configurations was either about 4 or about 61 seconds. Hence, a few samples were needed to learn the clustered execution times. By contrast, the black-box approaches needed to measure more configurations to build accurate models in the other subject systems, which had more complex performance behaviors.

We did not model the performance of Elevator with our approaches, as the taint analyses indicated that all options interact in the system (i.e., our approach equals a brute-force approach). However, the system was purposely built to have all options interact [Kim et al., 2013, Meinicke et al., 2016, Souto et al., 2017], and was used to showcase the benefits of the Family-Based approach.

Regarding **ConfigCrusher**'s and **Comprex**'s prediction error of Email, the system has a feature model [Apel et al., 2013] that describes its valid configurations. Since the invalid configurations were not executed, our approaches did not have all the information for each region to generate an accurate model. Despite missing information, our approaches were able to produce more accurate results than the other approaches, except for the Family-Based approach.

Only for Elevator and Email, the Family-Based approach remains the most efficient and accurate approach, but, at the same time, the most limited one in terms of which systems can be analyzed.

*Summary RQ1: **ConfigCrusher** and **Comprex** generate performance-influence models with similar accuracy, but **Comprex** is typically more efficient, and can also model the performance of medium- to large-scale systems. Additionally, the models produced with our approaches have comparable accuracy to those models generated with the most accurate and expensive black-box approaches, but **Comprex** is often more efficient. Furthermore, our approaches usually outperform other black-box approaches that produce linear models. Our approaches also generate models that are interpretable and can be mapped to specific regions.*

Contribution - White-box performance-influence modeling: We developed a white-box approach and two prototypes that combine the insights of *compositionality* and *compression* to efficiently build accurate and interpretable performance-influence models.

Thesis contribution: A white-box performance-influence modeling approach that efficiently and accurately models the performance of configurable software systems. The approach generates interpretable models, which can help users make informed configuration decisions to run configurable software systems more efficiently, thus reducing energy consumption and operational costs.

3.7 Discussion

Based on the results our empirical evaluation, we now discuss the impact of the design decisions made in **ConfigCrusher** and **Comprex** on the types of configurable software systems that can be analyzed and the cost to build models.

3.7.1 Static vs. Dynamic Taint Analysis

The type of taint analysis used resulted in different tradeoffs when modeling the performance of configurable software systems. On one hand, we executed the *static* taint analysis *once* in a few seconds, which reduced the cost to determine the configurations that we need to measure for building models. However, the static taint analysis was limited to relatively *small systems* and could not efficiently explore nested decisions. On the other hand, we executed *multiple* configurations to run the *dynamic* taint analysis, but we were able to analyze all subject systems of different sizes by executing the analysis with a reduced workload size. While both analyses allowed us to generate accurate models, our results indicate that a dynamic taint analysis can also be used to model the performance of large configurable software systems, in terms of SLOC.

Dynamic Taint Analysis Overhead

Reducing the size of the workload to reduce the cost of running the iterative dynamic taint analysis may produce inaccurate models if we miss some interactions. However, this decision will likely have a low impact in highly repetitive workloads. More importantly, our empirical evaluation suggest that inaccuracies on the regions' partitions resulted in, at most, minor accuracy degradation in our performance-influence models, given the consistently high accuracy achieved across all subject systems.

Furthermore, we used a debugging strategy to identify potential effects of inaccurate partitions. As discussed in Section 3.2.3, multiple executions of configurations within the same subspace of a region's partition must have the same performance behavior. Significant performance differences, beyond normal measurement noise, indicate that the region's partition might be inaccurate and not capturing all relevant configuration options and interactions.

Specifically, we analyzed the performance measurements of all regions, searching for regions with a significant performance influence (> 0.1ms in any observed configuration) and with a high variance among the execution times of the same subspace in a region (coefficient of variation of the execution times > 1.0). Among the 94 instrumented regions analyzed with **ConfigCrusher** and the 2771 method-level regions analyzed with **Comprex**, we found only 10 of such regions, 2 in Prevayler and 8 in Apache Lucene.

We executed the iterative dynamic taint analysis with the regular workload on all subject systems to determine whether we identify different partitions in the regions. In Prevayler, the regular workload resulted in the same partitions for the 2 regions with high performance variance, whereas in Apache Lucene, additional subspaces were partitioned in 7 out of 8 regions with high performance variance, due to slightly different taints in the shorter workload. We conjecture that, since Prevayler writes data to disk, there might be some interactions in system calls, which we do not analyze. While the missing subspaces in Apache Lucene slightly decreased the MAPE from 3.2 to 3.0 (as a result of slight changes in the coefficients in the model, not from new configuration options or interactions becoming performance relevant), the time to run the dynamic taint analysis with the regular workload is *extremely* expensive; 11 hours instead of 29 minutes to run the same 26 configurations. In fact, the dynamic taint analysis ran for over 6 hours in all but H2, Berkeley DB, and both versions of Density Converter, in which the analysis did not finish executing after 24 hours!

We argue that the extremely high cost and inability to use the regular workload does not outweigh a potential slight accuracy increase of the already highly accurate models that we generated. These results support our conjecture that inaccuracies on the regions' partition caused by using a small workload are only a minor issue in practice.

Summary: A static taint analysis can be executed only once, but is limited to relatively small systems and cannot efficiently explore nested decisions. A dynamic taint analysis requires multiple executions, but can analyze systems of various sizes, in terms of SLOC, when using a reduced workload, which does not affect the accuracy of the models that we generate.

Table 3.9: Number of regions and configurations to measure with compression at different region granularities.

	Control-flow		Method		System	
System	#Reg.	#Conf.	#Reg.	#Conf.	#Reg.	#Conf.
Pngtastic Counter	36	**4**	22	**4**	1	16
Pngtastic Optimizer	397	**8**	124	**8**	1	32
Elevator[1]	42	**64**	28	**64**	1	64
Grep	46	**36**	31	**36**	1	64
Kanzi	128	**20**	59	**20**	1	64
Email	60	**8**	35	**8**	1	8
Prevayler	147	**26**	78	**26**	1	32
Sort	166	**70**	89	**70**	1	256
H2	2483	**64**	932	**64**	1	256
Berkeley DB	2152	**144**	718	**144**	1	2048
Apache Lucene	1654	**26**	551	**26**	1	16384
Density Converter (Interface)	124	**88**	42	**88**	1	4608
Density Converter (Complete)	190	**88**	62	**88**	1	4608

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

[1] The taint analysis found that all configuration options interact in the system.

3.7.2 Granularity of Regions, Compression, and Measuring Performance

Considering different granularities of regions yielded different tradeoffs between compression potential and effort to measure 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 system. We overcame this issue by optimizing how we instrumented regions. 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 and the *overhead* to perform these measurements.

Number of configurations. To explore the number of configurations to measure at different granularities, 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 system by combining the partitions of all methods.

Table 3.9 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 system 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 does 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 8 out of 2771 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 configuration options). However, in all cases, the additional subspaces were already explored in other parts of the system. Hence, no additional configurations needed to be explored.

We conclude that fined-grained compression is highly effective, but that control-flow granularity does not appear to offer significant compression benefits over method granularity.

Measurement overhead. Measuring performance at different granularities requires different strategies, each with vastly different amounts of measurement overhead. Measuring at the *system* level is cheap, as a single end-to-end measurement is sufficient to measure the entire system (e.g., Unix `time`). At finer granularities, multiple measurements of different parts of the system are required. Instrumenting the system at *control-flow statements* and corresponding post-dominators leads to significant measurement overhead, as the measurement instructions are executed frequently (similar to an instrumentation profiler). To overcome this issue, we optimized how we instrument regions, but the calculation requires several minutes to run, and we need to process the execution time for each region.

By contrast, measuring the performance of numerous *methods* is inexpensive with a *sampling profiler*, for which we observed a mostly linear overhead of about 8% in our subject systems.

Summary: Compression at method and control-flow granularities is highly efficient to reduce measurement effort. Compression at method granularity provides a good compromise between compression potential and measurement overhead.

3.7.3 Limitations of our White-box Approach

While our white-box approach can generate similarly accurate models to some black-box approaches, there are several limitations with our approach, and several situations in which other approaches might be more applicable or preferable to use.

There is a lot of engineering effort to use our white-box approach. First, our approach needs access to the source code, which might be difficult, impossible, or not necessary to have in some situations or use cases. Additionally, a taint analysis needs to be setup and executed, which requires defining sources and sinks. In our prototypes, we manually defined sources and automatically instrumented control-flow statements to define sinks. If the static taint analysis is used, then large-scale systems cannot be analyzed. If the dynamic taint analysis is used, the workload of the system needs to be reduced. Considering control-flow statements as regions requires instrumenting the system to measure the performance of regions, which can cause measurement overhead. Regardless of whether performance is measured with instrumentation or an off-the-shelf profiler, there is a lot of infrastructure that we implemented to map performance measurements to individual regions, and to build local and global performance-influence models. By contrast, black-box approaches can simply measure the end-to-end performance of a compiled executable, using established sampling approaches, and use established machine learning algorithms to generate performance-influence models.

Due to the complicated setup of our white-box approach, we envision (and show in Chapter 4) that developers would be the primary consumers of our approach. Most end users would not want to go through a complicated setup to analyze the performance of a system. By contrast, end users could simply collect the end-to-end performance of a system and use machine learning packages, such as in Matlab or Scikit learn, to generate models to analyze the performance of a system.

While in Chapter 4 we show that interpretable local and global models help developers debug the performance of configurable software systems, there are some scenarios (discussed in Section 2.3.1) in which such type of models are probably not necessary.

In the simplest case, a user wants to optimize the performance of a system by selecting the fastest configuration. While global performance-influence models have been used for optimization [Guo et al., 2013, Nair et al., 2017, Oh et al., 2017, Zhu et al., 2017], other approaches 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 understanding the entire configuration space is not necessary.

In other scenarios, only accurate predictions of individual configurations are needed. Scenarios include *automatic reconfiguration* and *runtime adaptation*, where there is no human-in-the-loop and online search is impractical. In these scenarios, the model's prediction accuracy over the entire configuration space is important, but understanding the structure of the model is irrelevant. 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 preferable, as they build accurate models, with a large enough number of sampled configurations.

While running our white-box approach requires a complicated process, and there are other approaches that are preferable in some situations, the evaluation in this chapter (Section 3.6), as well as two user studies in Chapter 4 demonstrate that our white-box approach provides accurate and relevant information for the scenario that we address in this dissertation: Helping developers debug the performance of configurable software system.

3.8 Summary

In this chapter, we presented *compositionality* and *compression*, the key insights for efficiently and accurately modeling the global and local performance of configurable software systems. Additionally, we presented how we *tailor* a *taint analysis* to identify how configuration options and their interactions influence the performance of code regions. Based on different alternatives to implement our white-box approach, in terms of the type of taint analysis to use and the granularity of regions, we presented two prototypes: **ConfigCrusher** and **Comprex**. An empirical evaluation of our two prototypes demonstrated that a white-box analysis can 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 **Comprex** was implemented, can also scale the modeling to medium- and large-scale configurable software systems.

Our white-box approach contributes to our thesis goal of reducing the energy consumption and operational costs of running configurable software systems by providing users accurate and interpretable models, which can be used to make informed configuration decisions to run systems more efficiently.

In Chapter 2, we suggested using interpretable global and local performance-influence models to help developers debug, but noted limitations of existing approaches. Our white-box approach overcame these limitations. In Chapter 4, we describe how we tailor the interpretable global and local models that we generate, as well as CPU profiling and program slicing, to help developers debug the performance of configurable software systems.

Chapter 4

Tailoring Ingredients for Debugging Performance: Information Providers

In Chapter 2, we conducted an exploratory study, in which we identified that developers struggle to find relevant information to identify influencing options; the configuration options or interactions causing an unexpected performance behavior, locate option hotspots; the methods where configuration options affect the performance of the system, and trace the cause-effect chain; how influencing options are used in the implementation to directly and indirectly affect the performance of option hotspots. Based on these findings, we suggested ingredients (i.e., techniques and information sources) that can be tailored to support the above needs: Interpretable global and local performance-influence models can help developers identify influencing options and locate option hotspots, and CPU profiling and program slicing can help developers trace the cause-effect chain.

While we discussed that CPU profiling and program slicing can be tailored without major modifications, we noted the limitations of existing performance-influence modeling approaches, in terms of tradeoffs among the cost to build models, and the accuracy and interpretability of the models. Consequently, in Chapter 3, we presented and evaluated a white-box approach that overcame those limitations, as the approach efficiently builds accurate and interpretable local and global performance-influence models.

We now continue our human-centered approach, which started with our exploratory user study in Chapter 2, to identify solutions to support developers' actual needs in the process of debugging the performance of configurable software systems; specifically, presenting how we tailor the ingredients and evaluate that the information that we provide support developers' needs. In this chapter, we first describe how we *design* and *implement information providers* to support developers' needs, *tailoring* interpretable **G**lobal and **L**ocal performance-**I**nfluence **M**odels, CPU **P**rofiling, and program **S**licing, in a tool called **GLIMPS**. Afterwards, we conduct two user studies to *validate* and *confirm* that the designed information providers are useful to developers when debugging the performance of complex configurable software systems, in terms of supporting their needs and speeding up the debugging process.

In summary, we make the following contributions:

- The design of information providers, tailoring interpretable **G**lobal and **L**ocal performance-**I**nfluence **M**odels, CPU **P**rofiling, and program **S**licing, to support the information needs

Table 4.1: Ingredients that support developers' information needs when debugging the performance of configurable software systems.

Information Need	Description	Tailored Ingredients
Influencing Options	Which configuration options influence the performance of the system?	Interpretable global performance-influence models
Option Hotspots	Where do configuration options influence the performance of the system?	Interpretable local performance-influence models
Cause-Effect Chain	How are influencing options used in the implementation to directly and indirectly influence the performance of option hotspots?	CPU Profiling and Program Slicing

that developers have when debugging the performance of configurable software systems.

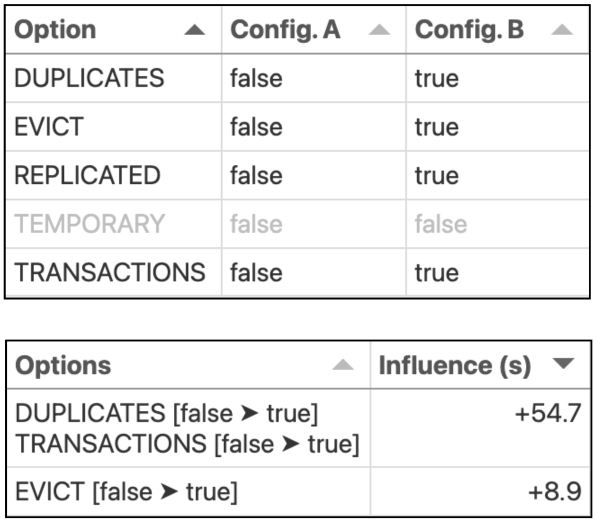
- Two empirical evaluations that demonstrate the usefulness of the designed information providers.
- A prototype tool, **GLIMPS**, that implements the designed information providers to help developers debug.

The rest of this chapter is organized as follows: We first describe how we *design* and *implement information providers* to support the information needs that developers have when debugging the performance of configurable software systems, *tailoring* the ingredients that we identified in Chapter 2 (Section 4.1). Then, we conduct a *validation* user study and a *confirmatory* user study to evaluate that the designed information providers actually support developers' information needs and speed up the process of debugging the performance of complex software systems (Section 4.2).

This chapter shares material with a conference submission under review at the time of writing: "On Debugging the Performance of Configurable Software Systems: Developer Needs and Tailored Tool Support" [Velez et al., 2021a].

4.1 Supporting Information Needs

We aim to support developers in identifying influencing options, locating option hotspots, and tracing the cause-effect chain. To this end, we design *information providers*, adapting *interpretable global* and *local performance-influence modeling*, *CPU profiling*, and *program slicing* to provide relevant information for supporting the above information needs. We implement the designed information providers in a cohesive prototype called **GLIMPS**, which can assist developers debug the performance of configurable software systems; primarily, by comparing the performance behavior of a system between a problematic configuration and a non-problematic configuration (see the results of our exploratory study in Section 2.2). Table 4.1 shows which information needs are supported by the ingredients that we tailor for designing information providers.



Option	Config. A	Config. B
DUPLICATES	false	true
EVICT	false	true
REPLICATED	false	true
TEMPORARY	false	false
TRANSACTIONS	false	true

Options	Influence (s)
DUPLICATES [false ➤ true] TRANSACTIONS [false ➤ true]	+54.7
EVICT [false ➤ true]	+8.9

Figure 4.1: Our tool highlights differences in the values of configuration options selected between two configurations, and shows the influencing options of the changes from one configuration to (▶) another configuration.

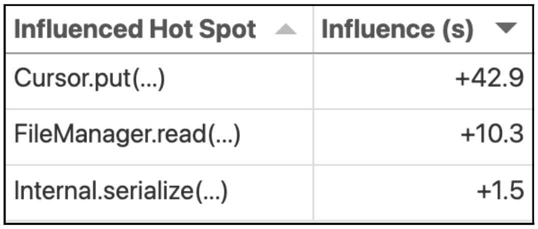
4.1.1 Identifying Influencing Options

As discussed in Section 2.3, to help developers identify the influencing options that cause an unexpected performance behavior, we use *interpretable global performance-influence models*, which we generate with our white-box approach described in Chapter 3. For instance, the model $\mathtt{m} = 4.6 + 54.7 \cdot \textsc{Duplicates} \cdot \textsc{Transactions} + 8.9 \cdot \textsc{Evict} + 3.5 \cdot \textsc{Temporary}$ explains the influence of the configuration options and their interactions on the performance of a system

We adapt this ingredient to design an *information provider* that shows developers influencing options; specifically, *which* and *how differences* between configurations (e.g., a problematic configuration and a non-problematic configuration) influence the performance of a system. In our implementation, this information provider highlights the differences in the values of configuration options selected between two configurations, and shows the influencing options between the configurations. If changes between the configurations are not shown, then the changes do not influence the performance of the system.[1]

Example. Figure 4.1 shows a screenshot of our tool highlighting the differences in the values of the configuration options selected between two configurations (e.g., a problematic configurations and a non-problematic configuration). Our tool also shows the influencing options between these two configurations. For instance, changing both DUPLICATES and TRANSACTIONS from `false` to `true` results in an interaction that increased the execution time by 54.7 seconds. Based on this information, most developers would consider DUPLICATES and TRANSACTIONS as influencing options that are causing an unexpected performance behavior.

[1] Any performance-influence model is shown relative to one configuration (e.g., the default configuration), which explains the impact of changes to that configuration. In our tool, developers can select that one configuration.



Influenced Hot Spot ▲	Influence (s) ▼
Cursor.put(...)	+42.9
FileManager.read(...)	+10.3
Internal.serialize(...)	+1.5

Figure 4.2: Our tool shows option hotspots affected by influencing options, and the influence on performance in each method.

Note that individual changes to DUPLICATES and TRANSACTIONS did not influence the performance of the system; only the *interaction* increased the execution time. Additionally, any other changes between the configurations – REPLICATED – do not influence the performance of the system. Likewise, the influence of TEMPORARY is not shown, as both configurations selected the same value.

Contribution - Design of information providers: We tailor interpretable global performance-influence models, generated with our white-box approach, to show developers the influencing options between a problematic configuration and a non-problematic configuration.

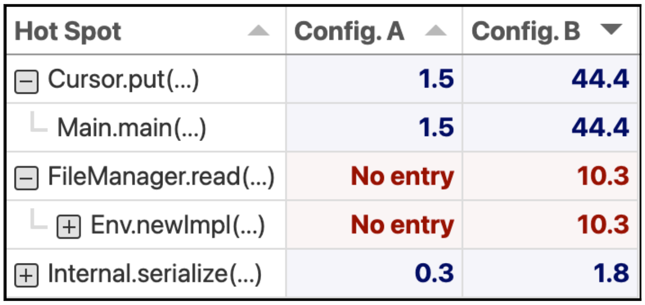
4.1.2 Locating Option Hotspots

After helping developers identify the influencing options, we help developers locate the option hotspots where these configuration options cause an unexpected performance behavior. As discussed in Section 2.3, we use *interpretable local performance-influence models*, which we generate with our white-box approach described in Chapter 3. For instance, the local model $\mathtt{m}_{\mathtt{put}} = 0.9 + 42.9 \cdot \text{DUPLICATES} \cdot \text{TRANSACTIONS}$ explains the influence of the configuration options and their interactions on the performance of the method `put`.

We adapt this ingredient to design an *information provider* that shows developers option hotspots; specifically, *where* and *by how much* configuration options and their interactions influence the performance of a system. In our implementation, this information provider shows (a) the *methods* whose performance is influenced by changes made between configurations (e.g., a problematic configuration and a non-problematic configuration) and (b) the *influence* of the changes on each method's performance.[2]

Example. Figure 4.2 shows as screenshot of our tool indicating the option hotspots where the influencing options DUPLICATES and TRANSACTIONS affect the performance of the system. Note that the influence on all methods equals the influence in the entire system (see Figure 4.1). Based on this information, most developers would consider `Cursor.put` as an option hotspot; the location where the *effect* of the influencing options is observed.

[2] Our tool allows developers to select a single configuration to analyze individual local performance-influence models.



Hot Spot ▲	Config. A ▲	Config. B ▼
⊟ Cursor.put(...)	1.5	44.4
└ Main.main(...)	1.5	44.4
⊟ FileManager.read(...)	No entry	10.3
└ ⊞ Env.newImpl(...)	No entry	10.3
⊞ Internal.serialize(...)	0.3	1.8

Figure 4.3: Our tool helps developers trace the cause-effect chain by highlighting the differences in the option hotspots' execution time and call stacks affected by influencing options. While the call stacks of `Cursor.put` are the same under both configurations, `FileManager.read` is only called under the second configuration.

Contribution - Design of information providers: We tailor interpretable local performance-influence models, generated with our white-box approach, to show developers option hotspots affected by influencing options, and the influence on performance in each method.

4.1.3 Tracing the Cause-Effect Chain

After helping developers identify the influencing options and locate the option hotspots, we help developers trace the cause-effect chain. As described in Section 2.3, we use *CPU profiling* and *program slicing*.

CPU Profiling

We use *CPU profiling* to collect the *hotspot view* of the problematic configuration and a non-problematic configuration. The hotspot view is the inverse of a call tree: A list of all methods sorted by their total execution time, cumulated from all different call stacks, and with back traces that show how the methods were called.

We adapt this ingredient to design an *information provider* that helps developers trace the cause-effect chain; specifically, *compare* the hotspot view of two configurations (e.g., a problematic configuration and a non-problematic configuration) to help developers determine whether the influencing options *affect* how option hotspots are called. In our implementation, this information provider highlights *differences* in the option hotspots' execution time and call stacks.[3]

CPU profiles are collected with most off-the-shelf profilers. Since our white-box approach to build global and local performance-influence models collect these profiles (Chapter 3), we use these profiles in our implementation.

Example. Figure 4.3 shows a screenshot of our tool, which helps developers trace the cause-effect chain by highlighting differences in the option hotspots' execution time and call stacks based on the influencing options DUPLICATES and TRANSACTIONS. For instance, the changes

[3] In our tool, developers can also analyze the CPU profile of one configuration.

increased `Cursor.put`'s execution time, but did not affect how the method was called. By contrast, `FileManager.read` is only executed under the problematic configuration. This information can help developers understand how the influencing options are used in the implementation to affect the option hotspots's performance.

Contribution - Design of information providers: We tailor CPU profiles, collected with our white-box performance-influence modeling approach, to help developers trace the cause-effect chain by comparing the hotspot view between a problematic configuration and a non-problematic configuration. This information can help developers determine whether the influencing options affect how option hotspots are called.

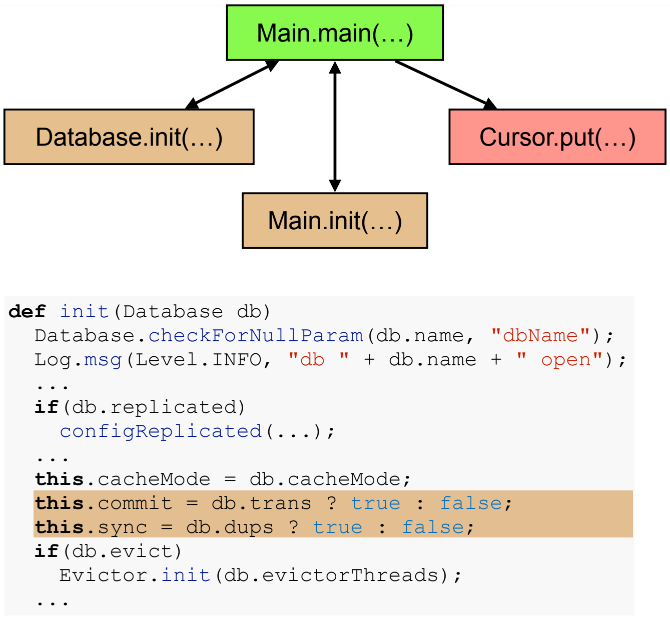
Program Slicing

We use *program slicing* to compute the relevant fragments for tracing the cause-effect chain. Specifically, we adapt this ingredient to design an *information provider* that helps developers *track how* influencing options are *used in the implementation* to directly and indirectly *influence* the performance of option hotspots. In our implementation, this information provider slices (chops) a system from the point where influencing options are first loaded into the system to the option hotspots, and shows (a) a *method-level dependence graph* and (b) *highlighted statements* of the slice in the source code.

Ideally, we would slice the program dynamically, as we are analyzing a system's dynamic behavior and to avoid approximations in the results. However, after exploring various dynamic and static slicing research tools, we settled on the state-of-the-art static slicer provided by JOANA [Graf et al., 2012], as it is the most mature option.

Example. Figure 4.4 shows a screenshot of our tool, which helps developers trace the cause-effect chain by showing a method-level dependence graph from the `main` method, in which the influencing options DUPLICATES and TRANSACTIONS are loaded into the system, to the option hotspot `Cursor.put`. The graph can help developers track dependences across methods in the system. When clicking on a method on the graph, our tool opens the file with the method, and highlights the statements in the slice, such as in Figure 4.4. The highlighted statements can help developers trace the cause-effect chain by tracking how influencing options are used in the implementation to directly and indirectly cause a performance issue in option hotspots.

Contribution - Design of information providers: We tailor program slicing to help developers trace the cause-effect chain by showing a method-level dependence graph and highlighting relevant statements. The information is derived by slicing (chopping) the system from the point where influencing options are first loaded into the system to the option hotspots. This information can help developers track how influencing options are used in the implementation to directly and indirectly influence the performance of option hotspots.



```
def init(Database db)
  Database.checkForNullParam(db.name, "dbName");
  Log.msg(Level.INFO, "db " + db.name + " open");
  ...
  if(db.replicated)
    configReplicated(...);
  ...
  this.cacheMode = db.cacheMode;
  this.commit = db.trans ? true : false;
  this.sync = db.dups ? true : false;
  if(db.evict)
    Evictor.init(db.evictorThreads);
  ...
```

Figure 4.4: Our tool helps developers trace the cause-effect chain by displaying a method-level dependence graph from the method where the influencing options are first loaded into the system (green box) to an option hotspot (red box). Other relevant methods are shown in brown boxes. When clicking on a box, our tool opens the file with the method and highlights the statements of the slice. The position of the nodes in the graph (left to right and top to bottom) does not represent the order of execution of methods.

4.1.4 Implementation

We implemented the information providers in a Visual Studio Code extension prototype called **GLIMPS**. Our prototype tailors interpretable **G**lobal and **L**ocal performance-**I**nfluence **M**odels, CPU **P**rofiles, and a program **S**licer. The first three items are collected prior to debugging, using an infrastructure where developers configure and run the system. Subsequently, developers use **GLIMPS** to identify influencing options, locate option hotspots, and trace the cause-effect chain.

GLIMPS is agnostic to the ingredients used to tailor and implement information providers. In fact, **GLIMPS** is entirely built on our existing infrastructure of white-box performance-influence modeling (Chapter 3) and the program slicer provided by JOANA, without major modifications. The novelty of **GLIMPS** is in the *design* and *integration of information providers*, from multiple *ingredients*, into a *cohesive* infrastructure and user interface, which can help developers debug the performance of configurable software systems.

The implementation that we evaluate uses Comprex, which we presented in Chapter 3, to build the interpretable global and local performance-influence models, using JProfiler [JPR, 2019] to collect the CPU profiles. We use JOANA to slice the system from influencing options to option hotspots using a fixed-point chopper algorithm, which first computes a backward slice from the option hotspots, and then computes a forward slice, on the backward slice, from the influencing options [Giffhorn, 2011]. For scalability and to reduce approximations, we modified JOANA to consider code coverage under the problematic configuration and a non-problematic configuration.

Contribution - GLIMPS: We integrated our designed information providers into a cohesive infrastructure and user interface that developers can use to debug the performance of configurable software systems.

4.2 Evaluating Usefulness of Information Providers

We evaluate the usefulness of our designed information providers to help developers debug the performance of configurable software systems. Specifically, we answer the following research question:

RQ1: To what extent do the designed information providers help developers debug the performance of configurable software systems?

We answer this research question with two user studies using different designs. We first evaluate the extent that our information providers support the information needs that we identified in our exploratory study presented in Chapter 2. To this end, we conduct a *validation* user study, in which we ask the same participants of our exploratory study to debug a *comparable* unexpected performance behavior using **GLIMPS** on the same subject system (Section 4.2.1). Afterwards, we replicate the study, intentionally *varying* some aspects of the design (theoretical replication [Juristo and Gómez, 2011, Schmidt]), to evaluate to which extent our information providers generalize for a more complex task with an interaction in a larger software system. Specifically, we conduct a *confirmatory* user study, in which we ask a *new* set of participants to debug a *more complex task* on a *more complex subject system*, (Sec 4.2.2). The validation and confirmatory studies, *together*, provide evidence that our information providers help developers debug the performance of complex configurable software systems, *because* the information providers *support* the information needs that developers have in this process.

4.2.1 Validating Usefulness of Information Providers

We first conducted a *validation* user study to evaluate the extent that the designed information providers support the information needs that we identified in our exploratory study (Chapter 2).

Method

Study design. We invited the participants from our exploratory study, after 5 months, to solve another problem in the same subject system, *Density Converter (Complete)*, which includes all Java dependencies, but now with the help of our information providers. This design can be considered as a within-subject study, where subjects perform tasks both in the control and in the treatment condition: Specifically, we consider our exploratory study as the *control* condition, in which participants debugged a system *without* **GLIMPS**, and consider the new study as the *treatment* condition, in which participants debug a *comparable* performance issue for 50 minutes in the same subject system with **GLIMPS**. Similar to the exploratory study, we use a think-aloud protocol [Jääskeläinen, 2010] to identify whether our information providers actually support the information needs that developers have when debugging the performance of the subject system.

Prior to the task, participants worked on a warm-up task for 20 minutes using **GLIMPS**. We tested the time for the warm-up task, as well as **GLIMPS**'s design and implementation in a pilot study with 4 graduate students from our personal network.

After the task, we conducted a brief semi-structured interview to discuss the participants' experience in debugging the system, as well as the usefulness of the information providers, and to contrast their experience to debugging without the information providers.

Due to the COVID-19 pandemic, we conducted the study remotely over Zoom. Participants used Visual Studio Code through their preferred Web browser. The IDE was running on a remote server and was configured with **GLIMPS**. We asked participants to share their screen. With the participants' permission, we recorded audio and video of the sessions for subsequent analysis.

Task and subject system. We prepared a comparable, *but different*, debugging task to the task in our exploratory study. Similar to the exploratory study, the task involved a user-defined configuration in *Density Converter (Complete)* that spends an excessive amount of time executing. We introduced a bug caused by the incorrect implementation of one configuration option (the system was using a larger scale of the input image instead of using a fraction). Participants were required to identify and explain which and how configurations options caused the unexpected performance behavior. In contrast to the exploratory study, the user-defined configuration, bug, and problematic configuration option, were *different*.

Participants. We invited the participants from our exploratory study to work on our task. After conducting the study with 8 participants, we observed a *massive effect size* between debugging with and without our information providers (correctly debugging within 19 minutes compared to failing after 50 minutes). Hence, we did not invite the remaining participants.

Analysis. We analyzed and compared transcripts of the audio and video recordings of the exploratory and validation studies to measure the time participants spend working on the task and their success rates. Based on our exploratory study, participants needed to identify influencing options, locate option hotspots, and trace the cause-effect chain to correctly debug the system. We also analyzed the interviews using standard qualitative research methods [Saldaña, 2015]. The author who conducted the study analyzed the sessions independently, summarizing observations and discussions during the task and interviews. All authors met weekly to discuss the observations.

Threats to Validity and Credibility. We invited the same participants and used the same subject system as in our exploratory study. Such a design might only validate the information needs when debugging performance in the selected subject system. Additionally, the exploratory and validation studies were conducted 5 months apart, which might result in learning effects that help participants in the latter study. Furthermore, there is the threat that the task in the validation study is simpler. While our later study varies these aspects to observe whether our solutions generalize to other tasks, generalizations about our results should be done with care.

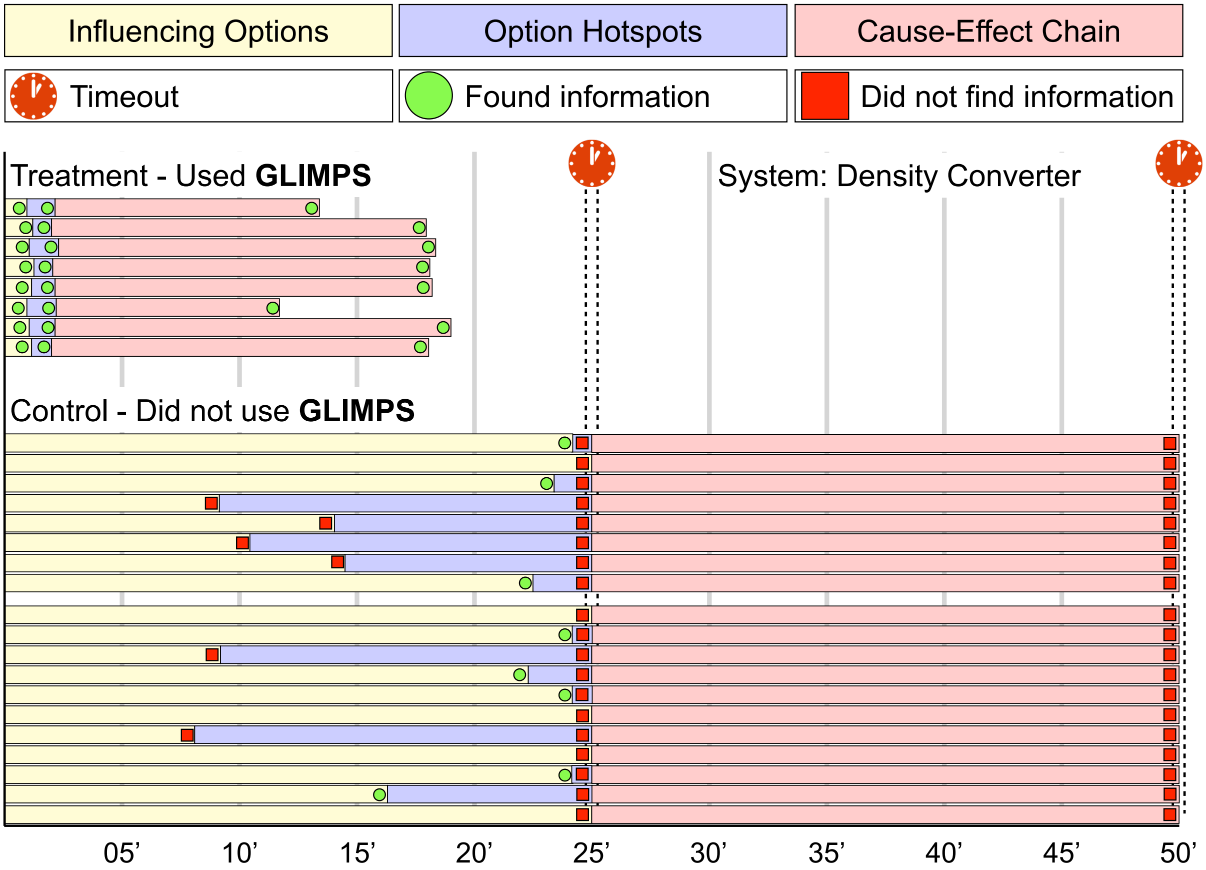


Figure 4.5: Time participants spent looking for each piece of information when debugging the performance of *Density Converter (Complete)* with and without GLIMPS. The first 8 participants who did not use GLIMPS are the same participants who used our tool. The data for the other participants who did not use GLIMPS is included for reference.

Results

Figure 4.5 shows the time that each participant spent looking for each piece of information when debugging the performance of *Density Converter (Complete)* with (treatment) and without (control) **GLIMPS**. Overall, all participants who used our information providers identified the influencing options, located option hotspots, and traced the cause-effect chain, and correctly explained the root cause of the performance issue in less than 19 minutes. By contrast, the 8 participants could not debug the unexpected behavior without our information providers in 50 minutes, when they struggled to find relevant information.[4]

All 8 participants who used our information providers identified the influencing options and located the option hotspots in a few minutes. Afterwards, all participants traced the cause-effect chain and explained how the influencing options caused the unexpected performance behavior in the option hotspots.

[4] We did not conduct a statistical significance test, since comparing completion rates is obvious: All participants correctly debugged with our tool, but no participants correctly debugged without our tool; comparing completion times cannot be done since nobody completed the task without our tool.

When these 8 participants did not use our information providers, no participant found a single piece of information in the same timeframe as they did when using our information providers. In fact, during a 25-minute window, only 3 participants found the influencing options, and no participant found any option hotspots. Furthermore, as described in our exploratory study, even when we *explicitly told* participants (a) the one influencing option that was causing the unexpected performance behavior and (b) the option hotspots whose execution times drastically increased as a result of the problematic configuration option, no participant could trace the cause-effect chain and find the root cause of the unexpected behavior within 25 minutes.

After completing the task, the participants discussed how the information providers helped them debug the performance of the system, and contrasted their experience to debugging without our tool. All participants mentioned that the information providers helped them obtain relevant information for debugging. The consensus was that the information providers *"helped me focus on the relevant parts of the system"* to debug the unexpected performance behavior. The participants contrasted this experience to the struggles that they faced when debugging without our tool. In particular, some participants remembered *"being lost"* on what methods to follow or not knowing *"which parts of the program are relevant."*

Summary RQ1: The validation study provides evidence that the designed information providers support the information needs that we identified in our exploratory study.

Contribution - Validation study: Our validation study demonstrates the usefulness of our designed information providers to support the needs that developers have to (a) identify influencing options, (b) locate option hotspots, and (c) trace the cause-effect chain when debugging the performance of configurable software systems.

Thesis contribution: We conducted an empirical evaluation that validates that our designed information providers support the needs that developers have when debugging the performance of configurable software systems. Supporting developers' needs helps them maintain configurable software systems to reduce the energy consumption and operational costs of running this type systems.

4.2.2 Confirming Usefulness of Information Providers

After validating that our information providers support the information needs that we identified, we conducted a *confirmatory* user study to evaluate the extent that the information providers can potentially generalize to support the information needs of debugging the performance of complex configurable software systems.

Method

Study design. We replicated the validation study, intentionally *varying* some aspects of the design: We used a *between-subject design* where we ask a *new* set of participants to debug a *more complex task* on a *more complex subject system*, all working on the same task, but using *different tool support*. With these variations, we evaluate that the results and *massive effect size* in our previous study are not due to, for example, a simpler task or learning effects, but rather, that the information providers help developers debug the performance of complex configurable software systems, *because* the information providers support the needs that developers have in this process.

As in our validation study, we conducted the confirmatory study using a think-aloud protocol [Jääskeläinen, 2010], to compare how two new sets of participants debug the performance of a complex configurable software system using different tool support in 60 minutes. The treatment group used **GLIMPS**, while the control group used a simple plugin, which profiles and provides the execution time of the system under any configuration. This information is the same that we gave participants in our exploratory study using a Wizard of Oz approach (see Section 2.2). For this confirmatory study, however, we did not use a Wizard of Oz approach, as we wanted both groups to access information for debugging using a tool and the same IDE.

Prior to the task, participants worked on a warm-up task for 20 minutes using either **GLIMPS** or the simple plugin to learn how to use the information providers or the components that provided performance behavior information, respectively. We tested the simple plugin's design and implementation in a pilot study with 4 graduate students from our personal network.

After the task, we conducted a brief semi-structured interview to discuss the participants' experience in debugging the system. In particular, we asked participants in the treatment group about the usefulness of the information providers and whether there was additional information that they would like to have in the debugging process. Similarly, we asked participants in the control group for the information that they would like to have when debugging the performance of configurable software systems.

Due to the COVID-19 pandemic, we conducted the study remotely over Zoom. Participants used Visual Studio Code through their preferred Web browser, which was running on a remote server, and was configured with **GLIMPS** and the simple plugin. We asked participants to share their screen. With the participants' permission, we recorded audio and video of the sessions for subsequent analysis.

Task and subject system. We prepared a more complex performance debugging task for a more complex configurable software system than the task and subject system in our exploratory and validation studies. Similar to the prior studies, the task involved a user-defined configuration that spends an excessive amount of time executing. Participants were required to identify and explain which and how configuration options caused the unexpected performance behavior. In contrast to the previous studies, we introduced a bug caused by the incorrect implementation of *an interaction of two configuration options* (The system spent a long time inserting duplicate data using transactions). We selected *Berkeley DB* as the subject system due to the following reasons: (1) the system is implemented in Java, is open source, and is more complex than *Density Converter (Complete)* (over 150K SLOC and 30 binary and non-binary configuration options) and (2) the system has a complex performance behavior (execution time ranges from a couple of seconds to a few minutes, depending on the configuration).

Participants. We recruited 12 graduate students, *independent* of our exploratory and validation studies, with extensive experience analyzing the performance of configurable Java systems.

When determining the number of participants for the control group, we made some ethical considerations, while also ensuring that we obtain reliable results. In our exploratory study, we observed 19 *experienced* researchers and professional software engineers who *could not debug* the performance of a *medium-sized* system with a performance bug caused by a *single configuration option* within 50 minutes (see Figure 4.5). With *Berkeley DB*, we want to observe how participants debug the performance of *a more complex system*; a significantly larger system, in terms of SLOC and configuration space size, in which the unexpected behavior is caused by an *interaction of two configuration options*. Based on (a) the fact that we have *strong empirical evidence* that debugging the performance of configurable software systems without relevant information is frustrating and is highly likely to not be completed under 60 minutes and (b) the massive effect size in our validation study between debugging with and without our information providers, we decided to minimize the number of participants that we expect to struggle and fail to complete the task, while still having a reasonable number participants in the control group.

Ultimately, we randomly assigned 4 out of the 12 participants to the control group, making sure to balance the groups in terms of the participants' debugging experience: The median programming experience for both groups is 6 years, a median of 3.5 years in Java, a median of 2.2 years of performance analysis experience, and a median of 2.7 years working with configurable software systems.

Analysis. We analyzed transcripts of the audio and video recordings to measure the time participants spend working on the task and their success rates. Based on our exploratory and validation studies, participants needed to identify influencing options, locate option hotspots, and trace the cause-effect chain to successfully debug the system. Additionally, we analyzed the interviews using standard qualitative research methods [Saldaña, 2015]. The author who conducted the study analyzed the sessions independently, summarizing observations and discussions during the debugging task and the interviews. All authors met weekly to discuss the observations.

Threats to Validity and Credibility. While we aimed to increase the complexity of the performance debugging task, readers should be careful when generalizing our results to other complex configurable software systems.

Our control group consisted of 4 participants. As argued previously, we did not recruit more participants due to the struggles that we observed in our exploratory study on a simpler system and the massive effect size in our validation study between debugging with and without our information providers. Nevertheless, readers should be careful when generalizing our results.

While the control group had access to the IDE's debugger and used a simple plugin, we might obtain different results if the participants had used other debugging tools and techniques.

Results

Figure 4.6 shows the time each participant spent looking for each piece of information while debugging the performance of *Berkeley DB* with **GLIMPS** (treatment) and the simple plugin

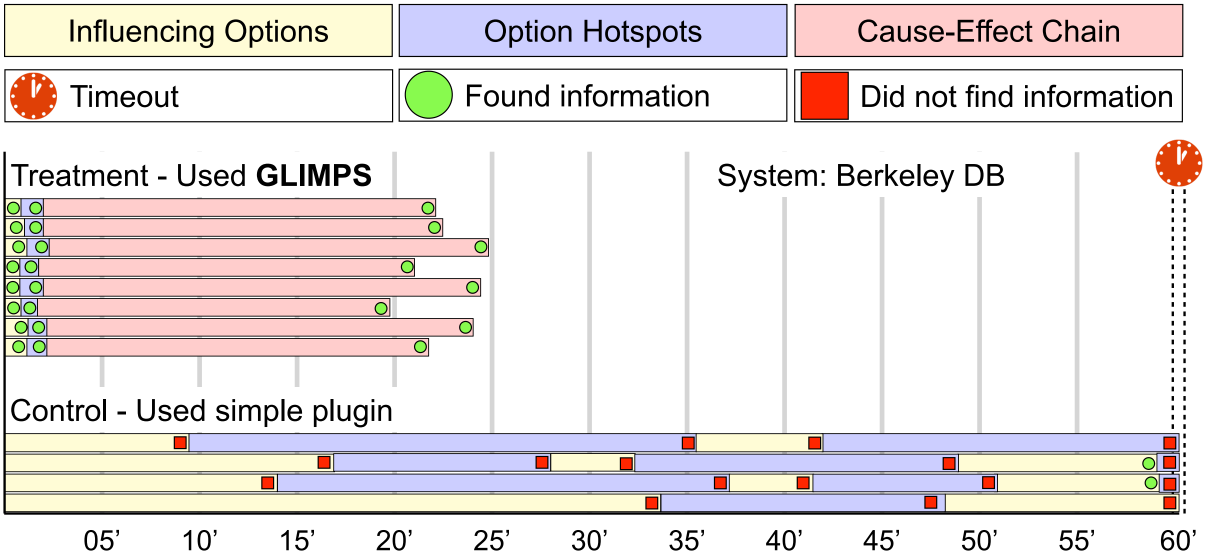


Figure 4.6: Time participants spent looking for each piece of information when debugging the performance of *Berkeley DB* with different tool support.

(control). Similar to our validation study, all participants who used our information providers identified the influencing options, located the option hotspots, and traced the cause-effect chain in less than 25 minutes. By contrast, the participants who did not use our information providers struggled for 60 minutes and could not debug the system.[5]

While working on the task, we observed the participants in the treatment group looking for the same information as in Table 4.1, and using our information providers similarly to the participants in the validation study to find information to debug the subject system. Likewise, the participants in the control group struggled while performing the same activities as those listed in Table 2.1 when trying to identify the influencing options and locate the option hotspots.

After working on the task, all participants discussed their experience in debugging the performance of the system using tool support. Similar to the discussion in our validation study, all participants in the treatment group commented how the information providers helped them identify influencing options, locate option hotspots, and trace the cause-effect chain. Likewise, the participants who used the simple plugin described similar struggles and barriers as those mentioned in our exploratory study (Section 2.2). All participants in this group mentioned that identifying the influencing options that cause the expected behavior is *“difficult”* and locating the option hotspots is *“challenging.”* However, none of the participants in this group commented on tracing the cause-effect chain, as they never got to that point in the debugging process.

Summary RQ1: The confirmatory study provides evidence that our information providers help developers debug the performance of complex configurable software systems.

[5] We did not conduct a statistical significance test, since comparing completion rates is obvious: All participants in the treatment group correctly debugged the system, but no participant in the control group did; comparing completion times cannot be done since nobody in the control group completed the task.

Contribution - Confirmatory study: Our confirmatory study demonstrates the usefulness of our designed information, because they support the needs that developers have when debugging the performance of complex configurable software systems.

Thesis contribution: We conducted an empirical evaluation that confirms that our designed information providers support the needs that developers have and speed up the process of debugging the performance of complex configurable software systems. Supporting developers' needs helps them maintain complex configurable software systems to reduce the energy consumption and operational costs of running this type systems.

4.3 Summary

In this chapter, we continued our *human-centered* approach, which started with our exploratory user study in Chapter 2, to identify solutions to support developers' actual needs in the process of debugging the performance of configurable software systems. We described how we *designed* and implemented *information providers*, *tailoring* our work on interpretable **G**lobal and **L**ocal performance-**I**nfluence **M**odeling, and use of CPU **P**rofiling described in Chapter 3, as well as tailoring program **S**licing. We integrated the information providers in a *cohesive* prototype called **GLIMPS**, which can help developers debug the performance of configurable software systems. Two *user studies*, with a total of 20 developers, *validate* and *confirm* that our designed information providers support the needs that developers have and speed up the process of debugging the performance of complex configurable software systems.

The information providers implemented in **GLIMPS** contribute to our thesis goal of reducing the energy consumption and operational costs of running configurable software systems by providing developers with tool support to help them debug and maintain their systems.

Chapter 5

Conclusions

Most of today's software systems are configurable. The flexibility to customize these systems, however, comes with the cost of increased complexity. Understanding how configuration options and their interactions affect performance, in terms of execution time, and often directly correlated energy consumption and operational costs, is challenging, due to the large configuration spaces of these systems. For this reason, developers often struggle to debug and maintain their systems when surprising performance behaviors occur.

In this dissertation, we took a human-centered approach to identify solutions to support developers' needs in the process of debugging the performance of configurable software systems. We tailored white-box analyses and techniques to provide relevant performance-behavior information for developers to understand how configuration options and their interactions cause performance issues.

In Chapter 2, we conducted an exploratory user study, in which we identified that developers struggle to find relevant information to identify influencing options; the configuration options or interactions causing an unexpected performance behavior, locate option hotspots; the methods where configuration options affect the performance of the system, and trace the cause-effect chain; how influencing options are used in the implementation to directly and indirectly affect the performance of option hotspots. Based on these findings, we suggested that interpretable global and local performance-influence modeling can help developers identify influencing options and locate option hotspots, and that CPU profiling and program slicing can help developers trace the cause-effect chain.

While the latter two techniques can be tailored to support developers' needs without major modifications, we noted some limitations with existing performance-influence modeling techniques that we needed to overcome to provide relevant information for debugging. Consequently, in Chapter 3, we presented a white-box approach to model the performance of configurable software systems. Our approach analyzes and instruments the source code to accurately capture configuration-specific performance behavior, without using machine learning to extrapolate incomplete samples. Our approach tailors a taint analysis to identify how configuration options and their interactions influence the performance of code regions. An empirical evaluation demonstrated that our white-box approach can efficiently build accurate and interpretable performance-influence models.

In Chapter 4, we continued our human-centered approach to identify solutions to help developers debug. We described how we designed and implemented information providers, tailoring interpretable global and local performance-influence models, CPU profiling, and program slicing, in a cohesive prototype called **GLIMPS**. Two user studies validated and confirmed that our designed information providers support the needs that developers have and speed up the process of debugging the performance of configurable software systems.

Thesis statement. We conclude that this dissertation provides substantial evidence that validates our thesis statement.

> ***Thesis Statement***: *Tailoring specific white-box analyses to track how configuration options influence the performance of code-level structures in configurable software systems helps developers to (1) efficiently build accurate and interpretable global and local performance-influence models and (2) more easily inspect, trace, understand, and debug configuration-related performance issues.*

To validate this hypothesis, we took a human-centered approach to identify how to analyze and obtain relevant information to help developers debug the performance of configurable software systems. We conducted an exploratory user study to identify the information that developers need when debugging the performance of configurable software systems. Afterwards, we identified the program analyses and techniques that can be tailored to support those needs. However, we noted that existing performance-influence modeling techniques have some limitations, that we needed to overcome to provide relevant information for debugging (Chapter 2). Consequently, we presented and evaluated a white-box performance-influence modeling technique, which tailored a taint analysis and overcame those limitations (Chapter 3). Afterwards, we described how we designed and implemented information providers, tailoring the white-box analyses that we identified, to support developers' needs; namely, interpretable global and local performance-influence modeling, CPU profiling, and program slicing. Finally, we conducted two users studies to validate and confirm that our designed information providers support the needs that developers have and speed up the process of debugging the performance of complex configurable software systems (Chapter 4).

In addition to validating our thesis statement, we conclude that this dissertation contributes to our larger goal of reducing the energy consumption and operational costs of running configurable software systems, by providing developers with targeted tool support to help them debug and maintain their systems.

5.1 Future Work

The goal of this dissertation is to identify and evaluate solutions to help developers debug the performance of complex configurable software systems. Reflecting upon our experiences and findings, we highlight some future directions.

Usability and Interpretability of Performance-Influence Models

We demonstrated that interpretable global and local performance-influence models provide relevant information to help developers debug the performance of configurable software systems. One possible future direction could further evaluate the usability and interpretability of these types of models.

As described in Section 2.1, one research area is devoted to improving the accuracy and reducing the cost of building performance-influence models [Ha and Zhang, 2019, Kolesnikov et al., 2018, Siegmund et al., 2015, Weber et al., 2021]. While one of the motivations in this area is to help users make informed configuration decisions, the models are often evaluated in terms of accuracy, and not usability or interpretability. Although the evaluation in Chapter 3 of our white-box approach also measured the cost to generate our models and their accuracy, we argued about the interpretability of our linear models. Furthermore, in Chapter 4, we conducted two user studies that demonstrated the usefulness of the tailored information provided by our linear models. Similar to this dissertation, future work could evaluate the extent that different representations of these models can help users make informed tradeoff decisions.

In recent work, Kolesnikov et al. [2018] briefly discussed the usability and interpretability of performance-influence models. When the authors presented linear models to 4 developers in high-performance computing, a developer mentioned "[being] surprised to see that [a configuration option] had only a small influence on system performance." This finding provides evidence that linear performance-influence models, such as the ones that we generate with our white-box approach, are preferable in situations where understanding how configuration options and interactions affect performance is important, such as debugging. A possible future direction could, more methodically, compare the usability and interpretability of different model representations.

While we argue that the performance-influence models that we generate with our white-box approach are interpretable, interpretability is an open research problem without a generally agreed measure for interpretability [Doshi-Velez and Kim, 2017, Lipton, 2018, Molnar, 2019]. Although the two users that we conducted in Chapter 4 provide evidence that linear models are helpful for developers to debug performance in configurable software systems, a future research direction could further empirically explore the topic of interprability in performance-influence models.

Tool Support Deployment in the Field

We conducted two user studies that validated and confirmed that the information providers that we implemented in **GLIMPS** support the needs that developers have and speed up the process of debugging the performance of complex configurable software systems. One possible future direction could evaluate the usefulness of the information providers and tool support to help developers debug real bug reports in their own configurable software systems.

One possible study design could involve developers independently using our tool to debug performance issues in their own systems. In this design, the tool would need to be instrumented to collect information of how developers use the tool for measuring usefulness. The study could be complimented with surveys or interviews, to further understand how developers used the tool during the debugging process. To conduct such a study, however, a significant amount of engineering effort would be required for developers to independently use the tool. Nevertheless, such a study would provide strong empirical evidence of the usefulness of the tool to debug performance issues “in the wild”.

Alternatively, another possible study design could involve working with a few developers to help them debug bug reports in their own configurable software systems. In this design, developers would be guided on how to set up and use our tool, while an experimenter observes how developers debug their own systems. The study could also be complimented with interviews to get the developers’ perspectives on the usefulness of the tool. This study design, however, most likely would be biased, as the experimenter would be interacting with the developers using the tool. Nevertheless, the design would provide evidence of the usefulness of the tool to help developers debug performance issues in real bug reports.

Another possible option would be to conduct case studies, in which researchers use our tool to debug performance issues in open-source configurable software systems. In this design, researchers would select bug reports from mailing lists or issue trackers, and debug the systems using our tool. Afterwards, the researchers would respond to the bug reports indicating any findings, misconfigurations, or bug fixes. While this study design does not involve interacting with developers, the design would provide some evidence that developers who are unfamiliar with a system can use our tool to debug the performance of configurable software systems.

Regardless of the study design the is used, such a study would help to further demonstrate the usefulness of the work in this dissertation. Furthermore, interacting with developers who are actively debugging performance issues in their own configurable software systems might uncover additional information needs. Consequently, researchers could identify additional techniques and approaches that can provide relevant information to further help developers debug.

Scalability of Taint Analyses

We observed some scalability issues with the static and dynamic taint analyses that we tailored in our white-box performance-modeling approach (Chapter 3). One possible future direction could explore how to overcome these limitations.

Scalability in static taint analysis is an open area of research, in which several different techniques have been developed and evaluated to analyze larger systems [Andreasen et al., 2017, Avdiienko et al., 2015, Barros et al., 2015, Bodden, 2018, Christakis and Bird, 2016, Do et al., 2017, Garbervetsky et al., 2017, Lerch et al., 2015, Späth et al., 2017, Zhang and Su, 2017]. Strategies include reducing the precision of the results [Avdiienko et al., 2015, Do et al., 2017], novel strategies to conduct the taint analysis itself [Andreasen et al., 2017, Bodden, 2018, Garbervetsky et al., 2017, Lerch et al., 2015, Zhang and Su, 2017], or only precisely tracking taints through specific constructs [Barros et al., 2015, Späth et al., 2017]. We hope that this community will continue advancing the state of the art, and these strategies are integrated in FlowDroid and other engines, to scale the analysis to larger systems.

In addition to the above strategies, the community could explore techniques from other research areas. For instance, a *barrier* is a concept in program slicing, which filters that parts of the system that should not be considered when computing a slice [Krinke, 2003]. We basically used barriers when we considered code coverage information when slicing our subject systems with JOANA (Chapter 4). One future direction could explore this and similar strategies to scale taint analyses to larger systems.

We also observed scalability issues when using the regular workload to run the dynamic taint analysis in our subject systems. While we overcame the issue by drastically reducing the workload size, a possible future direction could explore new or similar techniques, such as those in static taint analysis, to reduce the overhead in Phosphor of running a precise taint analysis.

Analyzing Highly-Configurable Software Systems

Similar to existing work, we analyzed the performance of a subset of the configuration options of our subject systems; ranging from 5 to 22 configurations. One possible future direction could explore the effort and practicality of analyzing systems with much larger configuration spaces.

While our larger subject systems have intractably large configuration spaces (e.g., more that 2^{22} configuration), there are several software systems that have hundreds or thousands of configuration options [Han and Yu, 2016, Mukelabai et al., 2018], which might require more effort to accurately analyze the functionality and behavior of those systems. For instance, at most, we measured the performance of 2K configurations of some subject systems to evaluate our white-box modeling approach (Chapter 3). However, there are scenarios and use cases in which a much larger number of configuration might need to be analyzed. For instance, Acher et al. [2019] analyzed about 95K configurations of the Linux kernel for measuring and predicting the kernel size. Likewise, Halin et al. [2018] reported the computational effort to exhaustively test over 26K configurations of JHipster. We hope that this community will continue considering larger configurations spaces in their empirical evaluations.

In addition to considering more options for a single system, there are some distributed systems that are composed of multiple configurable software systems. In these types of systems, configuration options might affect the functionality and quality attributes, including performance, of other components in addition to those in which the configuration options are defined [Krishna et al., 2020]. One future direction could explore how additional white-box analyses could be used and adapted to analyze and debug the performance of those systems.

Appendix A

Experimental Setup: Performance-Influence Modeling

A.1 Subject Systems

The following list describes each system in more detail, including the configuration options, scenario, and workload considered for the measuring performance:

- *Pngtastic Counter* is a component of Pngtastic[1]; an API for manipulating PNG images. This component counts the number of colors in an image. The system does not come with any tests or benchmarks, and only provides small sample images that are processed in a couple of seconds. Hence, we processed a publicly available 9118×5699 pixel 34.5MB PNG image. The configuration options that we considered are:
  - DIST_THRESHOLD
  - FREQ_THRESHOLD
  - LOG_LEVEL
  - MIN_ALPHA
  - TIMEOUT
- *Pngtastic Optimizer* is also a component of Pngtastic. This component optimizes PNG images to reduce file size. We processed the same PNG image as in *Pngtastic Counter*. The configuration options that we considered are:
  - COMPRESSION_LEVEL
  - COMPRESSOR
  - ITERATIONS
  - LOG_LEVEL
  - REMOVE_GAMMA
- *Elevator*[2] is a simulator of a configurable elevator system, purposely built to have all configuration options interact. The system comes with several specifications and scenarios of how the elevator should behave. We selected the default `Specification3`. The

[1] https://github.com/depsypher/pngtastic

[2] https://www.se.cs.uni-saarland.de/projects/family/Elevator.rar

configuration options that we considered are:

  - BASE
  - EMPTY
  - EXECUTIVE_FLOOR
  - OVERLOADED
  - TWO_THIRDS_FULL
  - WEIGHT

- *Grep* is the Java implementation of the Unix command line utility available in Unix4j[3] for searching plain-text data. The system does not come with any tests or benchmarks. Hence, we ran the command on a set of 45 text files of popular books totaling about 1 million lines. The configuration options that we considered are:
  - COUNT
  - FIXED_STRINGS
  - IGNORE_CASE
  - INVERT_MATCH
  - LINE_NUMBER
  - MATCHING_FILES
  - WHOLE_LINE
- *Kanzi*[4] is a modular, expandable, and efficient lossless data compressor. We executed the `BlockCompressor` scenario to compresses and archive all files in a Java 67.7MB `rt.jar` file. The configuration options that we considered are:
  - BLOCK_SIZE
  - CHECKSUM
  - ENTROPY
  - FORCE
  - LEVEL
  - TRANSFORM
  - VERBOSE
- *Email*[5] is a simulator of a configurable email client. The system comes with several scenarios that specify how the client should behave. We executed 9 scenarios sequentially. The configuration options that we considered are:
  - ADDRESS_BOOK
  - AUTO_RESPONDER
  - BASE
  - DECRYPT
  - ENCRYPT
  - FORWARD
  - KEYS
  - SING

[3] https://github.com/tools4j/unix4j
[4] https://github.com/flanglet/kanzi
[5] https://www.se.cs.uni-saarland.de/projects/family/Email.rar

  - VERIFY

- *Prevayler*[6] is an object-persistence database, in which business objects are kept live in memory, and transactions are journaled for system recovery. We executed a demo included with the system, `PrimeCalculator`, which calculates and stores the first 500K prime numbers. The configuration options that we considered are:
  - CLOCK
  - DEEP_COPY
  - DISK_SYNC
  - FILE_AGE_THRESHOLD
  - FILE_SIZE_THRESHOLD
  - JOURNAL_SERIALIZER
  - MONITOR
  - SNAPSHOT_SERIALIZER
  - TRANSIENT_MODE
- *Sort* is also the Java implementation of the Unix command line utility available in Unix4j for sorting or merging records of text and binary files. We ran the command on the same set of 45 text files as in *Grep*. The configuration options that we considered are:
  - CHECK
  - DICTIONARYORDER
  - GENERALNUMBERICSORT
  - HUMANNUMBERICSORT
  - IGNORECASE
  - IGNORELEADINGBLANKS
  - MERGE
  - MONTHSORT
  - NUMERICSORT
  - REVERSE
  - UNIQUE
  - VERSIONSORT
- *H2*[7] is a fast relational database system that can operate both in an embedded and a client-server setting. We executed the `RunBenchC` benchmark, included with the system, which is similar to the TPC-C benchmark. The benchmark includes multiple transaction types on a complex database with several tables and with a specific execution structure. The configuration options that we considered are:
  - ACCESS_MODE_DATA
  - ANALYZE_AUTO
  - ANALYZE_SAMPLE
  - AUTO_COMMIT
  - CACHE_SIZE
  - CACHE_TYPE

[6] https://prevayler.org/

[7] https://h2database.com/html/main.html

  - CIPHER
  - COMPRESS
  - DEFRAG_ALWAYS
  - FILE_LOCK
  - FORBID_CREATION
  - IF_EXISTS
  - IGNORE_UNKNWON_SETTING
  - MV_STORE
  - OPTIMAL_DISTINCT
  - PAGE_SIZE
- *Berkeley DB*[8] is a high-performance embeddable database providing key-value storage. We executed the `MeasureDiskOrderedScan` performance benchmark, included with the system, which populates a database with 500K key/value pairs. The configuration options that we considered are:
  - ADLER32_CHUNK_SIZE
  - CACHE_MODE
  - CHECKPOINTER_BYTES_INTERVAL
  - DUPLICATES
  - ENV_BACKGROUND_READ_LIMIT
  - ENV_IS_LOCKING
  - ENV_SHARED_CACHE
  - JE_DURABLE
  - JE_FILE_LEVEL
  - LOCK_DEADLOCK_DETECT_DELAY
  - LOCK_DEADLOCK_DETECT
  - MAX_MEMORY
  - REPLICATED
  - SEQUENTIAL
  - TEMPORARY
  - TXN_SERIALIZABE_ISOLATE
- *Apache Lucene*[9] is a library providing indexing and search features, as well as advanced analysis and tokenization capabilities. We executed the `IndexFiles` demo included with the system. Similarly to its nightly benchmarks that index large Wikipedia exports, we indexed a large publicly available 512.5MB Wikimedia dump with about 8 million lines. The configuration options that we considered are:
  - CHECK_PENDING_FLUSH_UPDATE
  - CODEC
  - COMMIT_ON_CLOSE
  - INDEX_COMMIT
  - INDEX_DELETION_POLICY

[8] https://www.oracle.com/database/berkeley-db/java-edition.html

[9] https://lucene.apache.org/

  - MAX_BUFFERED_DOCS
  - MAX_CFS_SEGMENT_SIZE_MB
  - MAX_TOKEN_LENGTH
  - MERGE_POLICY
  - MERGE_SCHEDULER
  - MERGED_SEGMENT_WARMER
  - NO_CFS_RATIO
  - RAM_BUFF_SIZE_MB
  - RAM_PER_THREAD_LIMIT
  - READER_POOLING
  - SIMILARITY
  - USE_COMPOUND_FILE

- *Density Converter*[10] is a popular image density converting tool, processing single or batches of images to Android, iOS, Windows, or CSS specific formats and density versions. The system does not come with any tests or benchmarks, and only provides small sample images that are processed in a couple of seconds. Hence, we processed a publicly available 5616×3744 pixel 3.3MB JPEG photo. The configuration options that we considered are:

  - ANDROID_INCLUDE_LDPI_TVI
  - ANDROID_MIP_MAP_INSTEAD_OF_DRAWABLE
  - ANTI_ALIASING
  - CLEAN
  - COMPRESSION_QUALITY
  - DOWNSCALING_ALGO
  - DRY_RUN
  - HALT_ON_ERROR
  - IOS_CREATE_IMAGESET_FOLDERS
  - KEEP_ORIGINAL_POST_PROCSSED_FILES
  - OUT_COMPRESSION
  - PLATFORM
  - POST_PROCESSOR_MOZ_JPEG
  - POST_PROCESSOR_PNG_CRUSH
  - POST_PROCESSOR_WEBP
  - ROUNDING_MODE
  - SCALE_IS_HEIGHT_DP
  - SCALE
  - SKIP_EXISTING
  - SKIP_UPSCALING
  - UPSCALING_ALGO
  - VERBOSE

  We considered two versions of this system: (1) the *Interface* version, in which we analyzed the original 1359 SLOC system that calls several libraries to process images, and (2) the

[10] https://github.com/patrickfav/density-converter

Complete version, in which we included and analyzed all Java dependencies.

A.2 Workloads for Dynamic Taint Analysis

We changed the workload to run the the iterative dynamic taint analysis in **Comprex** with a smaller workload by factors ranging from 20 to 5000, depending on the system:

- For *Pngtastic Counter* and *Pngtastic Optimizer*, we reduced the size of the image to process by 90%.
- For *Elevator*, we changed the number of times the elevator moved from 100 to 5.
- For *Grep*, we searched on one 1MB text file with 4440 lines.
- For *Kanzi*, we compressed the 5 smallest class files in `rt.jar`.
- For *Email*, we executed scenarios 1, 3, and 5.
- For *Prevayler*, we calculated and stored the first 10 prime numbers.
- For *Sort*, we sorted on three 1MB text files with 15 thousand lines.
- For *H*2, we changed the `size` parameter from 100 to 5.
- For *Berkeley DB*, we changed the `n_records` parameter from 500 thousand to 10.
- For *Lucene*, we trimmed the file to index from 8 million to 300 thousand lines.
- For *Density Converter*, we reduced the size of the photo to process by 90%.

Bibliography

Jprofiler 10, 2019. URL `https://www.ej-technologies.com/products/jprofiler/overview.html`. Cited in pages 6, 12, 45, 69

Visualvm, 2020. URL `https://visualvm.github.io/`. Cited in pages 6, 12

Iago Abal, Jean Melo, Ştefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. Variability bugs in highly configurable systems: A qualitative analysis. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 26(3):10:1–10:34, January 2018. ISSN 1049-331X. Cited in pages 16

Mathieu Acher, Hugo Martin, Juliana Alves Pereira, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Eddine Khelladi, Luc Lesoil, and Olivier Barais. Learning Very Large Configuration Spaces: What Matters for Linux Kernel Sizes. Research report, Inria Rennes - Bretagne Atlantique, October 2019. URL `https://hal.inria.fr/hal-02314830`. Cited in pages 83

Andrea Adamoli and Matthias Hauswirth. Trevis: A context tree visualization and analysis framework and its use for classifying performance failure reports. In *Proc. Int'l Symposium Software Visualization (SOFTVIS)*, page 73–82, New York, NY, USA, 2010. ACM. Cited in pages 6, 12

Hiralal Agrawal and Joseph R Horgan. Dynamic program slicing. *ACM SIGPlan Notices*, 25(6): 246–256, 1990. Cited in pages 6, 12, 13

Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. Incling: Efficient product-line testing using incremental pairwise sampling. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 144–155, New York, NY, USA, October 2016. ACM. Cited in pages 22, 33

Mohammad Mejbah ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proc. European Conference on Computer Systems (EuroSys)*, page 298–313, New York, NY, USA, April 2017. ACM. ISBN 9781450349383. Cited in pages 6

Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. Systematic approaches for increasing soundness and precision of static analyzers. In *Proc. Int'l Workshop State Of the Art in Program Analysis (SOAP)*, pages 31–36, New York, NY, USA, 6 2017. ACM. ISBN 978-1-4503-5072-3. doi: 10.1145/3088515.3088521. Cited in pages 82

David Andrzejewski, Anne Mulhern, Ben Liblit, and Xiaojin Zhu. Statistical debugging using latent topic models. In *Proc. European Conf. Machine Learning*, page 6–17, Berlin, Heidelberg, September 2007. Springer-Verlag. Cited in pages 6, 12, 13

Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer-Verlag, Berlin/Heidelberg, Germany, 2013. Cited in pages 1, 3, 51, 56

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 259–269, New York, NY, USA, June 2014. ACM. Cited in pages 14, 35, 36

Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. Workshop Programming Languages and Analysis for Security (PLAS)*, pages 113–124, New York, NY, USA, June 2009. ACM. Cited in pages 14

Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proc. Symp. Principles of Programming Languages (POPL)*, pages 165–178, New York, NY, USA, January 2012. ACM. Cited in pages 14

Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 426–436, Piscataway, NJ, USA, May 2015. IEEE. Cited in pages 36, 82

Paulo Barros, Rene Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo dAmorim, and Michael D. Ernst. Static analysis of implicit control flow: Resolving java reflection and android intents (t). In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 669–679, Washington, DC, USA, 2015. IEEE. Cited in pages 82

Farnaz Behrang, Myra B. Cohen, and Alessandro Orso. Users beware: Preference inconsistencies ahead. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)*, page 295–306, New York, NY, USA, August 2015. ACM. ISBN 9781450336758. Cited in pages 1

Jonathan Bell and Gail Kaiser. Phosphor: Illuminating dynamic data flow in commodity JVMs. *SIGPLAN Notices*, 49(10):83–101, October 2014. ISSN 0362-1340. Cited in pages 14, 35, 41

Cor-Paul Bezemer, J.A. Pouwelse, and Brendan Gregg. Understanding software performance regressions using differential flame graphs. In *Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER)*, pages 535–539. IEEE, March 2015. Cited in pages 6, 12

Eric Bodden. Self-adaptive static analysis. In *Proc. Int'l Conf. Software Engineering (ICSE): New Ideas and Emerging Results*, pages 45–48, New York, NY, USA, 2018. ACM. Cited in pages 36, 82

James Bornholt and Emina Torlak. Finding code that explodes under symbolic evaluation. *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2:149:1–149:26, October 2018. ISSN 2475-1421. doi: 10.1145/3276519. URL `http://doi.acm.org/10.1145/3276519`. Cited in pages 6, 12

Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information needs in bug reports: Improving cooperation between developers and users. In *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, page 301–310, New York, NY, USA, February 2010. ACM. Cited in pages 3, 5, 12

Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. Interactive record/replay for web application debugging. In *Proc. Symposium User Interface Software and Technology (UIST)*, page 473–484, New York, NY, USA, October 2013. ACM. Cited in pages 13

Pablo De Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. Cere: Llvm-based codelet extractor and replayer for piecewise benchmarking and optimization. *ACM Trans. Archit. Code Optim. (TACO)*, 12(1):6:1–6:24, April 2015. ISSN 1544-3566. Cited in pages 5, 6, 12

Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. Detecting missing information in bug descriptions. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)*, pages 396–407, New York, NY, USA, 9 2017. ACM. ISBN 978-1-4503-5105-8. Cited in pages 3, 5, 12

Shaiful Alam Chowdhury and Abram Hindle. Greenoracle: Estimating software energy consumption with energy measurement corpora. In *Proc. Int'l Conf. Mining Software Repositories*, page 49–60, New York, NY, USA, May 2016. ACM. Cited in pages 1

Maria Christakis and Christian Bird. What developers want and need from program analysis: An empirical study. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 332–343, New York, NY, USA, 2016. ACM. Cited in pages 82

Jürgen Cito, Philipp Leitner, Christian Bosshard, Markus Knecht, Genc Mazlami, and Harald C. Gall. PerformanceHat: Augmenting source code with runtime performance traces in the IDE. In *Proc. Int'l Conf. Software Engineering: Companion Proceeedings*, pages 41–44, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5663-3. Cited in pages 5, 6, 12

Charlie Curtsinger and Emery D. Berger. COZ: Finding code that counts with causal profiling. In *USENIX Annual Technical Conference (ATC)*, Denver, CO, USA, June 2016. USENIX Association. Cited in pages 5, 6, 12

Nils Dahlbäck, Arne Jönsson, and Lars Ahrenberg. Wizard of oz studies—why and how. *Knowledge-Based Systems*, 6(4):258–266, 1993. Cited in pages 15

Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. Just-in-time static analysis. In *Proc. Int'l Symp. Software Testing and Analysis (ISSTA)*, pages 307–317, New York, NY, USA, 2017. ACM. Cited in pages 36, 82

Z. Dong, A. Andrzejak, D. Lo, and D. Costa. ORPLocator: Identifying read points of configuration options via static analysis. In *Proc. Int'l Symposium Software Reliability Engineering (ISSRE)*, pages 185–195. IEEE, October 2016. Cited in pages 13

Finale Doshi-Velez and Been Kim. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*, 2017. Cited in pages 48, 81

Tayba Farooqui, Tauseef Rana, and Fakeeha Jafari. Impact of human-centered design process (hcdp) on software development process. In *Int'l Conf. Communication, Computing and Dig-*

ital systems (C-CODE), pages 110–114, March 2019. doi: 10.1109/C-CODE.2019.8680978. Cited in pages 4, 7, 11, 15

Xiaoqin Fu, Haipeng Cai, and Li Li. Dads: Dynamic slicing continuously-running distributed programs with budget constraints. In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)*, page 1566–1570, New York, NY, USA, 2020. ACM. URL `https://doi.org/10.1145/3368089.3417920`. Cited in pages 25

Diego Garbervetsky, Edgardo Zoppi, and Benjamin Livshits. Toward full elasticity in distributed static analysis: The case of callgraph analysis. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)*, pages 442–453, New York, NY, USA, 9 2017. ACM. ISBN 978-1-4503-5105-8. doi: 10.1145/3106237.3106261. Cited in pages 82

Erol Gelenbe and Yves Caseau. The impact of information technology on energy consumption and carbon emissions. *Ubiquity*, 2015(June), June 2015. Cited in pages 1

Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. *SIGPLAN Notices*, 42(10):57–76, October 2007. ISSN 0362-1340. Cited in pages 47

Dennis Giffhorn. Advanced chopping of sequential and concurrent programs. *Software Quality Journal*, 19(2):239–294, 2011. Cited in pages 69

Jürgen Graf, Martin Hecker, and Martin Mohr. Using joana for information flow control in java programs - a practical guide. Technical Report 24, Karlsruhe Institute of Technology, 2012. Cited in pages 68

Alexander Grebhahn, Norbert Siegmund, and Sven Apel. Predicting performance of software configurations: There is no silver bullet, 2019. Cited in pages 4, 21, 22, 23, 47

Brendan Gregg. The flame graph. *Commun. ACM*, 59(6):48–57, May 2016. ISSN 0001-0782. Cited in pages 5, 6, 12

Jiaping Gui, Ding Li, Mian Wan, and William G. J. Halfond. Lightweight measurement and estimation of mobile ad energy consumption. In *Proc. Int'l Workshop Green and Sustainable Software (GREENS)*, page 1–7, New York, NY, USA, May 2016. ACM. ISBN 9781450341615. Cited in pages 1

Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wąsowski. Variability-aware performance prediction: A statistical learning approach. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 301–311, New York, NY, USA, November 2013. ACM. Cited in pages 4, 14, 21, 22, 47, 61

Jianmei Guo, Dingyu Yang, N. Siegmund, S. Apel, Atrisha Sarkar, Pavel Valov, K. Czarnecki, A. Wasowski, and H. Yu. Data-efficient performance learning for configurable systems. *Empirical Software Engineering*, 23:1826–1867, 2017. Cited in pages 21, 22, 46, 47, 61

H. Ha and H. Zhang. Performance-influence model for highly configurable software with fourier learning and lasso regression. In *Proc. Int'l Conf. Software Maintance and Evolution (ICSME)*, pages 470–480, September 2019. Cited in pages 6, 14, 20, 21, 22, 61, 81

Huong Ha and Hongyu Zhang. DeepPerf: Performance prediction for configurable software with deep sparse neural network. In *Proc. Int'l Conf. Software Engineering (ICSE)*, page

1095–1106. IEEE, May 2019. Cited in pages 21, 22, 61

Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. Test them all, is it worth it? assessing configuration sampling on the JHipster web development stack. *Empirical Software Engineering*, July 2018. Cited in pages 1, 22, 46, 83

Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 145–155, Piscataway, NJ, USA, June 2012. IEEE. Cited in pages 12

Xue Han and Tingting Yu. An empirical study on performance bugs for highly configurable software systems. In *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*, pages 23:1–23:10, New York, NY, USA, September 2016. ACM. Cited in pages 3, 5, 13, 16, 17, 31, 83

Xue Han, Tingting Yu, and David Lo. Perflearner: Learning from bug reports to understand and generate performance test frames. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 17–28, New York, NY, USA, 9 2018. ACM. ISBN 978-1-4503-5937-5. Cited in pages 5, 12, 13

Xue Han, Tingting Yu, and Michael Pradel. Confprof: White-box performance profiling of configuration options. In *Proc. Int'l Conf. Performance Engineering (ICPE)*, page 1–8, New York, NY, USA, April 2021. ACM. Cited in pages 14

Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. Energy profiles of java collections classes. In *Proc. Int'l Conf. Software Engineering (ICSE)*, New York, NY, USA, May 2016. ACM. Cited in pages 1

Haochen He, Zhouyang Jia, Shanshan Li, Erci Xu, Tingting Yu, Yue Yu, Ji Wang, and Xiangke Liao. Cp-detector: Using configuration-related performance properties to expose performance bugs. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, November 2020. Cited in pages 5, 12, 13, 46

A. Hervieu, B. Baudry, and A. Gotlieb. PACOGEN: Automatic generation of pairwise test configurations from feature models. In *Proc. Int'l Symposium Software Reliability Engineering*, pages 120–129, November 2011. Cited in pages 22, 33

Aymeric Hervieu, Dusica Marijan, Arnaud Gotlieb, and Benoit Baudry. Optimal Minimisation of Pairwise-covering Test Configurations Using Constraint Programming. *Information and Software Technology*, 71:129 – 146, March 2016. Cited in pages 22, 33

Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 199–212, New York, NY, USA, March 2011. ACM. Cited in pages 13, 14

Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. A user survey of configuration challenges in Linux and eCos. In *Proc. Workshop Variability Modeling of Software-Intensive Systems (VAMOS)*, pages 149–155, New York, NY, USA, January 2012. ACM. ISBN 978-1-4503-1058-1. Cited in pages 1, 3

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proc. Int'l Conf. Learning and Intelligent Optimization*, pages 507–523, Berlin/Heidelberg, Germany, January 2011. Springer-Verlag. Cited in pages 4, 21, 61

Riitta Jääskeläinen. *Think-aloud protocol.* John Benjamins Publishing Amsterdam/Philadelphia, 2010. Cited in pages 15, 70, 74

Reyhaneh Jabbarvand, Alireza Sadeghi, Joshua Garcia, Sam Malek, and Paul Ammann. Ecodroid: An approach for energy-based ranking of android apps. In *Proc. Int'l Workshop Green and Sustainable Software (GREENS)*, pages 8–14, Piscataway, NJ, USA, May 2015. IEEE. Cited in pages 1

Pooyan Jamshidi and Giuliano Casale. 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)*, pages 39–48, September 2016. Cited in pages 4, 21, 61

Pooyan Jamshidi, Miguel Velez, Christian Kästner, Norbert Siegmund, and Prasad Kawthekar. Transfer learning for improving model predictions in highly configurable software. In *Proc. Int'l Symp. Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 31–41, Los Alamitos, CA, USA, May 2017. IEEE. Cited in pages 4, 21, 22

Pooyan Jamshidi, Miguel Velez, Christian Kästner, and Norbert Siegmund. Learning to sample: Exploiting similarities across environments to learn performance models for configurable systems. In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)*, pages 71–82, New York, NY, USA, November 2018. ACM. Cited in pages 4, 21

Dongpu Jin, Xiao Qu, Myra B. Cohen, and Brian Robinson. Configurations everywhere: Implications for testing and debugging in practice. In *Companion Proc. Int'l Conf. Software Engineering*, pages 215–224, New York, NY, USA, May 2014. ACM. Cited in pages 1

Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 77–88, New York, NY, USA, June 2012. ACM. Cited in pages 3, 5, 13, 16, 31

Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch me if you can: Performance bug detection in the wild. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, page 155–170, New York, NY, USA, October 2011. ACM. Cited in pages 3, 5

Natalia Juristo and Omar S. Gómez. *Replication of Software Engineering Experiments.* 2011. Cited in pages 70

C. Kaltenecker, A. Grebhahn, N. Siegmund, and S. Apel. The interplay of sampling and machine learning for software performance prediction. *IEEE Software*, 2020. Cited in pages 4, 21, 22, 23

Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. Distance-based sampling of software configuration spaces. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, May 2019. Cited in pages 14, 21, 22, 46, 48

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

Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don Batory, Sabrina Souto, Paulo Barros, and Marcelo d'Amorim. SPLat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)*, pages 257–267, New York, NY, USA, August 2013. ACM. Cited in pages 56

James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. ISSN 0001-0782. Cited in pages 12, 14

Andrew J. Ko and Brad A. Myers. Designing the whyline: A debugging interface for asking questions about program behavior. In *Proc. Conf Human Factors in Computing Systems (CHI)*, New York, NY, USA, April 2004. ACM. Cited in pages 13, 25

Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, Alexander Grebhahn, and Sven Apel. Tradeoffs in modeling performance of highly configurable software systems. *Software and System Modeling (SoSyM)*, February 2018. Cited in pages 4, 6, 14, 20, 21, 22, 49, 81

Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information processing letters*, 29 (3):155–163, 1988. Cited in pages 6, 12, 13

Jens Krinke. Barrier slicing and chopping. In *Int'l Workshop Source Code Analysis and Manipulation (SCAM)*, Amsterdam, Netherlands, September 2003. IEEE. Cited in pages 82

Rahul Krishna, Md Shahriar Iqbal, Mohammad Ali Javidian, Baishakhi Ray, and Pooyan Jamshidi. CADET: A systematic method for debugging misconfigurations using counterfactual reasoning, 2020. Cited in pages 3, 5, 13, 83

D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. *Introduction to Combinatorial Testing*. Chapman & Hall/CRC, 1st edition, 2013. Cited in pages 33

Fabian Lange. Measure Java performance – sampling or instrumentation?, January 2011. Cited in pages 35

T. D. LaToza and B. A. Myers. Visualizing call graphs. In *Symposium Visual Languages and Human-Centric Computing (VL/HCC)*, pages 117–124, November 2011. Cited in pages 12

J. Lerch, J. Späth, E. Bodden, and M. Mezini. Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths (t). In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 619–629, Washington, DC, USA, November 2015. IEEE. Cited in pages 36, 82

Chi Li, Shu Wang, Henry Hoffmann, and Shan Lu. Statically inferring performance properties of software configurations. In *Proc. European Conf. Computer Systems (EuroSys)*, New York, NY, USA, April 2020. ACM. ISBN 9781450368827. Cited in pages 6, 13

Ding Li, Yingjun Lyu, Jiaping Gui, and William G.J. Halfond. Automated energy optimization of http requests for mobile applications. New York, NY, USA, May 2016. ACM. Cited in pages 1, 5, 13

Max Lillack, Christian Kästner, and Eric Bodden. Tracking load-time configuration options.

IEEE Transactions on Software Engineering, 44(12):1269–1291, 12 2018. ISSN 0098-5589. Cited in pages 6, 13, 14, 46

Zachary C. Lipton. The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery. *Queue*, 16(3):31–57, June 2018. Cited in pages 48, 49, 81

Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proc. Int'l Conf. Software Engineering (ICSE)*, ICSE 2014, pages 1013–1024, New York, NY, USA, May 2014. ACM. Cited in pages 6, 12

Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017. Cited in pages 21, 49

Haroon Malik, Peng Zhao, and Michael Godfrey. Going green: An exploratory analysis of energy-related questions. In *Proc. Int'l Conf. Mining Software Repositories*, page 418–421. IEEE Press, May 2015. ISBN 9780769555942. Cited in pages 1

Irene Manotas, Christian Bird, Rui Zhang, David Shepherd, Ciera Jaspan, Caitlin Sadowski, Lori Pollock, and James Clause. An empirical study of practitioners' perspectives on green software engineering. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 237–248, New York, NY, USA, May 2016. ACM. Cited in pages 1

Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A comparison of 10 sampling algorithms for configurable systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 643–654, New York, NY, USA, May 2016. ACM. Cited in pages 22, 23, 46, 47

Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. On essential configuration complexity: Measuring interactions in highly-configurable systems. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 483–494, New York, NY, USA, September 2016. ACM. Cited in pages 6, 13, 14, 29, 30, 31, 56

Jens Meinicke, Chu-Pan Wong, Christian Kästner, and Gunter Saake. Understanding differences among executions with variational traces. *arXiv preprint arXiv:1807.03837*, 2018. Cited in pages 12, 13, 14, 16

Jean Melo, Claus Brabrand, and Andrzej Wąsowski. How does the degree of variability affect bug finding? In *Proc. Int'l Conf. Software Engineering (ICSE)*, page 679–690, New York, NY, USA, May 2016. ACM. ISBN 9781450339001. Cited in pages 1, 16

Jean Melo, Fabricio Batista Narcizo, Dan Witzner Hansen, Claus Brabrand, and Andrzej Wasowski. Variability through the eyes of the programmer. In *Proc. Int'l Conference Program Comprehension (ICPC)*, page 34–44. IEEE, May 2017. Cited in pages 1, 16

Christoph Molnar. *Interpretable Machine Learning*. 2019. `https://christophm.github.io/interpretable-ml-book/`. Cited in pages 21, 23, 48, 49, 81

Douglas C. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, 2006. Cited in pages 22

Mukelabai Mukelabai, Damir Nešić, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. Tackling combinatorial explosion: A study of industrial needs and practices for analyzing highly configurable systems. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 155–166, New York, NY, USA, 9 2018. ACM. Cited in pages 83

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

Brad A. Myers, Andrew J. Ko, Thomas D. LaToza, and YoungSeok Yoon. Programmers are users too: Human-centered methods for improving programming tools. 49(7):44–52, July 2016. ISSN 0018-9162. Cited in pages 4, 7, 11, 15

Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. Using bad learners to find good configurations. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)*, ESEC/FSE 2017, page 257–267, New York, NY, USA, August 2017. ACM. Cited in pages 4, 14, 21, 61

Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, page 89–100, New York, NY, USA, June 2007. ACM. Cited in pages 6, 12

James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software, 2005. Cited in pages 14

Thanhvu Nguyen, Thanhvu Koc, Javran Cheng, Jeffrey S. Foster, and Adam A. Porter. iGen dynamic interaction inference for configurable software. In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)*, Los Alamitos, CA, USA, November 2016. IEEE. Cited in pages 13, 14, 29, 30, 31

Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Comput. Surv. (CSUR)*, 43(2):11:1–11:29, February 2011. ISSN 0360-0300. Cited in pages 22, 23

Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *Proc. Int'l Conf. Mining Software Repositories*, pages 237–246, Piscataway, NJ, USA, May 2013a. IEEE. Cited in pages 3, 5, 12, 31

Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 562–571, Piscataway, NJ, USA, 5 2013b. IEEE. Cited in pages 6, 12

Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 902–912, Piscataway, NJ, USA, May 2015. IEEE. Cited in pages 6, 12, 31

Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. Finding near-optimal configurations in product lines by random sampling. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)*, pages 61–71, New York, NY, USA, September 2017. ACM. Cited in pages 4, 14, 21, 48, 61

Rafael Olaechea, Derek Rayside, Jianmei Guo, and Krzysztof Czarnecki. Comparison of exact and approximate multi-objective optimization for software product lines. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 92–101, New York, NY, USA, September 2014. ACM. Cited in pages 4, 21, 61

J. Park, M. Kim, B. Ray, and D. Bae. An empirical study of supplementary bug fixes. In *Proc.*

Int'l Conf. Mining Software Repositories, pages 40–49, June 2012. Cited in pages 3, 5

Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proc. Int'l Symp. Software Testing and Analysis (ISSTA)*, pages 199–209, New York, NY, USA, July 2011. ACM. Cited in pages 3, 5, 7, 16

Felix Pauck, Eric Bodden, and Heike Wehrheim. Do android taint analysis tools keep their promises? In *Proc. Int'l Symp. Foundations of Software Engineering (FSE)*, pages 331–341, New York, NY, USA, 11 2018. ACM. ISBN 978-1-4503-5573-5. doi: 10.1145/3236024.3236029. Cited in pages 36

Rui Pereira, Marco Couto, João Saraiva, Jácome Cunha, and João Paulo Fernandes. The influence of the java collection framework on overall energy consumption. In *Proc. Int'l Workshop Green and Sustainable Software (GREENS)*, page 15–21, New York, NY, USA, May 2016. ACM. Cited in pages 1

Gustavo Pinto and Fernando Castor. Energy efficiency: A new concern for application software developers. *Commun. ACM*, 60(12):68–75, November 2017. Cited in pages 1

Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, page 535–552, New York, NY, USA, October 2007. ACM. Cited in pages 13

Lina Qiu, Yingying Wang, and Julia Rubin. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In *Proc. Int'l Symp. Software Testing and Analysis (ISSTA)*, pages 176–186, New York, NY, USA, 2018. ACM. Cited in pages 36

Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 131–140, New York, NY, USA, May 2011. ACM. Cited in pages 13

Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 445–454. ACM, New York, NY, USA, May 2010. Cited in pages 13, 14, 29, 30, 31

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "Why should I trust you?": Explaining the predictions of any classifier. In *Proc. Int'l Conf. Knowledge Discovery and Data Mining (KDD)*, pages 1135–1144, New York, NY, USA, August 2016. ACM. Cited in pages 21, 49

Cynthia Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, (5):206–215, 2019. Cited in pages 21, 49

Johnny Saldaña. *The coding manual for qualitative researchers*. Sage, 2015. Cited in pages 16, 71, 75

J. P. Sandoval Alcocer, F. Beck, and A. Bergel. Performance evolution matrix: Visualizing performance variations along software versions. In *Conf. Software Visualization (VISSOFT)*, pages 1–11. IEEE, September 2019. doi: 10.1109/VISSOFT.2019.00009. Cited in pages 6, 12

Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. Cost-efficient sampling for performance prediction of configurable systems. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 342–352, Washington, DC, USA, November 2015. IEEE. Cited in pages 21, 22, 46, 47, 61

Stefan Schmidt. Shall we really do it again? the powerful concept of replication is neglected in the social sciences. *Review of General Psychology*. Cited in pages 70

Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 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)*, pages 317–331, Washington, DC, USA, May 2010. IEEE. Cited in pages 14

Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 167–177, Piscataway, NJ, USA, June 2012a. IEEE. Cited in pages 14, 21, 22, 46, 47

Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. SPLConqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal*, 20(3-4):487–517, September 2012b. ISSN 0963-9314. Cited in pages 14, 21, 22, 46, 47

Norbert Siegmund, Alexander von Rhein, and Sven Apel. Family-based performance measurement. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 95–104, New York, NY, USA, October 2013. ACM. Cited in pages 23, 31, 46, 47

Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. Performance-influence models for highly configurable systems. In *Proc. Europ. Software Engineering Conf. Foundations of Software Engineering (ESEC/FSE)*, pages 284–294, New York, NY, USA, August 2015. ACM. Cited in pages 3, 4, 6, 14, 20, 21, 22, 46, 47, 81

Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, page 561–578, New York, NY, USA, October 2014. ACM. Cited in pages 6, 12

Linhai Song and Shan Lu. Performance diagnosis for inefficient loops. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 370–380, Piscataway, NJ, USA, May 2017. IEEE. Cited in pages 5, 6, 12, 13

Sabrina Souto, Marcelo d'Amorim, and Rohit Gheyi. Balancing soundness and efficiency for practical testing of configurable systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 632–642, Piscataway, NJ, USA, May 2017. IEEE. Cited in pages 56

Johannes Späth, Karim Ali, and Eric Bodden. Ideal: Efficient and precise alias-aware dataflow analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA):99:1–99:27, October 2017. ISSN 2475-1421. doi: 10.1145/3133923. Cited in pages 82

Erik Štrumbelj and Igor Kononenko. Explaining prediction models and individual predictions with feature contributions. *Knowledge and information systems*, 41(3):647–665, 2014. Cited in pages 21, 49

Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv. (CSUR)*, 47 (1):6:1–6:45, June 2014. ISSN 0360-0300. Cited in pages 14

John Toman and Dan Grossman. Staccato: A bug finder for dynamic configuration updates. In *Proc. European Conf. Object-Oriented Programming (ECOOP)*, Dagstuhl, Germany, July 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. Cited in pages 6, 13, 14

John Toman and Dan Grossman. Legato: An at-most-once analysis with applications to dynamic configuration updates. In *European Conf. Object-Oriented Programming (ECOOP)*, Amsterdam, Netherlands, 7 2018. Cited in pages 13, 14

Jonas Trümper, Jürgen Döllner, and Alexandru Telea. Multiscale visual comparison of execution traces. In *Proc. Intl Conf. Program Comprehension (ICPC)*, pages 53–62, May 2013. Cited in pages 6, 12

Miguel Velez, Pooyan Jamshidi, Florian Sattler, Norbert Siegmund, Sven Apel, and Christian Kästner. Configcrusher: Towards white-box performance analysis for configurable systems. *Autom Softw Eng*, 2020. Cited in pages 28, 36

Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. On debugging the performance of configurable software systems: Developer needs and tailored tool support. In *Under review*, 2021a. Cited in pages 12, 64

Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. White-box analysis over machine learning: Modeling performance of configurable systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, May 2021b. Cited in pages 28, 36

Bo Wang, Leonardo Passos, Yingfei Xiong, Krzysztof Czarnecki, Haiyan Zhao, and Wei Zhang. Smartfixer: Fixing software configurations based on dynamic priorities. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 82–90, New York, NY, USA, 8 2013. ACM. ISBN 978-1-4503-1968-3. doi: 10.1145/2491627.2491640. Cited in pages 13

Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. Understanding and auto-adjusting performance-sensitive configurations. In *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 154–168, New York, NY, USA, March 2018. ACM. Cited in pages 4, 5, 13, 21

Yan Wang, Hailong Zhang, and Atanas Rountev. On the unsoundness of static analysis for android guis. In *Proc. Int'l Workshop State Of the Art in Program Analysis (SOAP)*, pages 18–23, New York, NY, USA, 2016. ACM. Cited in pages 36

Max Weber, Sven Apel, and Norbert Siegmund. White-Box Performance-Influence models: A profiling and learning approach. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, May 2021. Cited in pages 6, 14, 20, 22, 24, 46, 81

Mark Weiser. Program slicing. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 439–449, Piscataway, NJ, USA, March 1981. IEEE. Cited in pages 6, 12, 13

Claas Wilke, Sebastian Richly, Sebastian Götz, Christian Piechnick, and Uwe Amann. Energy consumption and efficiency in mobile applications: A user feedback study. In *Proc. Int'l Conf. Green Computing and Communications*, page 134–141. IEEE, 2013. Cited in pages 1, 5, 13

Chu-Pan Wong, Jens Meinicke, Lukas Lazarek, and Christian Kästner. Faster variational execution with transparent bytecode transformation. *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2, October 2018. Cited in pages 6, 13, 14, 29, 30, 31

Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005. Cited in pages 26

Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *Proc. Symp. Operating Systems Principles*, pages 244–259, New York, NY, USA, November 2013. ACM. Cited in pages 1, 3, 4, 13, 21

Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 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)*, pages 307–319, New York, NY, USA, August 2015. ACM. Cited in pages 1, 3

Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early detection of configuration errors to reduce failure damage. In *Proc. Conf.Operating Systems Design and Implementation (OSDI)*, pages 619–634, Berkeley, CA, USA, November 2016. USENIX Association. Cited in pages 6, 13

Tingting Yu and Michael Pradel. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *Proc. Int'l Symp. Software Testing and Analysis (ISSTA)*, page 389–400, New York, NY, USA, July 2016. ACM. Cited in pages 6

Tingting Yu and Michael Pradel. Pinpointing and repairing performance bottlenecks in concurrent programs. *Empirical Softw. Eng.*, 23(5):3034–3071, October 2018. ISSN 1382-3256. Cited in pages 5, 6, 12

Andreas Zeller. Yesterday, my program worked. today, it does not. why? *SIGSOFT Softw. Eng. Notes*, 24(6):253–267, October 1999. ISSN 0163-5948. Cited in pages 6, 12, 13

Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009. Cited in pages 3, 5, 13, 15

C. Zhang, A. Hindle, and D. M. German. The impact of user choice on energy consumption. *IEEE Software*, 31(3):69–75, 2014. doi: 10.1109/MS.2014.27. Cited in pages 1

Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. Encore: Exploiting system environment and correlation information for misconfiguration detection. In *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 687–700, New York, NY, USA, March 2014. ACM. Cited in pages 13

Qirun Zhang and Zhendong Su. Context-sensitive data-dependence analysis via linear conjunctive language reachability. In *Proc. Symp. Principles of Programming Languages (POPL)*, pages 344–358, New York, NY, USA, 1 2017. ACM. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009848. Cited in pages 82

Sai Zhang and Michael D. Ernst. Which configuration option should i change? In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 152–163, New York, NY, USA, May 2014. ACM. Cited in pages 6, 13

Sai Zhang and Michael D. Ernst. Proactive detection of inadequate diagnostic messages for software configuration errors. In *Proc. Int'l Symp. Software Testing and Analysis (ISSTA)*, pages 12–23, New York, NY, USA, July 2015. ACM. Cited in pages 13

Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. Bestconfig: Tapping the performance potential of systems via automatic configuration tuning. In *Proc. Symposium Cloud Computing (SoCC)*, pages 338–350, New York, NY, USA, 9 2017. ACM. ISBN 978-1-4503-5028-0. Cited in pages 4, 14, 21, 48, 61