Black-Box Performance Modeling of Configurable Software Systems

Dissertation zur Erlangung des Grades des

Doktors der Ingenieurwissenschaften (*Dr. Ing.*)

der Fakultät für Mathematik und Informatik der Universität des Saarlandes

vorgelegt von

Christian Kaltenecker

Saarbrücken, 2023

UNIVERSITÄT
DES
SAARLANDES

Dean of Faculty	Prof. Dr. Jürgen Steimle
Day of Colloquium	29.01.2024
Chair of the Committee	Prof. Dr. Jan Reineke
Reviewers	Prof. Dr. Sven Apel
	Prof. Dr. Mathieu Acher
Academic Assistant	Dr. Sebastian Biewer

Abstract

Software systems have become an important part of our daily lives, and a multitude of different application scenarios, user requirements, and hardware requirements have emerged. To handle these different requirements, most software systems offer some degree of configurability in terms of configuration options, allowing the user to adapt the software system to functional and non-functional requirements. Among non-functional requirements, the performance of the software system plays an important role to end-users.

It is often unclear which configuration options influence the performance of the system. Specifically, there is a gap in how to select configurations affecting the system's performance when no previous knowledge is available. Furthermore, little is known about how the influence of configuration options on the system's performance changes across different workloads and software evolution. To bridge this gap, performance modeling based on statistical learning has proved useful.

In this thesis, we follow three objectives in which we use or improve performance modeling of configurable software systems by statistical learning. First, we propose a novel sampling strategy, *distance-based sampling*, to improve the configuration selection (i.e., *sampling*) while also addressing the shortcomings of existing state-of-the art sampling strategies. To assess the advantages and limitations of distance-based sampling, we compare it to state-of-the-art sampling strategies on multiple real-world configurable software systems. Our results indicate that distance-based sampling outperforms other state-of-the-art sampling strategies in terms of accuracy, but also suggest that there is still room for improvement with regard to scalability.

Second, to assess how the influence of configuration options on the system's performance changes during software evolution, we use performance modeling on multiple real-world configurable software systems. This investigation delivers multiple valuable insights into the frequency of performance changes with implications to other research domains. We further investigate in how many cases the performance changes are documented by developers and find indications in which cases performance regressions are documented and when they are not.

Third, besides configurability and software evolution, we also assess the role of workload variability in an exploratory study of the configurable software system FastDownward. For this purpose, we propose a performance modeling approach to identify performance changes considering workload variability and investigate the accuracy of this approach by evaluating precision and recall. Our results show that our approach is able to identify most performance changes, but we also identify the limitations of our approach, leaving room for further improvement. Furthermore, our performance measurements proved helpful in that they enabled us to discover and report multiple performance regressions in a real-world configurable software system.

Overall, we contribute to performance modeling of configurable software systems by (1) proposing a novel sampling strategy designed to cover configuration spaces with regards to performance; (2) lifting how performance changes affect software configurability in practice and show implications on other research areas; (3) demonstrating how performance modeling can be used to find performance changes while including workload variability. To the best of our knowledge, we are the first to investigate configurability, evolution, and workload variability of configurable software systems together.

Zusammenfassung

Softwaresysteme sind zu einem wichtigen Bestandteil unseres täglichen Lebens geworden. Durch die Vielfalt der existierenden Softwaresysteme ist eine Vielzahl unterschiedlicher Anwendungsszenarien, Benutzeranforderungen sowie Hardwareanforderungen entstanden. Um diesen unterschiedlichen Anforderungen gerecht zu werden, bieten die meisten Softwaresysteme einen gewissen Grad an Konfigurierbarkeit, der es dem Benutzer ermöglicht, das Softwaresystem an funktionale und nicht-funktionale Anforderungen anzupassen. Unter den nicht-funktionalen Anforderungen spielt die Performance des Softwaresystems eine wichtige Rolle für die Endbenutzer.

Häufig ist unklar, welche Konfigurationsoptionen die Performance des Systems beeinflussen. Aktuelle Verfahren zur Auswahl von Konfigurationen, die die Performance des Systems beeinflussen, weisen Schwachstellen auf. Darüber hinaus ist wenig darüber bekannt, wie sich die Auswirkungen von Konfigurationsoptionen auf die Performance im Laufe der Softwareentwicklung und bei unterschiedlichen Arbeitslasten ändern. Zur Lösung dieser Probleme kann die Performancemodellierung eingesetzt werden.

In dieser Arbeit verfolgen wir drei Ziele, mit denen wir die Performancemodellierung von konfigurierbaren Softwaresystemen nutzen oder verbessern. Erstens stellen wir eine neuartige Sampling-Strategie vor, das *distanzbasiertes Sampling*. Damit kann die Auswahl der Konfigurationen (des *Samplings*) verbessert und gleichzeitig die Unzulänglichkeiten vermieden werden, die andere State-of-the-Art-Samplingstrategien aufweisen. Um die Vorzüge und Grenzen des distanzbasierten Samplings beurteilen zu können, vergleichen wir es mit modernsten Samplingstrategien auf mehreren realen, konfigurierbaren Softwaresystemen. Unsere Ergebnisse zeigen, dass distanzbasiertes Sampling andere moderne Samplingstrategien in Bezug auf die Genauigkeit übertrifft, aber auch, dass es in Bezug auf die Skalierbarkeit noch Raum für Verbesserungen gibt.

Zweitens: Um zu beurteilen, wie sich der Einfluss von Konfigurationsoptionen auf die Performance des Systems über die Evolution der Software hinweg verändert, verwenden wir Performancemodellierung für mehrere reale konfigurierbare Softwaresysteme. Diese Untersuchung liefert wertvolle Einblicke in die Häufigkeit von Performanceänderungen mit Auswirkungen auf andere Forschungsbereiche. Außerdem untersuchen wir, wie häufig die Performanceänderungen von den Entwicklern dokumentiert werden und finden Hinweise darauf, in welchen Fällen Performanceänderungen dokumentiert werden und wann nicht.

Drittens untersuchen wir in einer explorativen Studie zum konfigurierbaren Softwaresystem FastDownward neben der Konfigurierbarkeit und der Softwareevolution auch die Rolle der Arbeitslastvariabilität. Daher schlagen wir einen Ansatz zur Performancemodellierung vor, um Performanceänderungen unter Berücksichtigung der Arbeitslastvariabilität zu identifizieren, und untersuchen die Genauigkeit dieses Ansatzes, indem wir die Präzision und die Ausbeute auswerten. Unsere Ergebnisse zeigen, dass unser Ansatz in der Lage ist, die meisten

Performanceänderungen zu erkennen. Wir zeigen aber auch die Grenzen unseres Ansatzes auf, die zeigen, wo noch weitere Verbesserungen möglich sind. Darüber hinaus haben sich unsere Performancemessungen als hilfreich erwiesen, da sie es uns ermöglichten, mehrere Performanceregressionen in einem realen konfigurierbaren Softwaresystem aufzudecken und an die Entwickler zu melden.

Insgesamt leisten wir einen Beitrag zur Performancemodellierung konfigurierbarer Softwaresysteme, indem wir (1) eine neuartige Samplingstrategie vorschlagen, die darauf ausgelegt ist, Konfigurationsräume in Bezug auf die Leistung abzudecken; (2) aufzeigen, wie sich Performanceänderungen auf die Konfigurierbarkeit von Software in der Praxis auswirken und Auswirkungen auf andere Forschungsbereiche aufzeigen; (3) demonstrieren, wie die Performancemodellierung verwendet werden kann, um Performanceänderungen aufzudecken und zusammen mit der Variabilität der Arbeitslast zu berücksichtigen. Soweit uns bekannt ist, sind wir die Ersten, die Konfigurierbarkeit, Evolution und Arbeitslastvariabilität von konfigurierbaren Softwaresystemen gleichzeitig untersuchen.

Acknowledgments

Writing a dissertation is a challenging and long-lasting endeavor that is often underestimated. I would have not accomplished this dissertation without the support of many persons, which I want to thank in this section.

Foremost, I want to thank my girlfriend Kristela Kaja for her relentless support on writing this thesis and for accepting the invitation to read this thesis, knowing how much work it would be. I also want to thank the rest of my family for always lifting my spirit and supporting me with all the means to finally pursue a study program.

I want to express my gratitude to my advisor Sven Apel for his support in terms of motivation, knowledge, new ideas, his eager eye to spot inconsistencies, but also for having always a friendly ear. Sven was one of the main reasons for me to become a researcher and, ultimately, the reason to move from Passau to Saarbrücken.

I would also like to thank Norbert Siegmund for his special way of providing feedback, which helped me in improving my scientific writing skills. Furthermore, I want to thank Mathieu Acher for accepting the invitation to be my second examiner.

I want to thank my colleagues at the chair for the many activities that increased the team spirit and for supporting me in the last few months by overtaking administrative tasks, which are undenyingly time consuming. Especially, I want to thank Thomas Bock, whom I got to know early in my studies. He has supported me throughout my academic carrier in very different ways and always helped me by sharing his opinion on different important matters. I also want to thank Florian Sattler for sharing new ideas, plenty of helpful book recommendations, and for the collaboration on VaRA. Furthermore, I want to thank my colleagues and friends for proofreading my thesis and general support: Sebastian Böhm, Sandra Greiner, Christian Hechtl, Anna-Maria Maurer, and Kallistos Weis. In a similar vein, I also want to thank a former colleague, Alexander Grebhahn, for his ideas and support, especially in the beginning while joining the research team. I also want to thank Eva Reichhart and Friederike Repplinger for their organizational and administrative support.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation and Problem Statement

Software systems simplify our life by taking over various kinds of tasks, not only as part of embedded systems, such as coffee machines, dish washers, or autonomous cars, but also as part of end-user devices, such as smartphones, or the digital infrastructure such as database servers, or network devices. Since software systems are much faster in calculating and solving tasks than human beings, software systems overtake crucial tasks from our daily life (e.g., search engines, which makes manual searching in books obsolete) and, thus, save a lot of time. Ultimately, some sectors not only rely on software, but cannot even operate without them (e.g., banking, movie streaming).

Software systems must evolve constantly to adapt to changes of hardware and user requirements [152]. This evolution of software is mainly driven by the integration of new functionality, refactoring, and bug fixes [98]. Besides functionality, the performance of a software system may change considerably over time. For example, it has been observed that often software updates accidentally slow down specific parts of the software [44]. This is a major problem for software systems: Such performance regressions do not save time, but often waste time. For instance, a software incident in 2019 involving performance regressions at Salesforce[1] affected eight data centers and led to a negative user experience for many clients across many organizations[2]. Another more recent incident from 2022 affected the streaming service Netflix[3], where a performance regression degraded the latency[4] by 50% while upgrading the machine's hardware. Syncsort [146] estimated in 2018 that 59% of Fortune 500 companies (i.e., a magazine focusing on the 500 corporations with the highest turnover of the United States) have a minimum downtime of 1.6 hours per week due to software failure, which results in an estimated loss of $ 896 000 per week. Zaman et al. [154] report that many of the failures are due to performance issues; Han and Yu [44] report that most of these performance issues are configuration issues. This emphasizes the prominent role of performance issues in highly configurable software systems, which we will address in this thesis.

1 `https://www.salesforce.com/`, last accessed on 02/16/2023.

2 `https://martech.org/performance-degradation-affecting-salesforce-clients/`, last accessed on 02/16/2023.

3 `https://netflixtechblog.com/seeing-through-hardware-counters-a-journey-to-threefold-performance-increase-2721924a2822`, last accessed on 02/16/2023.

4 Latency is the delay between a user's action and the received response. Typically, latency appears in server–client software.

Configurable software systems Software systems became increasingly configurable in the past decades [88, 116]. A multitude of configuration options have been introduced to software systems to support higher numbers of users and environments by offering users and admins the possibility to adjust the software system to their environment and needs. Furthermore, these configuration options allow for optimizing the software's performance [112], to save time and resources. The downside of adding more and more configuration options is that it may confuse users. In this vein, Xu et al. [152] found that 48.5% of the occurring configuration problems originate from difficulties to find the right configuration option or to choose the right value for the respective configuration option. Moreover, they found that 51.9% of the configuration options are unused by a majority of the users.

Let us take a compression program as a simple example that provides two different algorithms for file compression; the algorithms exhibit different runtime behavior (i.e., some compression algorithms need less time to compress a file than others) and achieve different results in reducing the file size. If each algorithm can be tuned in addition by, for instance, adjusting the compression quality, the user has to determine not only which compression algorithm is best suited, but also which compression quality is most suitable.

A major consequence of maintaining a high number of configuration options is the resulting combinatorial explosion [94] of the configuration space (i.e., each additional configuration option that allows for switching functionality on or off doubles the number of configurations). In our example, a user who wants to shrink the file size of a certain file as fast as possible has to assess the differences between the algorithms first, and then the differences of their compression quality. Even worse, in many cases it is not clear from the documentation which of the algorithms performs best in terms of performance. To save time and resources, knowing which algorithm performs best is crucial when the software is deployed in a larger setting, such as in databases.

As a solution, domain knowledge from an experienced user or a developer of the software can help. That is, experienced users or developers of the software can tell which configuration options are known to have an impact on the performance of the software and which do not. However, such domain knowledge is only rarely (publicly) available. Without prior domain knowledge, the performance of all algorithms and their compression qualities have to be measured to identify the fastest among them. For a compression software providing two compression algorithms with ten configuration options for each compression algorithm, this results in measuring $2^{11} = 2\,048$ configurations. This combinatorial explosion of the configuration space turns measuring all configurations of larger software systems into an infeasible task, as the following thought experiment illustrates: A software with 300 configuration options has 2^{300} configurations in the worst case, which is more than the estimated number of atoms in the universe[5]. Even worse, some configuration options (e.g., compression level, maximum memory consumption) cannot only be selected or deselected, but also accept a certain numeric value. This poses a central problem for configuration testing [94], where the goal is to identify configuration issues such as undocumented invalid configurations or performance issues.

In the past, research [112, 133, 139] focused on approaches for identifying configuration-specific issues as they require special techniques to analyze the large number of configurations since measuring all configurations is infeasible [139]. Sampling of software configurations (i.e.,

5 https://www.thoughtco.com/number-of-atoms-in-the-universe-603795, last accessed on 02/16/2023.

selecting a small amount of configurations that can be measured in feasible time) and applying machine learning on the sample set serves as one possible solution. But multiple questions arise when selecting configurations for a sample set: How should we choose the configurations for performance modeling without domain knowledge? Are there any characteristics that can be exploited? In the past, different sampling strategies have been proposed and tested to identify a suitable set of configurations for performance prediction [48, 95, 112, 133, 139]. Missing domain knowledge and a lack of sampling strategies that produce random samples with a low computational effort are among the core problems.

Software Evolution It is almost a law: Over time, software evolves [98]. Changes might arise throughout the evolution of a software system, for instance, from adding configuration options to serve more stakeholders, or from changing existing functionality. In particular, these changes might alter the behavior of one or multiple configuration options or even all of them with respect to non-functional properties such as performance [44]. Han and Yu [44] found that 59% of the performance bugs of configurable software are related to configuration issues; 78% to 92% of these configuration issues are due to wrong parametrization (i.e., wrong values for the configuration options).

Understanding how and when software changes affect the performance of configuration options plays an important role in configurable software systems. This knowledge provides a base for devising novel approaches to detect performance changes early on and, thus, improve the configurable software system with regards to performance. In the past, approaches have been proposed to identify evolutionary software performance changes related to configuration options by relying on heuristics [102] or by reusing existing performance data [54, 93]. However, these approaches typically rely on the performance data of a small number of case studies [54, 93] or randomly sampled configurations [102]. Thus, they are unable to provide a comprehensive overview on how evolutionary changes actually affect the performance of configuration options in the wild.

Workload Variability Apart from configuration options and the evolutionary changes, workloads (i.e., the benchmark or input of the software, such as music files or text files for a compression software) also affect the behavior of a configurable software system and its performance [151]. Identifying the right set of workloads that is scalable, maximizes code coverage, and represents the real world is a challenging endeavor [56, 71]. Despite several studies and approaches on workload variability [56, 71], the impact of workloads in the presence of configuration options remains unclear. Recently, studies conducted by Mühlbauer et al. [103] and Lesoil et al. [79] highlighted the impact of workloads on the performance of configurable systems. They found that workloads induce performance variation that even depends on the configuration options. In other words, different configuration options might exhibit a different performance impact in different workloads and might even invert the performance impact of configuration options (i.e., previously important configuration options become irrelevant and vice versa).

Problem Statement While ongoing research sheds light onto the performance impact of the workloads on a given configurable software system, there is a lack of empirical results on combining all three dimensions to pin down performance changes: configurability, evolution, and workloads. It is important to note that, typically, all three dimensions are present at the same time in current software systems (e.g., database systems, web servers, compression tools, video encoders, etc.), but—to the best of our knowledge—have not been considered together in research. Empirical research considering all three dimensions is fundamental for future research in this area. That is, including all three dimensions enables us to find performance regressions that might have been missed by previous studies since the studies only considered one or two of the three dimensions. It is important to note that considering these dimensions separately is certainly not enough due to interactions between these dimensions, as recent studies demonstrate on the dimensions configurability and workloads [79, 103]. For instance, measuring the performance of different database releases using only read queries cannot be used to detect performance changes that affect only write queries. An analysis of these performance data can be used to pinpoint performance changes by tracking down in which configuration, release, and workload the performance changes are detected.

1.2 Goals

Overall, this thesis pursues the following three goals:

1. We aim at analyzing the shortcomings of state-of-the-art sampling strategies and we devise a novel sampling strategy to overcome these shortcomings.
2. We aim at understanding how performance changes interact with configurability and whether performance changes are documented by software developers.
3. We aim at understanding how performance changes interact with configurability, evolution, and workload variability together and at assessing how performance modeling of configurable software systems can be used to detect performance changes.

Sampling Performance modeling typically consists of two major parts [139]: a sampling strategy to select software configurations to measure and a machine-learning approach that models the performance information from the data. In essence, the machine-learning approach builds and fits a *performance model* based on the measured configurations. As a consequence, the quality of the machine-learning approach highly depends on the sampling strategy [75], since the machine-learning approach is capable of extracting only information that is actually present in the provided data. In other words, if the measurement data do not contain any information on a configuration option, the machine-learning approach will not be able to derive any information about this configuration option. For instance, measuring only compression algorithms in TAR, such as XZ, does not allow to infer knowledge about the performance of other compression algorithms, such as LZMA.

In the past, different sampling strategies have been devised that exploit different characteristics in terms of performance and quality. Siegmund et al. [139] propose different sampling strategies to identify a suitable set of configurations and tested the sampling strategies for performance prediction. Medeiros et al. [95] compare different sampling strategies for fault prediction. Because of the high number of sampling strategies, further questions arise: Which sampling strategy is the most efficient for performance modeling based on machine learning? Which sampling strategy should be used?

One established sampling strategy is random (or uniform) sampling [48]. Random sampling performs generally good for software testing [120] and performance prediction [38]. However, Liebig [87] showed that sampling by randomly selecting a configuration at a time and checking its validity is intractable for real-world configuration spaces due to the high number of constraints between the configuration options[6]. This approach produces a high number of invalid configurations at the expense of execution time [29, 120]. Other approaches aim at addressing this issue by relying on additional data structures [112, 133]. The idea is to embed all constraints in a corresponding data structure (e.g., a binary decision diagram [112]). Avoiding invalid configurations while using these data structures to select random valid configurations. However, this advantage comes at the expense of scalability [48]. In general, random sampling performed good in past experiments on performance prediction, but is infeasible in practice [38, 87].

To shed light onto these questions, we devise a novel sampling strategy that comes close to random sampling and is feasible in practice. To assess the performance of our sampling strategy, we compare it with existing state-of-the-art sampling strategies on the basis of a number of real-world configurable software systems to derive which sampling strategies are efficient and in which cases.

Detecting Performance Changes of Configurable Software Systems Another goal of this thesis is to address one fundamental question of identifying performance changes in configurable software systems: Which technique is able to reliably identify performance changes? Performance modeling has shown promising results at describing and understanding the impact of individual configuration options on the performance of a system [37, 137]. However, performance modeling has been applied on configurable software systems by considering either only a single release or only a single workload. In a first step, we mitigate this shortcoming and propose an approach to identify performance changes in one or more configuration options across multiple releases of configurable software systems. For this purpose, we rely on comparing performance models and tackle issues that arise when using multiple performance models, such as multicollinearity [27]. To assess the practicability of this approach with regard to the capability of identifying performance changes, we apply it to a number of real-world configurable software systems and analyze the results. We contrast our findings with the history of the software system—captured in the version control system and the change log—to verify these results.

We expand this approach further to support workload variability. Ultimately, we devise an approach to identify performance changes across different configurations, releases, and workloads. The data produced by this approach gives us the opportunity to analyze when and how often performance changes occur: Exploring (1) configurability, (2) software evolution, and (3) workload variability provides us data on how these dimensions interact with each other. Applying our approach on these data provides us with the opportunity to assess the limitations of our approach in terms of precision and recall.

6 When randomly selecting options of the Linux kernel, there was not a single valid configuration even after one million trials [86, 123].

1.3 Contributions and Key Results

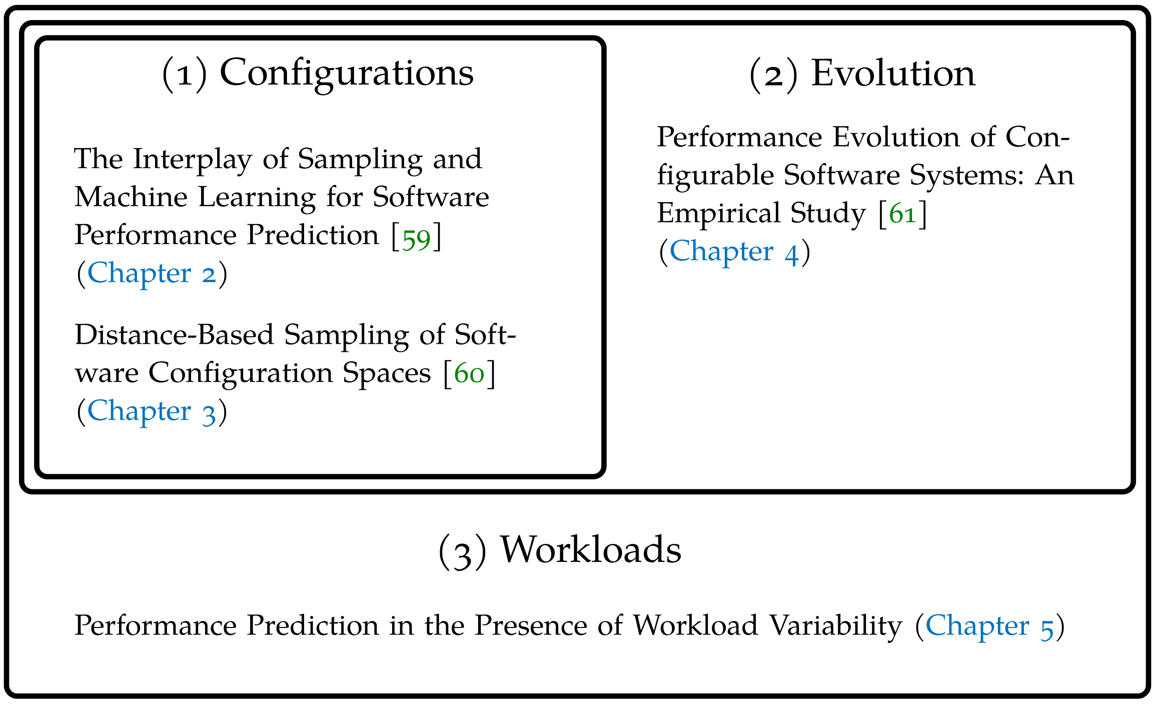

(1) Configurations
The Interplay of Sampling and Machine Learning for Software Performance Prediction [59] (Chapter 2)
Distance-Based Sampling of Software Configuration Spaces [60] (Chapter 3)
(2) Evolution
Performance Evolution of Configurable Software Systems: An Empirical Study [61] (Chapter 4)
(3) Workloads
Performance Prediction in the Presence of Workload Variability (Chapter 5)

Figure 1.1: Different dimensions of configurable software systems to which we contribute in this thesis. This includes contributions that consider (1) only configurations (the first one is included in the background), (2) a contribution focusing on evolution and configurations, and (3) a contribution considering workloads, evolution, and configurations.

Our contributions to performance modeling of configurable software systems are guided by the goals in Section 1.2. We provide an overview of the different dimensions mentioned before and the contributions to these dimensions in Figure 1.1. Note that some contributions are overarching multiple dimensions. For instance, „Performance Prediction in the Presence of Workload Variability"focuses on all three dimensions (i.e., configurability, evolution, and workloads). Overall, we make the following contributions:

1. We propose a novel sampling strategy—distance-based sampling—for performance modeling of configurable software systems that uses a distance metric to select configurations. To assess its practicability, we compare distance-based sampling with 4 different state-of-the-art sampling strategies on the basis of 10 widely used configurable software systems. We find that distance-based sampling outperforms state-of-the-art sampling strategies in terms of both accuracy and robustness, almost reaching the baseline of random sampling.

2. We propose a novel approach using performance modeling that enables us to identify performance changes of configuration options throughout history. For this purpose, we include additional metrics (such as the variance-inflation factor) to counter threats that arise while learning performance models and extract statistics on configurable

software systems. We apply this approach to 12 different popular configurable software systems covering up to 15 years of their evolution. Our results indicate that in 91% of the performance changes, multiple configuration options and interactions between them are involved. This emphasizes that focusing on the performance of single configuration options is certainly not sufficient. We traced the results of the performance changes to the change log and commit messages of the respective configurable software systems in a subsequent metadata analysis. We find that 68% of the releases mention performance changes in the change log or commit messages and 67% of them even mention the affected configuration option from our results.

3. We devise a novel approach to additionally support workload variability by refining our analysis from the previous contribution further. To assess the capabilities and limitations of our approach, we analyze the precision and recall. Thereby, we follow the *case-study research method* [127] (i.e., initial investigation on the considered phenomena) instead of a quantitative approach, by focusing on a single configurable software system (i.e., FastDownward). To keep our performance measurements feasible, we reduce the measurement effort by involving developers of FastDownward. We find that the precision of our approach is 88.2%, the recall is 59.4%, and identify limitations. In addition, we investigate the role of workload variability in the evolution of configurable software systems. We also find that 92.4% of the performance changes are detected only by a subset of the workloads.

1.4 Outline

This thesis is structured as follows. In Chapter 2, we lay the foundation that is needed to understand this thesis. In particular, we provide detail on configurable software systems, performance modeling, and sampling. Chapter 3 comprises the contributions related to configurability of software. That is, we present and evaluate a novel sampling strategy—distance-based sampling— for performance modeling and we compare it with other state-of-the-art approaches. In Chapter 4, we explain the adaption of performance modeling for detecting performance changes, apply this approach on 12 configurable software systems, and verify our results using a subsequent metadata analysis. We further expand performance modeling to support workloads in Chapter 5 and assess the accuracy in terms of precision and recall. Further, we investigate configurability, evolution, and workload variability at once. Finally, we summarize the thesis in Chapter 6 and discuss future work.

2 Background

This chapter (in particular Section 2.2) shares material with the following publication: Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, and Sven Apel. "The Interplay of Sampling and Machine Learning for Software Performance Prediction." In: *IEEE Software* 37.4 (2020), pp. 58–66 [59]

In this chapter, we lay the foundations of this thesis by introducing basic terminology and concepts. We begin with a description of configurable software systems, different types of features and their conversion, and a brief notion of non-functional properties in Section 2.1. Afterwards, in Section 2.2, we describe the process of performance modeling, which we use throughout this thesis.

2.1 Configurable Software Systems

A *configurable software system*—or simply *configurable system*—is a software system built with the possibility to tailor it to specific requirements of individual users [7]. These requirements can be functional (e.g., encrypted communication) or non-functional (e.g., performance, security requirements) and can arise from different environments (e.g., specific hardware or software environment) or user groups (e.g., administrators or users). In the following, we describe configurable software systems in more detail.

Configuration options—also called *features*—are knobs to adjust the behavior of a configurable system. These *configuration options* can be adjusted statically using, for instance, preprocessor statements (e.g., `ifdef`) or dynamically using configuration files or command-line parameters. In this work, we focus on dynamic configuration options, since the effect of static configuration options on performance have already been analyzed extensively in literature [11, 50, 63, 84, 85].

Each configurable system offers a multitude of different configuration options. Formally, we denote the set of all configuration options with regard to the configurable system S as $\mathcal{O}_S$. Further, we distinguish between the set of binary configuration options $\mathcal{O}_S^B$ and the set of numeric configuration options $\mathcal{O}_S^N$. These sets are disjoint (i.e., $\mathcal{O}_S^B \cap \mathcal{O}_S^N = \emptyset$) and together they represent all configuration options of the configurable system (i.e., $\mathcal{O}_S^B \cup \mathcal{O}_S^N = \mathcal{O}_S$). Each binary configuration option $o \in \mathcal{O}_S^B$ can be selected or deselected, whereas each numeric configuration option $o \in \mathcal{O}_S^N$ usually exposes multiple different numeric values that it can be assigned to (e.g., the compression level of a compression tool typically has a value from 0 to 9). We denote the value set of a numeric configuration option $o \in \mathcal{O}_S^N$ as r_o.

Moreover, the combination of one assignment for all configuration options (e.g., whether the configuration option is enabled) is called a *configuration*. A configuration is the end-product tailored upon the specific requirements of the end-user. We further denote $\mathcal{C}_S$ as the set of all configurations (i.e., configuration space) of the configurable system S. A configuration is an allocation for each configuration option $o \in \mathcal{O}_S$. Formally, a configuration $c \in \mathcal{C}_S$ maps each configuration option $o \in \mathcal{O}_S$ to its value $c : \mathcal{O}_S \to \mathbb{R}$. That is, for each binary configuration option $o \in \mathcal{O}_S^B$, $c(o) = 1$ if the configuration option o is selected in configuration c; otherwise $c(o) = 0$. For each numeric option $o \in \mathcal{O}_S^N$, c maps the configuration option to its specific value $v \in \mathbb{R}$: $c(o) = v$.

2.1.1 Configuration Issues

Although configuration options can be easily introduced into a system and provide a good opportunity to cover a broader range of users, they come with their own pitfalls. Previous work by Xu et al. [152] pointed out that the number of configuration options in configurable systems is typically increasing throughout time to provide even more flexibility to users. The addition of new configuration options increases complexity of configurable systems substantially although only a small portion of configuration options between 6.1% to 16.7% are used by the majority of users and up to 54.1% are rarely used. One immediate consequence of maintaining such a high number of configuration options are configuration issues [152]. Configuration issues arise when the user has specified a configuration $c \in \mathcal{C}_S$ (i.e., an allocation of configuration options) which triggers an unwanted behavior of the configurable system. To avoid such unwanted behavior, developers typically introduce constraints between configuration options (e.g., if encryption is selected, an encryption standard has to be chosen). Such constraints are often included only implicitly into the code of the configurable system. If even one of these constraints is violated, the configurable system might behave in an undesired way and could even produce wrong results. Formally, we denote $\mathbb{B}_S$ as the set of all constraints of configurable system S. Each constraint $\texttt{con} \in \mathbb{B}_S$ is a Boolean expression for one or more configuration options. We model a constraint $\texttt{con} : \mathcal{C}_S \to \{\texttt{false}, \texttt{true}\}$ as a function that takes a configuration and returns whether the constraint is satisfied (`true` if it is satisfied; `false` otherwise). A configuration $c \in \mathcal{C}_S$ is valid if and only if it satisfies all constraints:

$$\forall \texttt{con} \in \mathbb{B}_S : \texttt{con}(c) = \texttt{true}$$

Usually, constraint solvers such as CLASP[1], SAT4J[2], PICOSAT[3], or z3[4] are used to assess whether a configuration satisfies all constraints. In this thesis, we use z3 to search for valid configurations, as it supports constraints from feature modeling, but also specific constraints from sampling strategies, as we will explain in Section 2.2.2. We do not describe constraint solvers in more detail and recommend the Handbook of Satisfiability by Biere et al. [13] for further reading.

1 `https://potassco.org/clasp/`, last accessed on 07/07/2023.
2 `http://www.sat4j.org/`, last accessed on 07/07/2023.
3 `https://fmv.jku.at/picosat/`, last accessed on 07/07/2023.
4 `https://github.com/Z3Prover/z3`, last accessed on 07/07/2023.

For illustration purposes, we use an exemplary configurable system which we expand throughout this chapter. The exemplary configuration system is a compression program called Comp, which offers the binary configuration options $\mathcal{O}^{B}_{\text{Comp}}$: root for the base code that is always executed, Encryption (🔒) for encrypting a file, Compression (📦) for compressing a file, and two compression algorithms – A1 (⚙) and A2 (⛭). The binary configuration option root must always be selected. For brevity, we leave root in this example out. Another constraint is that exactly one of the compression algorithms has to be set if Compression is used. Furthermore, Comp offers a numeric configuration option CompressionLevel (☆☆). Formally, the following holds: CompressionLevel $\in \mathcal{O}^{N}_{\text{Comp}}$. CompressionLevel offers values from 0 to 2 where 0 (☆) indicates no compression, 1 (★) medium compression, and 2 (★★) maximum compression. The compression level can only be set to 1 or 2 if Compression is selected; otherwise it is 0.

Table 2.1: All valid configurations of the exemplary compression program Comp. Comp has the binary configuration options Encryption (🔒), Compression (📦), and two different compression algorithms: A1 (⚙) and A2 (⛭). Further, Comp provides CompressionLevel to adjust the compression: no compression (☆), medium compression (★), and maximum compression (★★). For brevity, we omitted root that denotes the execution of the base code.

	Configuration		Configuration
c_1	{☆}	c_6	{📦, ★★, ⛭}
c_2	{🔒, ☆}	c_7	{🔒, 📦, ★, ⚙}
c_3	{📦, ★, ⚙}	c_8	{🔒, 📦, ★★, ⚙}
c_4	{📦, ★★, ⚙}	c_9	{🔒, 📦, ★, ⛭}
c_5	{📦, ★, ⛭}	c_{10}	{🔒, 📦, ★★, ⛭}

Only configurations that fulfill all constraints specified before are *valid* configurations. In Table 2.1, we show the valid configurations of the compression program Comp. An invalid configuration would be {📦, ☆, ⚙, ⛭} since it violates two constraints. First, Compression (📦) is turned on with CompressionLevel 0. This is not possible since either Compression has to be turned on or CompressionLevel has to be 0, but not both. Second, both compression algorithms A1 (⚙) and A2 (⛭) are enabled, which contradicts the rule that exactly one of these algorithms can be turned on. In total, Comp has 5 configuration options and 10 valid configurations. To formally describe such subject systems with their configuration options and constraints, feature models are typically used. These feature models are described in the following section.

2.1.2 Feature Modeling

To model a configurable software system, there are many different concepts to formally describe the variability of a configurable software system as we show in Table 2.1. Some of these concepts originate from the area of feature-oriented software product-line engineering [7, 130]. In this work, we reuse the concepts of *feature models* and their visual representation in a tree-like structure, *feature diagrams*, which we describe in the following by creating a feature model for COMP.

A feature model consists of configuration options ($\mathcal{O}_S$) and the constraints ($\mathbb{B}_S$) among them. This information is sufficient to describe the configuration space ($\mathcal{C}_S$) of a configurable software system S. The configuration option that is present in every feature model is the binary option *ROOT*. It represents the root of the feature diagram, which stands for the base code (i.e., the code that is always executed and not related to any configuration option) and is also referred to as *BASE*. In Figure 2.1, we show the representation of the ROOT feature used throughout this paper. In our exemplary configurable software system COMP, this configuration option represents, among other functions, (1) the code that reads in the data to compress and (2) writes the compressed data into a file. Both of these operations have to be performed each time regardless of other configuration options. Formally, the configuration option ROOT is a binary configuration option: ROOT $\in \mathcal{O}^B_{\text{COMP}}$. Further, ROOT is always enabled, which we assure with the constraint `con` $\in \mathbb{B}_{\text{COMP}}$:

$$c(\text{ROOT})$$

Figure 2.1: Feature diagram representation of the feature ROOT in the software system COMP

A further constraint that describes the relationship between configuration options is the parent–child constraint. In a feature diagram, it is represented as a line between two configuration options, illustrated in Figure 2.2 with configuration options ROOT and COMPRESSION. In essence, the parent–child constraint assures that the parental configuration option is selected whenever the child is selected. Formally, the expression `con` $\in \mathbb{B}_{\text{COMP}}$ to express the parent–child relation between ROOT and ENCRYPTION in the configurable software system COMP is as follows:

$$c(\text{ENCRYPTION}) \Rightarrow c(\text{ROOT})$$

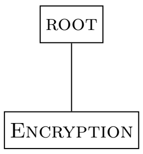


Figure 2.2: Parent–child relation between configuration options ROOT and ENCRYPTION.

In a feature diagram, we distinguish between OPTIONAL and MANDATORY configuration options and between OR and ALTERNATIVE groups [7]. An OPTIONAL configuration option is a configuration option that can be selected or deselected. A configuration option is marked as OPTIONAL in a feature diagram by a hollow circle symbol on top of the feature as shown in Figure 2.3 (a). The optional configuration option does not need any additional constraint. A MANDATORY configuration option is a configuration option that is selected in each configuration whenever the parent configuration option is selected. In a feature diagram, the symbol for a MANDATORY option is a filled circle on top of the configuration option as we illustrate in

Figure 2.3 (b). The corresponding constraint $\texttt{con} \in \mathbb{B}_{\text{Comp}}$ for the mandatory configuration option Algorithm is as follows: $c(\text{Compression}) \Rightarrow c(\text{Algorithm})$.

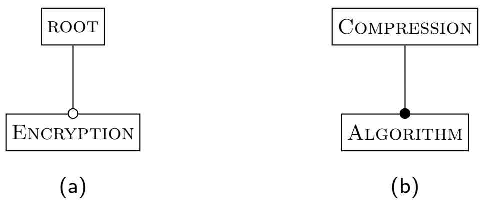


Figure 2.3: Optional option Encryption (a) and mandatory option Algorithm (b) of the compression tool Comp.

A special group of configuration options are alternative groups. In an alternative group, the parent node usually depicts functionality of the software that allows for different choices (i.e., exchanging one functionality with another). Among these choices, exactly one choice can be chosen. In a feature diagram such an alternative group is represented by a hollow arc in the parent configuration option.

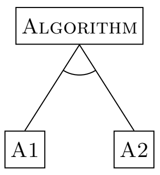


Figure 2.4: Alternative group offering different compression algorithms A1 and A2. Exactly one of both alternatives A1 and A2 have to be chosen.

In Figure 2.4, we depict such an alternative group in our exemplary configurable software system Comp. The parent node, Algorithm, has two child nodes, A1 and A2. This means that either A1 or A2 have to be chosen as an compression algorithm to compress the data. Multiple constraints $\texttt{con} \in \mathbb{B}_{\text{Comp}}$ are needed to express an alternative group:

$$c(\text{Algorithm}) \Rightarrow (c(\text{A1}) \vee c(\text{A2})) \tag{2.1}$$

$$c(\text{A1}) \Rightarrow \neg c(\text{A2}) \tag{2.2}$$

$$c(\text{A2}) \Rightarrow \neg c(\text{A1}) \tag{2.3}$$

The constraint in Equation 2.1 is needed to assure that at least one of the alternatives is selected whenever the parent node is selected. The constraints in Equation 2.2 and Equation 2.3 exclude the other configuration option to assure that only one of them can be selected at a time.

A similar group of configuration options is the or group as we show in Figure 2.5. In an or group, at least one configuration option has to be selected. The difference of an or group compared to multiple optional configuration options is that at least one option has to be selected. Both requirements of selecting at least one configuration option at a time is, however, very rare in feature modeling of configurable software systems. The only needed constraint is the following: $c(\text{OrGroup}) \Rightarrow (c(\text{A}) \vee c(\text{B}))$

Previously, we described binary configuration options ($\mathcal{O}_S^B$) and different characteristics of these. A common trait is that these options can only be selected or deselected. This, however, is

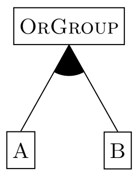


Figure 2.5: Representation of an or group in a feature diagram. When OrGroup is selected, at least A or B have to be selected.

often not sufficient for modeling a configurable system, since other options demand different numeric values instead of selecting or deselecting it. These options are called numeric options ($\mathcal{O}_S^N$). In our exemplary configurable system Comp, we have the possibility to adjust the compression level (i.e., the quality of the compression). In Figure 2.6, we represent this with the numeric configuration option CompressionLevel. The first line of the node contains the name of the configuration option; the second line contains the set of values; the third line contains the default value of the numeric configuration option (i.e., the value when it is not explicitly set). This type of configuration option adds one constraint `con` $\in \mathbb{B}$: $(c(\text{CompressionLevel}) = 0) \vee (c(\text{CompressionLevel}) = 1) \vee (c(\text{CompressionLevel}) = 2)$

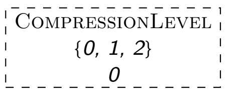
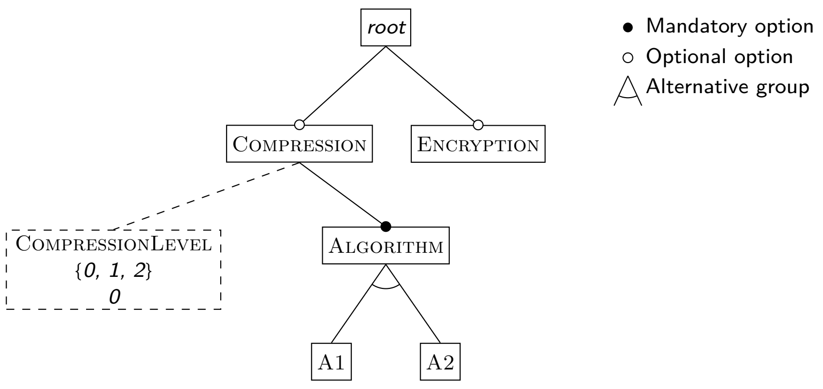


Figure 2.6: Numeric configuration option CompressionLevel in Comp. The first line represents the name of the configuration option. The possible values for CompressionLevel are shown in the second line. The last line defines the default value of the configuration option (i.e., the value this numeric option has when it is not used).

Other constraints, that are not implicitly encoded into the tree-like structure of a feature diagram, are called *cross-tree constraints*. The necessity of using additional constraints arises since the tree-like structure of feature diagrams and its implicit constraints do not suffice to express configuration spaces as proven by Knüppel et al. [65]. These cross-tree constraints are usually listed below the feature tree and are expressed by their formula as we show in the summarized feature diagram in Figure 2.7. In our case, we have two cross-tree constraints. The first cross-tree constraint, Compression ⇒ (CompressionLevel > 0), expresses that CompressionLevel can only have the values 1 or 2 if Compression is enabled. The second cross-tree constraint, CompressionLevel = 0 ⇒ ¬Compression, is additionally needed to assure that Compression is always turned off while CompressionLevel is 0.
The feature diagram in Figure 2.7 contains the feature diagram of our configurable software system Comp. In summary, the configuration options have the following meaning: root represents the base code of the configurable software system Comp and is always selected. Encryption represents the encryption functionality (i.e., encrypting the compressed data before writing the data to disk) and can be selected or deselected. Compression represents using the compression algorithm, which can be selected or deselected. The CompressionLevel has to be 1 or 2 if the compression is selected and 0 otherwise. Either A1 or A2 has to be chosen as an compression algorithm if Compression is selected.



$$\textsc{Compression} \Rightarrow (\textsc{CompressionLevel} > 0)$$
$$\textsc{CompressionLevel} = 0 \Rightarrow \neg\textsc{Compression}$$

Figure 2.7: Complete feature model of Comp. For the sake of simplicity, we exclude mentioning configuration c in the cross-tree constraints (e.g., we write Compression instead of c(Compression)).

In summary, the feature diagram in Figure 2.7 encodes the following constraints:

$$c(\mathit{root})$$
$$c(\textsc{Encryption}) \Rightarrow c(\mathit{root})$$
$$c(\textsc{Compression}) \Rightarrow c(\mathit{root})$$
$$c(\textsc{Algorithm}) \Rightarrow c(\textsc{Compression})$$
$$c(\text{A1}) \Rightarrow c(\textsc{Algorithm})$$
$$c(\text{A2}) \Rightarrow c(\textsc{Algorithm})$$
$$c(\textsc{Compression}) \Rightarrow c(\textsc{Algorithm})$$
$$c(\textsc{Algorithm}) \Rightarrow (c(\text{A1}) \vee c(\text{A2}))$$
$$c(\text{A1}) \Rightarrow \neg c(\text{A2})$$
$$c(\text{A2}) \Rightarrow \neg c(\text{A1})$$
$$(c(\textsc{CompressionLevel}) = 0) \vee (c(\textsc{CompressionLevel}) = 1) \vee (c(\textsc{CompressionLevel}) = 2)$$
$$c(\textsc{Compression}) \Rightarrow (c(\textsc{CompressionLevel}) > 0)$$
$$(c(\textsc{CompressionLevel}) = 0) \Rightarrow \neg c(\textsc{Compression})$$

With all these constraints, the configurations from Table 2.1 are the only configurations that satisfy all constraints. Interestingly, although our exemplary configurable software system has 7 configuration options and a configuration space of 192 possible configurations, only 10 ($\sim 5\%$) of them satisfy all constraints. In another configurable software system, the video encoder VP9, we focused on only 42 binary configuration options with a configuration space of $2^{42} \sim 10^{12}$ configurations. In this example, only 216 000 configurations were valid (i.e., 0.000021% of the configurations are valid). This high number of constrains illustrates one of the main challenges when handling configurable software systems: it is usually difficult to identify valid configurations [87]. We address this issue later in Section 2.2.2. For the remainder of this work, we denote the set of all valid configurations of a configurable system S as C_S and refer to it as *whole population*.

2.1.3 Non-Functional Properties

Each configuration of a configurable software system has measurable properties which can be acquired by building or executing the configurable software system with the particular configuration and a workload. In this section, we describe the so called *non-functional properties*. In literature, non-functional properties have received much attention in the last decades as Siegmund [138] points out since there are over 25 definitions of these terms. Additionally, there are multiple surveys on this topic due to the ambiguous definitions [138]. Similarly to Siegmund, we do not give an additional definition of non-functional properties and state only that non-functional properties are related to a quality or attribute. Exemplary non-functional properties of configurable software systems are *footprint* (i.e., the size of the binary after compiling it), *latency* (i.e., the time to send or process some information), *throughput* (i.e., the amount of information to send or process during a given time frame), *energy consumption* (i.e., the energy needed while performing a certain task), or *execution time* (i.e., the time needed to perform a certain task).

Execution time, in particular, has gained momentum in software engineering in the last years. Whole domains, such as the high performance computing (HPC) domain, are dedicated on performance optimization of hardware and software alike. Recently, Han and Yu conducted an empirical study on performance bugs by using bug repositories and change logs [44]; Leitner and Bezemer conducted an exploratory study on performance tests of configurable systems [78]. Siegmund et al. propose a machine-learning approach to measure only a subset of the configuration space and predict the rest [139] and later, Grebhahn et al. use this approach to explain and verify the performance behavior of configuration options [37]. Our work builds on the later two publications. In the remainder of this work, we focus mainly on the *execution time* as a non-functional property of configurable software systems and use *performance* as a synonym.

Formally, the performance measurements can be represented as a function $\mathbb{P}_S : \mathcal{C}_S \to \mathbb{R}$ that maps a given configuration of the configurable software system S to the measured performance value.

In Table 2.2, we show exemplary performance values of the configurable software system Comp. Taking a closer look on the table reveals the following: Running the base code without any configuration option takes about 5 seconds. Using only Encryption (🔒) in c_2 needs about 3 seconds more than c_1 (8 seconds in total). In c_3, using Compression (📦) with algorithm A1 (⚙) and CompressionLevel 1 (⭐) needs about 2 seconds more than without compression (7 seconds in total). One could assume that these performance offsets in c_2 and c_3 are additive, which would result in a performance of $5 + 3 + 2 = 10$ seconds. Interestingly, combining encryption with compression in c_7 does not yield the assumed result of 10 but only 9. The reason for this behavior are interactions between configuration options, which we describe in the following section.

2.1.4 Interactions

In configurable software systems, multiple configuration options might *interact* with each other. That is, two or more configuration options interact when they influence each other

Table 2.2: Measured performance values $\mathbb{P}$ in seconds for each configuration of the exemplary configurable software system Comp.

	Configuration	$\mathbb{P}$		Configuration	$\mathbb{P}$
c_1	{☆}	5	c_6	{📦, ★★, ☸}	7
c_2	{🔒, ☆}	8	c_7	{🔒, 📦, ★, ⚙}	9
c_3	{📦, ★, ⚙}	7	c_8	{🔒, 📦, ★★, ⚙}	10
c_4	{📦, ★★, ⚙}	9	c_9	{🔒, 📦, ★, ☸}	8.5
c_5	{📦, ★, ☸}	6	c_{10}	{🔒, 📦, ★★, ☸}	9

while executed together. The reasons for such interactions can be, among others, dependencies in the configuration option's implementation, such as control-flow or data-flow dependencies. In some cases, interactions even lead to program errors [42]. Apel et al. [7] describe the problem of detecting, managing, and resolving these interactions as the *feature-interaction problem*.

In Section 2.1.3, we pointed out that the measurements in Table 2.2 are not consistent when focusing on Encryption (🔒), A1 (⚙) and CompressionLevel 1 (★). We observe that the execution time was 1 second lower than expected. In fact, it turns out that there exist a data-flow dependency between encryption and compression in this case since Comp first compresses the data and afterwards encrypts the compressed data. This way, the encryption has less data to encrypt and obtains a speed up of 1 second. While this example is easy to explain and involves only two configuration options, in reality we encounter more complex interactions of up to 6 configuration options, as we describe in Chapter 4. Kolesnikov et al. [68] observed in an empirical study that considering interactions of up to 3 configuration options are sufficient to cover most of the influences; interactions with more than 3 configuration options are only marginally relevant. We describe the results of this study in more detail in Section 2.2.

In general, interactions are a cross-cutting problem throughout this thesis (i.e., they are present in each chapter). Interactions have implications on sampling as discussed in Section 2.2.2 and Chapter 3. Interactions also appear when it comes to explaining performance behavior on different releases and workloads as we do in Chapter 4 and Chapter 5.

2.1.5 One-Hot Encoding of Numeric Configuration Options

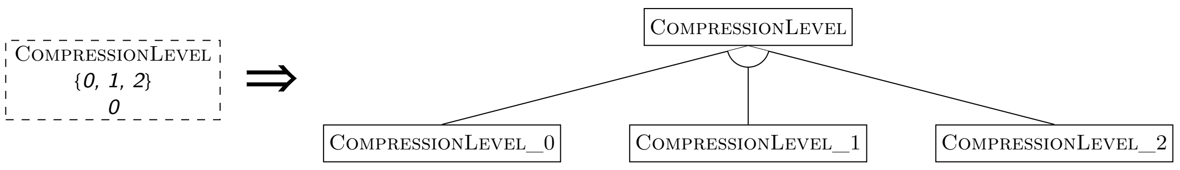


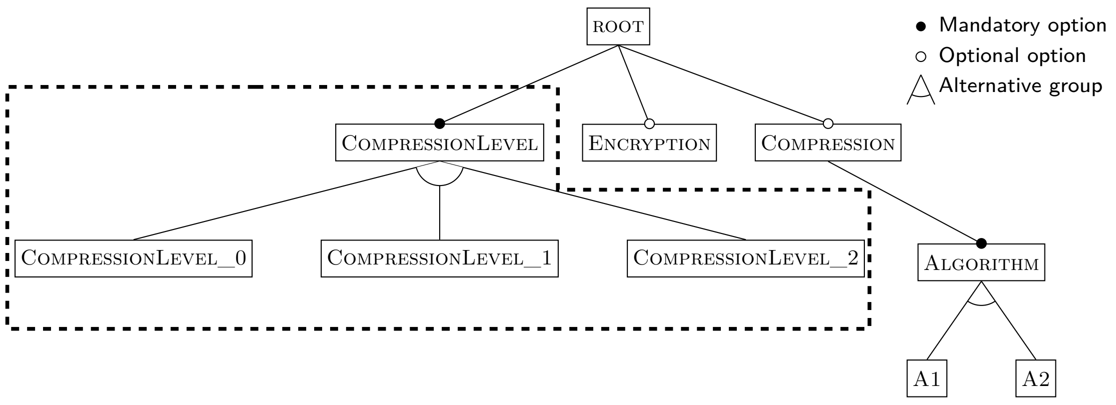
Figure 2.8: One-hot encoding of the configuration option CompressionLevel.

Typically, numeric configuration options are configuration options with a multitude of different numeric choices. However, numeric configuration options are often converted into binary configuration options since feature model formats used in practice (such as SXFM in S.P.L.O.T. [97] or UVL [145]) do not support numeric configuration options. But, to cover numeric configuration options nevertheless, numeric configuration options are often converted into binary configuration options [60, 139]. In literature, this conversion process is also referred to as *discretization* [60, 139]. However, the more precise term for this conversion in the machine-learning area is *one-hot encoding* [125]. In our configurable software system Comp, we use a numeric configuration option for the quality of our compression algorithm (i.e., the higher the compression level, the better the compression). Other numeric configuration options could adjust the hardware usage of the program (i.e., maximum memory consumption, number of threads) or the control flow of the program (i.e., number of iterations or the threshold of a certain algorithm). In the past, literature focused mainly on binary configuration options, since numeric configuration options can be converted into multiple binary configuration options. To use this conversion process, the values of a numeric options have to be a fixed number of possible values. Theoretically, numeric configuration options could be continuous and infinite, but in practice the value range of numeric configuration options are bound to data types such as `int` or `double` and, thus, more limited. In this work, we further limit the value range of numeric configuration options to keep the measurement effort of a configurable system feasible. Consequently, the prerequisite of categorical data (i.e., data with a limited and fixed number of possible values) is given for each numerical configuration option.

The general idea of one-hot encoding of numeric configuration options is as follows: Replace each numeric configuration option by a binary configuration option with the same name and create an alternative group, where each child represents one single value of the numeric configuration option. In Figure 2.8, we show the conversion of the numeric configuration option CompressionLevel. To obtain a valid feature diagram again, two additional steps have to be performed. First, the alternative group has to be a child of root to be able to model that CompressionLevel_0 is the default value and selected whenever Compression is not selected. Second, the cross-tree constraints referring to the choices of numeric configuration options have to be changed too. Therefore, each equation or inequality constraint is reformed by replacing them by the corresponding alternative configuration options as we show in Figure 2.9.

2.2 Performance Modeling

Performance plays a crucial role when it comes to configurable systems. From a developer's point of view, understanding and verifying their knowledge of configurable systems [37] might help, for instance, to detect performance bugs [44]. From a user's point of view, identifying the performance-optimal configuration to a certain preset (i.e., the user specifies that certain configuration options have to be selected or deselected) helps improving performance [139]. For instance, a user of Comp might be interested in identifying the performance-optimal configuration when A1 (⚙) is selected. A look on Table 2.2 reveals that c_3 is the best configuration for his needs (i.e., using only medium compression (★) and deselecting



Compression ⇒ (CompressionLevel_1 ∨ CompressionLevel_2)

CompressionLevel_0 ⇒ *not* Compression

Figure 2.9: Complete feature model of Comp after one-hot encoding. The change of the feature model is highlighted by a dashed line.

encryption (🔒)). However, in most cases it is not as trivial as in Comp because on the one hand, the feature interaction problem should be addressed by measuring and investigating all configurations [42] as described in Section 2.1.4. On the other hand, it is not feasible to measure all configurations due to the vast size of the configuration space. For illustration, a configurable system offering 30 binary configuration options without any constraints results in a configuration space of $|\mathcal{C}| = 2^{30} \approx 10^9$. However, real-world configurable systems have usually a lot more configuration options. For instance, SQLite, a small SQL database engine consists of more than 80 configuration options [138]. Larger configurable systems, such as the Linux kernel had more than 5 000 configuration options in the year 2010 [135] and recently more than 9 000 configuration options [1] with an estimated number of 10^{6000} configurations [93]. These factors make it difficult to either measure all configurations or to alternatively identify the performance-optimal configuration by measuring only a few configurations. To tackle this problem, an approach is needed that aims at identifying the performance-optimal configuration by only measuring a few configurations. To tackle the problem which we address as *performance modeling*, a joint approach of (1) sampling strategies to limit the amount of configurations to measure and (2) machine-learning technique to extract performance information is used [139]. Next, we provide an overview of how sampling and machine learning are combined to use it not only for predicting the configuration space, but also to convey the influence of different configuration options and interactions on the performance. After this overview, we provide more detailed information on the components of performance modeling.

2.2.1 Overview

In Figure 2.10, we illustrate the overall process of learning performance-influence models for configurable software systems. A sampling strategy is used to select (Step I) and measure (Step II) a tractable subset of configurations, which is then used to learn an influence model (Step III). The influence model can then be used to predict the performance of any configura-

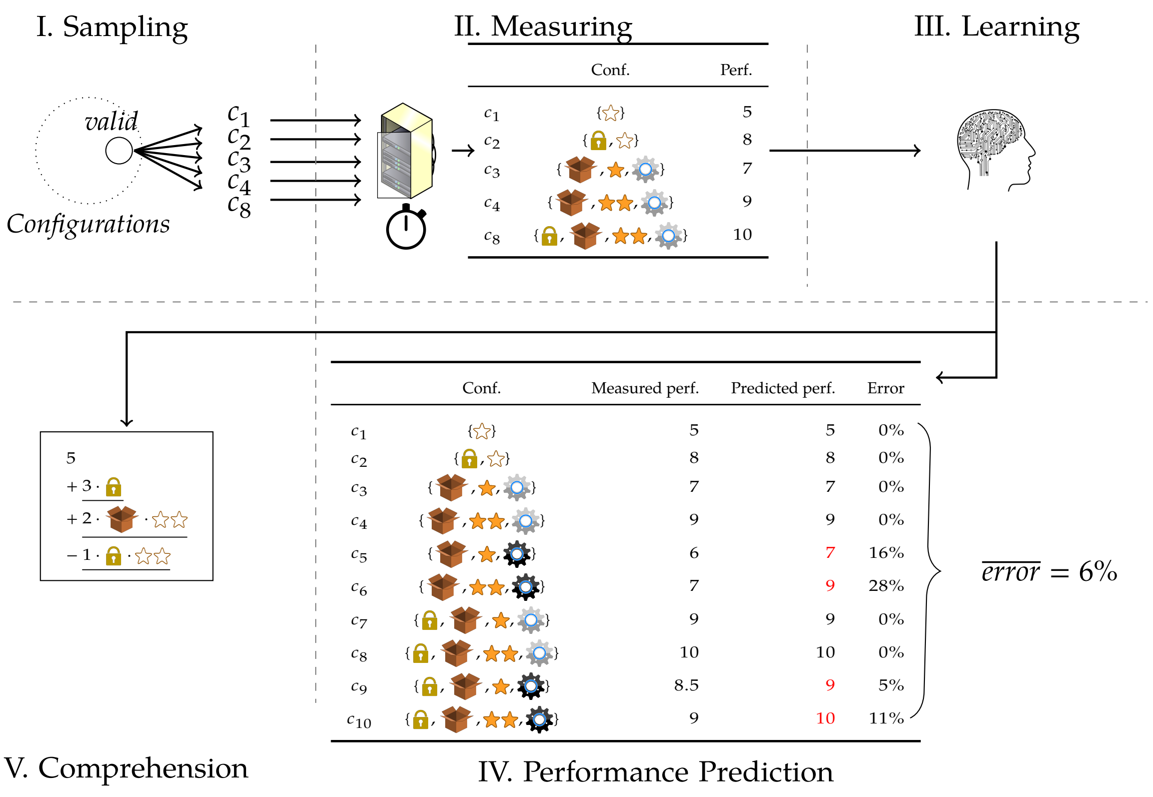


Figure 2.10: Predicting and comprehending the performance of configurations: The subject is the configurable software system Comp. The steps for performance prediction of configurations are *sampling* (I), *measuring* (II), *learning* (III), *performance prediction* (IV), and *comprehension* (V). In Step I, a sampling strategy selects a tractable number of configurations. These configurations are measured in Step II. In Step III, a machine-learning algorithm is used to learn an influence model. The learned influence model can be used in Step IV to predict the performance of any configuration. This way, one can compare the predicted performance to the measured performance and compute the prediction or model error. Alternatively, the influence model can be used in Step V to comprehend the performance behavior of configuration options; terms representing influences of options and interactions among options are underlined.

tion (Step IV). This can be specifically used to identify the performance-optimal configuration for a certain preset. Additionally, the influence model can also be used to comprehend the influence of different configuration options on the performance (Step V). In our example, the influence model captures the *relevant* influences of configuration options and interactions on performance as a linear combination of the respective terms [139]. An influence is *relevant* if it increases the predictive power of the model on configurations of the sample set [139]. The influence model from Figure 2.10 has four components:

- 5 represents the base performance (the *intercept*),
- 3 · 🔒 represents the influence of option ENCRYPTION,
- 2 · 📦 · ☆☆ represents the influence of the numeric option CompressionLevel, and
- −1 · 🔒 · ☆☆ represents the influence that arises from the interaction of ENCRYPTION and CompressionLevel

The accuracy of the model depends on two factors: (1) the sample set (Step I) and (2) the machine-learning technique (Step III). Note that other representations for the influence model (e.g., regression trees) are possible and may even result in improved accuracy (i.e., lower error) [129]. However, an important aspect for software engineers is *interpretability* of the model as we use it in Step V, which is why we resort to the simple, but yet reasonably expressive form of a linear, multi-variable regression model [37].

Let us start with a closer look at the sampling step. The goal of the sampling step is to select a sample set such that all *relevant* influences are covered. If an important option or interaction among multiple options is not present in the sample set (for example, the interaction among ENCRYPTION and COMPRESSION in our example), the learning step cannot possibly uncover it and include it into the influence model.

2.2.2 Sampling

Since measuring all configurations of a configurable software system is typically infeasible due to the sheer size of the configuration space, researchers and practitioners have developed a number of strategies to sample the configuration space of given configurable systems. Although incomplete (i.e., it selects only a subset $\mathbb{S} \subseteq \mathcal{C}$ of all configurations), sampling is the state of the art in practice [139]. For example, in Linux, code coverage sampling is used to create a sample set that covers all lines of source code [147]. Achieving completeness by exploiting parallelism to measure all configurations is infeasible, since the number of configurations grows exponentially with the number of options.

A prominent sampling strategy among researchers and practitioners is *random sampling*. The idea is to select configurations randomly from the configuration space in an unbiased way. More precisely, the probability of selecting a configuration c from configuration space $\mathcal{C}_S$ should be $\frac{1}{|\mathcal{C}_S|}$. Without any knowledge on relevant influences and interaction effects, random sampling is a reasonable choice. Nevertheless, random sampling may miss important information—there is no guarantee that a certain configuration option or interaction is covered in the sample set, which becomes more prevalent when the number of samples is very low compared to the size of the configuration space (i.e., the probability of covering a configuration option is lower). Furthermore, obtaining an unbiased random sample is computationally hard. This is due to possible constraints among configuration options. While there has been progress in this direction using binary decision diagrams or satisfiability solvers [19, 112], existing solutions are not ready for industrial adoption yet.

A more systematic sampling strategy is *t-wise sampling*. The idea is to select the sample set such that it covers *all* interactions among *all* combinations of t options. *Pair-wise sampling* ($t = 2$) is the most popular. It ensures that all *pairs* of configuration options are present (i.e., enabled), at least, once in the sample set. *Option-wise sampling* ($t = 1$) ensures that each individual option is selected, at least, once. Clearly, the larger t is, the more possible interactions we can catch, but the larger the sample set grows. Let us illustrate this fundamental tradeoff with an example—a function that contains preprocessor directives to realize configurability:

```
void encrypt(data) {
    #ifdef Compression
    data = compressData(data);        // Block 1: Compression
```

```
        #endif
        #ifdef Encryption
        data = encryptData(data);         // Block 2: Encryption
        #endif
    }
```

In the above code snippet, we illustrate the interplay of the options Encryption and Compression in the configurable system Comp at code level. Function `encrypt` uses two macros, Encryption (🔒) and Compression (📦), controlling the inclusion of configuration-dependent code. Block 1 is included only if Compression is selected. Block 2 is included only in configurations that have Encryption selected. It can be seen in the above code snippet that, when data are compressed, less data have to be encrypted, which speeds up the encryption process. Option-wise sampling would select only Block 1 or 2, not both blocks 1 and 2, missing the speedup due to the interaction of the two options. Pair-wise sampling would select an additional configuration that includes blocks 1 and 2, at the cost of a larger sample set, though.

While t-wise sampling is systematic, it comes with its own challenges. First, computing a sample set even for $t \geq 2$ is computationally expensive and even infeasible for configuration spaces of the size of the Linux kernel [123]. Second, stoically including all pairs (or triples, quadruples, etc.) may be unnecessarily expensive since practice has shown that not all interactions among options are relevant—actually only few are [68]. To address these challenges, hybrid sampling strategies have been proposed, which combine an element of randomness with coverage criteria such that certain combinations of interactions or certain parts of the code are selected [60, 118, 147]. Sampling strategies that are dedicated to numeric configuration options have been proposed as well [139]. In this work, however, we focus on binary configuration options. After this brief overview, we present the binary sampling strategies that are relevant in this thesis in more detail.

Binary Sampling Strategies

In literature, many publications focus on using binary sampling strategies [19, 95, 112, 118]. Binary sampling strategies are designed especially for binary configuration options. For covering also numeric configuration options, the numeric configuration options could be converted into multiple binary configuration options using one-hot encoding (see Section 2.1.5). These sampling strategies typically collect samples until they cover a certain goal (coverage based), use an off-the-shelf satisfiability solver to identify suitable configurations (solver based), or adopt a certain strategy to randomly select configurations. Next, we explain the binary sampling strategies in more detail and extend the feature model of Comp by adding the binary configuration option CheckIntegrity (📋) for checking the file integrity of the compressed file. CheckIntegrity is an optional configuration option and, thus, can be selected and deselected independently to the other configuration options. We refer to this extended feature model as CompExt and utilize it for a better demonstration of the binary sampling strategies. In total, CompExt has 20 configurations.

Coverage-Based Sampling *Coverage-based sampling strategies* focus on specific areas or properties of the configuration space, such as specific kinds of interactions as in t-wise

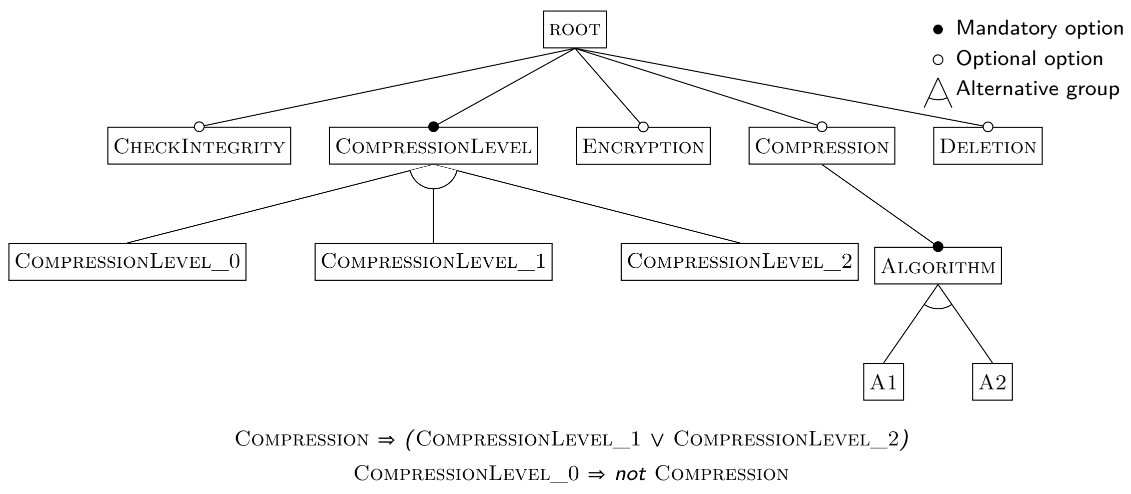


Figure 2.11: Feature model of CompExt, an extension of the feature model of Comp. In essence, CompExt has one more configuration option: CheckIntegrity (✓) for checking the file integrity of the compressed file.

sampling [140]. This might be the optimal way to sample if we know in advance where to sample, but that is usually not the case for large software systems.

In t-wise sampling, the idea is to select configurations such that all combinations of t configuration options appear at least once in the sample set. In many cases, additional constraints force the samples to include configurations with even more than t configuration options enabled. A valid sample that fulfills this goal might be the configuration where all configuration options are enabled. This, however, is less practical since a single configuration does not provide enough information to infer any knowledge about configuration options. As a countermeasure, Siegmund et al. [140] use t-wise sampling such that it minimizes the number of selected configuration options. In this thesis, we reuse the later sampling strategy. For illustration, we show the sample set for $t = 1$ in Table 2.4. In total, $t = 1$ needs 6 out of 20 configurations and thereby covers each configuration option of CompExt, at least, once.

Another strategy aims at a balanced selection and deselection of all configuration options in the sample set. Sarkar et al. showed that such a frequency-based sampling further improves the accuracy of performance models learned based on the sample set [129]. Other coverage-oriented sampling strategies are, for example, statement-coverage sampling [147] or most-enabled-disabled sampling [95]. Statement-coverage sampling is a white-box strategy, in which the configurations are selected such that every block of optional code from the software system is selected, at least, once; whereas most-enabled-disabled sampling selects just one configuration where all configuration options are selected and one where all configuration options are deselected. The main problem of these strategies is that they require prior knowledge to select a proper coverage criterion, which typically requires a domain expert (i.e., an experienced user or a developer) and, thus, this option is often not available.

One general shortcoming of coverage-based sampling strategies is that one cannot set an arbitrary number of configurations to sample. This is especially crucial when the measurement budget only allows for a very limited number of configurations. In t-wise sampling, adjusting t is not practicable in reality. For instance, for a configurable system with 300 op-

Table 2.3: All 20 configurations of CompExt.

	Configuration		Configuration
c_1	{☆}	c_{11}	{☆, 🗹}
c_2	{🔒, ☆}	c_{12}	{🔒, ☆, 🗹}
c_3	{📦, ★, ⚙}	c_{13}	{📦, ★, ⚙, 🗹}
c_4	{📦, ★★, ⚙}	c_{14}	{📦, ★★, ⚙, 🗹}
c_5	{📦, ★, ⛭}	c_{15}	{📦, ★, ⛭, 🗹}
c_6	{📦, ★★, ⛭}	c_{16}	{📦, ★★, ⛭, 🗹}
c_7	{🔒, 📦, ★, ⚙}	c_{17}	{🔒, 📦, ★, ⚙, 🗹}
c_8	{🔒, 📦, ★★, ⚙}	c_{18}	{🔒, 📦, ★★, ⚙, 🗹}
c_9	{🔒, 📦, ★, ⛭}	c_{19}	{🔒, 📦, ★, ⛭, 🗹}
c_{10}	{🔒, 📦, ★★, ⛭}	c_{20}	{🔒, 📦, ★★, ⛭, 🗹}

Table 2.4: Example for using t-wise sampling with $t = 1$.

	Conf.
c_1	{☆}
c_2	{🔒, ☆}
c_3	{📦, ★, ⚙}
c_4	{📦, ★★, ⚙}
c_5	{📦, ★, ⛭}
c_{11}	{☆, 🗹}

tional configuration options such as SQLite, option-wise sampling would yield about 300 configurations, whereas pair-wise would result in about $300^2 = 90\,000$ configurations.

Solver-Based Sampling *Solver-based sampling* strategies use an off-the-shelf constraint solver, such as SAT4J or z3, for sampling an arbitrary number of configurations. Naturally, these strategies do not guarantee true randomness [47] as in random sampling. Often the sample set consists only of the first k solutions provided by the constraint solver [18], and the internal solver strategy is typically to search in the "neighborhood" of an already found solution. Hence, the result is a locally clustered set of configurations. For illustration, we show this in an example in Table 2.5. Although the solver-based approach identifies 6 configurations, the configurations are similar to each other and do not contain the configuration options A2 (⛭) and CheckIntegrity (🗹). To weaken the locality drawback of solver-based sampling, Henard et al. [47] change the order of configuration options, constraints, and values in each solver run. This strategy, which we call henceforth *randomized* solver-based sampling, increases diversity of configurations, but it cannot give any guarantees about randomness or coverage. As we show in our evaluation in Chapter 3, this strategy requires to rebuild the entire solver model from scratch at each solver call (i.e. selection of one configuration), which is computationally expensive.

Table 2.5: Example for solver-based sampling to sample 6 configurations (i.e., the same number as $t = 1$).

	Conf.
c_1	{☆}
c_2	{🔒, ☆}
c_3	{📦, ★, ⚙}
c_4	{📦, ★★, ⚙}
c_7	{🔒, 📦, ★, ⚙}
c_8	{🔒, 📦, ★★, ⚙}

Random Sampling One way to perform *random sampling* is by randomly assigning either 0 or 1 to each configuration option for each configuration [39]. However, it is very likely that many invalid configurations are selected this way due to unsatisfied constraints, which makes this strategy inefficient. For instance, Liebig [87] sampled 1 000 000 configurations randomly using this approach for the configurable software system Linux, but none of them satisfied all constraints. Chakraborty et al. [18] use hash functions to split the configuration space recursively in multiple regions, and they select configurations from each of the regions. Still, this strategy produces many invalid configurations. Chen et al. [22] use a distance metric to find different test inputs for methods to uniformly cover the configuration space. However, they do not consider constraints among the input variables and, thus, produce many invalid configurations. Oh et al. [112] encode a system's configuration space using a binary decision diagram. This way, they can represent and enumerate all configurations in a compact way, such that they can randomly draw configurations. However, construction time and memory consumption of binary decision diagrams are high, and they do not scale to the largest configurable software systems [134]. Gogate and Dechter [34] propose a random sampling strategy that uniformly selects configurations without enumerating all configurations using the Monte-Carlo method [99]. This strategy also selects invalid configurations, though. Although difficult in practice, we resort in this thesis to enumerating all configurations of the configurable software systems using an off-the-shelf solver and randomly select configurations afterwards. For illustration, we show in Table 2.6 a random sample consisting of 6 configurations. Interestingly, every configuration option is covered, at least, once. Some interactions between configuration options are also covered.

2.2.3 Performance Measurements

After using sampling strategies in Step I to select a suitable amount of configurations, these configurations have to be measured in Step II. Performance measurements are exposed to different confounding factors that may lead to wrong results [33, 105] and inherently to

Table 2.6: Example for random sampling to sample 6 configurations (i.e., the same amount as $t = 1$).

	Conf.
c_1	{☆}
c_4	{📦, ★★, 🔍}
c_6	{📦, ★★, ⚙}
c_{16}	{📦, ★★, ⚙, 📋}
c_{18}	{🔒, 📦, ★★, 🔍, 📋}
c_{19}	{🔒, 📦, ★, ⚙, 📋}

drawing wrong conclusions. The main issue herein lies in the fact that measurement bias is unpredictable [105] and, thus, cannot be controlled completely. Measurement bias, however, could deflect the results of Step III–V and provide a misleading and wrong view on the configurable software system. This way, we cannot make any trustworthy observations to a system. To reduce the measurement bias, we deploy multiple steps as precautions to reduce measurement noise.

First, the hardware could produce a measurement bias which could blur our results. A way to avoid variation in hardware is to use only a single compute node for performance measurements. This, however, is unpractical when measuring thousands of configurations with multiple repetitions. For this reason, we use multiple compute nodes. However, different compute nodes can also have an effect on measurements. Using different hardware for our performance measurements (e.g., a different CPU with different clock speeds) affects the performance measurements (e.g., one hardware executes the measurements faster). Thus, we are using the same hardware for our performance measurements. That is, we are using multiple computing nodes with the same hardware to distribute the measurements on multiple machines. Although hardware may vary slightly [105] due to manufacturing issues, we ran a performance benchmark (i.e., SYSBENCH[5]) and compared the performance of all nodes to assure that the performance is similar on each computing nodes. Further measurement bias can also be induced by software [105]. For instance, using different versions of software (e.g., different LINUX kernel version by using different LINUX operating systems) can also affect performance due to, for instance, performance updates. Moreover, software running in the background but consuming the CPU also affects performance negatively. To counteract, we are running a minimum operating system image (UBUNTU or DEBIAN – depending on the case study) with only necessary packages (typically libraries) installed on the nodes. To further assure isolation from the execution of other tasks, we internally use a scheduling system called SLURM [153], which is often used in high performance computing environments. However, we cannot control all confounding factors completely (e.g., the environmental temperature), which gives rise to further measurement noise. To control the remaining noise, we measure each configuration 3 to 5 times (depending on the case study) and repeat the measurement of the configuration if the standard deviation exceeds 10%.

5 https://github.com/akopytov/sysbench, last accessed on 03/24/2023.

2.2.4 Performance-Influence Models

Performance-influence models allow us to model and predict the performance of all individual configurations of a configurable software system [139], as we show in Step III–V in Figure 2.10. We denote a performance-influence model as a function $\Pi_S : \mathcal{C}_S \rightarrow \mathbb{R}$, which takes a configuration $c \in \mathcal{C}_S$ of configurable system S and returns its predicted performance value.

However, performance-influence models can become quite complex (i.e., the number of terms of the performance-influence models is high), especially if the performance-influence model contains even very small and neglectable influences. Therefore, it is desirable in many cases that a performance-influence model contains only the most relevant influences, which can be achieved by adjusting the learning procedure at the cost of predictive power [68]. In any case, predictions of performance-influence models are rarely totally accurate, even if we included all possible configurations for learning the performance-influence models. As pointed out in Section 2.2.3, the measurement setup introduces systematic error, resulting in noisy data. Because of the random nature of measurement noise, this noise cannot be learned by machine-learning approaches. Other reasons are that the machine-learning approach can often not learn all types of influences, as we will point out later in this section. Performance-influence models are not specific to performance. They can be used to model any non-functional property that can be quantified on an interval scale. Performance-influence models have been applied to accurately predict execution time, throughput, memory consumption, binary footprint (i.e., the size of the binary), energy consumption, verification effort, and more [37, 66, 141].

In the past, many different approaches have arisen for performance modeling of configurable software systems. Different approaches, such as classification and regression trees, multiple linear regression, and deep learning have been shown to work for performance modeling [41, 129]. We provide more detail on different approaches in Section 2.2.4.

As Shmueli [137] points out, performance-influence models can be used for different purposes, such as for describing a certain behavior or to predict a certain behavior. Predictive modeling is used to predict the performance behavior of (unseen) configurations. This can be used as a heuristic to search for a performance-optimal configuration. However, to get reliable results, the prediction error has to be minimized. We deliver more details on predictive modeling in Section 2.2.4. In contrast, we use the descriptive modeling aspect of performance-influence models to derive the influence on the performance of different configuration options and interactions thereof. But to avoid drawing wrong conclusions, we need to assure that the models are reliable. Therefore, we have taken multiple precautions, which we describe in Section 2.2.4. To use machine-learning models to describe or to predict are two different goals with different focuses and different machine-learning models perform differently on predicting or describing a certain behavior [137]. In other words, one machine-learning approach can—in terms of accuracy—clearly outperform other machine-learning models when it comes to predicting performance behavior, but is bad in describing the observed performance behavior (e.g., a configuration option slows down the program by 3 seconds).

In this thesis, we use both predictive and descriptive modeling for different aspects.

State-of-the-Art Machine Learning Approaches

Literature has focused in the past on finding and improving approaches for performance modeling of configurable software systems [35, 38, 41, 129]. These approaches originate from the domain of machine learning and have a wide variety of different applications.

Table 2.7: Excerpt of different machine-learning strategies used for performance modeling.

Learning algorithm	Category	Used in
Classification and Regression Trees	Decision Tree	[35, 129]
DeepPerf	Deep Feedforward Neural Network	[35, 41]
k-Nearest Neighbours	Regression method	[35, 38]
Kernel-Ridge Regression	Regression method	[35, 38]
Multiple Linear Regression	Regression method	[35, 139]
Random Forest	Multiple decision trees	[35, 38]
Support Vector Regression	Regression method	[35, 38]

We show some prominent examples of machine learning approaches used for performance modeling in Table 2.7. In this thesis, we focus on multiple linear regression since this is in terms of accuracy among the best performing machine-learning approaches according to Grebhahn et al. [38], is also used in related work as we show in Table 2.7. Additionally, it allows for comprehending which configuration options and workloads influence the performance of a configurable software system. Next, we describe how the prediction error of a machine-learning model is determined and provide more detail into multiple linear regression.

Prediction Error The general goal of learning a performance-influence model on some data is to minimize the prediction error. Typically, a low prediction error indicates that the learned performance-influence model accurately depicts the given data, whereas a high prediction error indicates that the performance-influence model induces inaccurate predictions. Clearly, performance-influence models with a low prediction error on the provided data are desired. The prediction error is determined by using a statistical measure, also known as loss function. However, in the past, a variety of different loss functions have been proposed [4, 17, 106, 110] and used in literature [24, 40, 139]. In the domain of performance modeling, the mean absolute percentage error (MAPE) [37, 118, 139], mean squared error (MSE) [24, 41], and the minimal sum of squared error (MSSE) [40] are common choices.

In this thesis, we use the mean absolute percentage error, which is as follows:

$$\text{MAPE}_{\mathcal{C}_S}(\mathbb{P}_S, \Pi_S) = \frac{1}{|\mathcal{C}_S|} \cdot \sum_{c \in \mathcal{C}_S} \frac{|\mathbb{P}_S(c) - \Pi_S(c)|}{\mathbb{P}_S(c)} \cdot 100$$

where $\mathbb{P}$ denotes the function that maps the configuration to its measured performance value and Π_S denotes the performance-influence model that maps the configuration to its predicted performance value.

The rationale and advantages behind using MAPE are manifold. First, this loss function provides a percentage and, thus, is easier to interpret than metrics providing absolute values.

The most important advantage of using MAPE is that it can be used to compare the accuracy of the performance-influence models across different datasets; other metrics such as using absolute values do not offer this advantage. However, it is important to note that MAPE comes with a major drawback when handling with measured values ($\mathbb{P}$) near to zero. That is, that the percentage increases to a large value when the prediction deviates from the measured value and the measured value is near to zero. For instance, a configuration $c \in \mathcal{C}$ with a measured performance value of 0.1 and a predicted performance value of 2 would result in a prediction error of 1 900%. This phenomenon, however, occurs only rarely in the data of this thesis. In this thesis, the MAPE is used to assess the prediction error of the performance-influence models derived by multiple linear regression, which we describe next.

Multiple Linear Regression The resulting performance-influence model using multiple linear regression is a polynomial in which each additive term consists of a coefficient that describes either the base performance, the influence of a single configuration option (denoted as ϕ), or an interaction among multiple options (denoted as ψ) on the performance of the system.

For illustration, consider the configurable system from Table 2.2. A corresponding performance-influence model could be as follows:

$$\Pi(c) = 5 + \overbrace{3 \cdot c(\text{🔒})}^{\phi_{\text{E}}} + \overbrace{2 \cdot c(\text{⚙})}^{\phi_{\text{A1}}} - \overbrace{1 \cdot c(\text{🔒}) \cdot c(\text{⚙})}^{\psi_{\text{E,A1}}}$$

Notice that influences may be positive, negative, or negligible (close to 0). In our example, Encryption (🔒) increases the execution time by 3 (ϕ_{E}) and compression algorithm A1 (⚙) increases the execution time by 2 (ϕ_{A1}). Only if both Encryption and A1 are selected, the system is additionally sped up by 1, which is effectively an interaction between two configuration options ($\psi_{\text{E,A1}}$). The configuration-independent base performance is denoted by the polynomial's intercept 5. Again, the configuration-independent base performance also incorporates the performance of all mandatory options.

For multiple linear regression [6], performance models are of the following form:

$$\Pi_S(c) = \overbrace{\beta_0}^{\text{Base}} + \overbrace{\sum_{o \in \mathcal{O}} \beta_o \cdot c(o)}^{\text{Option influences}} + \overbrace{\sum_{o_1..o_i \in \mathcal{O}} \beta_{o_1..o_i} \cdot c(o_1) \cdot \ldots \cdot c(o_i)}^{\text{Interaction influences}}$$

For the rest of the thesis, we denote $\mathcal{T}_{\Pi_S}$ as the set of all additive terms of the performance-influence model Π_S of the configurable software system S.

The underlying problem of multiple linear regression is to solve the following equation:

$$\mathbf{y} = \mathbf{X}\beta + \varepsilon$$

where $\mathbf{X}$ denotes the input matrix in which each row corresponds to a configuration and each column represents a configuration option or interaction. β is a vector that encodes the influences of the configuration options and interactions; ε is a vector containing the prediction errors. Finally, $\mathbf{y}$ is a vector containing our dependent variable (i.e., our performance measurement results). The objective of multiple linear regression is to fit the vector β such that the error ε is minimal.

Table 2.8: Configurations with A1 and the three compression levels.

	Configuration	$\mathbb{P}$
c_1	{☆}	5
c_3	{📦, ★, ⚙}	7
c_4	{📦, ★★, ⚙}	9

For further illustration, we use the configurations with three compression levels and configuration option A1 as we show in Table 2.8 to fill the equation:

$$\begin{array}{c|ccccccc} & \text{ROOT} & \text{🔒} & \text{☆} & \text{★} & \text{★★} & \text{⚙} & \text{⛭} \\ \hline c_1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ c_3 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ c_4 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \end{array} \cdot \begin{pmatrix} \beta_0 \\ \beta_{\text{🔒}} \\ \beta_{\text{☆}} \\ \beta_{\text{★}} \\ \beta_{\text{★★}} \\ \beta_{\text{⚙}} \\ \beta_{\text{⛭}} \end{pmatrix} + \begin{pmatrix} \varepsilon_{c_1} \\ \varepsilon_{c_3} \\ \varepsilon_{c_4} \end{pmatrix} = \begin{pmatrix} 5 \\ 7 \\ 9 \end{pmatrix} \tag{2.4}$$

In this example, we have encoded only the base influence and all individual options, but not interactions. To also support interactions, the columns of the matrix $\mathbf{X}$ and the vector β in Equation 2.4 have to be expanded accordingly. Note that we do not use CompressionLevel as a numeric option, but use its one-hot encoded form with CompressionLevel_0 (☆), CompressionLevel_1. The first matrix of Equation 2.4 contains different configurations that were measured. Thereby, the rows represent the specific configurations and the columns stand for the configuration options and their assignment in the given configuration. This matrix is multiplied by a vector, whose values β_0, $\beta_{\text{🔒}}$, ... have to be determined by the multiple linear regression. The next vector contains the prediction errors and the right hand side of the equation contains the measured values for each configuration.

The overall idea of learning a performance-influence model is to refine a model iteratively until a user-defined threshold is reached [68, 139], as defined in Algorithm 1. Function `learn_model` receives the performance data and the feature model (i.e., information about the configuration options) as input. In Line 2–3, we initialize two variables, prediction error and the error improvement, which are used to check against the threshold for aborting the learning process. Line 5–19 contain the iterative procedure to perform multiple linear regression with feature forward selection [6, 75]. Therein, a list of different candidates (or features) is created in each step(Line 6). Each individual configuration option is a suitable candidate and so are interactions of configuration options with options that have already been added to the model. For instance, if a model contains the configuration option Encrpytion(🔒), then also interactions with c(🔒) such as c(🔒) · c(📦), c(🔒) · c(⚙), or c(🔒) · c(⛭) become candidates. The rationale of this iterative extension of the model is to counter the combinatorial explosion of combining all configuration options. This iterative approach is hierarchical in that it can add interactions for only those options that have been found in reducing the model error in prior iterations. For instance, if E and A1 interact with each other (i.e., c(🔒) · c(⚙)), the approach would

***Algorithm 1:** Learning a performance-influence model*

```
1  Function learn_model(feature_model, performance_data):
2      error ← $\infty$
3      error_reduction ← $\infty$
4      model ← $\emptyset$
5      while error > 1% and error_reduction > 0.1% do
6          candidates ← create_candidates(model, feature_model)
7          best_candidate_model ← $\emptyset$
8          best_candidate_error ← $\infty$
9          foreach candidate ∈ candidates do
10             candidate_model, candidate_error ← fit_and_predict(candidate, performance_data)
11             if candidate_error < best_candidate_error then
12                 best_candidate_error ← candidate_error
13                 best_candidate_model ← candidate_model
14             end
15         end
16         model ← best_candidate_model
17         error_reduction ← error − best_candidate_error
18         error ← best_candidate_error
19     end
20     model ← backward_selection(model)
21     return model, error
```

firstly include either c(🔒) or c(⚙) into the model and, in a later iteration, c(🔒) · c(⚙) if both together would reduce the prediction error for a hold-out set. After creating the candidates, each candidate is evaluated within a model that represents the state of the prior iteration (Line 9–15). To this end, we first fit the model to the performance data of a hold-out set (see Equation 2.4) in Line 10 returning the model including the candidate and the overall error (i.e., MAPE) of the corresponding model. In Line 11–14, the current candidate is selected as the best candidate if it reduces the error more than previous candidates. The best candidate of the current iteration is added to the model in Line 16. The reduction of the error resulting from the newly added candidate and the new error are then calculated. Note that choosing the best candidate represents a limitation of our approach since a worse performing candidate could lead to a better reduction of the error in future iterations. This iterative process is continued until one of the thresholds in Line 5 is no longer satisfied. Due to its hierarchical nature, the model can potentially include configuration options or interactions that may become irrelevant in later iterations. For instance, if only the interaction c(🔒) · c(⚙) is relevant for performance but the individual configuration options c(🔒) and c(⚙) are not, this approach would still include, at least, c(🔒) or c(⚙) as it reduced for some configuration the prediction error in previous iterations. To remove such unnecessary options and interactions, we apply a backward selection in Line 20. The backward selection removes all options and interactions that no longer improve the model error.

Prediction

One purpose for using machine-learning approaches is to predict the performance values of (unseen) software configurations. In this work, we use their predictive ability to assess the quality of different sampling strategies. Clearly, the choice of the sampling strategy affects the learning algorithm—what information it can extract from the sample set to be included in the influence model. As mentioned previously, there is a variety of sampling strategies

Table 2.9: Sampling strategies considered in the empirical study of Grebhahn et al. [38]. We distinguish between binary and numeric as well as structured and unstructured sampling.

Sampling strategy	Abbreviation	Numeric	Structured
Option-wise	OW	✗	✓
Negative Option-wise	NegOW	✗	✓
t-wise ($t \in \{2, 3\}$)	T2, T3	✗	✓
Random (Binary), three sizes (OW, T2, T3)	RB-OW, RB-T2, RB-T3	✗	✗
One Factor at a Time	OFAT	✓	✓
Box-Behnken Design	BBD	✓	✓
Central Composite Inscribed Design	CCI	✓	✓
Plackett-Burman Design	PBD	✓	✓
D-Optimal Design	DOD	✓	✓
Random (Numeric)	RN	✓	✗

Table 2.10: Subject systems of the empirical study of Grebhahn et al. [38].

Subject system	Application domain	# Options	# Configurations
DUNE MGS	Multigrid framework	11	2 304
Polly	Code optimizer (plugin for LLVM)	19	59 592
HSMGP	Multigrid framework	14	3 456
JavaGC	Java garbage collector	11	193 536
TriMesh	Scientific code library	13	239 360
VP9	Video encoder	20	216 000

for highly configurable systems. This has immediate implications for the learning step. The question is *whether* and *to what extent* the choice of the sampling strategy affects the ability of different learning algorithms to obtain accurate models. The canonical literature on machine learning and predictive modeling says not much about this issue. A recent empirical study conducted by Grebhahn et al. [38] provides first insights.

This study has analyzed the dependencies between 13 sampling strategies (see Table 2.9) and 6 learning algorithms (see Table 2.7) on 6 subject systems (see Table 2.10) in terms of prediction accuracy, stability, and measurement effort using SPL Conqueror [142]. The authors paid special attention to the fact that different learning algorithms provide different *hyper parameters* (i.e., different configuration options to tune the learning algorithm). As hyper parameters affect efficiency and accuracy of the learning procedure and even depend on the selected sampling strategy, we included an extension hyper-parameter optimization step, to ensure a fair comparison. For a detailed description, we refer to Grebhahn et al. [38].

In Figure 2.12, we summarize the results in the form of nested matrix plots. The outer plot shows the error rates achieved by different machine-learning algorithms. The inner plots show the different sampling strategies that have been used in combination with the respective learning algorithm, divided by binary (x axis) and numeric sampling strategies (y axis).

Note that the subject systems contain besides binary configuration options also numeric configuration options, such as page size or number of threads, which require dedicated numeric sampling strategies [139].

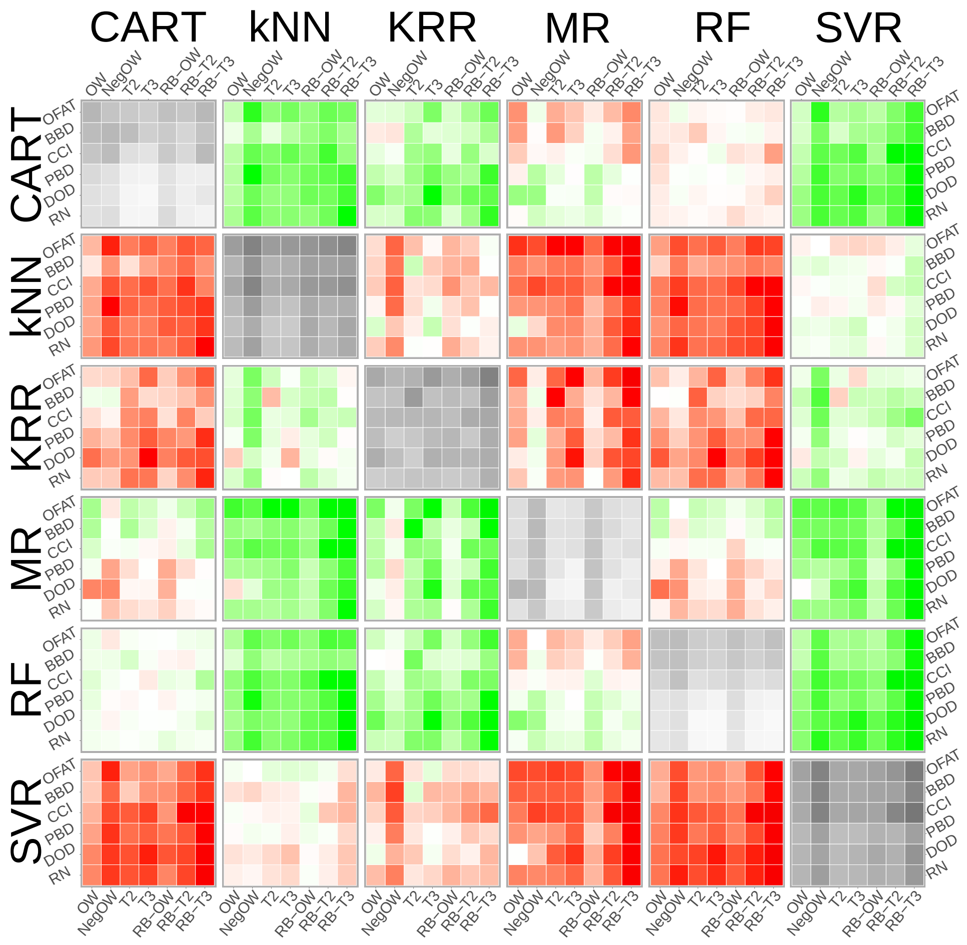


Figure 2.12: Comparison of combinations of machine-learning algorithms and sampling strategies in terms of predictions accuracy. Plots on the diagonal compare pairs of sampling strategies; the lighter the gray tone, the higher the prediction accuracy. Plots beyond the diagonal compare pairs of learning algorithms; shades of green (red) indicate that the learning algorithm in the row is more (less) accurate than the one in the column.

Note that we distinguish between plots that are on the diagonal of the top-level matrix and plots that are not: Each plot on the diagonal (gray-scale) compares pairs of sampling strategies given a specific learning algorithm. The lighter the gray tone, the higher the accuracy (the lower the error rate) of resulting influence model. For example, in the upper left plot of Figure 2.12, we show the error rates for Classification and Regression Trees (CART) when used with different combinations of sampling strategies.

The plots beyond the diagonal compare pairs of learning algorithms when combined with the different sampling strategies. Shades of green indicate that the learning algorithm in the row is more accurate than the one in the column (the more intense the green, the larger the effect), and shades of red otherwise. For example, in upper right plot, we compare the error rates achieved when using CART with the error rates of using Support Vector Regression (SVR). It is easy to see that CART outperforms SVR irrespective of the used sampling strategy.

Let us have a closer look at the results. First, let us focus on the machine-learning algorithms independently of the sampling strategy. Not surprisingly, the choice of the algorithm matters and has a significant effect on prediction accuracy. Although a clear ranking is not readily apparent, random forests (RF) outperform the other learning algorithms in most of the cases; Multiple Regression (MR) and CART also perform very well. Note that we use in this thesis multiple regression instead of random forests since performance-influence models, which use multiple regression can be used for comprehension (see Section 2.2.4).

Now let us look closer at combinations of sampling strategies and learning algorithms. Despite the rather clear picture that we obtained for learning algorithms, there seems to be no combination that is clearly superior in all cases. Although there is a trend to a specific combination of learning algorithm and sampling strategy, this does not hold for all cases. In many cases, random forests or multiple regression in combination with option-wise or random sampling perform best in terms of accuracy, but there are exceptions.

A further notable result, which is not directly visible in Figure 2.12, is that, despite the rather small sample sets selected by option-wise (e.g., 72 configurations for VP9), random forests and multiple regression are still able to learn comparatively accurate influence models. Increasing the size of the sample set (e.g., from 350 for option-wise to 3 780 for 3-wise for VP9) often increases the prediction accuracy only marginally (e.g., by only 1% for VP9). The paradigm "the more, the better" holds for the domain of configurable software systems only when the more configurations also provide new information (e.g., influences of interactions not seen before). However, due to the possibly exponential number of interactions with respect to the number of configuration options, it is unclear which additional configurations improve accuracy.

Comprehension

One major issue in performance prediction is that machine-learning models from different learning algorithms may not only differ in prediction accuracy but also in *interpretability*. After all, a key goal of influence models is not only prediction, but also *comprehension* [68]. Developers, administrators, and users would like to know *why* a certain configuration is fast, not only *whether* it is faster than another. Exploring the influence of learning on interpretability of influence models is clearly a rich and promising avenue that we follow for Chapter 4 and Chapter 5.

However, not all learning algorithms are suitable for comprehension [137] since the models are not human-readable. For instance, deep neural networks such as DeepPerf [41] represents a broad family of machine-learning algorithms inspired by the function of the human brain. Deep neural networks are complex constructs that achieve good results on predicting new data [41], but bear the caveat that they cannot be comprehended by humans.

Besides using multiple linear regression, researchers also rely on other approaches such as SHapley Additive Explanations (SHAP) values [89] and Local Interpretable Model-agnostic Explanations (LIME) to extract information from other machine-learning techniques [124]. In principle, these approaches extract feature information using the predictions and structure of certain models. This way, SHAP values exploit the structure of tree-based models to deliver even better insights. However, both approaches support any machine-learning approach that is capable of predicting. In this thesis, we focus on using the models from multiple linear regression since they were successfully used in the past for describing the influence of certain features [37].

Multiple linear regression relies on a model consisting of a polynomial that can be used both to predict and to comprehend. Multiple linear regression was already used for comprehending configurable software systems in literature [37]. For instance, the following performance-influence model could be used to predict the performance (in seconds) of compression algorithm A1 of Comp in Table 2.8:

$$\Pi(c) = 5 + 2 \cdot c(\text{⚙}) \cdot c(\text{★}) + 4 \cdot c(\text{⚙}) \cdot c(\text{★★}) \tag{2.5}$$

For the sake of simplicity, we ignore every other configuration option of Comp in this example. Using Equation 2.5, we derive that Comp without A1 needs about 5 seconds. When A1 is selected, then it depends on CompressionLevel. A1 and medium compression (★) needs about 2 seconds more, whereas maximum compression (★★) needs about 4 seconds more. In this thesis, we compare multiple polynomials to gain insights into how the performance of different configuration options and interactions evolve during time (see Chapter 4) and their effect in different workloads (see Chapter 5). An issue that needs to be addressed when using performance-influence models for comprehension is multicollinearity [27]. Multicollinearity represents a big challenge in regression analysis and refers to a situation, in which a term of a linear model can be linearly predicted by other terms. That is, multiple terms represent the same effect such that it becomes unclear, which of these terms has the true influence on the independent variable and to what extent.

For a comprehensive and an unambiguous analysis of a software system's evolution (see Chapter 4) and workload effect (see Chapter 5), we have to assure that the terms of our models are not multicollinear. Otherwise, we can end up with different performance-influence models all predicting the same value, but with diverging influences of options and interactions, threatening internal validity of our analysis. As a countermeasure, one can apply a variance inflation factor (VIF) analysis [53, 75, 104] and exclude terms that can be completely linearly predicted by other terms. For illustration, consider Table 2.8 and the following performance-influence models:

$$\begin{aligned}\Pi_1'(c) =& 10 + 15 \cdot c(\mathrm{C}) \cdot c(\mathrm{Z}) \\ \Pi_2'(c) =& 10 + 15 \cdot c(\mathrm{Z})\end{aligned}$$

Both performance-influence models predict the same performance values. The terms $c(\text{⚙}) \cdot c(\text{☆☆})$ and $c(\text{⚙}) \cdot c(\text{☆☆}) \cdot c(\text{📦})$ are perfectly multicollinear because when A1 (⚙) is selected in a configuration, Compression (📦) is also always selected. Hence, we cannot distinguish the influence of the interaction $c(\text{⚙}) \cdot c(\text{☆☆}) \cdot c(\text{📦})$ from the influence of the option $c(\text{⚙}) \cdot c(\text{☆☆})$. Having both terms in a performance-influence models would cause infinite possibilities of assigning coefficients to these terms, as demonstrated here:

$$\begin{aligned}\Pi_1'(c) =& 10 - 10 \cdot c(\mathrm{C}) \cdot c(\mathrm{Z}) + 25 \cdot c(\mathrm{Z}) \\ \Pi_2'(c) =& 10 + 10 \cdot c(\mathrm{C}) \cdot c(\mathrm{Z}) + 5 \cdot c(\mathrm{Z})\end{aligned}$$

Again, both performance-influence models make the same predictions but assign completely different coefficients to the terms. The VIF analysis detects such cases and declares the terms as multicollinear.

Variance Inflation Factor Analysis Next, we describe the VIF analysis [104] in more detail. The general idea behind the VIF analysis is to determine if a term from a performance-influence model can be predicted by other terms; or in other words: a term is multicollinear to the remaining terms of a performance-influence model. Identifying the multicollinear terms of a performance-influence model is performed by an iterative approach. We denote $\mathcal{T}_{\Pi_S}$ as the set of all terms of a performance-influence model Π_S. For the VIF analysis, an ordinary least square regression is applied for each $\mathsf{T}_i \in \mathcal{T}_{\Pi_S}$:

$$\mathsf{T}_i = \alpha_0 + \sum_{\mathsf{T}_j \in \mathcal{T} \wedge \mathsf{T}_j \neq \mathsf{T}_i} \alpha_j \cdot \mathsf{T}_j + \varepsilon, \tag{2.6}$$

where α_0 is the intercept (similar to β_0 in regression models), α_j are constants, and ε represents the error (similar to the error in multiple linear regression in Section 2.2.4).

Afterwards, the VIF factor is calculated for the term $\mathsf{T}_i \in \mathcal{T}_{\Pi_S}$ with the following formula:

$$\mathsf{VIF}_{\mathsf{T}_i} = \frac{1}{1 - R^2_{\mathsf{T}_i}}$$

where $R^2_{\mathsf{T}_i}$ is the coefficient of determination [108] of Equation 2.6. The coefficient of determination returns a value in the range of $[0, 1]$. A value of 0 for $R^2_{\mathsf{T}_i}$ (i.e., $\mathsf{VIF}_{\mathsf{T}_i} = 1$) represents that the term T_i does not correlate at all with the intercept or any of the remaining terms on the right-hand side of Equation 2.6. A value of 1 for $R^2_{\mathsf{T}_i}$ (i.e., $\mathsf{VIF}_{\mathsf{T}_i} = \infty$) represents that the term T_i can be completely estimated by the intercept and the remaining terms on the right-hand side of Equation 2.6[6].

Applying the VIF analysis on our previous example of Π'_1 would result in perfect multicollinearity. That is, the terms containing $c(\mathsf{Z})$ and $c(\mathsf{C}) \cdot c(\mathsf{Z})$ describe the same configurations since one term can be estimated by the other term:

$$c(\mathsf{Z}) = 1 \cdot c(\mathsf{C}) \cdot c(\mathsf{Z})$$

Consequently, the coefficient of determination would be 1, the VIF is ∞, and we remove the left-hand side term.

6 Note that we exclude only perfectly multicollinear terms, since perfect multicollinear terms are completely interchangeable. We do not use any threshold such as 5 [20, 113] as commonly used in literature because configuration options and their interactions can always be multicollinear to some extent due to overlap.

3

Distance-Based Sampling of Software Configuration Spaces

This chapter shares material with the following publication: Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. „Distance-Based Sampling of Software Configuration Spaces.“ In: Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE), IEEE/ACM, 2019, pp 1084–1094 [60]

In this chapter, we comprise the contributions to the configuration space of software systems. Measuring and analyzing the performance of the whole configuration space of software systems is difficult due to the combinatorial explosion of the configuration space in relation to the number of configuration options. As a consequence, research relied increasingly on performance-influence modeling in the past [40, 41, 139], where the idea is to measure only a low number of configurations using a sampling strategy and, afterwards, learn a performance-influence model through machine learning [39]. This performance-influence model can then be used to predict unseen configurations to find the performance-optimal configuration or to comprehend the impact of configuration options on the performance of the software. Inherently, the quality of the performance-influence model depends on the machine-learning approach and the sampling strategy (see Section 2.2). In this chapter, we focus primarily on the multitude of different sampling strategies. In essence, we propose a novel sampling strategy—distance-based sampling—for binary configuration options, which we present in the following.

Despite the benefits of configurability, identifying the performance-optimal configuration for a given setting is often a non-trivial task, due to the sheer size of configuration spaces [152] and potential interactions among configuration options [64, 68]. To identify the performance-optimal configuration of a configuration space, one can measure the performance of every valid configuration of the software system in a brute-force manner, which usually does not scale.

Instead, a more efficient approach that consists of sampling and machine learning is used (see Section 2.2). There, the main idea of sampling (see Section 2.2.2) is often to cover the configuration space such that one obtains a *representative* sample set, which, ideally includes both influential configuration options and interactions among options relevant to performance, so that accurate performance-influence models can be learned.

In general, a uniform coverage of the configuration space is desirable to obtain a representative sample set when no prior knowledge is available, since it tends to be unbiased when covering the configuration space. However, it is far from trivial to ensure unbiased uniformity if there are non-trivial constraints among configuration options. To achieve the goal in a light-weight way, we propose a novel sampling strategy, called *distance-based sampling*, that addresses the shortcomings of existing strategies, as we will discuss next.

The key idea behind distance-based sampling is to produce a sample set that covers the configuration space as uniformly as possible (or following another given probability distribution). To this end, distance-based sampling relies on a *distance metric* and assigns each configuration a *distance value*. It further relies on a *discrete probability distribution* to select configurations according to their distance values from the configuration space. It differs from other sampling strategies in that (1) it spreads the selected configurations across the configuration space according to a given probability distribution and is able to resemble the performance distribution of the whole population, (2) it does not require an analysis on the whole population, and (3) it uses internally a constraint solver for efficiency while avoiding locally-clustered sample sets. For illustration, we show in Figure 3.1 how distance-based sampling spreads the configurations in terms of performance. Other sampling strategies, such as 3-wise sampling sampling resemble the whole population worse in that 3-wise sampling covers primarily configurations with a lower performance.

In summary, our contributions are as follows:

- We define a new distance-based sampling strategy that is based on a given discrete probability distribution and a distance metric for configurations of software systems.
- We perform an empirical study to compare our sampling strategy with other widely-used sampling strategies learning performance-influence models for 10 popular real-world software systems. We find that distance-based sampling achieves better results in terms of prediction accuracy and robustness than other sampling strategies.
- To further improve the diversity of the sample set, we devise an optimization of distance-based sampling by iteratively forcing the selection of the least frequently selected configuration option for configurations with the same distance. The optimization leads to a significantly higher robustness and significantly better prediction accuracy of distance-based sampling.
- We show that distance-based sampling is more flexible since the probability distribution it relies on can be exchanged on demand. Further, we assess the scalability of distance-based sampling compared to other sampling strategies in an empirical study on 9 real-world software systems with huge configuration spaces.

All experiment and replication data of this chapter are available on a supplementary web site[1].

1 `https://github.com/se-passau/Distance-Based_Data/`, last accessed on 02/25/2023.

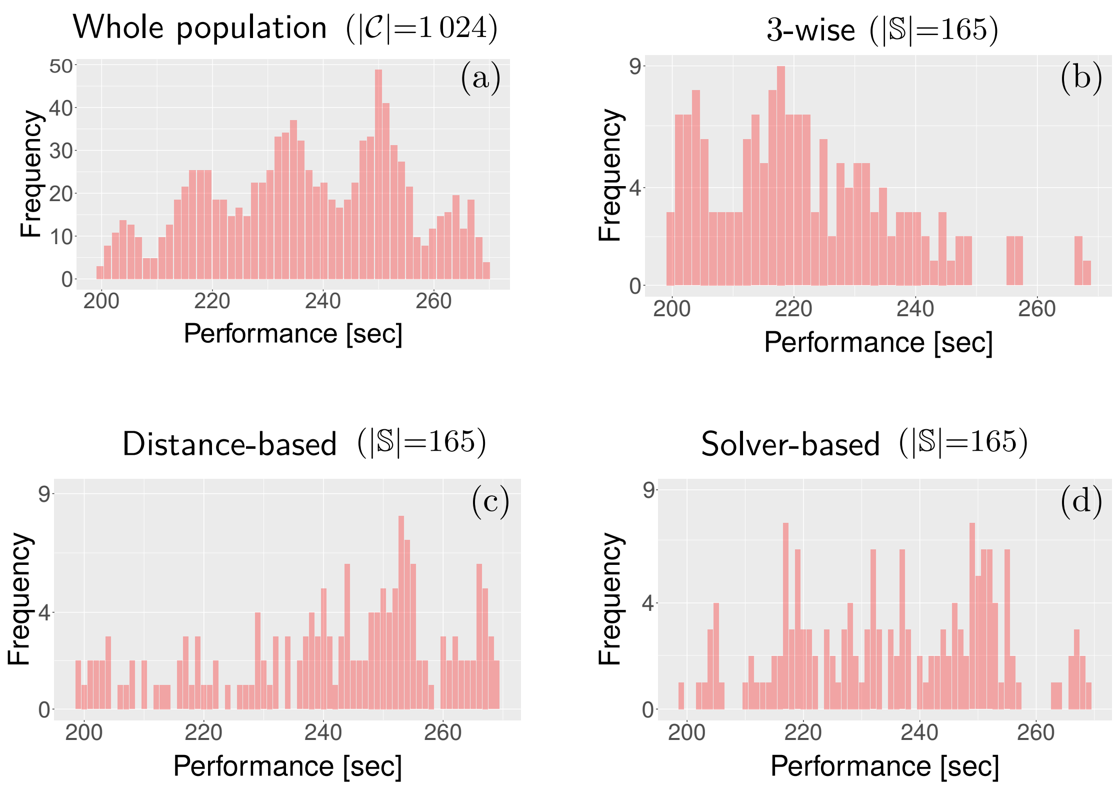


Figure 3.1: The performance distribution of (a) the whole population of LLVM (see Section 3.2.3), the sample set selected (b) by 3-wise sampling, (c) by distance-based sampling, and (d) by solver-based sampling. In 3-wise sampling, some high performance values are missed, leading to a skewed distribution, whereas distance-based sampling resembles the distribution from the whole population better.

3.1 Sampling

Sampling is the process of selecting a subset $\mathbb{S}_S \subseteq \mathcal{C}_S$ of all valid configurations $\mathcal{C}_S$ of a given configurable software system S. There are different strategies for sampling binary configuration options[2]: random sampling, solver-based sampling, and coverage-based sampling.

Random sampling: One way to create a sample set is by randomly assigning either 0 or 1 to each configuration option for each configuration [39]. However, it is very likely that many invalid configurations are selected this way due to unsatisfied constraints, which makes this strategy inefficient. Chakraborty et al. [18] use hash functions to split the configuration space recursively in multiple regions, and they select configurations from each of the regions. Still, this strategy produces many invalid configurations as also Plazar et al. point out [120]. Chen et al. [22] use a distance metric to find different test inputs for methods to uniformly cover the configuration space. However, they do not consider constraints among the input variables and, thus, produce many invalid configurations. Krishna et al. [73] use Markov-Chain Monte-Carlo methods in combination with a mutation and cross-over operator to draw

2 Note that we focus on binary sampling strategies (see Section 2.2.2) in this work.

random samples from the configuration space. Again, this approach produces many invalid configurations which have to be filtered out. Oh et al. [112] encode a system's configuration space using a binary decision diagram. This way, they can represent and enumerate all configurations in a compact way, such that they can randomly draw configurations. However, construction time and memory consumption of binary decision diagrams are high, and they do not scale to the largest configurable software systems [134]. Gogate and Dechter [34] propose a random sampling strategy that uniformly selects configurations without enumerating all configurations using the Monte-Carlo method. This strategy also selects invalid configurations, though.

Solver-based sampling: Many strategies use an off-the-shelf constraint solver, such as SAT4J[3], for sampling. Naturally, these strategies do not guarantee true randomness [47] as in random sampling. Often the sample set consists only of the first *k* solutions provided by the constraint solver [22], and the internal solver strategy is typically to search in the „neighborhood"of an already found solution. Hence, the result is a locally clustered set of configurations. To weaken the locality drawback of solver-based sampling, Henard et al. [47] change the order of configuration options, constraints, and values in each solver run. This strategy, which we call henceforth *randomized solver-based sampling*, increases diversity of configurations, but it cannot give any guarantees about randomness or coverage. As we will show in our evaluation, this strategy requires to rebuild the entire solver model from scratch at each solver call (i.e. selection of one configuration), which is computationally expensive.

Coverage-based sampling: Coverage-based sampling strategies optimize the sample set according to a specific coverage criterion. One prominent example is *t-wise* sampling [58, 77, 92]. This sampling strategy selects configurations to cover all combinations of t configuration options being selected. For instance, pair-wise (t=2) sampling covers all pair-wise combinations of configuration options being selected. To identify the influence of pairs of configuration options and not to be affected by influences of other configuration options, Siegmund et al. [140] improve *t-wise* sampling by additionally minimizing the number of other selected configuration options in each configuration. Another strategy aims at a balanced selection and deselection of all configuration options in the sample set. Sarkar et al. [129] showed that such a frequency-based sampling further improves the accuracy of performance-influence models learned based on the sample set. Other coverage-based sampling strategies are, for example, statement-coverage sampling [147] or most-enabled-disabled sampling [95]. Statement-coverage sampling is a white-box strategy, in which the configurations are selected such that every block of optional code from the software system is selected, at least, once; whereas most-enabled-disabled sampling selects just one configuration where all configuration options are selected and one where all configuration options are deselected. The main problem of these strategies is that they require prior domain knowledge to select a proper coverage criterion. Thus, depending on the coverage criterion, specific regions of the configuration space are emphasized, as shown in Figure 3.2: We see that a higher number of selected configuration options leads to higher performance values and contains information from interactions among multiple configuration options. Since t-wise sampling focuses only on a specific part of the configuration space (e.g., configurations with distances between 2 and 5), we miss certain performance values in the sample set (e.g., greater than 250 seconds).

3 `https://www.sat4j.org/`, last accessed on 02/23/2023.

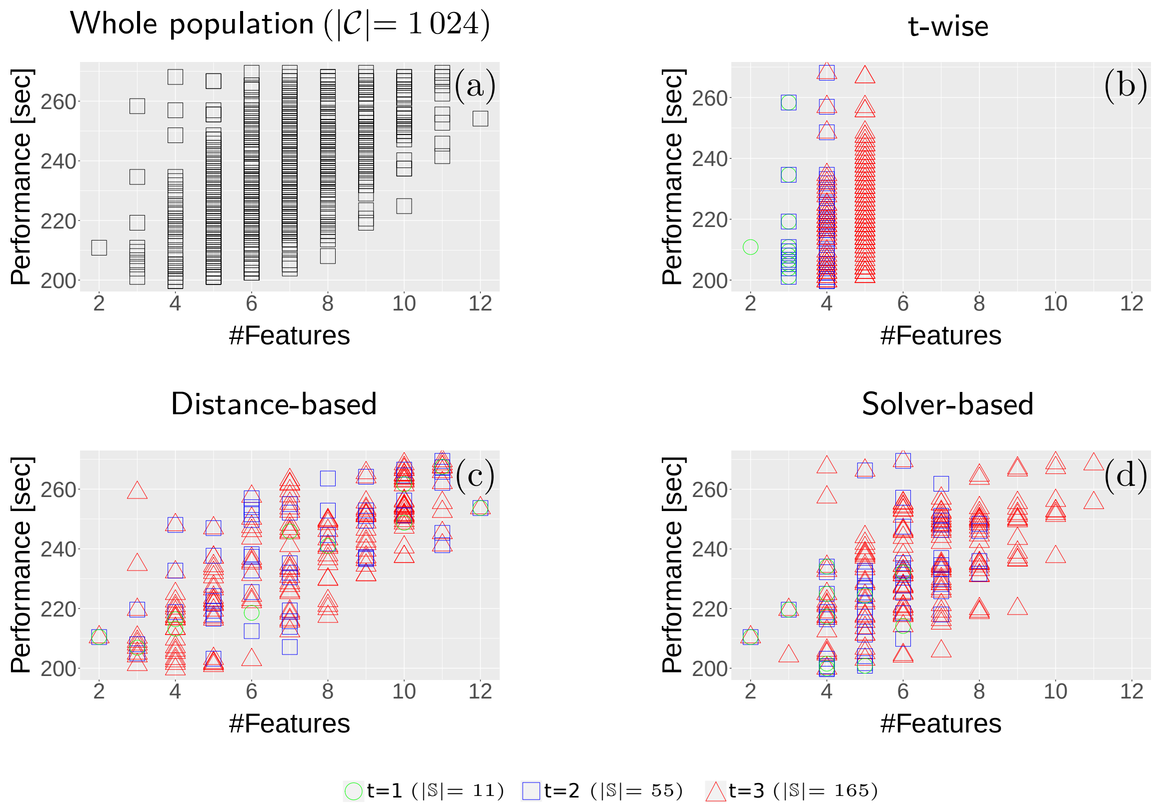


Figure 3.2: Distribution of configurations of LLVM (see Section 3.2.3) based on their distance value and their performance, for (a) the whole population, a sample set selected (b) by *t*-wise sampling, (c) by distance-based sampling, and (d) by solver-based sampling with the same sample size as *t*-wise with *t*=1, *t*=2, and *t*=3, respectively.

By contrast, a sample set that covers all performance values is more representative, as it resembles the whole population better. Moreover, not every configuration option interacts with any other configuration option, so not all pairs are relevant. So, the sample set is likely unnecessarily large.

3.1.1 Approach

Distance-based sampling aims at covering the configuration space by uniformly (or according to another given probability distribution) selecting configurations with different distance values (and therewith interaction degrees) without relying on a whole-population analysis. In Section 3.1.1, we describe the basic algorithm; in Figure 9, we present an optimization that further increases the diversity of the sample set.

Basic Algorithm

The key idea to spread the sample set across the configuration space to increase diversity in the sample set is to use a *distance metric* in combination with a *discrete probability distribution* (e.g., a uniform distribution or a binomial distribution).

Algorithm 2: *Distance-based sampling*

```
Input: $VM$, $numSamples$, $probabilityDistr$
Output: $sampleSet$
1 $sampleSet \leftarrow \emptyset$
2 while otherSolutionsExist($VM$, $sampleSet$) and size($sampleSet$) $< numSamples$ do
3     $d \leftarrow$ selectDistance($probabilityDistr$)
4     $c \leftarrow$ searchConfigWithDistance($VM$, $d$) ▷ Search for configuration $c$ with exactly $d$
          configuration options selected
5     if $c \neq \emptyset$ then
6         $sampleSet \leftarrow sampleSet \cup \{c\}$
7     end
8 end
9 return $sampleSet$
```

In Algorithm 2, we describe the algorithm behind distance-based sampling. It receives three parameters as input: the variability model (VM), the number of configurations to be selected ($numSamples$), and the probability distribution to use ($probabilityDistr$). Internally, we use a constraint solver that uses the variability model to determine the valid configurations. We assume that the solver is globally available to the algorithm.

The algorithm selects a distance d based on the probability distribution ($probabilityDistr$). The distance is passed as an additional numeric constraint (i.e., in addition to the constraints of the variability model) to the constraint solver, which searches for a solution with exactly d configuration options selected (Line 4). If a solution (i.e., a valid configuration) is found, it is included into the sample set (Line 6). If not, another distance d is selected until a valid configuration with this distance is found. This process is repeated until the sample set contains the desired number of valid configurations or there are no more solutions (Line 2).

In what follows, we define the distance metric and the discrete probability distribution that we use, and we describe the selection of a valid configuration in more detail.

Distance metric (selectDistance): Figure 3.2 illustrates that t-wise sampling covers only specific intervals in the range of possible distances. In fact, t-wise sampling misses information on interactions among more than t configuration options. By using a distance metric to diversify the sample set, we cover more regions of the configuration space, which leads to a more diverse sample set. In what follows, we use the Manhattan distance [72] of a configuration to the origin of the configuration space c_0 ($\forall o \in O : c_0(o) = 0$) as distance; another reference point would be possible, though.

So, let dist: $\mathcal{C} \rightarrow \mathbb{N}$ be the distance metric defined as follows:

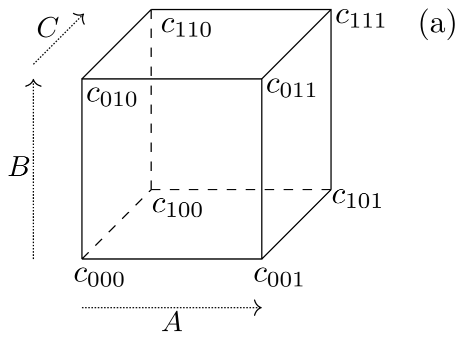
$$\text{dist}(c) = \sum_{o \in \mathcal{O}} c(o)$$

where $c \in \mathcal{C}$ is a valid configuration. Let $\mathcal{D}$ be the set of all distances:

$$\mathcal{D} = \{\text{dist}(c) \mid \forall c \in C\}$$

Note that, in the case of having only binary configuration options and using the Manhattan distance, the distance is just the number of selected configuration options of the configuration.

For instance, for d=2, we will search for a configuration that has exactly two configuration options selected and all remaining configuration options deselected.



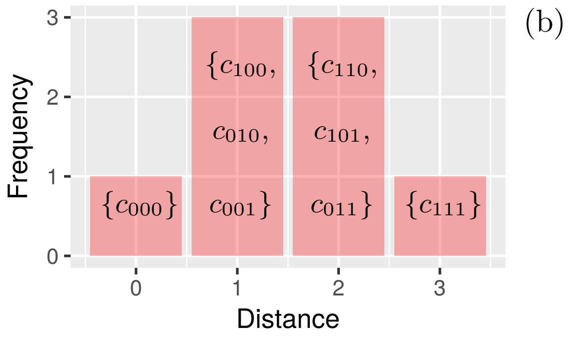


Figure 3.3: Example for applying the distance function to a software system with three configuration options A, B, and C without any constraints. In (a), we show the configuration space and in (b) we illustrate the distribution of distances.

Probability distribution (probabilityDistr): In Figure 3.3, we show the distances and the number of configurations (frequency) per distance for a system with three configuration options A, B, and C, without any constraints. In this small example, the distance distribution is easy to compute: We derive all valid configurations and apply the distance metric to each of them. However, deriving all valid configurations (i.e., the whole population) is infeasible for complex systems. Fortunately, it turns out that we do not need a data set following an exact probability distribution. Instead, we use pre-defined discrete probability distributions (e.g., uniform, geometric, binomial) to define the desired distribution of our sample set. These discrete probability distributions (*probabilityDistr*) are used to express the likelihood of choosing a certain value $u \in U$. In every discrete probability distribution P, we have $\sum_{u \in U} P(X = u) = 1$, where X is a random variable. In what follows, we use the discrete *uniform* distribution. So, we uniformly draw a distance $d \in \mathcal{D}$ to derive a configuration with d configuration options selected. Thus, we obtain:

$$P(X = d) = \frac{1}{\text{card}(\mathcal{D})}$$

where card is the cardinality function. In other words, each distance d is equally likely to be picked. While it is possible to use another discrete probability distribution, such as the geometric distribution or the binomial distribution, we fix this degree of freedom for now, to keep the discussion focused and the experiment design tractable. Nevertheless, we performed experiments with different distributions, which we discuss in Section 3.3.4.

Without computing the whole population or understanding all constraints in the variability model, we have no knowledge about the value domain of $\mathcal{D}$ and thus about $P(X = d)$. We can approximate the lower and the upper bound for $\mathcal{D}$, though, by using the number of mandatory configuration options and the number of all configuration options, respectively.

$$\max(\mathcal{D}) = \text{card}(\{o \mid o \in \mathcal{O}\})$$

$$\min(\mathcal{D}) = \text{card}(\{o \mid o \in \mathcal{O} \wedge o \text{ is mandatory} \wedge \text{all parents of } o \text{ are mandatory}\})$$

where card is the cardinality function. Note that all parents nodes (i.e., including transitive parent nodes) of a configuration option o have to be mandatory to increase the minimum distance. Mandatory configuration options can be easily computed based on the given constraints of the variability model [10].

Configuration selection (searchConfigWithDistance): After choosing a certain distance d, we select a configuration that has a distance of d (d options selected in our setting), for which we use a constraint solver. To this end, we add an additional constraint to the solver describing that exactly d options have to be selected in the configuration. If there is no more configuration with exactly d selected configuration options, another distance d is selected. This process is repeated until the sample set contains a given number of configurations or no further configurations are found. In cases where we select several times the same distance value, the constraint solver could generate locally clustered solutions. To address this problem, we propose an optimization to further increase diversity of our sample set, as we describe in Figure 9.

Time complexity: The most costly function in Algorithm 2 is searchConfigWithDistance, whose complexity is dominated by the time of computing a feasible solution by the constraint solver. Theoretically, a constraint solver, such as a SAT solver, has an exponential time complexity to solve a satisfiability problem [12]. Practically, state-of-the-art constraint solvers are able to handle thousands of variables and constraints efficiently [82]. In our experiments, the constraint solver considered up to 54 variables and up to 216 000 constraints, which resulted in less than 0.3 seconds to find a solution (i.e., a valid configuration) on average. But, to draw a clearer picture of how much overhead the constraint to find a solution with distance d induces into configuration spaces with large configuration spaces, we perform scalability experiments in Section 3.3.

Increasing Diversity

In preliminary experiments with Algorithm 2, we noticed that the produced sample sets may lack diversity in that some configuration options are selected in many configurations and some only in few or even none. Hence, to increase diversity of the sample set, we refine the configuration selection procedure of distance-based sampling by adding configurations that contain the least frequently selected configuration options. This way, we reduce the possibility of missing or underrepresenting certain configuration options in the sampling process. Technically, we determine a ranking over the frequency of configuration options, which is defined as follows:

$$\forall o \in \mathcal{O} : \mathrm{card}(\{c \mid c \in \mathbb{S}_S \wedge c(o) = 1\})$$

If there is no valid configuration with the given configuration option and distance, we select the next option in the ranking and so on.

In Algorithm 3, we show the optimized version of distance-based sampling, which we call *diversified distance-based sampling*. The novelty here is that we count the number of selections of each configuration option for each distance $d \in \mathcal{D}$. To this end, we define one map in Line 1 for each distance $d \in \mathcal{D}$ and update the map in Line 12 when a new configuration is added to the sample set.

The least frequent configuration option of the current distance d is selected using the map in Line 6 and used to retrieve the next configuration in Line 7. If there is no more configuration with the given distance d that contains the candidate, this candidate is removed from the distance's candidate map in Line 8 and the next configuration option is used. As we show in Line 5, another least frequent candidate is repeatedly chosen until a valid configuration is found.

Algorithm 3: Diversified distance-based sampling

```
Input: VM, numSamples, probabilityDistr
Output: sampleSet
1  candidates ← getAllOptions(VM)    ▷ Generates a list of candidates, one candidate for each configuration option
2  sampleSet ← $\emptyset$
3  while otherSolutionsExist(VM, sampleSet) and size(sampleSet) < numSamples do
4      d ← selectDistance(probabilityDistr); c ← $\emptyset$
5      while candidatesExist(candidates, d) and $c = \emptyset$ do
6          candidate ← getLeastFrequentCandidate(candidates, d)
7          c ← searchConfigWithDistance(VM, candidate, d)
8          if $c = \emptyset$ then removeCandidate(candidates, candidate, d)
9      end
10     if $c \neq \emptyset$ then
11         sampleSet ← sampleSet $\cup$ {c}
12         updateCandidateMap(c, d, candidates)
13     end
14 end
15 return sampleSet
```

3.2 Experiment Setup

In this section, we introduce our research questions regarding the comparison of distance-based sampling with other state-of-the-art sampling strategies. Furthermore, we describe how we attempt to answer the research questions and the software systems we use for the comparison.

3.2.1 Research Questions

The prediction accuracy of machine-learning techniques largely depends on the data set, which is defined by the sampling strategy, in our setting. Some sampling strategies, such as random sampling, are affected by randomness, which can have a considerable influence on the sample set and, consequently, on the prediction accuracy. Hence, we consider both the prediction accuracy and its robustness when comparing sampling strategies. To this end, we aim at answering two research questions:

Research Question 1

What is the influence of using distance-based, diversified distance-based, random, solver-based, randomized solver-based, and t-wise sampling on the accuracy of performance predictions?

Research Question 2

What is the influence of randomness of using distance-based, diversified distance-based, solver-based, randomized solver-based, and random sampling on the robustness of prediction accuracy?

Note that we have excluded t-wise sampling from RQ_2, as it is deterministic in our setting and does not lead to variations.

As pointed out earlier, distance-based sampling relies on a constraint solver to find a configuration with a certain distance d. Although being only one additional constraint, this constraint to find any configuration with exactly d out of n selected configuration options could be difficult to process for large configuration spaces. To this end, we aim at comparing the sampling strategies in terms of the time they need to find a certain number of solutions:

Research Question 3

What is the performance of using distance-based, diversified distance-based, solver-based, and randomized solver-based on searching a fixed number of solutions?

Note that we exclude t-wise sampling from RQ_3 since t-wise does not allow for sampling an arbitrary number of configurations. Further, since we focus on large configuration spaces, we cannot retrieve all valid configurations for the configuration spaces anymore. Since random sampling needs to assess all valid configurations, we further exclude random sampling. A similar investigation on the scalability of uniform sampling strategies was performed by Plazar et al. [120]. They, however, have focused on uniform sampling strategies that also sample invalid configurations, which is out of scope in this chapter.

3.2.2 Operationalization

To answer our research questions, we apply a state-of-the-art machine-learning technique that relies on multiple linear regression and feature-forward selection [139] to learn performance-influence models based on the sample sets defined by the different sampling strategies.

To answer RQ_1, we use the resulting performance-influence models to predict the performance of the whole population of each of our subject systems. We quantify the difference between the predicted performance $\Pi_S(c)$ and the measured performance $\mathbb{P}(c)$ by means of the *error rate* for all configurations $c \in \mathcal{C}$ as follows:

$$error_S(c) = \frac{|\Pi_S(c) - \mathbb{P}_S(c)|}{\mathbb{P}_S(c)} \tag{3.1}$$

where $\Pi_S(c)$ is the predicted performance of configuration c and $\mathbb{P}_S(c)$ the measured performance of configuration c. Lower error rates indicate a higher prediction accuracy and, thus, are better. We further determine the mean error rate of the whole population:

$$\overline{error}_{\mathcal{C}_S} = \frac{\sum_{c \in \mathcal{C}_S} error_S(c)}{\text{card}(\mathcal{C}_S)} \tag{3.2}$$

where card is the cardinality function. Note that we compute the error rate based on predictions for the whole population, including configurations from the sample set. The background is that we use regression learning as machine-learning technique, which may produce imperfect predictions even for configurations from the sample set, and we would like to take that into account. Also note that $\overline{error}$ corresponds to the loss function mean absolute percentage error (MAPE) presented in Section 2.2.4.

Further note that, initially, we compared the performance distributions of the sample sets and the whole population. But, as the similarity of distributions of a sample set and the whole population does not necessarily imply good predictions, we refrain from this evaluation method and decided for the more definitive method of comparing error rates (i.e., MAPE).

To answer RQ_2, we perform the sampling and machine-learning procedures 100 times per experiment run using different seeds for the random number generator, and we compute the variance across the error rates:

$$\widetilde{error}_{\mathcal{C}_S} = \mathrm{Var}(\{error(c) \mid c \in \mathcal{C}_S\}) \tag{3.3}$$

A lower variance indicates a higher robustness (i.e., is better).

So, in our experiments, the *independent variables* are the subject systems, the sample sizes, the sampling strategies, and the random seeds for the random number generator. To rule out influences of different sample sizes, we selected for RQ_1 and RQ_2 the same sample sizes for (diversified) distance-based, (randomized) solver-based, and random sampling such that their size equals the size for t-wise sampling with t=1, t=2, and t=3[4]. For RQ_3, we chose to use the sample sizes 10, 100, and 1 000 to draw a picture how the different sampling strategies perform. Choosing these sample sizes has the advantage that the results are more easily interpretable since t=1, t=2, and t=3 are relative to the case study. The *dependent variables* are, for RQ_1, the mean error rates of the performance predictions (i.e., $\overline{error}$), for RQ_2, the variance of the error rates of the performance predictions on the whole population (i.e., $\widetilde{error}_{\mathcal{C}_S}$), and for RQ_3, the performance (i.e., execution time) of the sampling strategies.

For both research questions RQ_1 and RQ_2, we perform a standardization on the error rates when considering different subject systems, to be able to answer the research questions without considering each subject system separately. For RQ_1, we use a Kruskal-Wallis test [74] to identify for every sample size t=1, t=2, and t=3 if the error rates of, at least, two sampling strategies differ significantly ($p < 0.05$). As proposed by Arcuri and Briand [9], we then perform pair-wise and one-sided Mann-Whitney U tests [91] to identify which sampling strategy leads to significant lower error rates than others. In addition to testing for statistical significance, we determine the effect size using the $\hat{A}_{12}$ measure by Vargha and Delaney [150]. Values of $\hat{A}_{12}$ of more than 0.56, 0.64, and 0.71 indicate small, medium, and large effect sizes, respectively.

To answer RQ_2, we use Levene's test [81] to identify whether the variances of, at least, two sampling strategies differ significantly from each other. If this is the case, we perform a pair-wise comparison using one-sided F-tests [144] to identify the sampling strategy with the lower variance.

For RQ_3, we perform a pair-wise and one-sided Mann-Whitney U tests [91] to assess for each subject system and sample size which sampling strategy leads to significant lower execution times.

4 In t-wise sampling, the size can be chosen only by using t, whereas in distance-based, solver-based, and random sampling, any positive number in $[1, |\mathcal{C}|]$ can be used for the specification of the sample size.

Technically, we implemented the (diversified) distance-based sampling strategy on top of the tool SPL Conqueror[5] and compared it with the implementations of t-wise sampling, (randomized) solver-based sampling, and random sampling of SPL Conqueror. t-wise sampling corresponds to the optimized t-wise strategy by Siegmund et al. [140]. For random sampling, SPL Conqueror selects randomly distributed configurations from the whole population, which guarantees a uniform distribution of configurations across the configuration space. That is, for the purpose of computing a baseline (for $\overline{error}$ and $\widetilde{error}$), we follow the non-scalable random sampling: we derive the whole population (i.e., all valid configurations) first, which is necessary to answer RQ_1. Then, we randomly draw configurations from the whole population to the sample set. Other random sampling strategies such as the Monte Carlo method or BDD-based sampling are not suitable, because of the disadvantages mentioned in Section 3.1. This design decision allows us to maximize internal validity, but requires to acquire the whole population. For larger subject systems (except for the subject systems in RQ_3), we spent in total more than a week of measurement per subject system. For (randomized) solver-based sampling, we used the z3 solver [100], which allows us to set a random seed. Specifying different random seeds influences the variable-selection heuristics and, thus, determines the location of the sample set in the configuration space. We performed each sampling for RQ_1 and RQ_2 100 times with different random seeds from 1 to 100; for RQ_3, we performed each sampling 10 times with different random seeds from 1 to 10. The measurements for RQ_1 and RQ_2 were performed on a cluster with an Intel Xeon E5-2690 and 64 GB RAM (Debian 9) and the measurements for RQ_3 were performed on a cluster with an Intel Xeon E5-2630 v4 and 256 GB RAM (Debian 11).

3.2.3 Subject Systems

In our experiments, we consider 10 real-world configurable software systems from different domains and of different sizes. We measured all configurations of all subject systems (i.e., the whole population) between 5 to 10 times until reaching a standard deviation of less than 10%, to control measurement bias. In total, the measurements took multiple years of CPU time. In Table 3.1, we provide an overview of the subject systems. Note, as we needed the whole population for every subject system to perform random sampling, the sizes of configuration spaces of potential subject systems was limited.0 We provide the variability models and the measurements of the subject systems on our supplementary web site. Note that we used one-hot encoding (see Section 2.1.5) to convert the numeric configuration options into multiple binary configuration options, due to the fact that our sampling strategy is currently designed especially for binary configuration options. Next, we describe the subject systems in more detail.

7-Zip (7z) is a file archiver written in C++. Configuration options include different compression methods, different sizes of the dictionary, and the use of single- or multithreading. We used version 9.20 of 7-Zip and measured the compression time of the Canterbury corpus[6] on an Intel Xeon E5-2690 and 64 GB RAM (Ubuntu 16.04).

5 http://www.fosd.de/SPLConqueror/, last accessed on 02/23/2023.
6 https://corpus.canterbury.ac.nz/, last accessed on 02/23/2023.

Table 3.1: Overview of the subject systems for RQ_1 and RQ_2 including domain, number of valid configurations ($|\mathcal{C}|$), number of configuration options ($|\mathcal{O}|$), and the performance metric to be predicted.

	Domain	$\|\mathcal{C}\|$	$\|\mathcal{O}\|$	Performance
7z	File archive utility	68 640	44	Compression time
BDB-C	Embedded database	2 560	18	Response time
Dune	Multigrid solver	2 304	32	Solving time
HIPA^{cc}	Image processing	13 485	54	Solving time
JavaGC	Garbage collector	193 536	39	Time
LLVM	Compiler infrastructure	1 024	11	Compilation time
lrzip	File archive utility	432	19	Compression time
Polly	Code optimizer	60 000	40	Runtime
VP9	Video encoder	216 000	42	Encoding time
x264	Video encoder	1 152	16	Encoding time

BerkeleyDB-C (BDB-C) is an embedded database engine written in C. We consider configuration options defining, for example, the page and cache size or the use of encryption. We measured the time of version 4.4.20 to answer different read and write queries on a machine with an Intel Core 2 Quad CPU 2.66 GHz and 4 GB RAM (Windows Vista).

Dune MGS (Dune) is a geometric multigrid solver for partial differential equations based on the Dune framework [14]. As configuration options, we consider different algorithms for smoothing and different numbers of pre-smoothing and post-smoothing steps to solve Poisson's equation. We performed all measurements with version 2.2 on an Intel i5-4570 and 32 GB RAM (Ubuntu 13.04).

HIPA^{cc} Solver (HIPA^{cc}) is an image processing framework written in C++. We included, for instance, different numbers of pixels calculated per thread and different types of memory (e.g., texture, local) as configuration options. We measured the runtime for solving partial differential equations on an nVidia Tesla K20 with 5 GB RAM and 2 496 cores (Ubuntu 14.04).

JavaGC is the garbage collector of the Java VM, which provides several configuration options, such as disabling the explicit garbage collection call, modifying the adaptive garbage collection boundary, and modifying the policy size. We measured the garbage collection time of Java 1.8 to execute the DaCapo benchmark suite[7] on a cluster with an Intel Xeon E5-2690 and 64 GB RAM (Ubuntu 14.04).

LLVM is a popular compiler infrastructure written in C++. Configuration options that we considered concern code optimization, such as enabling inlining, jump threading, and dead code elimination. We measured the compile time (using the Clang frontend) of version 2.7 for executing the opt-tool benchmark on an AMD Athlon64 Dual Core, 2 GB RAM (Debian GNU/Linux 6).

lrzip is a file compression tool. We consider configuration options that define, for instance, the compression level and the use of encryption. We used the uiq2[8] generator to generate a file (632 MB), and we measured the time for compressing this file with version 0.600 on a machine with AMD Athlon64 Dual Core, 2 GB RAM (Debian GNU/Linux 6).

7 http://dacapobench.sourceforge.net/, last accessed on 02/23/2023.
8 http://mattmahoney.net/dc/uiq/, last accessed on 02/23/2023.

Polly is a loop optimizer that rests on top of LLVM. Polly provides various configuration options that define, for example, whether code should be parallelized or the choice of the tile size. We used Polly version 3.9, LLVM version 4.0.0, and Clang version 4.0.0. As benchmark, we used the `gemm` program from `polybench` and measured its runtime on an Intel Xeon E5-2690 and 64 GB RAM (Ubuntu 16.04).

vpxenc (VP9) is a video encoder that uses the VP9 video coding format. It offers different configuration options, such as adjusting the quality, the bitrate of the coded video, and the number of threads to use. We measured the encoding time of 2 seconds from the Big Buck Bunny trailer on an Intel Xeon E5-2690 and 64 GB RAM (Ubuntu 16.04).

x264 is a video encoder for the H.264 compression format. Relevant configuration options included the number of reference frames, enabling or disabling the default entropy encoder, and the number of frames for ratecontrol and lookahead. We have measured the time to encode the Sintel trailer (734 MB) on an Intel Core Q6600 with 4 GB RAM (Ubuntu 14.04).

In Table 3.2, we show the subject systems for our scalability experiments in RQ_3. The rationale behind using these subject systems is that they are freely available and all of these subject systems use kConfig to describe the configuration options. Although kConfig also allows for numeric configuration options, we focus only on the binary configuration options. For the conversion from kConfig models to feature models, we used kmax[9] in a first step to derive models in conjunctive normal form in the DIMACS[10] format and, in a second step, converted this DIMACS file into a feature model using SPL Conqueror. It is important to note that through this conversion process, the configuration options in the feature model do not exactly correspond to their original configuration options in kConfig. In particular, tristate configuration options (i.e., configuration options with three different assignable values instead of two) are converted into multiple binary configuration options. This is also the reason why the number of configuration options of Linux does not correspond to the numbers of other literature, such as the recently reported 9 000 configuration options by Acher et al. [1].

Table 3.2: Overview of the subject systems for RQ_3 including domain and number of configuration options ($|\mathcal{O}|$).

	Domain	$\lvert\mathcal{O}\rvert$
axTLS	Embedded SSL library	143
Buildroot	Embedded system generator	13 559
BusyBox	Unix toolbox	772
Fiasco	Microkernel	130
Freetz-NG	Router firmware extension	15 921
Linux	Operating system kernel	27 318
toybox	Linux command line	69
uClibc-ng	C library	331

9 `https://github.com/paulgazz/kmax`, last accessed on 07/08/2023.
10 `https://jix.github.io/varisat/manual/0.2.0/formats/dimacs.html`, last accessed on 07/08/2023.

Table 3.3: Error rates of t-wise, (randomized) solver-based, (diversified) distance-based, and random sampling for all 10 subject systems. The bottom row contains the mean value across all subject systems. The best results per subject system and sample set size are highlighted in bold and green iff the Mann-Whitney U test reported a significant difference ($p < 0.05$).

	Coverage-based			Solver-based			Randomized solver-based			Distance-based			Diversified distance-based			Random		
	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$
7z	51.2 %	33.8 %	22.6 %	65.4 %	58.2 %	25.2 %	55.1 %	37.2 %	16.7 %	85.9 %	27.3 %	16.6 %	74.3 %	**16.3 %**	17.2 %	58.2 %	15.1 %	9.9 %
BDB-C	122.9 %	29.0 %	26.5 %	49.5 %	46.8 %	42.0 %	**45.1 %**	46.1 %	18.1 %	320.0 %	75.1 %	15.0 %	237.0 %	**12.7 %**	**9.3 %**	121.3 %	39.1 %	12.2 %
Dune	**15.5 %**	12.5 %	11.4 %	23.6 %	15.1 %	11.8 %	43.3 %	16.8 %	11.2 %	24.4 %	15.2 %	11.4 %	21.5 %	**11.8 %**	**11.0 %**	17.6 %	11.5 %	11.3 %
Hipacc	**26.2 %**	20.5 %	20.5 %	44.8 %	17.2 %	14.7 %	31.9 %	15.7 %	14.2 %	27.9 %	19.0 %	15.3 %	31.5 %	**14.5 %**	**14.0 %**	19.9 %	13.9 %	13.4 %
JavaGC	**36.7 %**	32.1 %	23.7 %	54.2 %	59.3 %	35.8 %	41.9 %	37.8 %	30.2 %	72.9 %	43.8 %	28.2 %	56.0 %	29.9 %	**13.2 %**	55.8 %	13.9 %	12.3 %
LLVM	6.2 %	6.2 %	5.8 %	9.5 %	5.5 %	5.2 %	5.6 %	5.2 %	5.4 %	5.8 %	**5.2 %**	5.3 %	5.9 %	5.3 %	5.2 %	5.6 %	5.2 %	5.2 %
lrzip	**27.2 %**	28.2 %	**13.4 %**	47.3 %	27.3 %	23.9 %	91.5 %	36.0 %	25.0 %	162.5 %	39.7 %	21.9 %	134.2 %	**25.1 %**	18.2 %	62.7 %	18.3 %	15.6 %
Polly	**19.7 %**	12.7 %	**7.3 %**	20.3 %	16.1 %	15.5 %	20.0 %	13.6 %	14.0 %	23.3 %	14.2 %	14.9 %	25.8 %	**10.5 %**	11.8 %	25.1 %	13.0 %	10.3 %
VP9	**100.3 %**	96.3 %	45.3 %	413.0 %	224.2 %	80.8 %	470.2 %	389.1 %	94.5 %	721.9 %	125.0 %	84.5 %	189.8 %	**66.5 %**	**32.0 %**	80.6 %	27.2 %	23.3 %
x264	20.9 %	11.9 %	10.9 %	26.2 %	40.4 %	42.2 %	18.5 %	22.2 %	33.2 %	14.7 %	10.0 %	9.4 %	**12.6 %**	**8.8 %**	**9.0 %**	13.5 %	9.2 %	9.1 %
Mean	**42.7 %**	28.3 %	18.7 %	75.4 %	51.0 %	29.7 %	82.3 %	62.0 %	26.2 %	145.9 %	37.4 %	22.2 %	78.9 %	**20.1 %**	**14.1 %**	46.0 %	16.6 %	12.3 %

3.3 Evaluation

In Section 3.3.1, we present the results regarding RQ_1 and in Section 3.3.2 the results regarding RQ_2. In Section 3.3.4, we discuss further findings, the computation effort, and our optimization. In Section 3.3.5, we discuss threats to validity.

3.3.1 Results RQ_1—Prediction Accuracy

In Table 3.3, we show the mean error rates for the different sampling strategies. We show the results of random sampling in the rightmost column. Again, random sampling requires the computation of the whole population and does not scale, but it serves as a base line for our experiments. We mark for each sample-set size the lowest, statistically significant error rate in green. That is, if two strategies perform similarly and have no statistically significant difference, we do not mark them. Additionally, we provide the mean error rate (bottom row) over all subject systems.

There are several observations: Diversified distance-based sampling performs best or similar to all other sampling strategies for t=2 and t=3. Distance-based sampling without optimization produces partially good results for some systems (e.g., for 7z and LLVM), but is outperformed for other systems (e.g., JavaGC, lrzip, and VP9).

Solver-based sampling results in inaccurate performance-influence models for most subject systems and sample-set sizes. Randomized solver-based sampling performs overall better than solver-based sampling; t-wise sampling perform best when only a very limited number of samples are considered (i.e., t=1).

When we compare the results to random sampling, we make two observations. First, it seems that a diverse coverage of the configuration (by random selection) yields most accurate performance-influence models, especially for systems with many configurations (e.g., 7z, JavaGC, and VP9). Second, we observe that the error rates of diversified distance-based sampling often come close to the base line of random sampling, especially when the size of the sample set increases.

When performing Kruskal-Wallis tests for all sample sizes (t=1, t=2, and t=3), we observe p values less than 0.05 (shown on our supplementary web site), indicating that, at least, two sampling strategies differ significantly for each sample size. To identify these sampling strategies, we apply one-sided Mann-Whitney U tests pair-wisely and, if significant ($p < 0.05$), report the effect sizes in Table 3.4. Specifically, we test whether the sampling strategy of the row in Table 3.4 has a significantly lower error rate than the sampling strategy of the column. The first row shows that t-wise sampling leads to significantly lower error rates than solver-based sampling, with a small effect size for all sample sizes, and to significantly lower error rates than distance-based sampling for t=1. In the fourth row, we see that distance-based sampling leads to lower error rates than t-wise sampling for t=2 and t=3, with a small effect sizes. Distance-based sampling has also lower error rates than solver-based sampling for t=2 and t=3 with a small effect size and t=3 with a medium effect size. Solver-based sampling performs significantly better than distance-based sampling for t=1, which is also negligible

Table 3.4: p values from a one-sided pair-wise Mann-Whitney U test, where we tested pair-wisely whether the population from the row is smaller than the population from the column for different sample sizes after standardization. The effect size is included for every significant result ($p < 0.05$), where we consider differences as small, medium, and large when $\hat{A}_{12}$ is over 0.56, 0.64, and 0.71, respectively.

Mann-Whitney U test [p value ($\hat{A}_{12}$)]	Coverage-based			Solver-based			Randomized solver-based			Distance-based			Diversified distance-based			Random		
	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$
Coverage-based				10^{-25} (0.63)	10^{-12} (0.59)	10^{-09} (0.57)	10^{-11} (0.59)	10^{-04} (0.54)	10^{-06} (0.56)	10^{-55} (0.70)			10^{-28} (0.64)					
Solver-based										10^{-04} (0.54)								
Randomized solver-based				10^{-06} (0.56)	10^{-06} (0.56)	10^{-06} (0.56)				10^{-17} (0.61)			10^{-05} (0.55)					
Distance-based		10^{-07} (0.56)	10^{-11} (0.58)		10^{-19} (0.61)	10^{-30} (0.65)		10^{-10} (0.58)	10^{-25} (0.63)									
Diversified distance-based		10^{-151} (0.84)	10^{-107} (0.78)		10^{-147} (0.83)	10^{-119} (0.80)		10^{-141} (0.83)	10^{-146} (0.83)	10^{-07} (0.57)	10^{-78} (0.74)	10^{-49} (0.69)						
Random	10^{-03} (0.54)	10^{-196} (0.89)	10^{-155} (0.84)	10^{-30} (0.65)	10^{-175} (0.86)	10^{-151} (0.84)	10^{-15} (0.60)	10^{-175} (0.86)	10^{-187} (0.88)	10^{-50} (0.69)	10^{-119} (0.80)	10^{-83} (0.75)	10^{-27} (0.64)	10^{-11} (0.58)	10^{-10} (0.58)			

Table 3.5: p values from a one-sided pair-wise F-test, where we tested pair-wisely whether the variances of the population from the row is smaller than the one from the column, for different sample sizes after standardization.

F-test (p value)	Solver-based			Randomized solver-based			Distance-based			Diversified distance-based			Random		
	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$
Solver-based															
Randomized solver-based	10^{-20}	10^{-46}	10^{-45}				10^{-09}	10^{-16}	10^{-25}						
Distance-based	10^{-04}	10^{-10}	10^{-05}												
Diversified distance-based	10^{-36}	10^{-195}	10^{-114}	10^{-04}	10^{-67}	10^{-21}	10^{-21}	10^{-132}	10^{-81}					10^{-09}	
Random	10^{-100}	10^{-137}	10^{-141}	10^{-37}	10^{-32}	10^{-34}	10^{-74}	10^{-83}	10^{-105}	10^{-21}		10^{-03}			

due to the small effect size. Randomized solver-based sampling performs significantly better than distance-based sampling for t=1 with small effect size.

When comparing the error rates of diversified distance-based sampling with t-wise sampling, randomized solver-based sampling, and solver-based sampling, we see that t-wise sampling, solver-based sampling, and randomized solver-based sampling lead to higher error rates for t=2 and t=3. The effect size in comparison to t-wise sampling is large for diversified distance-based sampling. Moreover, diversified distance-based sampling performs better than solver-based and randomized solver-based sampling with large effect sizes. Comparing diversified distance-based sampling to random sampling, we see that random sampling has significantly lower error rates with small to medium effect sizes. This result indicates that we can reach nearly the same low error rates using distance-based sampling as the computationally intractable random sampling.

Summary of Research Question 1

Diversified distance-based sampling outperforms all other sampling strategies for t=2 and t=3, almost reaching the accuracy of the base line of random sampling, but without relying on the whole population. For small sample sets (t=1), t-wise sampling is superior.

3.3.2 Results RQ$_2$—Robustness

Based on 100 runs per experiment, we obtained a distribution of mean error rates for each sampling strategy, which we further aggregated to compute their variances, $\widetilde{error}$. We compared the variances as follows: First, we performed Levene's test (shown on our supplementary web site), which checks the existence of significantly different variances between, at least, two sampling strategies over all sampling sizes. Then, we performed pair-wisely one-sided F-tests. We show the results in Table 3.5. In the second row, we can see that randomized solver-based sampling has a significantly lower variance than distance-based sampling. In the third row, we can see that distance-based sampling has a significantly lower variance than solver-based sampling on all sample sizes. The last row shows that random sampling has the lowest variance. When it comes to the diversified variant of distance-based sampling, it leads to a significantly lower variance compared to plain distance-based sampling. The optimization has a significantly lower error rate than random sampling for t=2, which requires, however, a whole-population analysis. Regarding randomized solver-based sampling, the variance of diversified distance-based sampling is significantly lower for all sample sizes.

We explain these observations as follows: Solver-based sampling either relies also on a random seed or produces a deterministic set of configurations (not considered here). In the case of random seeds, the seed defines the first solution found by the solver. Since neighboring solutions are produced very likely in subsequent solver calls, the sample set will be locally clustered around the first solution. Hence, for different seeds, different clusters are sampled such that high variations occur in the error rate depending on the representativeness of the cluster. Randomized solver-based sampling avoids building clusters, and thus the variance is significantly lower than solver-based sampling. Distance-based sampling uses also a solver to obtain configurations, but only with a certain distance. Having multiple configurations with the same distance might lead to clusters similar to solver-based sampling.

This is why we observe a lower variance than for solver-based sampling (we might obtain one cluster per distance, but not a single cluster in total), but a higher variance than random sampling, randomized solver-based sampling, and diversified distance-based sampling. Reducing clustering, diversified distance-based sampling yields even lower variances. As diversified distance-based sampling avoids clusters such as randomized solver-based sampling, the variances are rather similar. Hence, we conclude that our optimization of distance-based sampling is effective to increase the variety of configurations and thus lowers the variance of the prediction error.

Summary of Research Question 2

Diversified distance-based sampling is more robust than other sampling strategies except for random sampling, but at the benefit of lower computational effort.

3.3.3 Results RQ_3—Performance

In Table 3.6, we show the performance of distance-based sampling, diversified distance-based sampling, randomized solver-based sampling, and solver-based sampling on larger configurable software systems. We mark for each sample-set size the lowest, statistically significant performance in green. Note, however, that these performance values depend heavily on the constraint solver to search for configurations. Exchanging the constraint solver would result in other performance values. There are several observations: Clearly, using an off-the-shelf solver to find any arbitrary configuration outperforms the other sampling strategies by far when using z3. Distance-based sampling and diversified distance-based sampling run into a timeout in all sample sizes for 5 out of 8 subject systems and have difficulties beginning with 331 configuration options. The randomized solver-based sampling approach, where the constraint solver has to be reinitialized for each configuration, runs less often into a timeout sample sizes than distance-based and diversified distance-based sampling.

For completeness, we also show the performance values of the subject systems used in RQ_1 and RQ_2 in Table 3.7. In these smaller configurable systems, the picture changes slightly. We observe that in the configurable systems in RQ_1 and RQ_2, the distance-based sampling and diversified distance-based sampling both outperform randomized solver-based sampling.

Summary of Research Question 3

Distance-based sampling and diversified distance-based sampling currently do not scale well on large feature models using z3. Only solver-based sampling finds samples for each feature model in reasonable time, even for the Linux kernel.

3.3.4 Discussion

Computational effort: The computational effort of distance-based sampling and its diversified variant is lower than the effort of random sampling because we include every configuration found by the solver into the sample set instead of enumerating all valid configurations and discarding later a large part of it (i.e., configurations not used in the sample set—see

Table 3.6: Performance of distance-based sampling, diversified distance-based sampling, randomized solver-based sampling, and solver-based sampling on large subject systems for sampling 10, 100, or 1 000 configurations, respectively. >3h means that the sampling strategy ran into a timeout of 3 hours.

	Distance-based			Diversified distance-based			Randomized solver-based			Solver-based		
	10	100	1 000	10	100	1 000	10	100	1 000	10	100	1 000
axTLS	80s	4m 01s	4m 03s	2m 42s	34m 31s	> 3h	10s	15m 37s	> 3h	< 1s	< 1s	2s
Buildroot	> 3h	> 3h	> 3h	> 3h	> 3h	> 3h	> 3h	> 3h	> 3h	7s	49s	8m 25s
BusyBox	> 3h	> 3h	> 3h	> 3h	> 3h	> 3h	4m 55s	> 3h	> 3h	< 1s	< 1s	8s
Fiasco	21m 49s	> 3h	> 3h	21m 47s	> 3h	> 3h	8s	13m 00s	> 3h	< 1s	< 1s	2s
Freetz-ng	> 3h	> 3h	> 3h	> 3h	> 3h	> 3h	> 3h	> 3h	> 3h	11s	67s	11m 24s
Linux	> 3h	> 3h	> 3h	> 3h	> 3h	> 3h	> 3h	> 3h	> 3h	21s	2m 46s	28m 02s
toybox	< 1s	1s	2s	< 1s	1s	2s	2s	3m 41s	> 3h	< 1s	< 1s	< 1s
uClibc-ng	> 3h	> 3h	> 3h	> 3h	> 3h	> 3h	53s	1h 23m	> 3h	< 1s	< 1s	4s

Table 3.7: Performance of distance-based sampling, diversified distance-based sampling, randomized solver-based sampling, and solver-based sampling on the other subject systems used for RQ_1 and RQ_2.

	Coverage-based			Solver-based			Randomized solver-based			Distance-based			Diversified distance-based			Random
	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	WP
7z	< 1s	3s	43s	< 1s	< 1s	11s	5s	17m 32s	> 3h	10s	11s	15s	11s	12s	17s	47m 34s
BerkeleyDBC	< 1s	< 1s	2s	< 1s	< 1s	< 1s	< 1s	6s	77s	< 1s	< 1s	< 1s	< 1s	< 1s	< 1s	3s
Dune	< 1s	1s	11s	< 1s	< 1s	< 1s	2s	109s	29m 24s	< 1s	< 1s	1s	< 1s	< 1s	1s	4s
Hipacc	< 1s	7s	82s	< 1s	< 1s	13s	11s	39m 50s	> 3h	2s	3s	8s	3s	3s	9s	109s
JavaGC	< 1s	3s	40s	< 1s	1s	9s	4s	8m 41s	> 3h	92s	93s	95s	92s	93s	94s	> 3h
LLVM	< 1s	< 1s	< 1s	< 1s	< 1s	< 1s	< 1s	1s	8s	< 1s	< 1s	< 1s	< 1s	< 1s	< 1s	< 1s
Polly	< 1s	3s	41s	< 1s	< 1s	4s	2s	4m 50s	> 3h	13s	14s	16s	15s	16s	21s	43m 44s
VP9	< 1s	5s	52s	< 1s	< 1s	11s	4s	10m 54s	> 3h	2m 24s	2m 25s	2m 27s	2m 25s	2m 26s	2m 30s	> 3h
lrzip	< 1s	< 1s	2s	< 1s	< 1s	< 1s	< 1s	6s	22s	< 1s	< 1s	< 1s	< 1s	< 1s	< 1s	1s
x264	< 1s	< 1s	2s	< 1s	< 1s	< 1s	< 1s	2s	22s	< 1s	< 1s	< 1s	< 1s	< 1s	< 1s	< 1s

Section 3.1). Moreover, to reduce computational effort, we do not perform an expensive optimization as in t-wise sampling (i.e., minimizing the set of selected configuration options that are not of interest in the current configuration), but rather include additional constraints to the constraint solver that define, for example, the number of selected configuration options. However, as we observed in the evaluation on subject systems with large configuration spaces, the constraint to find any configuration with exactly d configuration options selected, becomes difficult to process for z3. Clearly, exchanging the constraint solver with another solver that is designed to process such constraints fast, would also change the picture. The computational effort of randomized solver-based sampling is high, since the solver has to be reinitialized from scratch to permute (1) the constraints, (2) the literals, and (3) initial assignment. Interestingly, distance-based sampling performs better than randomized solver-based sampling on the subject systems of RQ_1 and RQ_2 (see Table 3.7), but worse on the subject systems with large configuration spaces (see Table 3.2). This is because of the lower number of configuration options of the subject systems, where the overhead of reinitializing the constraint solver for each configuration exceeds the overhead of searching for a configuration with a distance d.

Probability distributions: In our experiments, we used exclusively the discrete uniform distribution for selecting distances, but our algorithm can be parameterized (*probabilityDistr*). Preliminary experiments with binomial and geometric distributions suggest that a uniform coverage of the configuration space is superior, though (see the supplementary web site).

Diversity: Comparing plain distance-based sampling with diversified distance-based sampling, we observe that optimizing for diversity indeed pays off in terms of prediction accuracy and robustness. Aiming at diversity is optimal in a black-box strategy, where no domain knowledge is available. However, diversified distance-based sampling only pays off with larger sample sizes, because the smaller sample sizes do not suffice to cover all configuration options, at least, once for each distance.

3.3.5 Threats to Validity

Internal validity: To rule out errors in our implementation of (diversified) distance-based sampling, we have thoroughly tested it. We verified that the produced sample set follows the given distribution of the configuration distances. We found deviations only when all configurations of a specific distance were already selected, which occurred only in few cases.

External validity: To increase external validity, we have selected software systems from different domains. We consider software systems ranging from systems with 432 configurations to systems with 216 000 configurations. We have excluded larger systems because it would be computationally infeasible for t-wise sampling and random sampling [95]. However, the study of Pereira et al. [118] confirms a part of our results on the prediction error. In detail, the study by Pereira et al. applied different sampling strategies on x264 to study how different sampling strategies perform for different non-functional properties such as performance prediction. In this study, distance-based sampling performed well for performance prediction, but not for predicting the encoding size of a file.

The selection of the machine-learning technique to learn a performance-influence model may threaten external validity. We used deliberately the same machine-learning technique for all experiments to increase internal validity. But, other machine-learning techniques have other strategies to derive information from the sample set and, thus, may lead to different results. In a parallel line of experiments presented in Section 2.2.4, we compared 6 different machine-learning techniques and observed that multiple regression is often as accurate as classification and regression trees and random forests[11], which are often used for learning performance-influence models of configurable software systems. So, we are confident that our results generalize to other machine-learning techniques.

Another threat to validity is the choice of the constraint solver, as different solvers adopt different search heuristics and, thus, performs differently in terms of prediction error and execution time. This, however, might represent a further reason not to rely on solver-based sampling, as this way the generation of the sample set remains intransparent. For instance, we observed even worse results for solver-based sampling when using the solver of the Microsoft solver foundation.

3.4 Summary

Measuring every configuration of a software system to identify the performance-optimal configuration is often unfeasible due to the sheer size of the configuration space. Addressing this problem, machine learning based on statistical learning is used to predict the performance of individual (or all) configurations by deriving information from a small and representative sample set. Finding a tractably small and representative set of configurations is an important but difficult task. To this end, different sampling strategies, such as t-wise sampling, solver-based sampling, and random sampling have been proposed, which focus on different aspects with different strengths and weaknesses. To address the weaknesses, we propose distance-based sampling, which is based on a user-defined discrete probability distribution and a distance metric. The key idea is that distance-based sampling spreads the selected configurations across the configuration space based on a given probability distribution while not relying on an expensive analysis of the whole population. To compare distance-based sampling with the state of the art, we learn performance-influence models for 10 real-world software systems using 6 different sampling strategies and compare the accuracy of the performance-influence models. Furthermore, we apply different sampling strategies on 8 subject systems with large configuration spaces to assess the execution time of the sampling strategies.

Our results demonstrate that distance-based sampling, when used in combination with a diversity optimization, leads to significantly lower error rates than state-of-the-art strategies, especially for larger sample sizes (t=2, t=3), and the predictions are more stable than solver-based sampling with respect to multiple runs using different random seeds. Our results demonstrate that, based on a distance metric and a probability distribution, we can effectively sample diverse configurations across the configuration space and without the need for a whole-population analysis, which makes random sampling unfeasible for highly configurable software systems. Further, distance-based sampling performs well on smaller subject systems, but runs into a timeout of 3 hours for all subject systems with more than 150 configuration options. Exchanging the constraint solver could change this picture, but is an avenue for

11 `https://github.com/se-passau/Distance-Based_Data/`, last accessed on 02/23/2023.

further work. This work provides a new view on sampling based on probability distributions and paves the way for further research in this area. For instance, using other metrics or distributions could lead more accurate predictions or improve the prediction robustness.

4 Performance Evolution of Configurable Software Systems: An Empirical Study

This chapter shares material with the following publication: Christian Kaltenecker, Stefan Mühlbauer, Alexander Grebhahn, Norbert Siegmund, and Sven Apel. "Performance Evolution of Configurable Software Systems: An Empirical Study." In: *Empirical Software Engineering*, 28.6 (2023), 152:1–152:41 [61]

One of our goals in this thesis is to identify performance changes in the evolution of configurable software systems. A *performance change* refers to a situation in which the execution time (or another property such as throughput) of a software system degrades (performance regression) or improves (performance fix or performance optimization) compared to previous releases.

There is a substantial corpus of previous work on analyzing, detecting, and reverting performance changes [15, 21, 43, 101], considering only a single or few default configurations across multiple releases of the software. However, performance changes may be *configuration-dependent*, that is, they appear only in a subset of configurations of the system in question [44]. As such, configuration-dependent changes could be easily missed by considering only the default configuration. Given that contemporary software systems are often configurable [44], this calls for investigating performance changes not only across *multiple releases*, but simultaneously across *multiple configurations*.

So far, there is no clear picture of how severe and frequent performance changes are in configurable software systems and whether individual configurations or configuration options play a central role in the evolution of a system's performance behavior. A systematic analysis of performance changes of configurable software systems holds the promise of providing insights beyond just studying default configurations or average performance behavior. Developers and users are interested in which specific configurations exhibit diverging performance behavior and which configuration options (or interactions among options) are responsible for this. At a conceptual level, insights on the nature and prevalence of configuration-dependent performance changes can be used to improve configuration sampling and performance modeling techniques, where only a representative subset of all software configurations is used for performance prediction [54, 60, 119, 139].

To learn about performance changes in configurable software systems, we conduct an empirical study on the performance evolution of 12 popular configurable open-source software systems from different domains across multiple releases and covering the entire configuration space. To pin down the performance changes to configuration options, we make use of the structure of performance-influence models (obtained by machine learning—see Section 2.2.4).

In particular, we address the following research questions:

- *$RQ_{1.1}$: What is the fraction of the configuration space containing performance changes between consecutive releases?*
- *$RQ_{1.2}$: How stable is the relative performance of configurations in the presence of performance changes between consecutive releases?*
- *$RQ_{2.1}$: How frequent and how strong are changes of performance influences of individual configuration options and interactions between consecutive releases?*
- *$RQ_{2.2}$: How stable is the relative influence of configuration options and interactions in the presence of performance changes between consecutive releases?*

To answer these research questions, we examine the prevalence and properties of performance changes at two levels of abstraction:

- *Configuration-level*: performance of individual configurations.
- *Option-level*: performance influence of individual configuration options and interactions.

Thereby, $RQ_{1.1}$ and $RQ_{1.2}$ are related to the configuration level, whereas $RQ_{2.1}$ and $RQ_{2.2}$ are related to the option level. In a deeper analysis, we contrast the performance change information from both levels to the change log and commit messages of the respective software systems.

Overall, we make the following contributions:

- A novel approach to use performance-influence models to identify performance changes in specific configuration options.
- An empirical study of 12 popular configurable software systems involving their complete configuration spaces for a series of releases considering up to 11 years of evolution.
- Insights on what role configurability plays in performance evolution of configurable software systems, which (kinds of) options and interactions cause performance changes, and which performance changes are documented.

In a nutshell, we found that almost all 190 releases that we analyzed exhibit, at least, one performance change in, at least, one configuration. Most performance changes (75%) affect less than half of the configurations of a system, and most of the performance changes (91%) affect multiple options (up to 6). Notably, despite the prevalence of performance changes, the performance ranking of configurations and influences of individual options are in many cases not affected. That is, developers and users can assume a certain stability of configuration-dependent performance behavior. About 43% of the performance changes are documented in change logs and 64% in commit messages. Specific configuration options were mentioned in 67% of the cases.

Our results have direct implications for configuration sampling, performance modeling, and transfer learning in the area of configurable software systems. That is, for instance, some performance changes affect only 1% of the configurations and demand for comprehensive performance measurements to spot performance changes. Additionally, we found that the relative influence of configuration options and interactions on the performance is stable in 80% of the releases. That is, performance engineers can assume a certain stability also on the options' influences while performing transfer learning across different releases [54]. A deeper analysis of change logs and commit messages shows that using a configuration-aware performance testing pipeline could help in identifying configuration-specific performance changes early. Our measurement and analysis framework provides a solid foundation for further experiments on different software systems and non-functional properties. All results along with analysis scripts and further information are available at a supplementary web site[1] .

4.1 Related Work

In this section, we discuss related work with respect to (1) the role and evolution of software performance, (2) methods to analyze the performance changes, and (3) the evolution of software configurability.

Performance & Software Evolution Root causes of performance changes and their effect on maintainability have been studied before. Zaman et al. conducted an analysis of over 400 bugs from Mozilla Firefox and Google Chrome [155]. They found that performance bugs often require more effort to fix and, therefore, are more costly than fixing functional bugs. A study on Mozilla Firefox, Apache, and MySQL found a strong relation between configurability and performance: 113 out of 193 bugs were configuration related [44].

Alcocer et al. studied the performance evolution of 19 software systems' releases. By analyzing the performance of multiple benchmarks, they found that one third of releases introduced performance bugs. The authors identified 9 patterns for performance changes [3], which include performance improvements, due to removing redundant method calls or caching, as well as performance regressions arising from the composition of collection operations. Our work links both research directions—software configuration and software evolution—and explores performance of software systems across their configuration spaces and along their development histories.

Performance Change Detection The detection of performance changes has been approached from different angles, such as using different statistical methods, and taking one or more performance characteristics of the software system into account. For example, statistical process control charts were used to capture changes of an observed metric, such as the performance of the system, and provide thresholds, which, when exceeded by accumulated change, indicate a performance degradation [76, 90, 111]. Other statistical approaches rely on testing and determining whether two observations are statistically different. For example, Heger et al. compare the performance distributions for different releases with ANOVA [45]. Reichelt et al. apply different statistical tests to identify performance anomalies from performance histories [122].

1 `https://github.com/se-passau/Distance-Based_Data/tree/dissertation/`, last accessed on 07/20/2023.

Aside from considering only a single performance measure, previous work considers multiple measures and their relations. Foo et al. mine repositories regarding performance regression tests and automatically detect performance changes by tracking the correlation of performance measures over time [31]. Malik et al. analyze performance regression by automatically selecting a subset of performance measures that describe system performance [90]. Using principal component analysis, they correlate the measures to obtain a performance fingerprint, which then can be compared across releases.

All this work illustrates that performance changes can manifest in many ways. However, it does not consider configurability and to what extent individual configuration options or interactions cause performance changes, which is the focus of this paper.

Evolution: Configurability & Performance Mühlbauer et al. devised a prediction technique for performance changes in software repositories, across releases and configurations [107]. This work is the closest but complementary to ours: While we study the prevalence and properties of performance feature interactions in the wild, they propose a technique to discover them with little effort. In principle, we could have used their technique to collect the data for our study. But, as their approach only approximates performance changes with iterative sampling, we analyze the configuration space as a whole for accuracy.

Several studies have observed and categorized recurring patterns in the evolution of variability models, such as the introduction or removal of new configuration options (often called *features*) or splitting generic options into more precise ones. There are three relevant patterns: a new feature is added, a mandatory feature becomes optional, or a mandatory/optional feature is split into alternative features [115–117, 132]. Our study considers only configuration options that exist in all releases of the software system, which is the majority, though. However, for the interpretation of our results (see $RQ_{2.2}$), these patterns provide some context that can help map changes in the performance influence across releases. Recent work by Jamshidi et al. explores the applicability of transfer learning to adapt performance-influence models to different environments [54]. Their key insight, after investigating 4 configurable software systems, is that only a subset of configuration options and interactions among them have a strong influence on performance and that the performance influence is generally preserved across environments and software releases.

Workload Dependence Clearly, the performance of a software system may change depending on the workload. There is a substantial corpus of work studying this phenomenon and providing models and solutions that incorporate workload-dependent performance [30, 83]. The work of Costa et al. and Leitner et al. focuses on studying and improving performance tests in Java-based open source projects [26, 78]. Our work is complementary in that we study system configurability, which is a further dimension that influences a system's performance. To increase internal validity, we fixed the workload per system in our experiments. Ultimately, our approach and previous work on workload-dependent performance behavior shall be combined.

4.2 Software Evolution

Software systems must evolve constantly to adapt to changes of hardware and user requirements [152]. Software evolution is driven by the integration of new functionality or libraries, refactoring, and bug fixes. Besides functionality, the performance of the system may change considerably.

Version control systems help developers to keep track of code changes that arise during software evolution. For this purpose, most version control systems provide the concept of revisions. A *revision* is effectively a view on the code base at a certain point in time. In what follows, $\mathcal{RV}$ denotes the set of revisions of a software system. To highlight revisions that (1) contain prominent changes, (2) are assumed as running stable, or (3) mark major milestones, a revision can be tagged as *release*, with $\mathcal{R} \subseteq \mathcal{RV}$ denoting the set of releases. In our study, we consider only releases (1) to focus on important revisions, (2) to keep measurement effort feasible, and (3) releases are usually the revisions that are used in production. Intermediate revisions are not guaranteed to compile/run without errors since those revisions typically are incremental modifications and „work in progress“. The rationale behind this is that older software releases do not compile and run anymore on current operating systems, which limits the time span that we can observe. Furthermore, we do not measure each minor release in each software system since measuring each release would require to measure all configurations of the configuration space again. In this case, we opted to distribute the releases in similar time frames (e.g., one release per half year) to cover each time frame equally.

***Algorithm 4:** Learning a performance-influence model*

```
1  Function learn_comparable_models(feature_model, releases, release_performance_data):
2      terms ← ∅
3      foreach release ∈ releases do
4          model ← learn_model(feature_model, release_performance_data[release])
5          terms ← include_terms_from_model(terms, model)
6      end
7      terms ← variance_factor_analysis(terms)
8      models ← ∅
9      foreach release ∈ releases do
10         model ← fit(terms, release_performance_data[release])
11         models ← models ∪ {model}
12     end
13     return models
```

For the option-level of our study, we learn one performance-influence model for each release. These performance-influence models, however, have the caveat that they cannot be used for comparison between different releases because of multicollinearity (see Section 2.2.4). To mitigate this, we follow the approach of Algorithm 4 to bring the performance-influence models into a comparable form (i.e., all performance-influence models contain the same terms). In Line 2–6, we learn a performance-influence model (see multiple linear regression in Section 2.2.4) for each release. This is necessary to identify the performance-relevant configuration options and interactions. These configuration options and interactions are included as *terms* into the model in Line 5. This way, we obtain a set containing all relevant configuration options and interactions among them. However, this set cannot be immediately used as a performance-influence model since this step includes configuration options and interactions that might be multicollinear. Hence, we remove multicollinear terms by applying a VIF analysis [28] in Line 7. Note that this does not affect our prediction error since we remove only perfectly multicollinear terms (i.e., terms that are completely interchangeable). After this step, we use the same terms and fit them for each release in Line 10. These performance-influence models contain the same configuration options and interactions and can now be compared.

4.3 Experiment Setup

In this section, we discuss our research questions and how we attempt to answer them. In particular, we discuss and motivate our research questions in Section 4.3.1. Later, in Section 4.3.2, we present 12 subject systems we use for our study and discuss the used workloads in Section 4.3.3. Last, we describe in detail how we evaluate our research questions in Section 4.3.4.

4.3.1 Research Questions

Our overarching goal is to understand the performance evolution of configurable software systems. To this end, we study the characteristics of performance changes and their relation to configurability. For a detailed analysis, we consider two levels of abstraction: *configuration level* and *option level*.

Configuration level As a first approximation, we address our goal at the level of individual configurations. In particular, we are interested in (1) whether performance changes affect typically many or only a few configurations and (2) whether performance changes alter typically the overall ranking of configurations with regard to their performance optimality.

For the first research question ($RQ_{1.1}$), we compare for each pair of releases each configuration with its successor in terms of the extent to which the performance has changed. This will allow us to make quantitative statements about how many performance changes exist in practice and what fractions and kinds of configurations are affected. These insights can inform sampling strategies and maintenance activities by prioritizing specific configurations that likely exhibit performance changes.

Research Question 1.1

What is the fraction of the configuration space containing performance changes between consecutive releases?

For the second research question ($RQ_{1.2}$), we analyze to what extent performance changes affect the ranking of configurations with regard to their performance. That is, the slowest configuration has the lowest rank, the fastest configuration the highest rank, etc. Often developers and users are less interested in the actual performance values, but rather in their *relative importance*, including which configurations are performance-optimal and which fall below a certain threshold [109]. It might be that performance changes exist but that most of them do not alter the performance ranking of configurations. That is, the *performance ranking* of configurations is *stable*. This would be useful for researchers (e.g., for transfer learning of performance models [54, 55]) and practitioners (so they can rely on a certain stability in the relative performance influences).

Research Question 1.2

How stable is the relative performance of configurations in the presence of performance changes between consecutive releases?

Option level Besides knowing which configurations are affected by a performance change, we would like to know which configuration options or interactions among options are responsible for this change. As with configurations, we are interested in (1) whether typically many or only few options or interactions cause performance changes and (2) whether performance changes alter typically the overall ranking of performance influences of options and interaction. To obtain information on the influences of options and their interactions, we learn a performance-influence model per release and compare their terms and coefficients (see Chapter 2). Since we use linear regression to learn our performance models, multicollinearity might occur between multiple terms (see Section 2.2.4). As a countermeasure, we apply a VIF analysis and remove all terms causing perfect multicollinearity. By doing so, 14 out of 702 terms were removed leaving the predictions of our performance-influence models unaffected.

For the first research question ($RQ_{2.1}$), we compare for each pair of releases each influence of each model term with its successor regarding the extent to which its influence has changed. This will allow us to make quantitative statements about how many options and interactions are responsible for performance changes. Knowing whether many or only few options are responsible for performance changes helps to understand root causes of these changes and to guide corresponding actions. Identifying patterns here can inform performance engineers to guide and improve the detection and tracing of performance bottlenecks [32]. Comparing each pair of releases further gives us the opportunity to assess the distribution of relative influences of the configuration options on performance (i.e., all options have a similar influence on performance, or a few influence performance the most).

Research Question 2.1

How frequent and how strong are changes of performance influences of individual configuration options and interactions between consecutive releases?

For the second research question ($RQ_{2.2}$), we analyze to what extent performance changes affect the global ranking of performance influences of configuration options and interactions. As with configurations, it is often sufficient to know which configuration options have a strong influence on performance without knowing exact performance values. For instance, when optimizing for performance, a user may concentrate on the configuration options having a strong influence on performance and ignore others [152]. When optimizing for performance in a compression software, the performance-influence model might point out to consider low instead of high compression levels and to neglect debug options. For a developer, it might be interesting to confirm own expectations of how configuration options perform, as shown in a former study [37].

Research Question 2.2

How stable is the relative influence of configuration options and interactions in the presence of performance changes between consecutive releases?

Table 4.1: Overview of the subject systems, including application domain, number of thousands of lines of code (KLOC) in the last measured release, number of valid configurations ($|\mathcal{C}|$) in each release, configuration options ($|\mathcal{O}|$), releases ($|\mathcal{R}|$), and performance metric.

Name	Domain	KLOC	$\vert\mathcal{C}\vert$	$\vert\mathcal{O}\vert$	$\vert\mathcal{R}\vert$	Performance
brotli	Compression	30	181	30	12	Compression time
FastDownward	Planning system	90	374	39	9	Solving time
HSQLDB	Database	194	864	29	19	Response time
lrzip	Compression	16	1 440	27	22	Compression time
MariaDB	Database	1 969	972	21	22	Response time
MySQL	Database	2 792	972	21	20	Response time
OpenVPN	VPN software	80	512	24	12	Response time
opus	Audio encoder	54	6 480	31	12	Encoding time
PostgreSQL	Database	1 160	864	18	22	Response time
VP8	Video encoder	324	2 736	27	15	Encoding time
VP9	Video encoder	324	3 008	25	7	Encoding time
z3	Constraint solver	415	1 024	13	18	Solving time

4.3.2 Subject Systems

For our experiments, we selected 12 real-world configurable software systems based on the following criteria: (1) different sizes (number of configurations and configuration options) to evaluate scalability, (2) different application domains to increase external validity, (3) different application architectures (e.g., client-server vs. desktop) to cover different performance aspects, and (4) actively maintained systems to detect historical changes in a realistic context; see Table 4.1, for an overview. As of 2023, all systems in our selection are actively maintained, and we consider lifetimes of 21 months (PostgreSQL) to 137 months (OpenVPN). From the respective development histories, we extracted all releases, which we identified based on Git tags and respective documentation. All considered configuration options represent run-time configuration options. We provide all variability models, selected releases, measurements, results from our deeper analysis, and a complete description of the configuration options on our supplementary web site. It is important to note that we performed the performance measurements on multiple machines in parallel to keep the measurement time manageable. While we use different machines across different subject systems, we use equally equipped machines for the measurements in each subject system. Parallelizing our performance measurements this way was possible, since we only compare revisions and configurations in subject systems and not across subject systems.

Brotli is an open-source file compression tool by Google written in C. We considered 30 configuration options that give rise to 181 configurations, including configuration options setting the window size and compression level. We used uiq2[2] to generate a general workload for compression (see Section 4.3.3 for more detail). As performance measure, we used compression time. The measurements took place on machines with Intel Core i7-4790 CPUs at 3.60 GHz with 16 GiB RAM (Debian 9). Overall, we considered 12 releases, from release 0.3.0 to 1.0.7, covering almost 3 years of history.

FastDownward is an open-source domain-independent planning system for optimization. To identify performance-relevant configuration options and a proper workload, we contacted a domain expert. Based on the feedback, we considered 39 configuration options that give rise to 374 configurations. 7 out of 39 configuration options control different search heuristics; all other configuration options represent parameters for these heuristics. Here, we mainly consider different heuristics to solve the planning task. Each heuristic comes with its own parameters (i.e., configuration options). We measured the time to find an optimal solution for the planning task. All measurements were conducted on machines with Intel Xeon E5-2630 v4 at 2.20 GHz with 256 GiB RAM (Debian 11). Overall, we considered 9 revisions chosen in cooperation with the domain expert. In total, we cover 5 years of history.

HSQLDB is a lightweight database engine. We considered 29 configuration options that give rise to 864 configurations. Configuration options include support for different encryption algorithms, transaction control settings, and incremental backup. We measured throughput with the benchmarking tool PolePosition[3]. We have used multiple thousands of read, insert, and update queries. We also considered nested queries. The tool emulates realistic user interaction by performing a number of insertions, deletions, updates, and queries. All measurements were conducted on machines with Intel Core i5-4590 CPUs at 3.30 GHz with 16 GiB RAM (Debian 9). Overall, we considered 19 releases, from release 2.1.0 to 2.4.1, covering over 7 years of history.

lrzip is an open-source file compression tool. We considered 27 configuration options that give rise to 1 440 configurations. Relevant configuration options are, for instance, different compression algorithms, compression levels, and processor numbers. We used the same setup as for Brotli. All measurements were conducted on machines with Intel Xeon E5-2650v2 CPUs at 2.60 GHz with 128 GiB RAM (Debian 10). Overall, we considered 22 releases, from release 0.530 to 0.631, covering almost 6 years of history.

MariaDB and MySQL are open-source relational database management systems. For both subject systems, we considered 21 configuration options that give rise to 972 configurations. Among others, we included different buffer pool sizes, table sizes, and flush methods. We measured throughput with the benchmarking tool PolePosition. All measurements were conducted on machines with Intel Xeon E5-2650v2 CPUs at 2.60 GHz with 128 GiB RAM (Debian 10). For MariaDB, we considered 22 releases, from release 5.5.23 to 10.4.7, covering over 7 years of history. For MySQL, we considered 20 releases, from release 5.6.10 to 8.0.17, covering over 6 years of history.

OpenVPN is an open-source software that provides secure communication between computers using virtual private networks. We considered 24 configuration options that give rise to 512 configurations. We included, for instance, support for compression, different encryption

2 `http://mattmahoney.net/dc/uiq/`, last accessed on 02/18/2023.
3 `http://polepos.org`, last accessed on 02/18/2023.

ciphers, authentication methods, and renegotiation settings. We set up an experiment with one client and one server exchanging files to measure the throughput of the application. All measurements of OpenVPN were conducted on machines with Intel Xeon E5-2650v2 CPUs at 2.60 GHz with 128 GiB RAM (Debian 10). Overall, we considered 22 releases, from release 2.1.0 to 2.4.6, covering over 11 years of history.

opus is a codec for lossy audio compression. We considered 31 configuration options, giving rise to 6 480 configurations. Configuration options include choices of bit rates, sample rates, and numbers of channels. We measured the performance of opus by repeatedly encoding a test vector, which has been used to validate the implementation against opus's file format specification. All measurements were conducted on machines with Intel Xeon E5-2620v4 CPUs at 2.10 GHz with 256 GiB RAM (Debian 10). Overall, we considered 12 releases, from release 1.0.0 to 1.3.1, covering almost 7 years of history.

PostgreSQL is an open-source relational database management system. We considered 18 configuration options that give rise to 864 configurations. As configuration options, we include synchronous commits as well as different sizes of buffers and working memory. As with HSQLDB, we used the benchmarking tool PolePosition for measurements. All measurements were conducted with machines with Intel Xeon E5-2650v2 CPUs at 2.60 GHz with 128 GiB RAM (Debian 9). Overall, we considered 22 releases, from release 9.6.3 to 11.2, covering almost 2 years of history.

vpxenc (vp8/vp9) is a video encoder that can be customized with different codecs, of which we study vp8 and vp9. We considered 27 and 25 configuration options that give rise to 2 736 and 3 008 configurations for vp8 and vp9, respectively. vpxenc provides a variety of configuration options, for instance, to adjust the quality or bitrate of the encoded video and multithreading operation. We used the raw trailer from the movie "Sintel" (480p, y4m format) as a benchmark and measured the encoding time of both codecs, respectively. vp8 was measured on machines with Intel Core i5-4590 CPUs at 3.30 GHz with 16 GiB RAM. vp9 was measured on machines with Intel Xeon E5-2650v2 CPUs at 2.60 GHz with 128 GiB RAM (Debian 10). For vp8, we considered 15 releases, from release 0.9.1 to 1.8.0, covering almost 9 years of history. For vp9, we considered 7 releases, from release 1.3.0 to 1.8.0, covering over 5 years of history.

z3 is an open-source SMT solver from Microsoft Research. We considered 13 configuration options that give rise to 1 024 configurations. Configuration options include the generation of proofs, model validation, and model simplification. As a benchmark, we selected four scenarios from the International SMT Competition (LRA, QF_FP, QF_LRA, and QF_UFLRA). We measured and report the execution time for solving these tasks. z3 was measured on machines with Intel Core i5-4590 CPUs at 3.30 GHz with 16 GiB RAM (Debian 11). Overall, we considered 18 releases, from release 4.3.2 to 4.8.13, covering more than 7 years of history.

4.3.3 Workloads

For each subject system, we selected one benchmark originated by the respective system developers or community to obtain a representative workload, this way, increasing external validity (see Section 4.4.4 and Section 4.1).

Audio Encoding (Opus): For the audio encoding, we used test vectors provided by the developers of Opus[4]. Test vectors are designed to test all aspects of the implementation of the audio encoder.

Compression (brotli/lrzip): We used the tool uiq2[5] to generate a large text compression workload. It creates a generic and general purpose compression workload of a specified size. The generated data was the same for both subject systems and has a size of about 100 MB.

Database (HSQLDB/MariaDB/MySQL/PostgreSQL): Each of the database systems supports SQL queries. We used the SQL benchmark PolePosition[6], which was also used in multiple publications [121, 157]. The benchmark enables us to generate different types of queries, such as SELECT, UPDATE, nested queries, and complex queries.

Planning System (FastDownward): We applied the workload data-network-opt18-strips/p05[7] that was suggested by an experienced user of FastDownward as a general workload. In addition, this workload does not contain specific characteristics that make the benchmark unsolvable for certain heuristics.

Solver (z3): We selected multiple benchmarks from the Satisfiability Modulo Theories Library[8] having different types of logics LRA, QF_FP, QF_LRA, and QF_UFLRA. These benchmarks cover floating point, linear real arithmetic, free sort and function symbols, formulas with and without quantifier, and satisfiable and unsatisfiable formulas, thus, covering a large range of options provided by z3.

Video Encoding (VP8/VP9): We used the Sintel trailer as a well-established workload when assessing the quality of different encoders. The Sintel trailer is listed in the Xiph repository[9] and has been used in different publications [118, 131].

VPN (OpenVPN): Similar to compression, we created a generic general purpose file using uiq2 with a size of 1 400 MB. We opted for uiq2 since it generates compression workloads for the lzo compression, which is a functionality enabled by an option in OpenVPN. We adjusted the size of the file as suggested by a community guide for performance testing[10].

4.3.4 Operationalization

To answer our research questions, for each release, (1) we measured all configurations of a subject system and (2) learn a performance-influence model on the entire set of configurations, resulting in one model per system and release. $\mathcal{S}$ refers to the set of subject systems. For a system $s \in S$, $\mathcal{C}_s$ refers to its set of configurations (see Section 2.1) and $\mathcal{R}_s$ to its set of releases.

4 https://opus-codec.org/docs/opus_testvectors-rfc8251.tar.gz, last accessed on 02/18/2023.
5 http://mattmahoney.net/dc/uiq/, last accessed on 02/18/2023.
6 http://polepos.org/, last accessed on 02/18/2023.
7 https://github.com/aibasel/downward-benchmarks/blob/master/data-network-opt18-strips/p05.pddl, last accessed on 02/18/2023.
8 https://smtlib.cs.uiowa.edu/benchmarks.shtml, last accessed on 02/18/2023.
9 https://media.xiph.org/, last accessed on 02/18/2023.
10 https://community.openvpn.net/openvpn/wiki/PerformanceTesting#Testcases, last accessed on 02/18/2023.

$\mathcal{M}_s^r : \mathcal{C}_s \rightarrow \mathbb{R}$ maps the configurations $c \in \mathcal{C}_s$ of release $r \in \mathcal{R}_s$ to their *measured* performance values in $\mathbb{R}$. Π_s^r denotes the performance-influence model for revision $r \in \mathcal{R}_s$ of system s.

Configuration level Conducting performance measurements on the history of a configurable software system raises the question of whether the addition and removal of configuration options across releases should be considered. To simplify the analysis, we resort to a fixed set of options that is available across all releases of a subject system. While this way we might miss some interesting cases, our data set is still large and diverse enough to answer reliably our research questions.

The independent variables for $RQ_{1.1}$ and $RQ_{1.2}$ are (1) the subject system s, (2) the release r, and (3) the configuration c. The dependent variable is the performance value $\mathcal{M}_s^r(c)$. A confounding factor is measurement noise caused by particularities of the hardware and software platform [71, 105]. To control for this factor, we measured all configurations multiple times (3 to 5 times depending on the subject system) until the coefficient of variation (i.e., standard deviation divided by the mean) of the repetitions is lower than 10%.

To answer $RQ_{1.1}$, we determine the performance values $\mathcal{M}_s^r(c)$ for each configuration $c \in \mathcal{C}_s$ and each release $r \in \mathcal{R}_s$. We consider a performance change between a configuration of two consecutive releases relevant if:

$$|\,\mathcal{M}_s^{r_i}(c) - \mathcal{M}_s^{r_{i+1}}(c)\,| > 2 \cdot \max\left(\mathrm{sd}_s^{r_i}(c), \mathrm{sd}_s^{r_{i+1}}(c)\right) \quad (4.1)$$

where $\mathrm{sd}_s^r(c)$ denotes the standard deviation of performance values of a configuration across repeated measurements. In other words, if a performance change does not exceed twice the larger standard deviation of the two releases, it is not further considered. The rationale for this conservative threshold is to filter out measurement noise and tiny performance changes.

Table 4.2: All valid configurations of our exemplary system from Section 2.1, their predicted performance values for two different releases, and the performance ranking of the configurations of the exemplary compression tool. The last column indicates whether the performance change is relevant according to Equation 4.1.

	Release 1		Release 2		
Configuration	$\Pi(c)$	Rank $(\Pi(c))$	$\Pi'(c)$	Rank $(\Pi'(c))$	Relevant
c_1	5	1	5	1	✗
c_2	8	4	8	4	✗
c_3	7	3	7	3	✗
c_4	9	5	9	5	✗
c_5	6	2	6	2	✗
c_6	7	3	7	3	✗
c_7	9	6	9.5	6	✓
c_8	10	7	10	7	✗
c_9	8.5	5	9	5	✓
c_{10}	9	6	9.5	6	✗

To answer RQ$_{1.2}$, we rank the configurations of each release r_i by their performance value. For illustration, we show the performance ranking of our exemplary compression tool for two releases in Table 4.2. c_2 represents the fastest configuration in both releases and c_6 the slowest configuration. Further, instead of directly comparing the rankings of two consecutive releases, we first filter out irrelevant performance changes according to our definition in Equation 4.1. That is, the ranking order of the second release is affected only by relevant changes. In Table 4.2, we show in the last column which configurations are relevant according to Equation 4.1, assuming a relative standard deviation of 1%. After filtering, the ranking of only c_2 and c_5 would be compared, resulting in a perfect correlation, since both configurations maintain their ranking in both releases (i.e., $c_2 < c_5$ holds).

To quantify the similarity of two rankings (i.e., the performance rankings of the configurations of the current and the previous release), we use the Kendall's Tau correlation coefficient [62]. A correlation value of 1 indicates perfect correlation, a value close to 0 means no correlation, and -1 indicates that the rankings are fully opposed (i.e., the configuration with the highest rank in release r_i has the lowest rank in release r_{i+1}, the configuration with the second highest rank in release r_i has the second lowest rank in release r_{i+1}, etc.). In other words, a high correlation indicates that the performance ranking of configurations remains stable across releases, whereas a low correlation indicates that the ranking changes considerably. We omit computing Kendall's Tau for releases where the rank changes for less than two configurations. Calculating the correlation of the relevant configurations in Table 4.2, we would obtain a perfect correlation of $\tau = 1.0$.

Option level In RQ$_{2.1}$ and RQ$_{2.2}$, we aim at identifying the configuration options and interactions that are responsible for the performance change that we observed at the configuration level. To identify changes of the performance influence of an individual configuration option or interaction, we build on previous work by Siegmund et al. [139]: We use *multiple linear regression* with *feature forward selection* to create for each revision $r \in R_s$ a performance-influence model Π_s^r of the form described in Section 2.2. Note that we do not follow a sample-based learning approach (i.e., one that uses only a subset of configurations). Instead, we learn models on the whole configuration space. This would be impractical in practice but gives us the most accurate results. So, the independent variables for RQ$_{2.1}$ and RQ$_{2.2}$ are (1) the subject system s and (2) the release r; the dependent variable is the corresponding performance-influence model Π_s^r for $r \in \mathcal{R}_s$.

To answer RQ$_{2.1}$, we determine for each $r \in \mathcal{R}_s$ the performance influences $\beta_s^r(t)$ of all terms $t \in \Pi_s^r$. A term can either consist of the base term (i.e., β_0 in Section 2.2.4), a configuration option (i.e., $\beta_o \cdot c(o)$ for $o \in \mathcal{O}$), or an interaction among multiple options (i.e., $\beta_{o_1..o_i} \cdot c(o_1) \cdot \dots \cdot c(o_i)$ for $o_1, \dots, o_i \in \mathcal{O}$). Function $\beta_s^r(t)$ returns the coefficient of the term. Similar to RQ$_{1.1}$, we consider a performance change between two coefficients relevant if:

$$|\beta_s^{r_i}(t) - \beta_s^{r_{i+1}}(t)| > 2 \cdot \max(\overline{\text{sd}}_s^{r_i}, \overline{\text{sd}}_s^{r_{i+1}}).$$

where $\overline{\text{sd}}_s^{r_i}$ denotes the mean standard deviation of all configurations of release $r_i \in R_s$. As with RQ1.1, if a change of performance influence does not exceed twice the larger average standard deviation of the two releases, it is not further considered. The rationale of using the *maximum* of the *mean* standard deviation is that we use the entire configuration space for learning performance models and thus accumulate the standard deviation over all configurations.

To answer RQ$_{2.2}$, we rank the terms $t \in \Pi_s^r$ based on their coefficients $\beta_s^r(t)$. Similar to RQ$_{1.2}$, the most influential term has the highest rank, the second most influential term has the second rank, and so on. As in RQ$_{1.2}$, we quantify to what extent the ranks between two releases r_i and r_{i+1} differ by using the Kendall's Tau correlation coefficient.

4.4 Evaluation

In this section, we summarize our results (Section 4.4.1). We use these results in subsequent metadata analysis (Section 4.4.2) and discuss the results along with further observations (Section 4.4.3) and potential threats to validity (Section 4.4.4).

4.4.1 Results

In what follows, we refer to the plots given in Figure 4.1 and Figure 4.2. For each subject system, there is one plot per figure: the plots in Figure 4.1 show the number of changes (red line) and the stability of the performance ranking (blue line) at configuration level; and the plots in Figure 4.2 show the number of changes (red line) and performance ranking stability (blue line) at the option level.

RQ$_{1.1}$: What is the fraction of the configuration space containing performance changes between consecutive releases?

In Figure 4.1, we show the fraction of configurations containing performance changes across consecutive releases (red lines)—the larger the value, the higher the fraction of configurations involved in a performance change. In Figure 4.3 (blue line), we provide a cumulative overview that shows how many of the 178 consecutive releases have a performance change in at least a certain fraction of configurations. For instance, we see that in more than 40% of the releases the performance changed in at least 20% of the configurations. Notably, 176 out of 178 (99%) releases have, at least, one configuration with a performance change.[11] Further, 2 (1%) performance changes are observed in the entire configuration space, 133 (75%) performance changes are observed in less than half of the configuration space, and 26 (15%) performance changes are observed only in 1% of the configuration space.

In Figure 4.4, we show the *intensity* of performance changes for VP9. Red color indicates performance degradation, blue color indicates performance improvement. For releases 1.4.0 and 1.6.0, we observe that the performance behavior of a considerable number of configurations (30%) of VP9 has changed substantially (i.e., the blue and the red colored configurations)—much more than our threshold of twice the standard deviation used in Figure 4.4.

11 We have detected no configurations with performance changes between releases 9.2.0 and 9.2.4 of PostgreSQL and between releases 2.2.1 and 2.2.2 of OpenVPN.

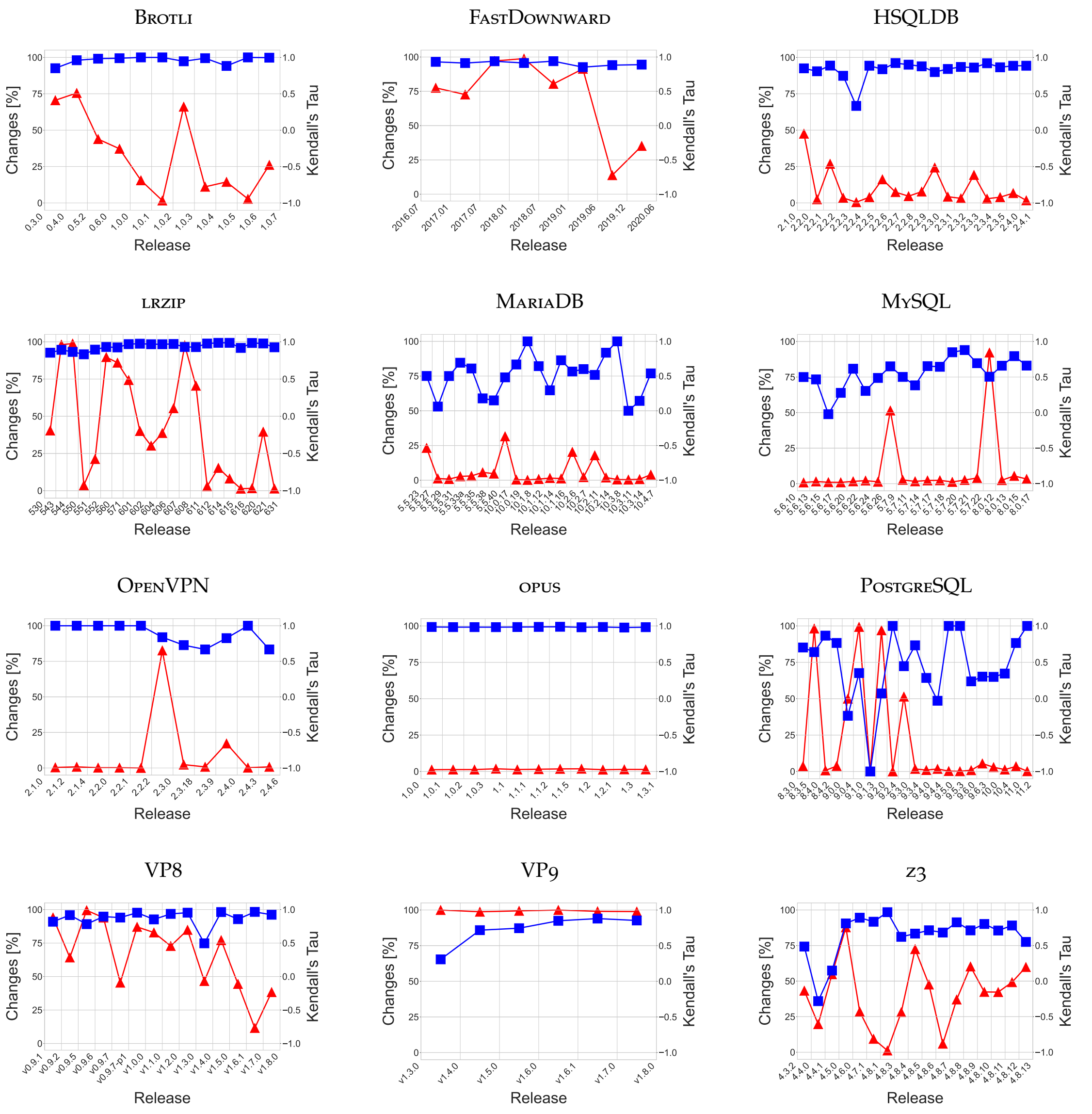


Figure 4.1: Fraction of performance changes and stability of performance ranking at configuration level. The red line indicates the fraction of configurations of the whole configuration space containing performance changes (in %); the blue line indicates the stability of the ranked configuration performance as measured by Kendall's Tau.

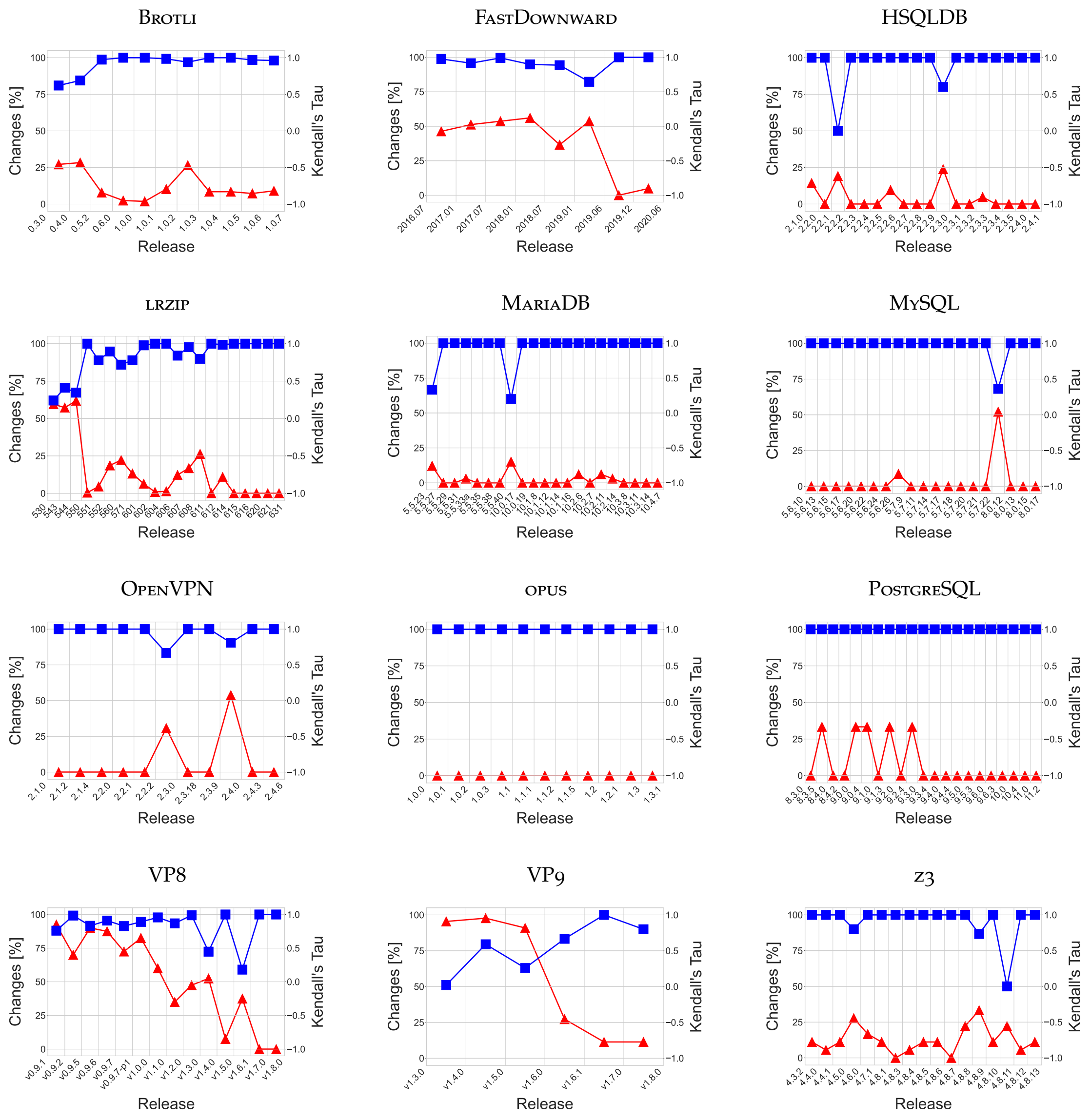


Figure 4.2: Fraction of performance changes and stability of performance ranking at option level. The red line indicates the fraction of options containing performance changes (in %); the blue line indicates the stability of the ranked options performance as measured by Kendall's Tau.

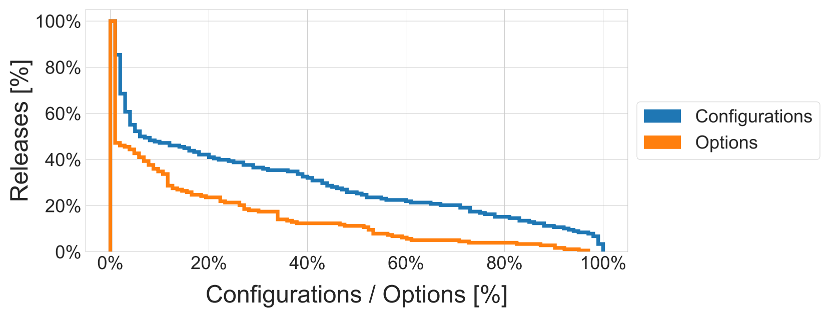


Figure 4.3: Cumulative plot on the fraction of involved configurations (blue) or options (orange) in all performance changes of $RQ_{1.1}$ and $RQ_{2.1}$, respectively.

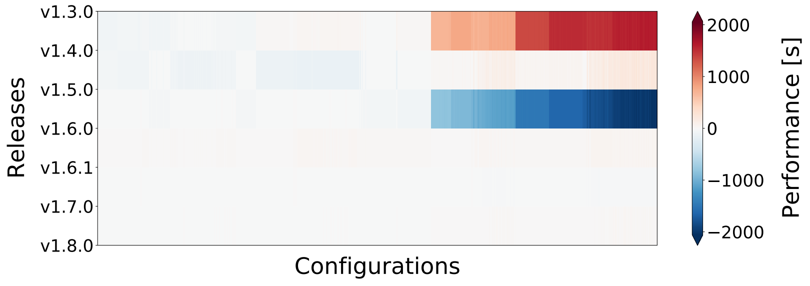


Figure 4.4: Performance changes of VP9 across all configurations (x-axis) and releases (y-axis). We use a color palette to illustrate performance degradation (> 0, red) and performance improvement (< 0, blue). The configurations are sorted in ascending order according to their mean performance over all releases. There are 3 008 configurations on the x-axis; axis ticks have been omitted for readability.

Summary of Research Question 1.1

Almost every release of every subject contains, at least, one performance change in some configuration. The majority of performance changes affects less than half of the configurations.

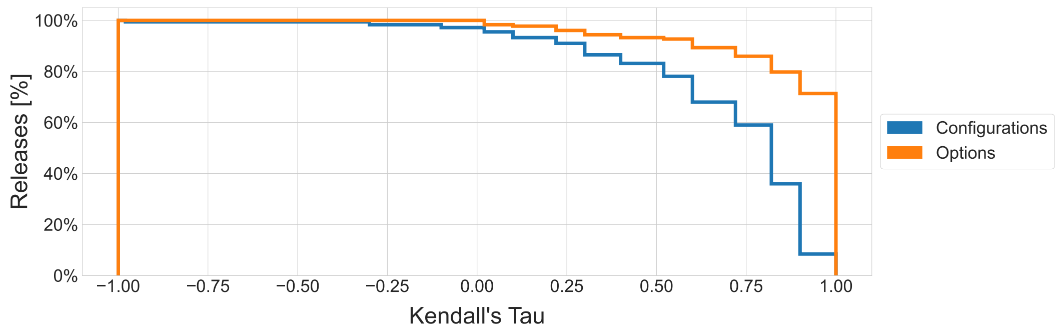


Figure 4.5: Cumulative plot on the stability of configurations (blue) or options (orange) in all performance changes of $RQ_{1.2}$ and $RQ_{2.2}$, respectively.

$RQ_{1.2}$: How stable is the relative performance of configurations in the presence of performance changes between consecutive releases?

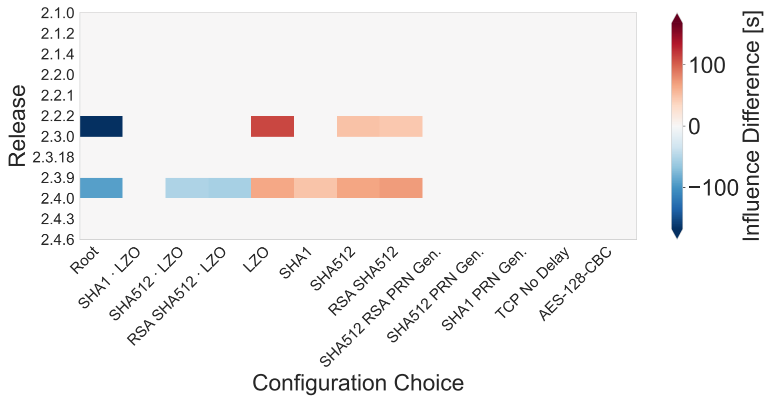


Figure 4.6: Performance influence of options and interactions (x-axis) of OpenVPN across all releases (y-axis). A color palette illustrates performance degradation (> 0, red) and improvements (< 0, blue).

In Figure 4.2, we show the stability of the performance ranking of configurations, as quantified by Kendall's Tau (blue lines). A high value indicates high stability: the performance ranking of configurations changes only slightly (i.e., the fastest configurations stay the fastest, etc.). Across all systems and releases, the ranking is largely stable: $\overline{\tau} = 0.74$. In Figure 4.5, we provide an overview of the stability (blue line) between all 178 consecutive releases. 148 (83%) releases have a τ value higher than 0.5, 105 (59%) releases have a τ value higher than 0.80, and 64 (36%) releases have a τ value higher than 0.90. Opus is most stable ($\overline{\tau} = 0.98$), PostgreSQL is least stable ($\overline{\tau} = 0.36$).

Summary of Research Question 1.2

The performance ranking of configurations is largely stable across consecutive releases ($\overline{\tau} = 0.74$), with some notable exceptions.

RQ$_{2.1}$: *How frequent and how strong are changes of performance influences of individual configuration options and interactions between consecutive releases?*

In Figure 4.2, we show the fraction of how many options or interactions have changed from one release to another (red line). As explained in Section 4.3.4, the influences were determined by learning a performance-influence model per release. It is important to note that the prediction errors of the models were generally low (3.9%, on average), so we are confident that the influences are accurate.

Frequency: The fraction of configuration options and interactions involved in performance changes ranges from 0.45% (e.g., lrzip) to 95% (e.g., VP8). In Figure 4.3 (orange line), we provide a cumulative overview that shows how many of the consecutive releases have a performance change in at least the certain fraction of configuration options. For instance, we see that about 12% of the consecutive releases indicate a change on more than 40% of the configuration options and interactions. On average, the influence of 28% of the configuration options and interactions change across all releases. While, in most of the changes (91%), multiple configurations options and interactions are involved, there are cases where just a single option is responsible for a performance change (PostgreSQL). Figure 4.6 shows the

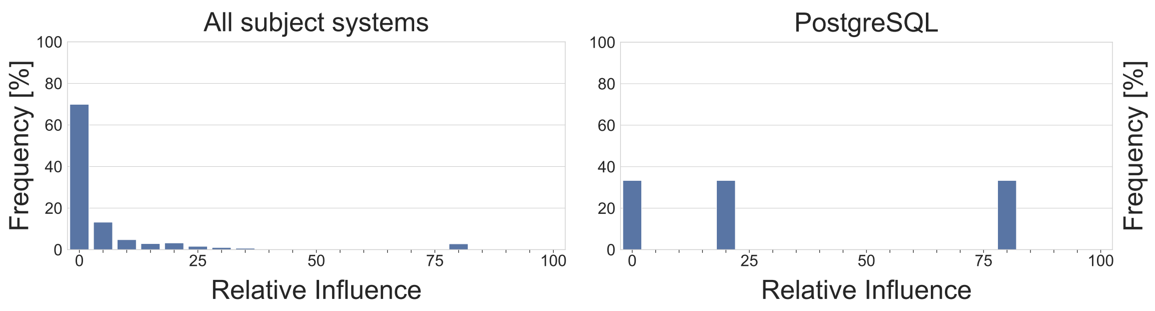


Figure 4.7: Distribution of the relative influences of model terms across all subject systems (left) and for PostgreSQL (right).

intensity of performance influences of individual configuration options and interaction for OpenVPN: In releases 2.3.0 and 2.4.0, we note substantial performance changes, each of which is caused by only a subset of options, some of which interact causing the effect (e.g., SHA512 and LZO).

Distribution: In Figure 4.7, we show the distribution of relative performance influences across all subject systems and releases. 83% of the model terms (options or interactions) have only a very small influence on performance (less than 7.5%), which is in line with theoretical considerations of influencing factors in sensitivity analysis [128]. Only 3% have an influence of 80% and more on the system's performance. That is, the influence on the performance is mostly distributed over all configuration options and interactions. A notable exception is PostgreSQL, where only three terms are relevant, namely the base term, fsync (which enables synchronized writes), and trackActivities (which enables the collection of information on the executed commands).

Summary of Research Question 2.1

There is a substantial number of cases where influences of individual configuration options or interactions change across releases, but only few have a substantial influence on performance. Most performance changes (91%) are caused by multiple options and interactions, but there are cases where only a single option is responsible.

RQ$_{2.2}$: How stable is the relative influence of configuration options and interactions in the presence of performance changes between consecutive releases?

In Figure 4.2, we show the stability of the performance ranking of individual influences of options and interactions, as quantified by Kendall's Tau (blue lines). We included a cumulative overview in Figure 4.5 (orange line). In comparison to RQ1.2, stability is much higher: $\overline{\tau} = 0.91$. 151 (85%) have a τ larger than 0.8, and 142 releases (80%) have a τ larger than 0.9. For two subject systems (Opus and PostgreSQL), the performance ranking is stable across all releases. The performance model ranking (i.e., blue line of the right plot) of the consecutive releases 1.3.0 and 1.4.0 in VP9 contain slightly negative values, which indicate larger fluctuations and even a partial reversal of the ranking (see change of ranking of first and fourth option between 1.3.0 and 1.4.0 in Figure 4.8).

For illustration, we show in Figure 4.8 the evolution of the ranking of the 5 most influential configuration options or interactions of VP9. The ranking changes considerably over time,

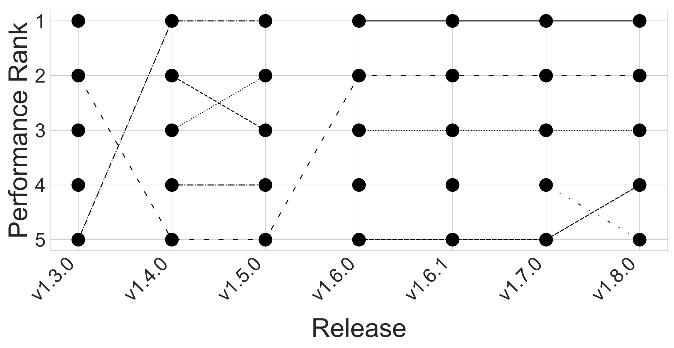


Figure 4.8: Evolution of the performance ranking of the 5 most important model terms of VP9. Connected nodes illustrate the change of ranking from one release to another. An unconnected node means that the ranking in the next release is lower than 5.

where the most changes are in between 1.3.0 and 1.4.0. The reason is a performance regression in the options realtime and quality encoding, which was fixed in 1.6.0.

Summary of Research Question 2.2

The performance ranking of influences of individual configuration options and interactions is largely stable across consecutive releases ($\overline{\tau} = 0.91$), with some exceptions.

4.4.2 Metadata Analysis

To triangulate the results of Section 4.4.1, we have conducted a deeper analysis that aligns the identified performance changes and influential model terms with reported cases in change logs and commit messages of the respective subject systems. In particular, we are interested in to what extent the learned performance models are able to pin down configuration options or interactions that are involved in a performance change.

Conduct In Figure 4.9, we show the steps of our deeper analysis. In Step I, we check the performance change of each consecutive release at the configuration level and the option level (see Figure 4.1 and Figure 4.2). We consider a release as relevant if the performance change at option or configuration level of one release exceeds 5% of the previous release. We exclude releases for which only the performance of the base program (i.e., the term base) has changed. There are two reasons for this: (1) a code change to the common base code affects all configurations; (2) a code change affects an option that is not included in our analysis. For instance, changing the default value of an unconsidered configuration option (e.g., by enabling it by default) can be the reason for performance changes in base. This scenario occurred only in PostgreSQL, in which in 4 out of 5 relevant releases, only the term base has a changed performance value.

Applying both filters, 79 out of 181 (43%) releases are relevant for our investigation. opus is the only subject system with no detectable performance changes. Thus, opus will not be considered in this analysis. By contrast, all releases of VP8 and VP9 are included in our analysis.

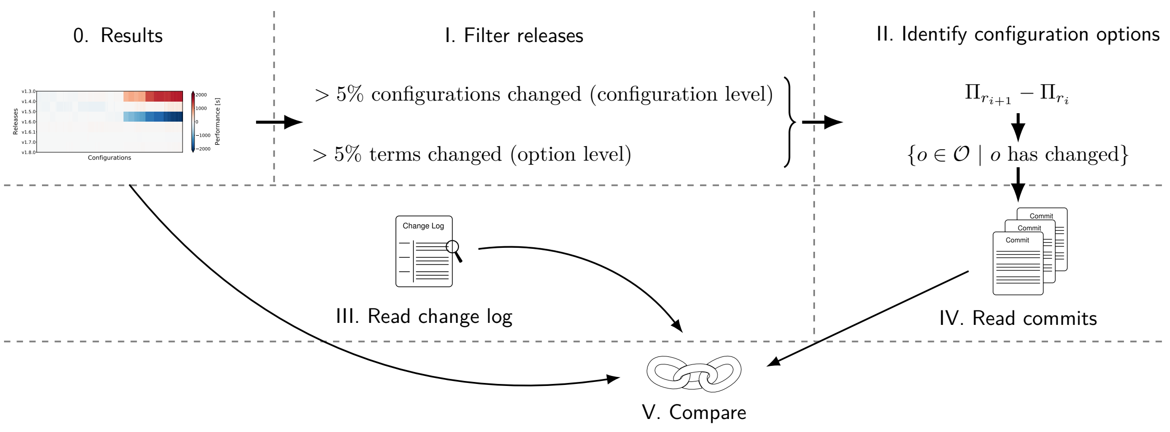

0. Results
I. Filter releases
II. Identify configuration options
> 5% configurations changed (configuration level)
> 5% terms changed (option level)
$\Pi_{r_{i+1}} - \Pi_{r_i}$
$\{o \in \mathcal{O} \mid o \text{ has changed}\}$
Change Log
III. Read change log
Commit
IV. Read commits
V. Compare

Figure 4.9: Methodology of our deeper analysis. Step 0 includes our previously discussed results. In Step I, we select consecutive releases with certain degrees of performance change. Afterwards in Step II, we identify the configuration options with a changed performance influence from one release r_i to another r_{i+1}. In Step III, we read change logs for documented performance changes to find the cause and extract for each release whether performance changes were documented or not. In Step IV, we read commit messages of the relevant consecutive releases and include the changed configuration options from Step II to our analysis to aid finding the cause. In this step, we obtain for each release whether a performance change was documented in the commits and whether at least one affected configuration option was mentioned or not. Last, in Step V, we compare the results from Step 0, Step III, and Step IV. In particular, we show in which cases the change log and commit messages correspond or differ from our results and in which cases the configuration option is mentioned.

In Step II, we inspect performance-influence models of Section 4.4.1 in more depth to gather information on which configuration options and interactions thereof have actually changed. Based on this information, we search for documented performance changes in the entire change log between each pair of relevant consecutive releases including the change log for the current release for documented performance changes in Step III.

In Step IV, we analyze the commit messages between each pair of relevant consecutive releases. Fortunately, our selected subject systems are open source relying on publicly accessible version control systems (mostly git). Since reading all commit messages is infeasible for larger projects, we filter the commit messages using the following keywords similar to other studies [23, 57]: *slow*, *fast*, *time*, *perf*(ormance), *optim*(ize), and *regression*. Additionally, we added the name of the configuration options that we identified in Step II and check whether a configuration option is mentioned. If one of these keywords matches, we analyzed the commit message in detail.

Finally, in Step V, we contrast the obtained information by comparing them with each other. In particular, we report in how many cases the commit messages reported a performance change in comparison to the change log and in how many cases the configuration option was mentioned. For brevity, we provide only a summary of our analysis in Table 4.3; the full set of results is available on our supplementary web site.

Table 4.3: Overview of the number of relevant releases (RR) and releases reporting speed-ups (⬆) or slow-downs (⬇) in the change log and commit messages. The last column indicates the number of releases where at least one affected configuration option is mentioned in the commit messages.

System	#RR	Change log		Commit		Option
		⬆	⬇	⬆	⬇	
BROTLI	9	2	0	4	0	5
FASTDOWNWARD	9	0	0	2	1	5
HSQLDB	8	4	0	1	0	1
LRZIP	15	7	0	7	1	9
MARIADB	5	2	0	5	0	5
MYSQL	3	2	0	3	0	3
OPENVPN	2	2	0	2	0	2
POSTGRESQL	1	0	0	0	0	1
VP8	14	7	0	10	0	12
VP9	6	6	0	6	1	5
Z3	16	1	2	15	2	11

To reduce interpretation bias, two authors of the original paper[12] performed the analysis of Step III and Step IV independently. After the analysis, they compared their results and discussed the differences to reach a consensus. Only in 3 pairs of releases of MARIADB, where the commit messages were larger than 10 MB, the third author checked and confirmed the results of the first author's manual analysis.

Results In Table 4.4, we list an excerpt of the results of our deeper analysis. We provide the complete list of results on our supplementary web site[13]. Details on each result are also included on our supplementary web site[14]. We show which of the consecutive releases have reported a speed-up or a slow-down in change logs or in commit messages, and whether the affected configuration option has been mentioned. The table also includes information which fractions of configurations improved or decreased performance.

7 out of 88 (8%) consecutive releases do not include a change log. In summary, in 35 out of 81 (43%) consecutive releases, the change log reported a performance change, whereas 2 reported a slow down and 33 a speed-up. In 56 pairs of releases (64%), the commit messages reported a performance change. Comparing change log and commit messages, we found that in 48 out of 81 (59%) consecutive releases, the change log and commit messages correspond to each other. In the remaining 33 consecutive releases (41%), 26 (32%) list other (and more)

12 Christian Kaltenecker and Stefan Mühlbauer

13 `https://github.com/ChristianKaltenecker/PerformanceEvolution_Website/blob/master/MetadataAnalysis/AnalysisTable.md`, last accessed on 02/20/2023.

14 `https://github.com/ChristianKaltenecker/PerformanceEvolution_Website/blob/master/MetadataAnalysis/MetadataAnalysis.md`, last accessed on 02/20/2023.

Table 4.4: Excerpt of the 88 relevant consecutive releases, the fraction of sped up and slowed down configurations, whether speed-ups (⬆) or slow-downs (⬇) are mentioned in the change log or in the commit message, and whether changes in the identified options/interactions are reported.

System	Release	Speed-up (%)	Slow-down (%)	Change log	Commit	Option
BROTLI	0.3.0–0.4.0	54.4	16.1	⬆	⬆	✓
	0.4.0–0.5.2	66.1	9.4	✗	⬆	✓
	0.5.2–0.6.0	36.1	7.7	⬆	⬆	✓
	0.6.0–1.0.0	10.0	27.2	✗	⬆	✗
	1.0.2–1.0.3	51.6	14.4	✗	✗	✓
	1.0.6–1.0.7	26.1	0.0	✗	✗	✗
HSQLDB	2.1.0–2.2.0	0.3	47.2	⬆	✗	✗
	2.2.1–2.2.2	26.6	0.2	⬆	⬆	✗
	2.2.5–2.2.6	0.9	15.2	✗	✗	✓
LRZIP	530–543	3.8	36.6	⬆	⬆	✗
	543–544	25.0	73.1	⬆	⬆	✓
	544–550	95.3	3.4	⬆	⬆	✓
	552–560	7.7	81.9	⬆	⬆	✗
	560–571	85.2	0.6	⬆	⬆⬇	✓
MariaDB	5.5.23–5.5.27	22.5	0.7	⬆	⬆	✓
	5.5.35–5.5.38	1.6	4.0	✗	⬆	✓
	5.5.40–10.0.17	5.9	25.6	⬆	⬆	✓
	10.1.16–10.2.6	5.9	25.6	✗	⬆	✓
	10.2.7–10.2.11	17.8	0.1	✗	⬆	✓
MySQL	5.6.26–5.7.9	0.0	51.4	✗	⬆	✓
	5.7.22–8.0.12	0.0	92.2	⬆	⬆	✓
	8.0.13–8.0.15	2.3	3.0	⬆	⬆	✓
PostgreSQL	9.0.0–9.0.4	50.0	0.0	✗	✗	✓
VP8	1.3.0–1.4.0	40.3	6.2	✗	⬆	✓
VP9	1.3.0–1.4.0	34.31	65.6	⬆	⬆⬇	✓
	1.6.0–1.6.1	0.0	100.0	⬆	⬆	✓
Z3	4.8.7–4.8.8	8.1	28.9	⬇	⬆⬇	✓
	4.8.8–4.8.9	16.2	44.0	⬇	⬆	✓

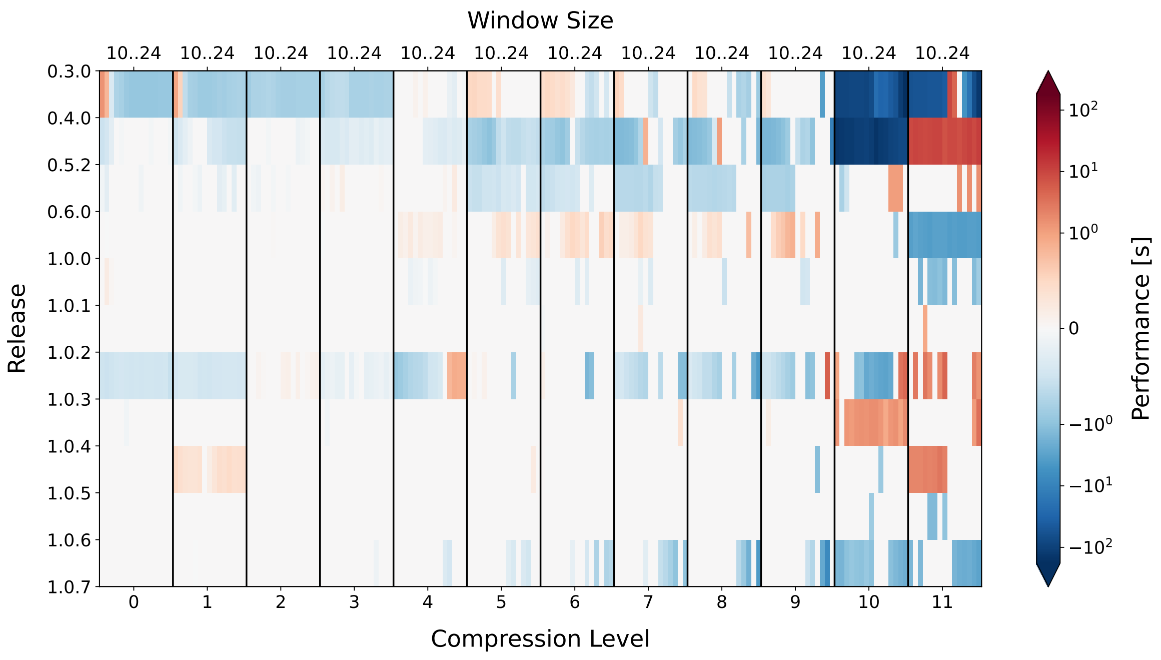


Figure 4.10: Performance changes of BROTLI across all releases (y-axis). The color code highlights performance degradation (> 0, red) and performance improvement (< 0, blue).

performance-relevant information in the commit messages than in the change log. The change log delivers more performance-relevant information in only 5 consecutive releases (6%). In total, 60 out of 88 (68%) consecutive releases mention a performance change in the change log or commit message.

In 4 cases (5%), speed-ups and slow-downs were reported in commit messages. At least one affected configuration option was mentioned in 59 cases (67%), out of which 14 pairs of releases (16%) mention only changes in the configuration option's code base but no performance changes in the change log or commit messages. In 29 of the cases (33%), no affected configuration option is mentioned. Moreover, in 7 cases (8%), some configurations show a minor but relevant performance change while the performance-influence model does not (i.e., the performance-influence models are similar in these cases). In 12 cases (15%), the change log or commit messages report speed-ups without mentioning a configuration option.

Details To provide in-depth insight into our deeper analysis, we show in Figure 4.10 the configuration options WindowSize and CompressionLevel of BROTLI which control the compression rate of files. A blue color represents performance increase and a red color a decrease from one release to another. In the first pair of consecutive releases, 0.3.0 – 0.4.0, an increase in performance of compression levels 0 – 3 can be observed, which is also mentioned in the change log and the commit message. However, the speed-up of compression levels 10 and 11 are not directly mentioned and may be a product of memory improvements, which was another focus of release 0.4.0. In release 0.5.2, the performance is improved for 66% of the configurations, which is not mentioned in the change log. One commit message, however, addresses speed and the affected configuration options: *„new hasher - improved speed, compression*

and reduced memory usage for q:5-9 w:10-16“[15]
Note that *q* stands for compression level (or quality) and *w* for the window size. The slow-down in compression level 11, however, is not addressed until the next release 0.6.0 and mentioned there as fixed. We can see the fix for the compression level 11 only later in release 1.0.0. In release 0.6.0, the developer also report optimizations for mid-level compression levels (5–9). Another interesting pair are releases 1.0.2 and 1.0.3. Although more than half of the configurations experience a performance change in this range, there are no direct relations to these performance changes in the change log or the commit messages. Only a fix in compression level 10 is reported. The changes are a consequence of a new dictionary generator that was introduced in this release. In the latest release 1.0.7, where a quarter of the configurations was sped up but no configuration was slowed down, nothing relevant is reported in the change log and the commit messages. The changes focus on optimizations on the ARM architecture. Some of these changes may also affect the x86 architecture where our experiments were performed on.

Between releases 2.1.0 to 2.2.0 of HSQLDB in Table 4.4, we measured a slow-down in 47% of the configurations and a speed-up of only 0.3% of the configurations, whereas the change log reports only a speed-up. With the option-level analysis, we could relate the slow-down to the configuration option logSize, which controls the size of the log file before an automatic checkpoint occurs. A deeper analysis of commit messages did not confirm any evidence of a slow-down.

In Table 4.4, we show notable cases for lrzip. In the pair 530–543, more than 36% of configurations show a slow-down and more than 3% show a speed-up. The change log and commit messages only mention the latter. In the option-level analysis, we find a slow-down in different compression algorithms, compression levels, and in multi-threading. The commit messages mention changes on multi-threading and compression algorithms, but in relation to decompression, which was not measured. In the next pair of releases, 543–544, we have a similar situation, with 73% of configurations showing a slow-down and 25% of the configurations showing a speed-up. According to our option-level analysis, similar configuration options as in the release pair 530–543 are affected. Commit messages report that the way how threads are spawned has been changed to improve the performance of compression[16]. However, this slow-down is addressed between 544–550, where the respective commit was completely reverted[17]. Another situation appears in the release pair 552–560. Change logs and commit messages report only speed-ups and no slow-downs. Again, multiple configuration options, such as compression algorithms, compression levels, and multi-threading are affected. Moreover, the commit messages do not mention any of the affected configuration options, only in relation to another operating system (Mac OSX). Later, in the release pair 560–571, more than 85% of the configurations are sped up and less than 1% have a slow-down. Both change log and commit messages report speed-ups in multi-threading, whereas only the commit messages also report a minor slow down.

MariaDB and MySQL are also included in the excerpt in Table 4.4 since the first is a fork of the latter. Both projects use semantic versioning and introduce new functionality in new major releases that may break backward compatibility. In the major release of MariaDB between

15 `https://github.com/google/brotli/commit/2048189048`, last accessed on 02/20/2023.
16 `https://github.com/ckolivas/lrzip/commit/688aa55c7930`, last accessed on 02/20/2023.
17 `https://github.com/ckolivas/lrzip/commit/8dd9b00`, last accessed on 02/20/2023.

releases 5.5.40–10.0.17 and MySQL between releases 5.7.22–8.0.12, the InnoDB engine was updated and, in the case of MySQL, some refactoring was applied. Further refactoring of logging and binlogging was applied in MySQL, between releases 5.6.26–5.7.9, which resulted in a slow-down. Releases 8.0.13–8.0.15 of MySQL contain further bug fixes that result in speed-ups. Between releases 5.5.35–5.5.38, MariaDB applied several bug fixes and speed-up fixes. Later, between releases 10.1.16–10.2.6, the InnoDB engine was updated. Between releases 10.2.7–10.2.11, MariaDB reverted an InnoDB fix from MySQL[18] and performed code optimization.

Interestingly, we observed that MariaDB and PostgreSQL have the same fixes between releases 5.5.23–5.5.27 and 9.0.0–9.0.4, respectively. There, forcing *fdatasync* for physical data synchronization on Linux causes an improvement in performance and assures that the files are synchronized on the physical storage, which is important for data recovery in case of system crashes. Interestingly, MariaDB reports speed-ups in the change log and commit messages, whereas PostgreSQL does not.

Another interesting case in Table 4.4 includes VP8 and VP9. Both video encoders are developed in the same repository and VP9 represents the successor of VP8. The consequence is that the developers compare VP9 with its predecessor in terms of performance, which applies to the pair 1.3.0–1.4.0. There, the developers report a regression in the commit messages in comparison to VP8: „*Was 20% faster than speed -5 of vp8. Now 20% slower but adds motion search*(...)“[19]. This change demonstrates that VP9 comes with additional functionality at the cost of deviating from the performance of VP8. Interestingly, VP9 contains the single consecutive release 1.6.0–1.6.1 where all configurations indicate a slow down. To increase confidence in this particular findings, we have additionally executed all configurations of releases 1.6.0–1.6.1 on another current setup (i.e., another hardware and current operating system[20]) and were able to observe the slow-down too. The change log and the commit messages, however, report only speed-ups. Our performance-influence model related the changes to multiple configuration options and interactions, some of which are mentioned in the commit messages.

z3 also contains pairs of consecutive releases (i.e., 4.8.7–4.8.8 and 4.8.8–4.8.9) where the developer reported a regression already in the change log and the commit message. The reason behind lies in nightly performance tests that are performed for z3 on different platforms and, thus, the developers of z3 are informed early about performance changes. However, the affected configuration options are not mentioned in these releases.

Summary of the Metadata Analysis

In most consecutive releases (68%), the developers mention performance changes in the change log or commit messages. In a similar amount of releases (67%), the developers mention the affected configuration option in the commit message, but there are cases (16%) where no performance change but changes in affected configuration options have been reported.

18 https://github.com/MariaDB/server/commit/cb9648a6b5, last accessed on 02/20/2023.
19 https://github.com/webmproject/libvpx/commit/ea8aaf15b55, last accessed on 02/20/2023.
20 Intel Core i5-4590 CPU with 16 GiB RAM (Debian 11)

4.4.3 Implications

Insight: Need for prioritization of configurations for testing Our study shows that change in performance behavior is not the exception but the rule (i.e., 99% of the releases contain a performance change in $RQ_{1.1}$) as also confirmed by others [56, 107]. What is interesting is that most performance changes (78%) affect less than half of the configuration space and a non-negligible number (16%) only 1% of the configuration space. This is bad news for developers as, this way, performance problems are more difficult to spot with standard methods, such as testing default or random configurations (we will get back to this shortly). Only in few (1%) cases, the whole configuration space is affected by a performance change, which is easy to discover by measuring the default configuration for instance. This result is notable and corroborates the need for performance modeling and testing methods that incorporate configurability. Random testing is unlikely sufficient to reveal cases where only few configurations are affected by a change. Furthermore, we found that, in 7% of the releases with a performance change, functional changes on the affected configuration options are reported but not observable with our models (i.e., a speed-up or slow-down). Combining configuration testing with performance modeling could help in such cases.

Insight: Mixed-strategy sampling Another notable result is that performance changes are often caused by *multiple* configuration options (i.e., in 91% of the changes in $RQ_{2.1}$). This includes (1) cases where the performance change is a *cumulation* of the individual influences of several options and (2) cases where multiple options *interact* and, this way, cause a performance change. Both cases are interesting as they demonstrate that configuration sampling methods based on simple structured coverage criteria (e.g., *t*-wise sampling) or simple random sampling are doomed to fail. The distribution of influences of options and interactions shows that only a *combination* of random and structured sampling methods is able to sufficiently cover the configuration space. That is, our results demonstrate that simple pair-wise sampling would miss many relevant interactions—in z3, we found even a performance-relevant interaction among 6 configuration options! At the same time, pair-wise sampling would consider way too many pair-wise interactions that are irrelevant, rending the whole approach expensive or even intractable in practice [123]. A random approach would likely miss important interactions, too. For example, in the case of PostgreSQL, a single option is responsible for a substantial performance change between 9.0.0 and 9.0.4. Our results (in particular, distributions of influences) shall inform recent developments in combining structured and random sampling to improve sample quality and reduce cost. In the past, the application of such a combined sampling strategy, distance-based sampling, already outperformed other sampling strategies with regards to performance [60, 118].

Insight: Configuration sensitivity A further notable result is that, in about 80% of the releases (see $RQ_{2.2}$), the ranking of configuration options and interactions is stable ($\tau > 0.8$). This is good news, as developers and users can assume a certain stability of the relative performance of individual configurations. In other words, there is no immediate need for reconfiguring the system after a new release. However, there are exceptions such as PostgreSQL, where the performance ranking changes considerably over time (see Figure 4.1). Knowing about this general behavior sheds light onto the *sensitivity* of the system's performance behavior on configuration. Our results suggest that this sensitivity varies across systems and developers need to know that for performance testing and tuning.

At the level of individual configuration options and influences, we observe a similar picture. The sensitivity of individual options regarding performance differs across systems and may change over time. An option that influences performance to a large extent in one release may have only a minor influence in the next release. This finding has implications for configuration sampling across revisions [148] and transfer learning [54, 80]: In both cases, a set of options is selected based on few revisions and then applied to other revisions (for further sampling or learning transfer). Our results indicate that this approach may work for most of the cases, but is too simplistic for the general case, as the set of relevant options and interactions may change considerably (e.g., VP9). For most cases nevertheless, focusing on the configuration options or interactions with the highest influence could be a promising way when using sampling, since their relative influence remains largely the same.

Insight: Diverging performance behavior An interesting aspect of our selection of subjects is that VP8 and VP9 share some of their history and are still developed in the same repository. One might expect that this leads to similarities in performance behavior and evolution, since fixes and optimizations might be transferred easily. Our data do not confirm this expectation. On the contrary, we even found an opposing performance regression in 1.3.0–1.4.0: VP8 was sped up for 40% of the configurations and slowed down for only 6.2% of the configurations whereas VP9 shows a massive slow-down for 65.6% of the configurations. The same holds for MariaDB and MySQL, where the first is a fork of the later. Both show different performance changes in their evolution. While this does not have to be a problem per se, our analysis framework provides proper means for developers to identify such divergences.

Insight: Main-effects sampling still necessary, but not sufficient Moreover, our results contribute to the *new feature-interaction challenge* [8]. The idea is that there are different kinds of feature interactions, at different levels of abstraction, including functional and non-functional interactions that manifest in externally observable or internal behavior. The goal is to collect data from many different cases and triangulate results on interactions between options or features to learn about their nature and to predict one kind of interaction based on information about another kind [69]. Our results in $RQ_{2.1}$ and $RQ_{2.2}$ provide real-world data on likelihood and properties of performance feature interactions; our measurement and analysis framework offers a blueprint for conducting further experiments on other kinds of interactions (e.g., regarding memory utilization or energy consumption).

Insight: Configuration awareness Another interesting issue of our empirical study is whether we are able to reveal new information in terms of performance changes in addition to what is already documented and thus well-known among developers and users. To investigate whether performance changes are explicitly documented by developers (i.e., the developers added the performance change intentionally), we manually analyzed the change logs (if available) of 6 out of 12 systems (i.e., FastDownward, HSQLDB, lrzip, VP8, VP9, z3) in Section 4.4.2. Several performance changes have been documented by developers, but not all. We found that developers often report speed-ups in commit messages and change logs but only rarely slow-downs. The reason may be that developers become aware of these

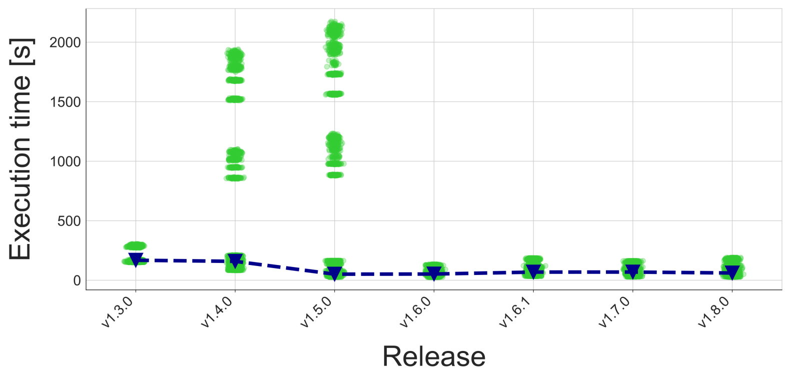


Figure 4.11: The performance of all configurations (green dots) and the default configuration (blue dotted line) of VP9. The x-axis shows the releases and the y-axis the execution time in seconds.

slow-downs only after deployment, as several cases indicate in which the slow-down was encountered and fixed one or two releases later. Such issues could be detected early by a configuration-aware continuous performance testing pipeline. Although some software systems, such as z3 and VP9 use performance tests, these are not configuration-aware. This could explain why these subject systems report regressions, but only to a certain extent. Our results suggest that configuration-aware performance testing can indeed provide new information in an automated manner and simultaneously validates our findings. Interestingly, in some of the performance changes, we observed a slow-down, although the change logs reported a speed-up. In particular, release 1.4.0 of VP9 promises faster encoding in change log, although the change results in a slow-down of 265%, which can be considered as an unintentional slow-down. The reason behind this discrepancy is that the change log referred only to the default configurations; all other configurations, however, were affected by a massive slow-down, possibly untested and unaware by the developers. For illustration, we contrast in Figure 4.11 the performance of the default configuration and the mean performance of all configurations. Notably, in release 1.6.0 the performance regression has been fixed resulting in a speed-up of all configurations; the performance of the default configuration, however, remains largely unchanged. This performance optimization was achieved by avoiding and reordering some of the processor instructions for Intel chips and is mentioned in the change logs. This is an interesting aspect, since such cases demonstrate the importance of automated support and paves the way for further research in this area.

4.4.4 Threats to Validity

Construct validity To guarantee comparability across releases and to simplify benchmarking, we selected options that are available in all releases. While we may have missed interesting cases, this way, we increase internal validity by ruling out effects from option-specific benchmarks. Moreover, while performance changes could affect newly included configuration options that are enabled by default, this would affect either the whole configuration space or certain configuration options if the configuration option does depend on another configuration option. Either way, this would be visible in the performance-influence models. This affected also our deeper analysis and is the reason for why we have excluded consecutive releases where only the base code changes. In the end, only 4 pairs of consecutive releases of PostgreSQL were excluded by this filter. In all other cases, the performance-influence model shows changes in certain configuration options or interactions or does not change at all. Another threat to validity arises from the selection of the keywords for filtering commit messages. Choosing another set of keywords may yield other results. However, all selected keywords were used in related publications [23, 57] that focus on identifying performance regressions in commit messages or issue lists. One reason for the low number of reported configuration options is that developers may state configuration options under different names (e.g., *q* or *quality* for the compression level in brotli). We have encountered few cases in which very specific parts of the code were addressed in a commit message, but a clear relation to a configuration option is hard to discover without domain knowledge and code inspection. Another reason could be data-flow dependencies between the configuration options. For instance, in HSQLDB, the configuration option blowfish was not mentioned a single time in any commit message when a performance change occurred. When other configuration options affect the data that has to be encrypted by blowfish, then we relate the change to blowfish as the effect occur here, but the cause resides in code of another option.

Internal validity Measurement noise is not only caused by software but also by hardware [105]. Measurement noise, however, can blur the results of our performance measurements and lead to wrong conclusions. To limit measurement noise, we used identical hardware per subject system, running with a minimum Debian installation. Furthermore, we preceded the measurements with a CPU warm-up phase. The measurements of the Java-based database (HSQLDB) are additionally preceded by a complete benchmark execution because of the JIT compilation as proposed by Georges et al. [33]. Additionally, we isolated the benchmark execution of client-server software (i.e., HSQLDB, MariaDB, MySQL, OpenVPN, and PostgreSQL) by running the server on a different node than the client(s) running the benchmark. To avoid wrong benchmark results, Costa et al. [26] observe and solve different bad practices in method-level performance tests. Since we measure the system as a whole and not individual methods by, for instance, issuing SQL queries to the database system, we are not affected by these bad practices. We varied the hardware across subject systems, since we do not need to compare measurements among systems. Furthermore, we used, if possible, the same release of the libraries over all releases, and we repeated our measurements three to five times until the relative standard deviation of the repetitions was lower than 10%. To control measurement noise, we used the standard deviation to pin down performance changes.

The choice of the learning algorithm may threaten internal validity. Other learning algorithms could have produced other results for $RQ_{2.1}$ and $RQ_{2.2}$. We used multiple linear regression with feature forward selection [139] because the structure of models enables us to compare releases by comparing the coefficients of model terms. Further, choosing always the best candidate in the feature forward selection (see Lines 11–14 in Algorithm 1) represents another limitation of our approach, since choosing a worse performing candidate in one iteration might lead to much better performing candidates in a later iteration. In other words, our learned models could not represent the optimum models. However, the prediction error of the models was 3.7% on average, which indicates that the models cover nearly all influences of options and interactions on performance accurately. To reduce spurious terms, which are only an artifact of the measurement and learning procedure, we checked the documentation (i.e., commit messages and change logs, if available) of our subject systems.

Additionally, we have used the variance inflation factor analysis to reduce variance in the performance-influence models as described in Section 2.2.4. This step removed a few terms by maintaining the error rate of the performance-influence models. Removing terms that are not perfectly multicollinear but exceed these thresholds removes important terms needed to predict specific parts of the configuration space and, thus, the error rate decreases. In a pre-study, we have applied the variance factor analysis by using the commonly threshold of 5 [136] on the subject system lrzip. From 230 terms, 160 were removed by the variance factor analysis but at the cost of increasing the error rate of the performance-influence model from 6 % to 60 %. In our setup, we removed only terms with perfect collinearity. In Table 4.5, we show the number of terms of the performance-influence models before and after the VIF analysis. Overall, we removed 14 out of 702 terms while the performance-influence models' error rate remained constant.

Table 4.5: The number of terms of the performance-influence model per subject system before and after the variance inflation factor (VIF) analysis.

System	#Initial Terms	#Terms after VIF
brotli	166	166
FastDownward	44	41
HSQLDB	21	21
lrzip	220	220
MariaDB	35	33
MySQL	25	23
OpenVPN	13	13
opus	66	66
PostgreSQL	3	3
VP8	40	40
VP9	51	44
z3	18	18
Total	702	688

Finally, our metrics for identifying performance changes may threaten internal validity, since other metrics would identify other performance changes. For instance, the work of Costa et al. [26] investigates the performance change of some bad practices at method level of one single configuration and uses the Wilcoxon non-parametric test and Cliff´s Delta effect size to identify significant performance changes of their benchmark results. We refrained from using statistical tests to assess a significant performance change because the number of performance values per configuration (i.e., 3 or 5 performance values; 1 from each repetition) is far too low for a significance test and the suggested effect size metric, whereas the work of Costa et al. had at least 100 performance values due to a high number of repetitions. Increasing the number of repetitions on a similar level is infeasible despite the number of releases and configurations, we measured. Instead, we have used the standard deviation as an effect size to express the variance of measurement noise across multiple repetitions.

Due to the absence of a baseline, we need to resort to an automated approach, which we complemented, though, by studying commit messages and change logs manually (see above).

External validity To increase external validity, we chose configurable software systems from different domains, including throughput-intensive applications (compression tools, video encoders) and client-server applications (Web servers, databases). In total, our corpus contains software systems ranging from 181 to 6 480 configurations and 7 to 22 releases.

To keep experiment effort feasible, we limited the selection of configuration options to a tractable number. This limitation is due to our experiment setup, which aimed for high internal validity, and is not a principal limitation of our analysis framework. Considering more configuration options would require to sample the configuration space for learning performance influence models, instead of considering the whole space. While learning performance-influence models on small sample sets works well in practice [59], we aimed for high internal validity, ruling out possible inaccuracies.

The choice of the workload for performance measurement poses another threat to external validity. We have fixed the workload/benchmark across configurations and releases, this way gaining internal validity for external validity—see the discussion above. However, we used established community or developer workloads to catch typical scenarios, which already provided numerous interesting insights (see Section 4.3.3). For instance, the selection of the developer workload might be a reason why we found no performance changes in OPUS. Varying the workload shall bring even more insights in further studies.

4.5 Summary

Although performance evolution has been extensively studied in the literature, prior work concentrated on single or few default configurations. Since most software systems are configurable, performance changes can easily be missed this way. Specifically, we are interested in the role of *configurability* for performance evolution, for example, whether specific configurations exhibit diverging performance behavior and what configuration options (or interactions among options) are responsible for this.

In an empirical study, we analyzed performance changes of 12 real-world configurable software systems across 190 releases that span a total of 11 years of history. We found that almost every release of every subject system exhibits performance changes in some of their configurations. Notably, the majority of performance changes affects only a small subset of the configuration space, and most performance changes affect multiple options (up to 6), either by accumulation of influences or interactions among options.

A deeper analysis of these configurable software systems shows that performance changes are reported in the change log or the commit messages in most cases. Similarly often, changes regarding affected configuration options have been mentioned.

Our results confirm prior beliefs that configuration-dependent performance changes are the rule, not the exception. This has direct implications for configuration sampling, performance modeling, and transfer learning in the area of configurable software systems. For example, our results confirm assumptions that simple random configuration sampling is not sufficient to catch all relevant performance changes. Likewise, structured sampling strategies likely overestimate the prevalence of performance-relevant interactions among options. Our results clearly indicate that combined sampling strategies such as *distance-based sampling* hit a proper sweet spot.

A further notable insight is that, despite the prevalence of performance changes, the performance ranking of configurations and influences of individual options are in many cases not affected. That is, developers and users can assume a certain stability of configuration-dependent performance behavior. Still, we found cases where the performance ranking fluctuates considerably across releases. This phenomenon seems to be application- or domain-specific and is worth further exploring, as it has implications for transfer learning of performance behavior across releases since more stable applications or domains could focus on the most relevant configuration options; in other applications and domains such approaches are doomed to fail. Additionally, our deeper analysis demonstrates that using a configuration-aware performance testing pipeline could help in identifying configuration-specific performance changes early. Our measurement and analysis framework offers a solid basis for exploring these and related issues.

5 Performance Prediction in the Presence of Workload Variability

The overarching focus of our work is performance modeling. In Chapter 3, we focused on the configurability of software systems and improved performance modeling by devising a new sampling strategy for configuration spaces. Afterwards, in Chapter 4, we used performance modeling to identify performance changes of certain configuration options and interactions thereof across multiple releases. In this chapter, we go a step further and devise a new approach by taking into account (1) configurability, (2) evolution, and (3) workloads. Our focus in this chapter is to assess the limitations of our new approach and to investigate workload variability in the presence of configurability and evolution.

By workloads, we refer to the input on which configurable systems are executed. For instance, a compression program can be executed on multiple different files (e.g., music files, source code, etc.) and can achieve a different file compression efficiency on different types of files. To give another example with regard to performance, Pereira et al. [118] demonstrate that the video encoder x264 shows different performance characteristics for the 17 different video inputs. This means, focusing on only a single workload can obscure the results and lead to wrong conclusions about the performance of a software system. Even worse, Mühlbauer et al. [103] and Lesoil et al. [79] both show in recent studies that the impact of configuration options and interactions thereof depend on the workload. So, considering workload variability in configurable software systems is substantial. However, the survey by Jiang and Hassan [56] points out that identifying an appropriate set of workloads that covers all performance-relevant functionality is a hard endeavor which requires knowledge of the configurable system.

To the best of our knowledge, the study we present in this chapter is the first one that explores all three dimensions (i.e., configurability, evolution, and workloads) together to identify performance changes. Because of the exploratory nature of this study, we use the *case-study research method* as proposed by Runeson et al. [127]. This method is used for an initial investigation on the considered phenomena. In this study, we focus on a single configurable software system (i.e., the planning software FastDownward [46]) instead of multiple different software systems. This way, we are sacrificing external validity to the benefit of internal validity. Moreover, the focus on a single configurable software system enables us to identify potential limitations of our approach for finding performance changes and relying on domain knowledge from developers of the configurable software system.

Besides workload variability, FastDownward exposes two new characteristics that were not present in Chapter 4. First, some functionality of FastDownward (e.g., support for linear programming solvers) was only added in later releases. Since this functionality represents performance-critical parts of the software, we consider some configurations only as soon as they are introduced. In other words, we ignore configurations in older releases if they are not present. For instance, the linear program solver Soplex[1] was only integrated in a release at the end of 2021. Thus, we ignore configurations using Soplex for releases prior to the end of 2021. So, compared to Chapter 4, we now vary the configuration space across different releases. Supporting different configuration spaces across different releases is especially useful to consider options that are integrated or removed in the configurable software system at a later stage. In Chapter 4, we missed such configuration options. Second, compared to subject systems in Chapter 4, some configurations timeout in some workloads and releases. That is, in a setting of FastDownward, the user specifies a timeout for aborting the run. Specifying a timeout corresponds to a typical use case in planning competitions [149]. This, however, can lead to cases where a configuration runs into a timeout in one release, but not in the subsequent release. Both characteristics represent additional difficulties for our approach. We address these characteristics by considering different configuration spaces in our analysis and use imputation [75] for the configurations with a timeout (i.e., we assign the timeouted configuration to a constant performance value).

Another arising problem while considering configurability, evolution, and workloads together is the extended combinatorial explosion, since all configurations have to be measured in all releases and workloads. Even worse, taking into account multiple configurable software systems adds another multiplying factor to the measurements. In Chapter 4, we take into account multiple configurable systems, different configurations, and releases, which took multiple years of CPU time for the measurements that we have presented. For the study in this chapter, we cannot perform all measurements in a reasonable amount of time and therefore need to reduce the number of measurements to a reasonable number. To further improve internal validity and to address the issue of domain knowledge for workload selection [56], we involve developers of FastDownward in the experiment setup of our study. Consequently, we are able to reduce the number of configurations by incorporating domain knowledge from developers. The configurations suggested by the developers are selected in such a way that they cover different functionality not already covered by other configurations.

A further problem is the high number of workloads of FastDownward (i.e., more than 1 800 workloads). However, developers do not know upfront how the workloads perform for different configurations. So, to reduce the number of workloads to a reasonable number, we perform preliminary performance measurements. With the results of the preliminary performance measurements, we filter out workloads that run into a timeout in most configurations and select workloads with a reasonable performance. This way, we reduce the number of measurements to a feasible amount. In the end, we consider 49 different configurations, 56 workloads, and 6 releases of FastDownward for this study.

After addressing the setup for the performance measurements of FastDownward, we need to analyze this data to identify performance changes. Since, to the best of our knowledge, there is currently no analysis approach that considers all three dimensions, we devise a new approach based on performance-influence models that identifies performance changes across workloads and takes in consideration different configuration spaces. Similar to Chapter 4, our approach identifies performance changes at the option level. That is, instead of pointing out which configurations are affected by a performance change, our approach shows which

1 `https://soplex.zib.de/`, last accessed on 03/29/2023.

configuration options or interactions among them are actually affected by the performance change. This way, performance changes can be spotted and addressed early on by developers instead of manually analyzing the regressions to detect which configuration option or interaction is actually affected.

To assess the feasibility and limitations of the approach, we apply it on the performance data of FastDownward to identify performance changes. Similar to Chapter 4, we use the configuration level (i.e., the performance of all configurations) and the option level (i.e., the performance of configuration options and interactions thereof). Thereby, we compare the performance changes found at the option level to the performance changes found at the configuration level. We use the configuration level as a point of reference.

To assess the quality of our approach, we focus on precision (i.e., how many of the detected performance changes at the option level are visible at the configuration level?) and recall (i.e., how many of the actual performance changes at the configuration level were identified at the option level?). Thus, we aim at answering the following research questions:

- $RQ_{1.1}$: What is the fraction of identified performance changes at the option level that can be confirmed at the configuration level?
- $RQ_{1.2}$: What is the fraction of identified performance changes at the configuration level that can be confirmed at the option level?

It is important to note that our primary focus is not on achieving the best possible precision and recall rate, but to investigate wether and which performance changes cannot be identified. This way, we can reveal possible limitations of our approach.

Further, to assess which role workload variability plays in finding performance changes, we investigate in an additional research question, how many workloads are necessary to identify a performance change. This way, we can determine whether multiple workloads should be used while searching for performance changes. Therefore, we aim at answering the following research question:

- RQ_2: What is the fraction of workloads that are involved in identifying performance changes at the configuration level?

In summary, our contributions in this chapter are as follows:

- A novel analysis approach to identify performance changes of configuration options across workloads and different configuration spaces.
- Application of our analysis on the configurable software system FastDownward (following the case-study research method) measuring 49 configurations, 56 different workloads, and 6 releases of FastDownward.
- Insights into the limitations of our approach by assessing the recall and precision of our approach.
- Insights into which role workload variability plays in the presence of configurability and software evolution.

In a nutshell, we identified 7 464 performance changes at the configuration level and 2 674 performance changes at the option level, leading to a precision of 88.2% and a recall of 59.4%. In a more thorough investigation, we could determine factors that considerably influence the outcome of the application of our approach for finding performance changes. Further, most performance changes (92.4%) could be identified only in a subset of the workloads and one performance change could even be identified by only 4 (7%) of the workloads.

Our results have multiple implications: Most notably, we can confirm the role that workload variability plays in the evolution of configurable software systems. We found multiple cases where only some specific workloads could identify a performance change. In only 7.6% of the performance changes all workloads identified the performance change. Another notable insight is that most performance regressions are not fixed in the following release, but, at least, 2 releases later. Further, we have helped to uncover 3 persisting performance regressions in FastDownward. All results along with our analysis scripts are available at a supplementary web site[2].

5.1 Related Work

Before we discuss workloads in more detail in Section 5.2, we discuss related work in this section. In particular, we discuss related work with the focus on workload variability, since we have already discussed related work of configurability in Chapter 3 and related work of configurability and evolution of configurable software systems in Chapter 4.

There is a substantial corpus of work studying the phenomenon of workloads on the performance of a software system [3, 54, 79, 83, 103, 118]. In an empirical study using x264 with 17 different workloads, Pereira et al. [118] emphasize the role of workload variability in configurable software systems on different non-functional properties, such as performance. That is, Pereira et al. demonstrate that different workloads can change the behavior of the configurable software in such a way that the outcome of the software changes drastically for all software configurations. In subsequent studies, Lesoil et al. [79] and Mühlbauer et al. [103] both investigated the effect of multiple workloads on the performance of different configuration options using different machine-learning techniques. Both studies show examples where the influence of a configuration option is different depending on the workload. That is, by using one workload, a certain configuration option might be very influential on the overall performance of the software; using another workload might well nullify or even negate this influence. These studies emphasize that the selection of workloads for performance measurements cannot be neglected in particular for configurable software systems. To counteract, other studies devise approaches to account for workload variability for configurable software systems [54, 83]. Jamshidi et al. [54] propose an approach for transfer learning, where the idea is to minimize the effort of applying performance prediction on configurable software systems by using and extrapolating existing performance measurements. Since their approach is very general, their approach cannot only be applied for different workloads, but also on different hardware or software versions. Liao et al. [83] propose an approach for predicting the performance for configurable software systems using varying workloads. That is, their approach is able to predict the performance of even unseen workloads. This is established by relying on log files that include the operations that had to be performed by each workload. For instance, a log file of a web server would indicate which web requests have been executed. The line `GET /index.html` indicates that the web page `index.html` has been accessed. Serving

2 `https://github.com/ChristianKaltenecker/PerformanceEvolutionAndWorkloads_WebSite`, last accessed on 07/25/2023.

this web page (i.e., sending it from the server to the client) takes a certain time. So, the idea is to describe each workload by the actions in the log file. Instead of predicting the influence on the performance of a workload as a whole, Liao et al. use the operations caused by a workload to predict the performance. Although this approach was successfully used to find performance regressions for a mail server (i.e., Apache James) and an open-source health care system (i.e., OpenMRS), this approach is limited to subject systems which log their performed actions into a log file.

Researchers already investigated the effect of workloads on the performance of a software system and even interactions among configurability and workloads in program verification [67] or algorithm selection [70]. In these domains, approaches to learn performance models that map a configuration and a workload to its performance value with the ability to generalize over workloads have been devised and successfully applied [67, 70].

Other publications concentrate only on the workload dimension. For instance, Kounev et al. [71] and Menascé et al. [96] present how to characterize and design different workloads. Moreover, Kounev et al. [71] provide an overview of the different quality attributes of benchmarks based on past literature. Among others, they provide insights into the *relevance* of a benchmark [51, 143].

Alcocer et al. [3] focus on studying the performance evolution of 19 software systems' releases by analyzing the performance while using multiple workloads. They found, that one third of the releases introduces performance bugs and identify 9 patterns for performance changes. However, they measure multiple releases of software systems and workloads, but do not consider different software configurations.

To the best of our knowledge, the interplay of all three dimensions (i.e., configurability, evolution, and workload variability) has not been studied before.

5.2 Workloads

Software systems typically process one or more workloads and need a certain processing time, which we refer to as performance. By workloads, we refer to user-specified input that is fed into the software system. In many cases, workloads are executed by benchmarks, which are special tools designed to test and compare software systems [71]. For instance, a compression program is usually applied on a certain user-specified file with the goal to reduce the size of the file through the compression process. As another example, we refer to the setting that we have used in Section 4.3, in which a benchmark such as PolePosition or sysbench can be used to test the performance of SQL databases such as MySQL, MariaDB, or PostgreSQL using a fixed number of SQL queries.

The selection of the "right" workload is typically not trivial because (1) the theoretical number of workloads is infinite—the configuration space and number of releases are finite[3]—and (2) it is difficult to assess which workloads depict the user's behavior best [56, 71, 143]. According to Kounev et al. [71], one of the most important characteristics of a benchmark and its workloads is *relevance*, which depicts how similar the benchmark behavior mimics the behavior of users in the real world. If a benchmark and its workload do not mimic the

3 The Linux kernel is one of the largest configurable systems. Although the configuration space was recently estimated to offer about 10^{6000} valid configurations [93], the set of configurations are finite.

behavior of users, the results cannot be used for further optimization of the software system. Benchmarks that mimic the behavior of users accurately typically do not scale [56, 143]. Consequently, while selecting a suitable benchmark and its workloads, the trade-off between real-world behavior and scalability has to be taken into account. Moreover, in many cases, there are no default workloads (as, for instance, in the subject systems in Section 4.4.3), but rather an always increasing number of benchmarks that are meant to mimic real-world workloads. However, the absence of default workloads and the high number of real-world benchmarks makes it especially difficult to select workloads that depicts the real world best. Instead, such workloads are constructed from usage patterns [16], by comprising multiple different workloads [56], or by real-world inspired examples [46, 149]. For instance, workloads for compression programs stem from the real world (e.g., source code of open-source software, blog articles, music) and can even be combined into one single file. Another way to produce workloads for compression is to use file generators such as UIQ2, which we have used previously in Section 4.3.3. UIQ2 uses knowledge from the compression algorithm to create an average compressible file of an arbitrary size. Using such an average compressible file might be a good decision in many cases, but does not necessarily depict a real-world workload. This is because the compressibility of a file depends on patterns such as repeating words. A compression program exploits such patterns by encoding these occurrences in a representation that uses less bits and, thus, reduces its size. For example, files containing source code of a program are easily compressible since they contain reappearing keywords; images, however, are difficult to compress since the image formats (e.g., JPEG or PNG) store the picture information already in a compressed format.

When comparing workload variability to configurability and evolution of software, workload variability is known to interact with the other dimensions (i.e., a workload might perform faster or slower while changing the configuration and/or the release of the software system), as pointed out in related work [79, 103] (see Section 5.1). Consequently, workloads are important means that have to be considered while investigating performance changes. Since considering only one workload was one threat to validity in Chapter 4, we address this threat in this chapter by suggesting a novel approach for the analysis of performance changes across configurations, releases, and workloads. Therefore, we denote $\mathcal{W}$ as the set of all workloads of a software system for the remainder of this work.

For our study in this chapter, we use the approach in Algorithm 5 to learn the performance-influence models and assess the performance changes. In essence, this algorithm is based on Algorithm 4 in Section 4.2 with a few changes, which we describe next. The algorithm needs another argument—*workloads*—containing the set of measured workloads of the configurable software system. The last argument—*data*—correspondingly contains multi-dimensional data including releases, workloads, configurations, and the corresponding performance value in this algorithm. In a similar vein, we are now iterating over the different workloads in Lines 5–8. That is, we are now learning performance-influence models over different releases and workloads, and collect the relevant terms. A term is relevant if the term has an influence on the performance in, at least, one workload. After learning the performance-influence models, we apply the variance inflation factor analysis [28] for each workload and release in Lines 10–14 and remove perfect multicollinear terms. We perform the variance factor analysis for each workload and release since the configuration spaces among the workloads and releases might differ (i.e., some configurations are not available in the respective release).

Algorithm 5: *Learning a performance-influence model over consecutive releases and different workloads*

```
1  Function learn_comparable_models_over_workloads(feature_model, releases, workloads, data):
2      terms ← ∅
3      models ← ∅
4      foreach release ∈ releases do
5          foreach workload ∈ workloads do
6              learned_model ← learn_model(feature_model, data[workload][release])
7              terms ← include_terms_from_model(terms, learned_model)
8          end
9      end
10     foreach release ∈ releases do
11         foreach workload ∈ workloads do
12             terms[workload] ← variance_factor_analysis(terms, data[workload])
13         end
14     end
15     foreach release ∈ releases do
16         foreach workload ∈ workloads do
17             fitted_model ← fit(terms[workload], data[workload][release])
18             models ← models ∪ {fitted_model}
19         end
20     end
21     return models
```

This typically affects not only specific configurations, but inherently configuration options. For instance, the linear program solver SOPLEX[4] was only integrated in a release at the end of 2021 and, thus, is completely missing in some releases of our data. To avoid considering absent features in these releases, we remove such configuration options by executing the variance factor analysis (see Section 2.2.4) for each workload in Line 12 [75, 104]. This enables us to focus on configuration options as soon as they are integrated into the configurable software system.

In the final step in Lines 16–19, we learn performance-influence models for the remaining terms. This way, we obtain unambiguous and comparable performance-influence models for each pair of workload and release. This step is similar to the approach in Chapter 4.

5.3 Experiment Setup

In this section, we discuss and motivate our research questions and how we evaluate them. In detail, we first present the research questions in Section 5.3.1. In Section 5.3.2, we present the case study FASTDOWNWARD and the corresponding selection of the configurations, workloads, and releases. Later, in Section 5.3.3, we describe how we answer our research questions.

5.3.1 Research Questions

Our overarching goals in this chapter are (1) to assess the feasibility and limitations of our approach based on performance-influence models for finding performance changes across configurations, releases, and workloads and (2) to investigate the interplay between workload

4 https://soplex.zib.de/, last accessed on 03/29/2023.

variability, configurability, and software evolution. Due to the exploratory nature of our study, we follow the *case-study research method* [127], which is an initial investigation on the considered phenomena. Runeson et al. [127] define the case-study research method in software engineering as follows:

> Case-study in software engineering is an empirical enquiry that draws on multiple sources of evidence to investigate one instance (or a small number of instances) of a contemporary software engineering phenomenon within its real-life context, especially when the boundary between phenomenon and context cannot be clearly specified.

The case-study research method is primarily used for exploratory purposes to seek new insights. In other words, the focus of this research method is more in-depth and qualitative by focusing on a single case study than a broad and quantitative investigation on a higher number of case studies as we previously performed it in Chapter 4. The case-study research method is used in multiple similar publications, such as Grebhahn et al. [36, 37].

In this chapter, we adopt this method by applying our approach on finding performance changes across configurations, releases, and workloads by focusing on a single subject system. This way, we decrease our external validity but improve our internal validity since this approach enables us to investigate the identified performance changes in much more detail than in Chapter 4.

Our Approach In this chapter, we now inspect the reported performance changes in terms of precision and recall. The precision and recall analysis enables us to confirm the identified performance changes or, alternatively, identify further improvements and limitations of our approach. Similar to Chapter 4, we evaluate our approach on two different abstraction levels—*configuration level* and *option level*. That is, we assess between each consecutive release and for each workload which configurations and which configuration options/interactions change. Afterwards, we use the identified changes at the configuration level as a point of reference to check whether and when our performance-modeling approach detects performance changes. Although we demonstrated the feasibility of our approach in Section 4.4.1, we do not know about the precision and recall of our approach and, thus, how many of the performance changes are correctly identified and in how many cases we miss performance changes. To close this gap, we focus at the option level to identify performance changes of configuration options across workloads.

In a first step, we aim at identifying the precision of our approach to detect performance changes in configuration options across workloads. To obtain this information, we learn performance-influence models for each release and workload. Afterwards, we compare their terms and coefficients (see Section 2.2.4). Similar to Chapter 4, multicollinearity might occur while learning performance-influence models. To mitigate this, we apply a variance inflation factor (VIF) analysis and remove terms with perfect multicollinearity (see Section 2.2.4 and Section 5.2). Then, for each workload, we compare the performance-influence models across all releases. This allows us to gain more information on the performance changes of configuration options across releases and workloads. To ultimately assess the precision of our approach, we compare the identified performance changes by our performance-modeling approach from the option level to the performance changes at the configuration level. That is, we assess in how many cases we can confirm the performance changes identified by our approach. This provides us with the possibility to investigate cases for pointing out limitations of our proposed approach, similar to other exploratory studies (e.g., Grebhahn et al. [36]).

Research Question 1.1

What is the fraction of identified performance changes at the option level that can be confirmed at the configuration level?

In a second step, after determining the precision of our approach, we assess the recall of our approach. That is, we now assess how many of the performance changes at the configuration level are covered by our approach at the option level. This way, we can point out cases in which our approach is unable to identify an performance change as such.

Research Question 1.2

What is the fraction of identified performance changes at the configuration level that can be confirmed at the option level?

Workload Variability Since we consider the workload variability in addition to configurability and software evolution from Chapter 4, we investigate how many workloads could detect a performance change at the configuration level. Although this does not reveal any more knowledge about our approach, this investigation enables us to assess the role of workload variability while identifying performance changes. For instance, we could find out whether all workloads identify all performance changes, or some workloads identify some performance changes. Investigating the role of workload variability in FastDownward is necessary to assess whether the observations made in other studies [79, 103] apply in FastDownward. Such insights clarify whether it suffices to measure only one workload rather than a variety of different workloads. Note that for investigating workload variability, we use the performance changes at the configuration level instead of performance changes at the option level due to the fact that our approach at the option level could miss some performance changes.

Research Question 2

What is the fraction of workloads that are involved in identifying performance changes at the configuration level?

5.3.2 Fast Downward

In this chapter, we introduce FastDownward[5] as our subject system for our evaluation. FastDownward is a classical planning software based on different search heuristics [46]. In essence, FastDownward is used to solve deterministic planning problems, such as transportation problems [46] or the towers of Hanoi [156]. A planning problem defines an initial state and a final state (i.e., the goal) that has to be reached. Further, a planning problem belongs to

5 https://www.fast-downward.org/, last accessed on 04/23/2023.

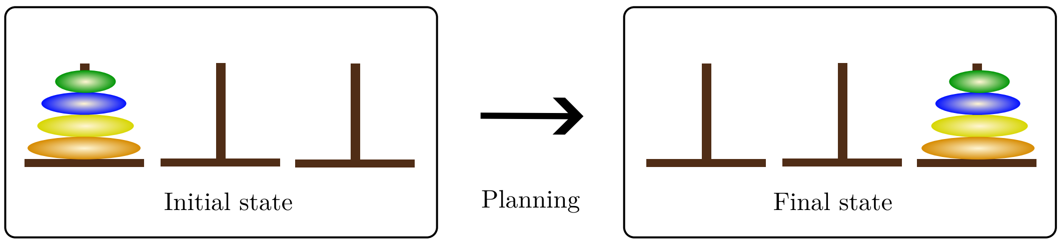


Figure 5.1: The tower of Hanoi planning problem. The left side presents the initial state and the right side the final state that has to be reached by the planner.

a *domain* and a domain can contain multiple planning problems. A domain typically defines different actions to change the state with a certain cost. The planner then has to find a series of actions to reach the final state from the initial state of the problem. The planner cannot only be used to find a series of actions to reach the final state, but also to find an optimal solution with respect to the cost. Each planning problem represents a workload.

For demonstration, we refer to the tower of Hanoi planning problem in Figure 5.1. The tower of Hanoi is a planning problem with multiple pegs and disks; three pegs and four disks in our case. These disks are initially arranged on one peg as we show in Figure 5.1, arranged bottom-up in descending size order. The goal is to arrange all disks in the same order on another peg. The initial state and the final state in Figure 5.1 represent the planning problem. To reach the final state, the domain defines that one disk can be moved from one peg to another if it fulfills multiple constraints [49]:

- Only a single disk can be moved at a time.
- Only the top disk on any peg can be moved.
- Larger disks cannot be stacked on smaller disks.

In essence, each movement represents an action. In an optimal solution for this planning problem, a minimal set of movements has to be found to reach the final state. That is, a planning system can be applied on this planning problem to find the minimal number of steps to reach the goal. The tower of Hanoi planning problem can then be further adjusted by adding more pegs and disks. These different problems of the tower of Hanoi belong to the same domain, but have different difficulties.

A planning problem from another domain is the transportation planning task [46] that represents package delivery on a street network represented by a graph. We show the transportation planning task in Figure 5.2. The graph consists of vertices and edges. The vertices represent the drop-off and pick-up points of the packages and the edges represent the streets connecting the drop-off or pick-up points or cities. The goal of this planning task is to deliver the packages at their destination. To reach this goal, the planner can perform different actions:

- Move a car from one node to another if the nodes are connected and the nodes are in the same city.
- Move a truck from one city to another if the cities are connected.
- A car/truck picks up a package if the current node contains a package.

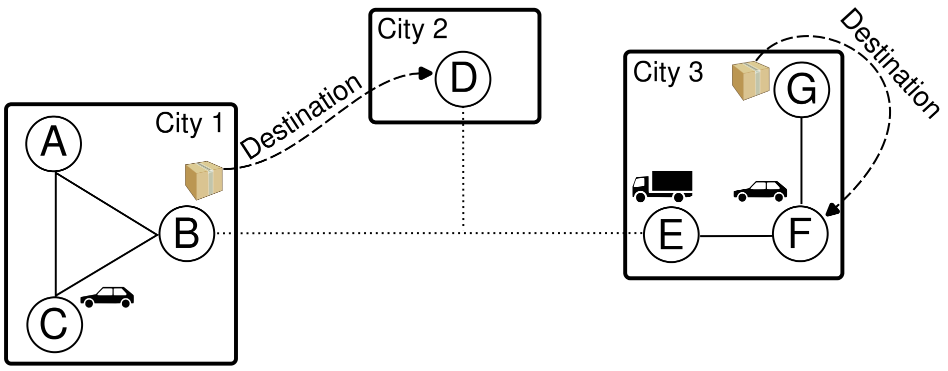


Figure 5.2: The transportation problem for package delivery consists of multiple cities interconnected by roads (dashed line), interconnected pick-up and drop-off points in the cities, cars for the transportation inside the city, trucks for the transportation between cities, and boxes with a certain destination.

- A car/truck drops off the carried package at the current node.

Each action has a certain cost. Similar to the towers of Hanoi, the optimal solution would utilize as less actions as possible. For example, the only action for the package in City 1 from Figure 5.2 is to transport it with a truck from City 1 to City 2. Every other choice that involves this package would not result in an optimal solution. The transportation problem instance from Figure 5.2 is only one instance; further instances differ in the number of cities, nodes, cars, packages, packages' destinations, and connections. This means, each different setting forms a different workload.

Planning systems such as FastDownward can be applied to find any solution or even the optimal solution of such planning problems. Although multiple different planning systems can find an optimal solution, the time until the solution is determined, plays a crucial role. One example are planning competitions [149], which compare different planning software in terms of different characteristics, such as performance. Such competitions emphasize the role of performance of such planning systems. Our contribution to identify performance changes can help by enabling us to find performance regressions in FastDownward. Such performance regressions could then be addressed by the developers of FastDownward to pinpoint or even improve the performance of the planning system.

Experiment Setup

For our experiment setup and our evaluation, we employ domain knowledge by contacting developers of FastDownward[6]. This way, we obtain access to domain knowledge, which represents a better means than the official documentation for selecting the configurations and workloads for FastDownward. Our overall goal is to find performance changes across configurations, workloads, and releases.

Configuration selection One important fact about FastDownward that further emphasizes the role of configurability is that the planning software does not represent a single

6 The developers that help us in our experiment setup are Jendrik Seipp, Silvan Sievers, and Florian Pommerening.

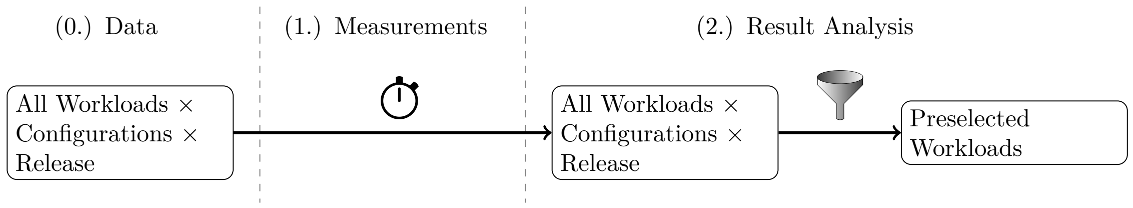

(0.) Data
(1.) Measurements
(2.) Result Analysis
All Workloads × Configurations × Release
All Workloads × Configurations × Release
Preselected Workloads

Figure 5.3: Overview of our preliminary measurements to select a suitable set of workloads. First, we measure more than 1 800 workloads and configurations for a specific release. Then, we use the results to filter out workloads that are too slow (i.e., run into a timeout for most configurations) and select one workload from each domain.

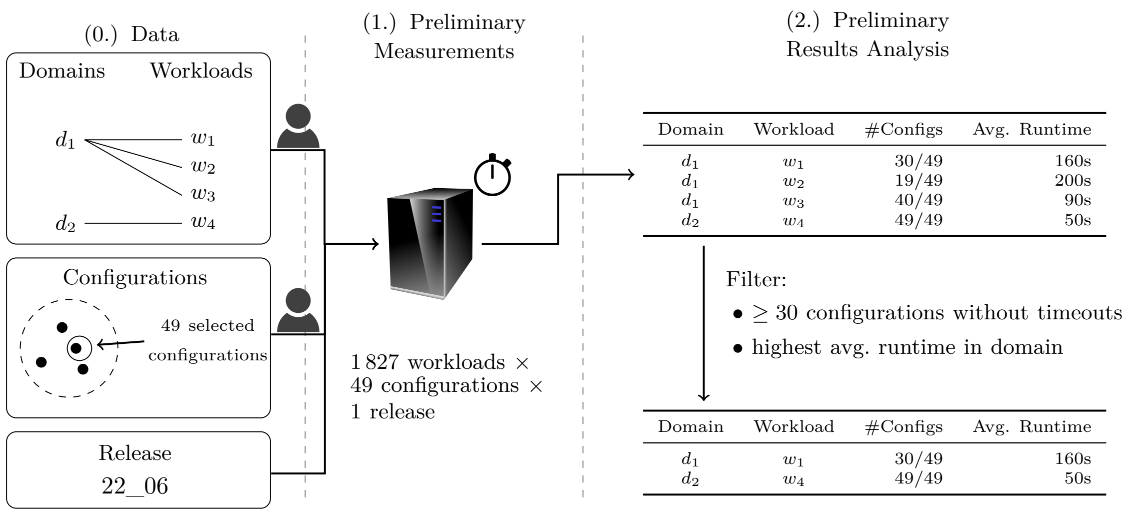
planning heuristic, but can be thought of as a collection of different planning heuristics. In other words, FastDownward is a configurable software system that allows the user to select between different heuristics for finding a solution. Furthermore, these heuristics can be fine-tuned by providing further parameters specific to the heuristic. Since our aim is to measure different configurations of different releases and several workloads, we reduced the number of configurations to a reasonable number. Therefore, we contacted the developers of FastDownward and selected 49 different configurations that cover different heuristics and different performance-relevant functionality in heuristics (e.g., we used two different linear program solvers CPLEX[7] and SoPlex[8]). Note that these 49 configurations were chosen in a collaboration with the three developers (i.e., all three developers were involved while selecting the configurations). In summary, we covered 11 different heuristics suggested by the developers. Furthermore, some heuristics offer random seeds that can influence their performance. We decided to use only a single random seed since our aim is not to assess how stable heuristics are with regards to random seeds, but to cover different performance-relevant source code. Choosing other random seeds would not change the amount of performance-critical source code that is executed. In the end, only 2 out of 11 heuristics use a random seed, which appear in 4 out of 49 configurations. We show the feature model of FastDownward in Figure A.1 in the appendix.

Another heuristic (i.e., counter-example-guided abstraction refinement [25]) allows for multiple combinations of parameters. In this case, we decided to measure each parameter in isolation, which enables us to pinpoint possible performance changes in these parameters more easily.

The parameters of the heuristics can also be used to tune the heuristic to the workload. We decided to leave this out since our aim is to get a general view of performance changes and, thus, use the default values where possible. This way, we achieve a similar setup to planning competitions [149]. Note that default values might change across different releases of the software and therefore, could be the cause for a performance change. To rule out performance changes induced by changes to the default values, we compared the documentation containing every heuristic and its parametrization and assured that the default values stayed the same between releases.

In our measurements, we investigate all releases of FastDownward from January 2019 to December 2022. In total, this results in 6 different releases in the investigated time period.

7 `https://www.ibm.com/de-de/analytics/cplex-optimizer/`, last accessed on 04/23/2023.
8 `https://soplex.zib.de/`, last accessed on 04/23/2023.


(0.) Data
Domains
Workloads
d_1
d_2
w_1
w_2
w_3
w_4
Configurations
49 selected configurations
Release
22_06
(1.) Preliminary Measurements
1 827 workloads ×
49 configurations ×
1 release
(2.) Preliminary Results Analysis
Filter:
• ≥ 30 configurations without timeouts
• highest avg. runtime in domain

Domain	Workload	#Configs	Avg. Runtime
d_1	w_1	30/49	160s
d_1	w_2	19/49	200s
d_1	w_3	40/49	90s
d_2	w_4	49/49	50s

Domain	Workload	#Configs	Avg. Runtime
d_1	w_1	30/49	160s
d_2	w_4	49/49	50s

Figure 5.4: Detailed overview of our preliminary measurements to select a suitable set of workloads. In the initial Step (0.), we select the workloads, configurations, and the release of FastDownward we measure. Note that each workload belongs to one planning domain. We highlighted the dimensions with developer involvement using the symbol 👤. We measure all workloads, given configurations in the specified release of FastDownward in Step (1.). Afterwards, in Step (2.), we analyze the results and obtain information about the fraction of configurations without a timeout (column #Configs) and the average runtime over all configurations without timeout. Finally, we filter these results by selecting workloads for which (1), at least, 30 configurations do not have timeouts and (2) which have the highest average runtime in the domain.

Workload Preselection After determining the configurations and releases in our performance measurements, we discuss the workloads next. Our goal is to shrink the number of workloads to a suitable number for our final performance measurements. Due to the high number of workloads, the workload preselection cannot be done with domain knowledge, but has to be done using preliminary measurements. For further illustration, we show this workload selection process in Figure 5.3. Overall, we first measure all configurations and all workloads in Step (1.). After measuring the data, we use the results to filter out workloads running into a timeout and to select a set of workloads fulfilling our criteria, afterwards.

In detail, we show the workload selection process in Figure 5.4. In the case of FastDownward, each planning problem represents a workload. Each workload belongs to exactly one domain (e.g., towers of Hanoi or transportation planning problem), as shown in Step (0.) in the figure. A large number of different workloads are comprised in a repository[9]. In summary, the repository contains 4 330 different planning problem instances; each associated to one of 133 domains. To reduce the number of domains, we focused on domains where an optimal solution (e.g., the minimal number of movements in tower of Hanoi) has to be found. The rationale behind this filter is that the performance on problems where an optimal solution has to be found is much more stable with regard to performance than for finding an arbitrary solution. After applying the filter, we had 1 827 different planning problem instances in 65 domains. Since this still results in an amount that is computationally infeasible to measure, we had to further reduce the number of workloads. This, however, was not possible without

9 `https://github.com/aibasel/downward-benchmarks/`, last accessed on 04/23/2023.

executing all configurations on all heuristics, at least, once, to assess which workloads result in a timeout of 5 minutes and which find an optimal solution (depicted in Step (1.)). The timeout was suggested by one of the developers that helped us and corresponds to the typical setup following in planning competitions [149]. It is important to note that the different heuristics of FastDownward perform quite different on the workloads. Some heuristics find an optimal solution in seconds, whereas other heuristics fail to find an optimal solution in 5 minutes[10]. The performance of a heuristic on a workload cannot be fully determined by the developer. Therefore, we executed all configurations of the latest release (i.e., December 2022 – Release 22.12[11]) on all workloads, resulting in 89 524 preliminary measurements. Afterwards, in Step (2.), we selected the workloads where, at least, 30 configurations found a solution in the specified time frame of 5 minutes. Furthermore, we selected the workload with the highest average execution time from each remaining domain. The rationale behind selecting the workloads with the highest average execution time is that many runs achieved an average execution time of less than 1 second. Since our performance measurements contain measurement noise [71], it is less likely to find performance changes in workloads with a low execution time than in workloads with a higher execution time. In 9 out of 65 domains, our filter could not find any fitting problem because most of the configurations timeouted in all workloads of these domains. Overall, we identified 56 different workloads from 56 different domains for our measurements.

In summary, we selected 49 configurations, 6 releases, and 56 workloads of FastDownward for our further measurements and repeat each execution 5 times to counteract to measurement noise [71].

Timeout for Final Measurements Previously, we used a timeout of 5 minutes in the preliminary measurements. To further reduce the amount of configurations running into a timeout in our final measurements, we set the timeout of the final measurements to 15 minutes. Despite increasing the timeout for our final measurements, some configurations can still indicate a timeout in some workloads and releases and should be considered in our performance-modeling approach. If timeouts would not be considered, we would certainly miss important performance changes. In theory, timeouts represent a form of censored data, in which we know that a configuration needs, at least, a certain time (i.e., at least, 15 minutes), but cannot determine exactly how much time the configuration needs. In predictive modeling, censored data is typically handled as missing data, which can be treated in multiple ways [75]. One possible treatment is to remove the data of some configurations for some workloads. Removing the data of some configurations, however, results in even more different configuration spaces. Consequently, it might happen that we miss data on whole configuration options in some workloads due to timeouts and, thus, would ignore it. Another possibilty is to use imputation, where the missing data is replaced by another value (e.g., mean performance or median value) [5]. In our setting, we opted to impute a value higher than our timeout (i.e., 30 minutes). This enables us to identify regressions if the execution time of some configurations is close to a timeout in one release and timeouts in another, which would not be possible in the case of mean performance values.

10 Not only the instance size, but also special workload characteristics might turn a workload unsolvable for a heuristic. For instance, some greedy heuristics might not perform well with actions with no cost.

11 https://github.com/aibasel/downward/releases/tag/release-22.12.0/, last accessed on 04/23/2023.

Performance Measurements We executed all measurements on computing nodes using the Non-Uniform Memory Access (NUMA) architecture equipped with two Intel Xeon E5-2630 v4, 256 GB of RAM and an NVME SSD. Since the computing nodes consist of two Intel processors, each with its own RAM bus, further measurement noise could arise while using both processors and RAM buses interchangeably. To reduce measurement noise, we limited our computing resources on using the first processor with its associated RAM bus. Our preliminary measurements were executed with no repetitions because of the high number of measurements. All other measurements were executed with 5 repetitions and were repeated again if the relative standard deviation was higher than 10% or the performance was lower than 0.1 seconds. The incentive behind this is that we expect our performance measurements to indicate a certain level of measurement noise, which is especially present in very short measurements.

5.3.3 Operationalization

To answer our research questions, for each release, (1) we measure the configurations of FastDownward specified in the previous section and (2) learn a performance-influence model on the entire set of configurations, resulting in one model per workload and release. For the subject system FastDownward, $\mathcal{C}_{\text{FastDownward}}$ refers to the set of configurations (see Section 2.1), $\mathcal{R}_{\text{FastDownward}}$ to its set of releases, and $\mathcal{W}_{\text{FastDownward}}$ to its set of workloads. Note that we will omit the subscript FastDownward in the following definitions for brevity.

Configuration Level We use the configuration-level measurements as our point of reference for finding performance changes. Although we might miss some performance changes throughout the history of FastDownward, we contacted the developers to cover the most important functionality. Compared to other subject systems in this thesis, the smaller configuration space of 49 configurations in FastDownward also enables us to inspect our approach in more detail, which is the ultimate goal in our exploratory study.

$\mathcal{M}^{r,w} : \mathcal{C} \to \mathbb{R} \cup \{\bot\}$ maps the configurations $c \in \mathcal{C}$ of release $r \in \mathcal{R}$ and workload $w \in \mathcal{W}$ to their *measured* real-valued performance values or to $\bot$ in case the configuration does not exist in the release, yet. In contrast to Chapter 4, where $\mathcal{C}$ was fixed across all releases of the subject systems, we change this for FastDownward in this chapter since not all configurations are available in all releases. Overall, this affects the first 3 out of 6 releases, where not all configurations based on linear program solvers are available. In the first two releases (releases 19.06 and 19.12), measured only 39 out of the 49 selected configurations, and in the third release (release 20.06), we measured 44 out of 49 configurations.

We consider a performance change between a configuration of two consecutive releases relevant if:

$$|\mathcal{M}^{r_i,w_j}(c) - \mathcal{M}^{r_{i+1},w_j}(c)| > 2 \cdot \max(\text{sd}^{r_i,w_j}(c), \text{sd}^{r_{i+1},w_j}(c)), \tag{5.1}$$

where $\text{sd}^{r,w}(c)$ denotes the standard deviation of performance values of a configuration across repeated measurements. In other words, if a performance change does not exceed twice the larger standard deviation of the two releases, it is not further considered. The rationale for this conservative threshold is to filter out measurement noise and tiny performance changes.

The independent variables for the configuration-level data used as a point of reference in $RQ_{1.1}$ and $RQ_{1.2}$ and used in RQ_2 are the three dimensions: (1) the configuration $c \in \mathcal{C}$, (2) the release $r \in \mathcal{R}$, and (3) the workload $w \in \mathcal{W}$. The dependent variable is the performance value $\mathcal{M}^{r,w}(c)$. As a confounding factor, measurement noise caused by particularities from the software and hardware biases our results [71, 105]. To counteract, we repeat each measurement 5 times until the coefficient of variation (i.e., standard deviation divided by the mean) of the repetitions is lower than 10% or the performance is lower than 0.1 seconds.

Option Level Our performance-modeling approach allows us to detect performance changes at the option level, as formerly demonstrated in Chapter 4. Still, our goal in this exploratory study is to find out in how many cases we can detect the performance changes and in which cases we do not.

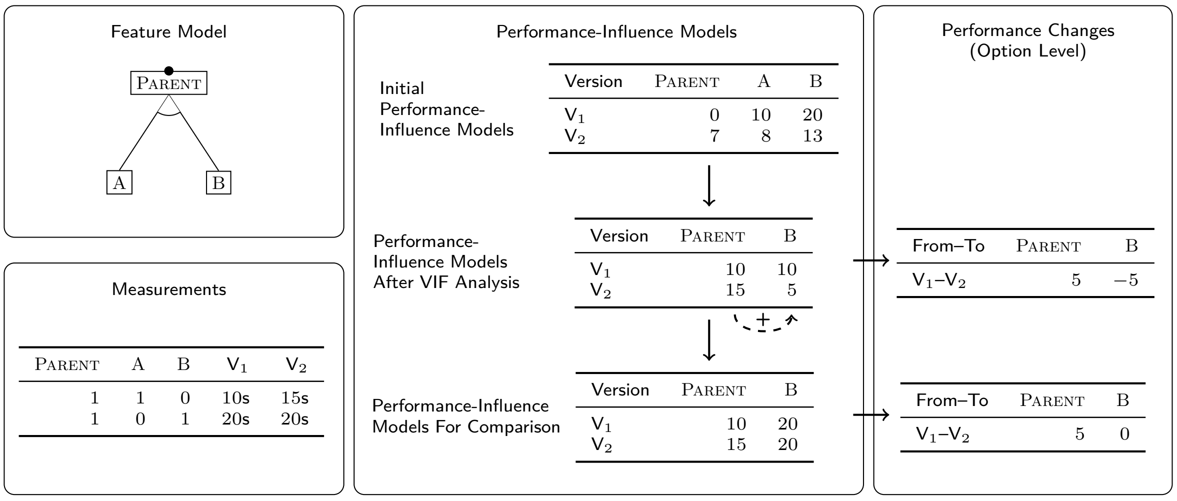
Formally, $\Pi^{r,w}$ denotes the performance-influence model for revision $r \in \mathcal{R}$ and workload $w \in \mathcal{W}$ of FastDownward. Further, $\mathcal{T}_{\Pi^{r,w}}$ denotes the terms of the performance-influence model $\Pi^{r,w}$. Note that we do not follow a sample-based learning approach (i.e., one that uses only a subset of configurations as in Chapter 3). Instead, we learn models on the whole configuration space defined together with the developers. This would be impractical in practice but gives us the most accurate results since applying a sampling approach would likely miss important configuration options such as heuristics. So, the independent variables for our performance-modeling approach from the option level investigated in $RQ_{1.1}$, $RQ_{1.2}$ are (1) the workload $w \in \mathcal{W}$ and (2) the release $r \in \mathcal{R}$; the dependent variable is the corresponding performance-influence model $\Pi^{r,w}$ for $r \in \mathcal{R}$ and $w \in \mathcal{W}$.

To find the performance changes at the option level, we determine for each $r \in \mathcal{R}$ and $w \in \mathcal{W}$ the performance influences $\beta^{r,w}(t)$ of all terms $t \in \mathcal{T}_{\Pi^{r,w}}$. A term can either consist of the base term (i.e., β_0 in Section 2.2.4), a configuration option (i.e., $\beta_o \cdot c(o)$ for $o \in \mathcal{O}$), or an interaction among multiple options (i.e., $\beta_{o_1..o_i} \cdot c(o_1) \cdot \cdots \cdot c(o_i)$ for $o_1, \dots, o_i \in \mathcal{O}$). Function $\beta^{r,w}(t)$ returns the coefficient of the term. We consider a performance change between two coefficients relevant if:

$$|\beta^{r_i,w_j}(t) - \beta^{r_{i+1},w_j}(t)| > 2 \cdot \max(\overline{\mathrm{sd}}^{r_i,w_j}, \overline{\mathrm{sd}}^{r_{i+1},w_j}), \tag{5.2}$$

where $\overline{\mathrm{sd}}^{r_i,w_j}$ denotes the mean standard deviation of all configurations of release $r_i \in \mathcal{R}$ and workload $w_j \in \mathcal{W}$. If a change of performance influence does not exceed twice the larger average standard deviation of the two releases, it is not further considered. The rationale of using the *maximum* of the *mean* standard deviation is that we use the entire configuration space for learning performance models and thus accumulate the standard deviation over all configurations.

Reducing Wrong Detections Before diving into the operationalization of the research questions, we discuss an issue that appears with optional configuration options and alternative groups in FastDownward (see corresponding feature model in Figure A.1). Through this issue, a direct comparison of performance-influence models would produce too many invalid performance changes. For the demonstration of the issue, we use the example in Figure 5.5 with an alternative group. There, we demonstrate a general problem that arises with mandatory alternative groups (see top left of the figure). In the bottom left, we show exemplary performance measurements using the alternatives A and B. Deriving a feature


Feature Model
Parent
A
B
Measurements
Performance-Influence Models
Initial Performance-Influence Models
Performance-Influence Models After VIF Analysis
Performance-Influence Models For Comparison
+
Performance Changes (Option Level)

Measurements

Parent	A	B	V_1	V_2
1	1	0	10s	15s
1	0	1	20s	20s

Initial Performance-Influence Models

Version	Parent	A	B
V_1	0	10	20
V_2	7	8	13

Performance-Influence Models After VIF Analysis

Version	Parent	B
V_1	10	10
V_2	15	5

Performance-Influence Models For Comparison

Version	Parent	B
V_1	10	20
V_2	15	20

Performance Changes (Option Level)

From–To	Parent	B
V_1–V_2	5	−5

From–To	Parent	B
V_1–V_2	5	0

Figure 5.5: Minimal example for the adjustments performed on the performance-influence models. The issue appears in mandatory alternative groups (shown in the top left) and optional configuration options. In the bottom left, we show the according performance measurements (in seconds) for two releases V_1 and V_2 of this example. We show the performance-influence models in the center. The first performance-influence model represents the performance-influence model using multiple linear regression. Afterwards, we apply the VIF analysis to eliminate multicollinearity. Comparing the coefficients (right) of these performance-influence models would still detect performance changes in unaffected configuration options (i.e., configuration option B has not changed from V_1 to V_2) wrongly. To reduce these wrong detections, we add the values of Parent to B.

model from the performance data of releases V_1 and V_2 would result in the influence model in the top center. Each row contains one performance-influence model per release and each column the coefficients of the terms. Since the mandatory option Parent is multicollinear to A and B (i.e., Parent is only selected in configurations where either A or B are selected), the performance-influence models are no longer unique (see Section 2.2.4 for further information on multicollinearity). In particular, the coefficient values can shift between A and B, and the mandatory option Parent since Parent can be represented by A and B (i.e., either A or B are selected on when Parent is selected). So, Parent is perfectly multicollinear to A and B. We depict this shift in the first performance-influence model in Figure 5.5, where V_1 assigns 0 to Parent, whereas the value 7 is shifted from A and B to Parent in V_2. Note that we cannot exactly determine the influence of Parent, A, and B since there is no configuration where only Parent is selected and A and B are not. Overall, this value shifts would bias our comparison and is ultimately the reason for the application of the VIF analysis. However, the VIF analysis would remove either A or B to resolve the multicollinearity. In our case, the VIF analysis removes A. Note that now the only term that relates to the first configuration is term Parent. Also note, that because of the removal of A by the VIF analysis, B could also be represented as an optional configuration option. That is, B can either be selected or deselected. Typically, A would be selected if B is deselected. But, because of the removal of A, this is no longer the case from the data point of view. So, the following flaw does also apply on optional configuration options.

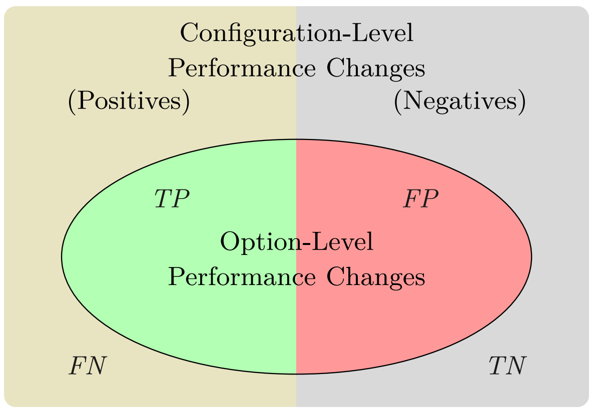


Figure 5.6: Visualization of *True Positives* (*TP*), *False Positives* (*FP*), *True Negatives* (*TN*), and *False Negatives* (*FN*). The positives (left) are the performance changes identified on the configuration level; the negatives (right) are performance changes that are not identified at configuration level. The ellipsis in the middle represents the identified performance changes by our approach at the option level.

The flaw that makes a subsequent comparison of the performance-influence models difficult is the following: Performance changes of the removed alternative are propagated to the other alternatives. This is because the column Parent relates to both configurations in the measurement data, but the performance of only one configuration changes. However, to model the performance change from 10 seconds to 15 seconds in the first configuration, the only term that addresses this configuration is Parent. But, to compensate that there is no performance change in the second configuration, the multiple linear regression now fits a value that is $10 - 15 = -5$ seconds smaller than in the previous release. A subsequent comparison of these performance-influence models would wrongly indicate that B changed. To solve the problem, we reverse this effect by adding the influence value of the term Parent to the other alternative terms for the comparison (see dashed arrow in the figure). This step invalidates our performance-influence models in general (i.e., the performance-influence models cannot be used for performance prediction anymore), but eliminates the described flaw in mandatory alternative groups and for optional options. These were the only modifications necessary to improve the performance-influence models $\Pi^{r,w}$ for the comparison to identify performance changes at the option level. By applying this approach in our analysis, we can reduce the number of invalid performance changes detected by our approach at the option level.

Research Questions After presenting the necessary changes for our comparison, we present the operationalization for the research questions, next. To provide a better understanding of precision and recall for $RQ_{1.1}$ and $RQ_{1.2}$, we show the *True Positives* (*TP*), *False Positives* (*FP*), *True Negatives* (*TN*), and *False Negatives* (*FN*) in Figure 5.6. In particular, we interpret the notions as follows:

True Positives (TP): The performance changes identified at the configuration level and the option level.

False Positives (FP) : The performance changes identified at the option level but not at the configuration level.

True Negatives (TN) : The performance changes identified neither at the option level nor at the configuration level.

False Negatives (FN) : The performance changes identified at the configuration level but not at the option level.

Note that we use the performance changes identified at configuration level as our positives since we use the performance changes identified at configuration level as the point of reference. Hence, the negatives are all performance changes that are not identified at configuration level.

$RQ_{1.1}$: Since we have already described how we determine the performance changes from the point of reference at the configuration level above, we now present how we assess the precision of our approach. The precision is a performance metric which is used to assess in how many cases a certain approach is correct. In other words, we determine the fraction of true positives (TP) in all found performance changes of the option level. That is, we assess in how many cases the performance change is correct.

For the changes at the configuration level, we denote $\mathcal{CC}^{r_i,r_{i+1}}$ as the set containing all configurations indicating a performance change between consecutive releases $r_i, r_{i+1} \in \mathcal{R}$ in, at least, one workload:

$$\mathcal{CC}^{r_i,r_{i+1}} = \{c \,|c \in \mathcal{C} \wedge \exists w \in \mathcal{W} : |\mathcal{M}^{r_i,w_j}(c) - \mathcal{M}^{r_{i+1},w_j}(c)| > 2 \cdot \max(\mathrm{sd}^{r_i,w_j}(c), \mathrm{sd}^{r_{i+1},w_j}(c))\}$$

For the changes at the option level, we denote $\mathcal{TC}^{r_i,r_{i+1}}$ as the set containing all terms indicating a performance change between consecutive releases $r_i, r_{i+1} \in \mathcal{R}$ in, at least, one workload and $\mathcal{T}_{\Pi^{r,w}}$ as the set of terms of the performance-influence model $\Pi^{r,w}$ of release r and workload w:

$$\mathcal{TC}^{r_i,r_{i+1}} = \{t\,|\,t \in \mathcal{T}_{\Pi^{r_i,w}} \cap \mathcal{T}_{\Pi^{r_{i+1},w}} \wedge \exists w \in \mathcal{W} : |\beta^{r_i,w}(t) - \beta^{r_{i+1},w}(t)| > 2 \cdot \max(\overline{\mathrm{sd}}^{r_i,w}, \overline{\mathrm{sd}}^{r_{i+1},w})\}$$

Note that we only check for terms that are present in both consecutive releases. This is necessary since some configurations are not available in some releases. Also note that we do not distinguish between performance changes that appear in one and performance changes that appear in multiple or all workloads in our research questions since we are interested in performance changes in general. However, we take a closer look into performance changes and the workloads in RQ_2.

We denote the function $\mathrm{affectedConfigs}^{r_i,r_{i+1}}$ as a function that maps a term (i.e., an option or an interaction) to the set of configurations corresponding to the term between releases $r_i, r_{i+1} \in \mathcal{R}$:

$$\mathrm{affectedConfigs}^{r_i,r_{i+1}} : \mathcal{T}_{\Pi} \rightarrow \mathcal{P}(\mathcal{C})$$

where $\mathcal{T}_{\Pi}$ represents the set of all terms that are relevant in, at least, one release:

$$\mathcal{T}_{\Pi} = \bigcup_{r_i, r_{i+1} \in \mathcal{R}} \mathcal{T}_{\Pi^{r_i,r_{i+1}}}$$

Next, to determine the confirmed performance changes, we split this set into multiple different sets: $\mathcal{TC}_{\text{TP}}^{r_i,r_{i+1}}$ denotes the confirmed performance changes (i.e., *TP* in Figure 5.6), and $\mathcal{TC}_{\text{FP}}^{r_i,r_{i+1}}$ denotes the performance changes that were wrongly identified as such (i.e., *FP* in Figure 5.6). We confirm the terms indicating a performance change by inspecting the affected configurations in the set $\mathcal{CC}^{r_i,r_{i+1}}$ using the function affectedConfigs:

$$\mathcal{TC}_{\text{TP}}^{r_i,r_{i+1}} = \{t \,|\, t \in \mathcal{TC}^{r_i,r_{i+1}} \wedge \exists c \in \text{affectedConfigs}^{r_i,r_{i+1}}(t) : c \in \mathcal{CC}^{r_i,r_{i+1}}\}$$

If none of the affected configurations is included in $\mathcal{CC}^{r_i,r_{i+1}}$, we mark the term as a false positive (i.e., *FP* in Figure 5.6):

$$\mathcal{TC}_{\text{FP}}^{r_i,r_{i+1}} = \mathcal{TC}^{r_i,r_{i+1}} \setminus \mathcal{TC}_{\text{TP}}^{r_i,r_{i+1}}$$

The precision over all releases is then assessed as follows:

$$\text{Precision} = \frac{\sum_{r_i,r_{i+1} \in \mathcal{R}} \text{card}(\mathcal{TC}_{\text{TP}}^{r_i,r_{i+1}})}{\sum_{r_i,r_{i+1} \in \mathcal{R}} \text{card}(\mathcal{TC}^{r_i,r_{i+1}})}$$

where card denotes the cardinality of a set.

$RQ_{1.2}$: Next, to assess the sensitivity of our approach, we determine the recall of our approach. The recall is another performance metric that assesses how many of all cases a certain approach covers. In other words, we determine all positives (i.e., all performance changes in our case) and verify, how many of them have been covered.

We define the function affectedTerms$^{r_i,r_{i+1}}$ as a function that maps a configuration to the terms that contain this configuration between releases $r_i, r_{i+1} \in \mathcal{R}$:

$$\text{affectedTerms}^{r_i,r_{i+1}} : \mathcal{C} \rightarrow \mathcal{P}(\mathcal{T}_{\Pi^{r_i,r_{i+1}}})$$

For instance, the map would return the terms B and Parent for the second configuration $\{\text{Parent}, \text{B}\}$ in the example of Figure 5.5. For the first configuration, it would return only Parent since this is the only term containing the configuration.

Similar to $\text{RQ}_{1.1}$, we split the set of all performance changes on configuration level $\mathcal{CC}^{r_i,r_{i+1}}$ into two sets: $\mathcal{CC}_{\text{TP}}^{r_i,r_{i+1}}$ denotes the set of all configurations that could be confirmed at the option level (i.e., *TP* in Figure 5.6), and $\mathcal{CC}_{\text{FN}}^{r_i,r_{i+1}}$ denotes the set of all configurations that could not be confirmed at the option level (i.e., *FN* in Figure 5.6). We confirm the configurations indicating a performance change by inspecting the terms corresponding to the configuration:

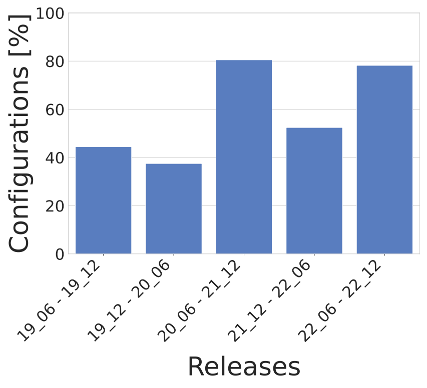
$$\mathcal{CC}_{\text{TP}}^{r_i,r_{i+1}} = \{c \,|\, c \in \mathcal{C}, \exists t \in \text{affectedTerms}^{r_i,r_{i+1}}(c) : t \in \mathcal{TC}^{r_i,r_{i+1}}\}$$

If none of the corresponding terms is included in $\mathcal{TC}^{r_i,r_{i+1}}$, we mark the configuration as a false negative:

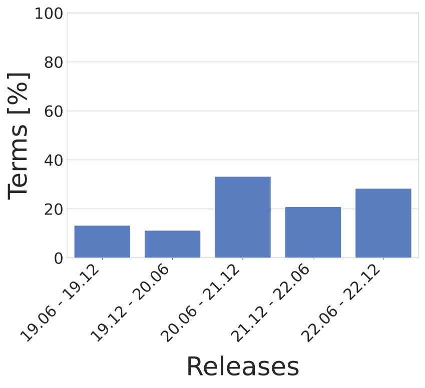
$$\mathcal{CC}_{\text{FN}}^{r_i,r_{i+1}} = \mathcal{CC}^{r_i,r_{i+1}} \setminus \mathcal{CC}_{\text{TP}}^{r_i,r_{i+1}}$$

The recall over all releases is assessed as follows:

$$\text{Recall} = \frac{\sum_{r_i,r_{i+1} \in \mathcal{R}} \text{card}(\mathcal{CC}_{\text{TP}}^{r_i,r_{i+1}})}{\sum_{r_i,r_{i+1} \in \mathcal{R}} \text{card}(\mathcal{CC}^{r_i,r_{i+1}})}$$



(a) Configuration-level changes



(b) Option-level changes

Figure 5.7: Relative number of identified performance changes at the configuration level (a) and of the identified performance changes at the option level (b).

where card denotes the cardinality of a set.

RQ_2: Next, we focus on the workload-specific research question, which assesses the number of workloads that are involved in the identification of a performance change. We use the changes at the configuration level to avoid bias that is introduced by our approach at the option level. We denote $\mathcal{WF}$ as a function that maps a configuration $c \in \mathcal{C}$ between a pair of releases $r_i, r_{i+1} \in \mathcal{R}$ to the number of workloads that identified a performance change:

$$\mathcal{WF}^{r_i,r_{i+1}} : \mathcal{C} \to \mathbb{N}$$

$$\mathcal{WF}^{r_i,r_{i+1}}(c) = \text{card}(\{w \,|\, w \in \mathcal{W} : |\mathcal{M}^{r_i,w}(c) - \mathcal{M}^{r_{i+1},w}(c)| > 2 \cdot \max(\text{sd}^{r_i,w}(c), \text{sd}^{r_{i+1},w}(c)\})$$

In other words, $\mathcal{WF}^{r_i,r_{i+1}}$ returns the number of workloads between consecutive releases r_i and r_{i+1} in which the configuration c exceeds the threshold of twice the maximum standard deviation (see Equation 5.1). It is important to note that $\mathcal{WF}^{r_i,r_{i+1}}(c)$ would return 0 if there is no performance change in $c \in \mathcal{C}$ between releases $r_i, r_{i+1} \in \mathcal{R}$.

5.4 Evaluation

In this section, we present the results of our exploratory study on detecting performance changes from FastDownward in Section 5.4.1. Later, in Section 5.4.2, we discuss the applicability and limitations of our approach as well as further insights. We discuss threats to validity in Section 5.4.3.

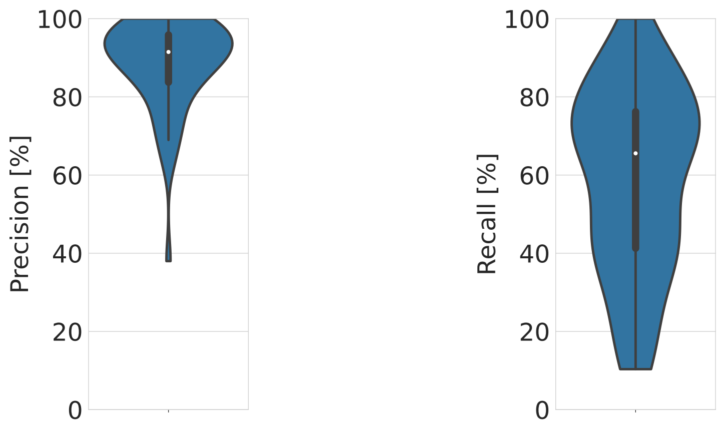


Figure 5.8: Overview of the precision (left) and the recall (right) of our approach at the option level compared to the performance changes at the configuration level over different workloads.

5.4.1 Results

In what follows, we summarize the performance changes identified at the configuration level and the performance changes identified at the option level in Figure 5.7. In Figure 5.7 (a), we show the relative number of changed configurations for all workloads and for each pair of consecutive releases (x-axis). We see that between each pair of releases, 38% to 81% of configurations had a performance change at the configuration level. In total, 7 464 performance changes are identified at the configuration level. In Figure 5.7 (b), we show the relative number of changed options/interactions for all workloads and for each pair of consecutive releases (x-axis). Clearly, the number of terms changing from one release to another is lower than at the configuration level, ranging between 2% of the options/interactions for releases 19.06 to 19.12 until 17% for releases 20.06 to 21.12. In total, our approach identified 2 674 performance changes at the option level. Note that the changes at the configuration level cannot be directly compared to the changes at the option level since many of the configuration-level changes can be comprised in a single term at the option level. For instance, a single performance change affecting every configuration at the configuration level can be comprised using only the option ROOT.

RQ$_{1.1}$: *What is the fraction of identified performance changes at the option level that can be confirmed at the configuration level?*

For the evaluation of this research question, we check in how many cases the identified performance changes of the option level can be confirmed at the configuration level. In Figure 5.8 (left), we show the precision of our approach for each workload. That is, we show in how many cases our approach is „right"(in regards to the configuration level). The exact values can be found in Table A.1 in the appendix. The precision per workload spans between 37.98% for the workload MOVIE/PROB29 and 100% for the workload DRIVERLOG/P08 with a standard deviation of 11.5%. Overall, the average precision of the identified performance changes of our approach is 88.2%. That is, 88.2% of the identified performance changes at the option level could be confirmed at the configuration level. In other words, the 88.2% of the identified performance changes at the option level are true positives (TP); the rest are false positives (FP).

Summary of Research Question 1.1

The precision of our approach per workload is between 37.98% and 100%. On average, the majority (88.2%) of the identified performance changes at the option level could be confirmed at the configuration level. That is, our approach identifies most of the performance changes „right“(with regards to the performance changes at the configuration level).

RQ$_{1.2}$: *What is the fraction of identified performance changes at the configuration level that can be confirmed at the option level?*

For this research question, we check in how many cases the identified performance changes at the configuration level can be confirmed at the option level. So, compared to RQ$_{1.1}$, we check how many of the performance changes were missed by our approach. In Figure 5.8 (right), we show the recall of our approach for each workload. Again, the exact values can be found in Table A.1 in the appendix. The recall spans between 10.3% for the workload gedopt14strips/d76 to 100% for the workload movie/prob29 with a standard deviation of 23.3%. Overall, the average recall of the identified performance changes of our approach is 59.4%. That is, a performance change identified at the configuration level is identified in more than half of the cases at the option level.

Summary Research Question 1.2

The recall of our approach per workload is between 12.6% and 100%. On average, the bare majority (59.4%) of the identified performance changes at the configuration level could be confirmed at the option level. That is, our approach identifies more than half of all performance changes.

RQ$_2$: *What is the fraction of workloads that are involved in identifying performance changes at the configuration level?*

In Figure 5.9, we show the frequency of how many workloads identified a performance change at the configuration level. In total, 225 performance changes are identified at the configuration level in, at least, one workload. The highest number of performance changes (i.e., 17 out of 225) are revealed by all 56 workloads. A single performance change was identified by a minimum of 4 out of 56 (7.6%) workloads. The majority of the performance changes (208 out of 225—92.4%) could not be identified by all workloads. On average, 33 out of 56 (59%) workloads identified a performance change.

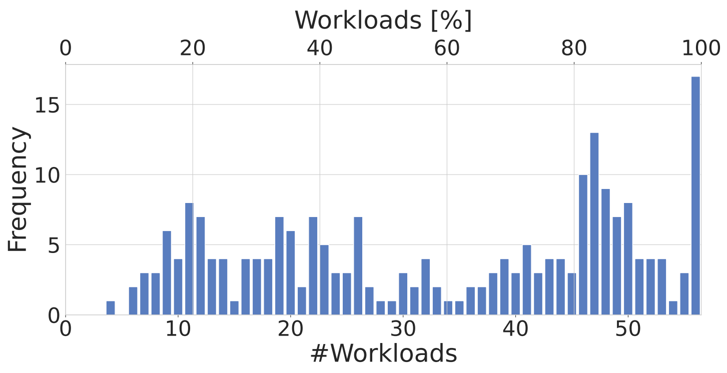


Figure 5.9: Overview of how many performance changes (y-axis) have been found by a certain number of workloads (x-axis). The higher the number of workloads, the more workloads could identify a performance change.

Summary Research Question 2

At least 4 and at most all 56 workloads identified performance changes. The highest frequency of performance changes are revealed by all 56 workloads, but the majority of 92.4% of performance changes could not be identified by all workloads.

5.4.2 Discussion

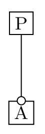
In this section, we discuss the limitations of our proposed approach for identifying performance changes at option level in Section 5.4.2. Thereby, we discuss the results from $RQ_{1.1}$ and $RQ_{1.2}$. In Figure 5.4.2, we discuss RQ_2 and further insights that we obtained in this study.

Limitations of Our Approach

In this section, we address the limitations of our approach. Therefore, we inspect our results to pinpoint reasons for these results and point out limitations of our approach.

Optimizations and Regressions Until now, we focused on identifying and comparing performance changes at the configuration level and at the option level. But, we have not distinguished between a performance optimization and a performance regression. However, distinguishing between a performance optimization and a performance regression is crucial for understanding performance changes of the system. Identifying an actual performance regression as a performance optimization would mislead users and developers. So, we assessed the precision and recall while also considering the direction of the change. That is, a performance optimization or regression at the option level is now only considered if the configuration level also indicates a performance optimization or regression, respectively. Considering also the direction of a performance change, decreases our precision from 88.2% to 84.5% and the recall from 59.4% to 43.5%. Overall, we found that the decrease in precision and recall is in many cases due to the learning error and because of low performance values.

Feature Model



Performance-Influence Model

Version	P	A
V_1	1	300
V_2	1	304

Performance Measurements (Configuration Level)

P	A	V_1	sd^{V_1}	V_2	sd^{V_2}	Result	Relevant
1	0	1s	0.01s	1s	0.01s	$0 \not> 0.02$	✗
1	1	300s	3s	304s	3.04s	$4 \not> 6.08$	✗

Performance Changes (Option Level)

$\overline{\text{sd}}^{V_1}$	$\overline{\text{sd}}^{V_2}$
1.505	1.525

Term	V_1–V_2	Result	Relevant
P	0	$0 \not> 3.05$	✗
A	4	$4 > 3.05$	✓

Figure 5.10: Example for a performance change that is identified at the option level but not at the configuration level. In this example, A is by 4 seconds slower in V_2 than in V_1. The performance-influence model contains this performance regression (see second line and right column in the performance-influence model table) and the option level identifies this regression as a relevant performance change. We added the standard deviation values and the results when applying the metrics at the configuration level and the option level, respectively. Note that we have omitted the workload since it is not relevant in this example.

Precision For assessing the precision, we analyzed how many performance changes at the option level could be confirmed at the configuration level. We obtain a precision of 88.2% and identified an important reason that influences this result.

While investigating the false positives, one important reason for the false positives we identified are the metrics that we used to identify performance changes at the configuration level (see Equation 5.1) and at the option level (see Equation 5.2). In the metric at the configuration level, we use the standard deviation ($\text{sd}^{r_{i+1},w}$); at the option level, we use the average standard deviation of all configurations ($\overline{\text{sd}}^{r_i,w}$). The rationale behind this conservative metric at the option level is that we are using the entire configuration space for learning the performance-influence models. Since we are learning performance-influence models on all configurations, we correspondingly used the average standard deviation of all configurations. This discrepancy between the metrics, however, can lead to neglecting many performance changes at the configuration level. In particular, some of the configurations of FastDownward need less than a second of runtime for certain workloads, whereas other configurations need multiple minutes. In particular, we have seen an interesting effect for configurations that indicate a runtime above the average runtime. In this case, the following holds for the specific configuration $c \in \mathcal{C}$, the workload $w \in \mathcal{W}$, and the consecutive releases $r_i, r_{i+1} \in \mathcal{R}$:

$$2 \cdot \max(\overline{\text{sd}}^{r_i,w}, \overline{\text{sd}}^{r_{i+1},w}) < 2 \cdot \max(\text{sd}^{r_i,w}(c), \text{sd}^{r_{i+1},w}(c))$$

where the left-hand side is the threshold for identifying performance changes at the option level and the right-hand side is the threshold for identifying performance changes at the configuration level. So, the option level would identify smaller changes than the configuration level. In Figure 5.10, we show an example of this discrepancy. In this example, there are only two options, P and A and two releases V_1 and V_2. We omitted the workload since the workload is not relevant for this example. The measurements of V_1 and V_2 indicate a regression of 4 seconds when A is selected. We assume a relative standard deviation of 1% in

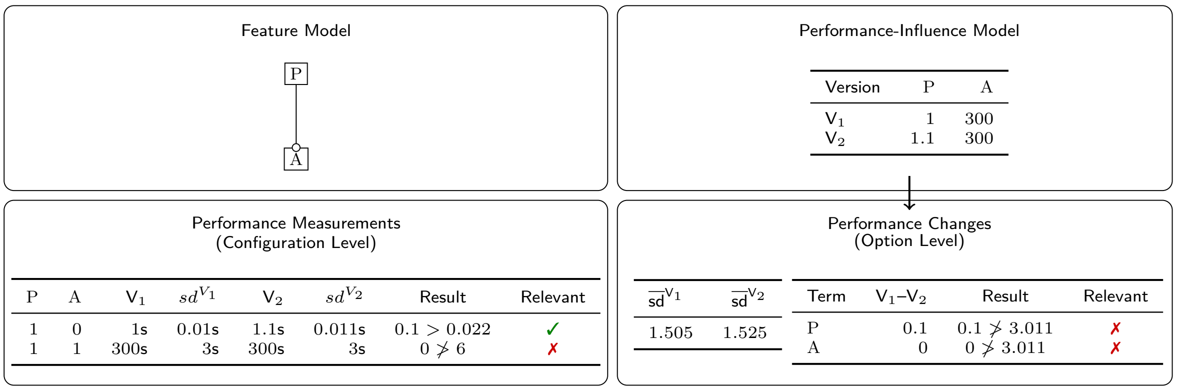
this example and show the standard deviation values at the configuration level and the option level, respectively. At the option level (right bottom), we can see that A is slower by 4 seconds while P remains unchanged. Applying the metric for the option level from Equation 5.2 on this change (mean value is 152.5) results in $4 > 3.05$ and, thus, would identify the performance change. However, the performance change would not be identified at the configuration level (left bottom). When applying the metric from Equation 5.1, it would result in $4 \ngtr 6.08$ and, thus, the configuration level neglects the change.
To determine how many of the false positives are affected by using different metrics, we opted to adjust the metric of how performance changes at the configuration level are identified. In detail, we changed the right-hand side of Equation 5.1 to match to Equation 5.2:

$$|\mathcal{M}^{r_i,w}(c) - \mathcal{M}^{r_{i+1},w}(c)| > 2 \cdot \max(\overline{\text{sd}}^{r_i,w}, \overline{\text{sd}}^{r_{i+1},w})$$

where the configuration is any configuration that is affected by the term $c \in \text{affectedConfigurations}(t)$. The function affectedConfigurations returns a set of configurations, $\mathcal{M}^{r_i,w}(c)$ denotes the performance value in release $r_i \in \mathcal{R}$ and $\overline{\text{sd}}^{r_i,w}$ returns the mean performance value of all configurations of release r_i and workload w. This way, we can determine how many of the false positives are due to the discrepancy in the metrics. By applying the adjusted metric, we found that more that more than half of the false positives (i.e., 190 out of 316 false positives) could then be found, which increases the precision by 7.1% to 95.3%.

We also found that about 78 of the remaining 126 false positives relate to configurations indicating a performance value of less than 0.1 seconds. In many of these configurations, we observed measurement deviations of more than 10%. Additionally, learning performance-influence models introduces a learning error, which makes the performance-influence models more imprecise. We found that the performance-influence models of 2 workloads indicated a error higher than 10% (see Section 5.4.3), which was mainly because of a major drawback of our learning error metric (i.e., mean absolute percentage error). For the remaining 54 workloads, the average learning error is always below 10% and 0.2% on average. A combination of both the measurement deviations and the learning error of the performance-influence models are another reason for the remaining false positives. So, considering the learning error of a model in our metrics would be another step that decreases the amount of false positives, since this affects all false positives.

Recall For assessing the recall, we analyzed how many performance changes at the configuration level could be confirmed at the option level. In more than half of the performance changes at the configuration level, we could confirm the performance changes at the option level. We identified two reasons that influence this result. The most important reason are the different metrics at the configuration level and at the option level. This discrepancy between the metrics can lead to neglecting many performance changes at the option level, too. Interestingly, this effect is the opposite to when calculating the precision. Again, some of the configurations of FastDownward need less than a second of runtime for certain workloads, whereas other configurations need multiple minutes. Using the average performance for all configurations then leads to neglect very small changes. In the case of configurations



Performance-Influence Model

Version	P	A
V_1	1	300
V_2	1.1	300

Performance Measurements (Configuration Level)

P	A	V_1	sd^{V_1}	V_2	sd^{V_2}	Result	Relevant
1	0	1s	0.01s	1.1s	0.011s	$0.1 > 0.022$	✓
1	1	300s	3s	300s	3s	$0 \not> 6$	✗

Performance Changes (Option Level)

$\overline{\mathrm{sd}}^{V_1}$	$\overline{\mathrm{sd}}^{V_2}$
1.505	1.525

Term	V_1–V_2	Result	Relevant
P	0.1	$0.1 \not> 3.011$	✗
A	0	$0 \not> 3.011$	✗

Figure 5.11: Example for a performance change that is identified at the option level but not at the configuration level. In this example, the configuration where A is deselected is by 0.1 seconds slower in V_2 than in V_1. The performance-influence model contains this performance regression (see second line and right column in the performance-influence model table) and the option level identifies this regression as a relevant performance change. We added the standard deviation values and the results when applying the metrics at the configuration level and the option level, respectively. Note that we have omitted the workload since it is not relevant in this example.

indicating a lower runtime than the average runtime, the following holds for the specific configuration $c \in \mathcal{C}$, the workload $w \in \mathcal{W}$, and the consecutive releases $r_i, r_{i+1} \in \mathcal{R}$:

$$2 \cdot \max(\overline{\mathrm{sd}}^{r_i,w}, \overline{\mathrm{sd}}^{r_{i+1},w}) > 2 \cdot \max(\mathrm{sd}^{r_i,w_j}(c), \mathrm{sd}^{r_{i+1},w_j}(c))$$

where the left-hand side is the threshold for identifying performance changes at the option level and the right-hand side is the threshold for identifying performance changes at the configuration level. In detail, we show another example in Figure 5.11. Again, our example consists of two configuration options, P and A and two consecutive releases, V_1 and V_2. In this case, the regression is introduced at V_2 in the configuration where A is deselected. The regression is about 0.1 seconds and we assume a relative standard deviation of 1%. At the configuration level, the metric would identify the performance change because it relates to the performance value instead of the average performance value of all configurations. In particular, when applying Equation 5.1 on our example, it results in $0.1 > 0.022$. However, at the option level, we relate to the average performance values. When applying Equation 5.2) on our example, it results in $0.1 \not> 3.011$ and, hence, this performance change is not relevant on the option level.

To determine how severe the metric discrepancy affects our analysis, we executed our recall analysis again but used the standard deviation and mean value of the respective configuration $c \in \mathcal{C}$:

$$|\beta^{r_i,w}(t) - \beta^{r_{i+1},w}(t)| > 2 \cdot \max(\mathrm{sd}^{r_i,w}(c), \mathrm{sd}^{r_{i+1},w}(c)).$$

where $t \in \mathrm{affectedTerms}(c)$ denotes any term affecting this configuration, affectedTerms is a function that returns a set of terms affected by the configuration, $w \in \mathcal{W}$ denotes the workload, and $r_i, r_{i+1} \in \mathcal{R}$ denote the consecutive releases. By adjusting the metric to identify performance changes at the option level, most of the false negatives become true positives, increasing the recall by 35.7% from 59.4% to 95.1%. So, only adjusting the metric at the option level accounts for most of the false negatives.

Another reason for a part of the remaining false negatives is the learning error of the performance-influence models. We observed that in 2 out of 56 workloads the learning error was higher than 10% (see Section 5.4.3). This suggests that using performance-influence models with multiple linear regression for finding performance changes at option level might not always work and can lead to wrong conclusions. In total, the high learning error accounts for another 1.5% of the recall.

Summary To sum up, we have addressed multiple limitations in this section and investigated them. First and foremost, we observed that the different metrics to identify a performance change at the configuration level and at the option level represent the main reason for the false positives in the precision and the false negatives in the recall. In our investigation, we determined that matching the metrics of identifying performance changes at the configuration level and at the option level would largely increase the precision and recall to over 95%. So, using different metrics at the configuration level and option level represents one limitation. Other limitations are that low performance values in combination with the learning error is currently not considered, which again affects precision and recall. A further limitation of our approach is distinguishing between optimizations and regressions.

Besides, we found no indication on false negatives or false positives that are caused by different configuration spaces and, thus, conjecture that our approach can be used with different configuration spaces. This is essential to cover software evolution in practice since configuration options are typically added or removed over time [114].

Moreover, we also found that using imputation was a good choice since it enabled us to immediately see a performance change when a configuration was running into a timeout in a newer release or recovering from a timeout. Removing such configurations completely, would have lead to missing many performance changes.

Implications

Insight: Workload sensitivity In RQ_2, we investigated how many different workloads could identify a performance change. Interestingly, 92.4% of the performance changes at the configuration level were identified only in a subset of the workloads. Thus, different workloads identify different performance changes. This insight confirms findings of recent publications [79, 103]. Besides, for a further categorization of which workloads identify similar performance changes at the configuration level, we created a dendrogram. In Figure 5.12, we show a dendrogram using the performance change information at the configuration level and also consider the direction (i.e., regression or optimization). In general, the dendrogram indicates how similar some workloads are to each other by clustering them. The x-axis is the average distance between groups of workloads and the y-axis contains the 56 workloads that we measured. The higher the distance, the more different performance changes have been identified by the different groups of workloads. We observe that the dendrogram mostly clusters domains with similar names together. For instance, OPENSTACKSOPT08STRIPS/P12 and OPENSTACKSOPT11STRIPS/P07 are similar domains and, according to the dendrogram, identify similar performance changes. A further observation is that some other workloads such as AIRPORT/P07AIRPORT2P2 and TIDYBOTOPT11STRIPS/P01 differ much more from the rest than other workloads. That is, that these workloads identify other performance changes than the rest of

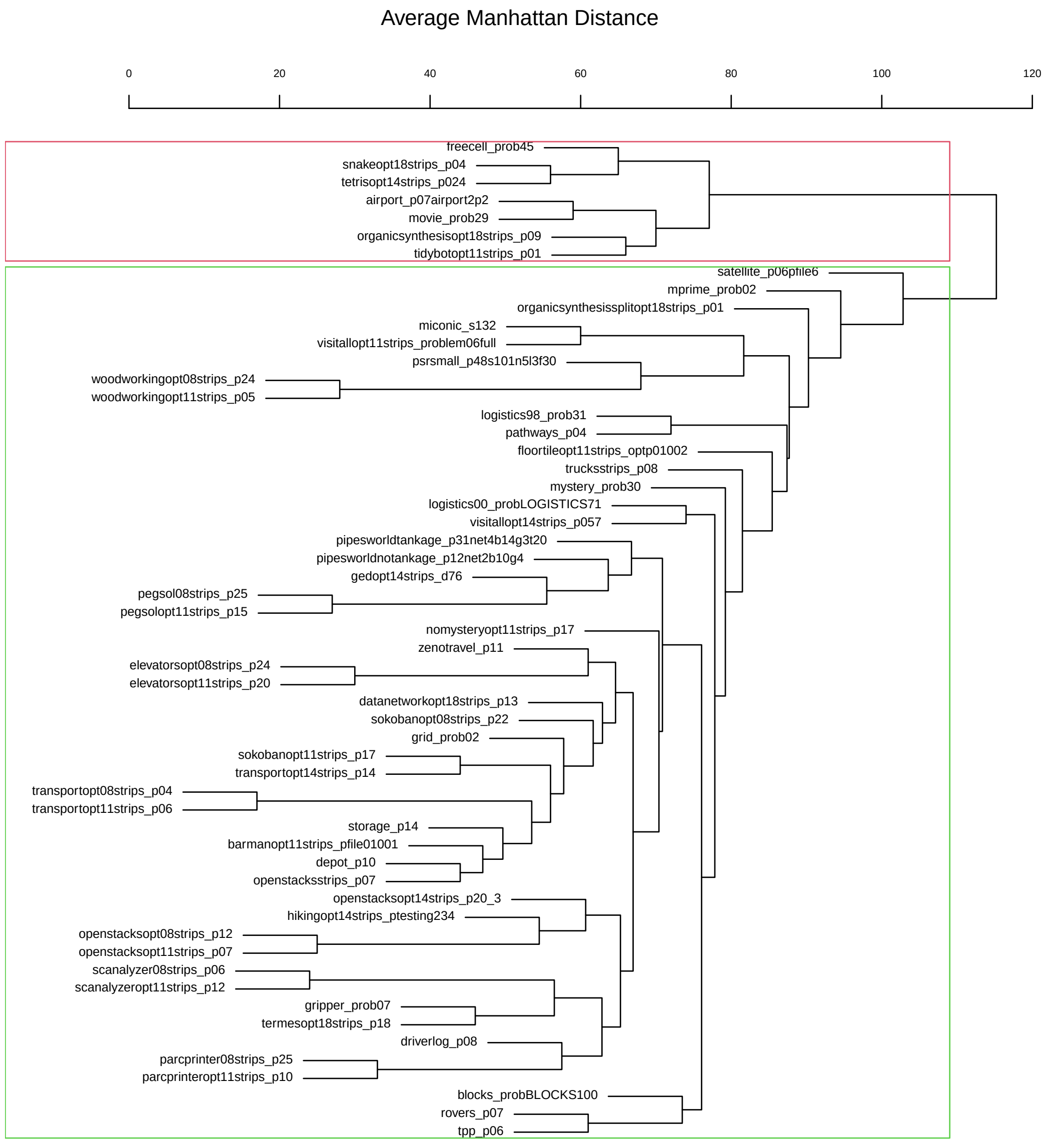


Figure 5.12: Dendrogram for clustering all 56 workloads (y-axis) with regards to the performance changes the respective workloads have identified at the configuration level. The x-axis is the average Manhattan distance between clusters. The higher the Manhattan distance, the more different performance changes have been identified by these clusters. Each of the 2 clusters is surrounded by a differently colored rectangle.

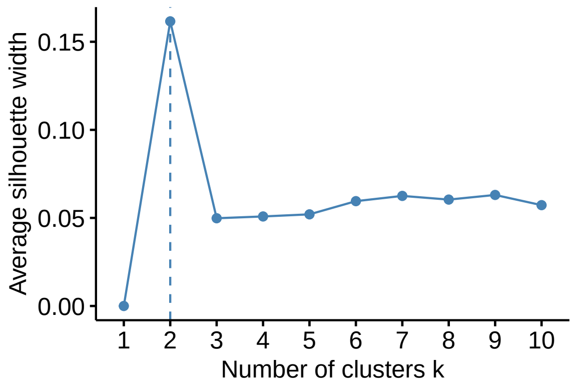


Figure 5.13: Overview of the average silhouette width [126] on different number of clusters. According to the average silhouette width, $k = 2$ is the best choice.

the workloads. Here, using a diverse set of workloads in terms of the performance changes they identify, could help practitioners to identify performance changes early on. In essence, selecting workloads from different clusters would help in diversifying the workload variability with respect to uncovering performance changes. To select a suitable set of workloads using the dendrogram, the number of workloads to measure has to be identified. We denote this number as k, split the dendrogram in k different clusters, and choose one workload from each cluster. For demonstration, we use the average silhouette approach [126] in Figure 5.13 to determine a suitable number of clusters. In essence, the average silhouette approach measures the quality of a clustering using the distances within clusters and between clusters [126]. The higher the value, the better. In our case, the average silhoette approach determines that $k = 2$ is the optimal number of clusters. We show the different clusters in Figure 5.12 and would then select one workload for each of both clusters. For instance, AIRPORT/P07AIRPORT2P2 and SATELLITE/P06PFILE6 are very dissimilar to each other, belong to different clusters and, thus, would be suitable candidates for performance testing. Choosing only these 2 workloads already find 154 of a total of 398 (38.7%) different performance changes at the configuration level (when considering the direction).

In further experiments, we observed that with $k = 7$ workloads, we find 255 out of 398 (64%) performance changes and with $k = 10$ workloads, we find 292 out of 398 (73.3%) performance changes.

Insight: Duration of performance regressions The introduction of a performance regression is often not detected immediately. In many cases, it takes some time until the performance regression is notified by users or developers and, afterwards, fixed. In Chapter 4, we observed many cases where regressions have been addressed by the developers only a few releases later. To investigate this, we assessed the number of releases until a performance regression is optimized again and obtains nearly the original performance (depending on the standard deviation of the measurement). In Figure 5.14, we show for FASTDOWNWARD the number of releases until a configuration obtained its original performance. In total, we found 418 performance regressions that were fixed after a number of releases. In most cases (i.e., 261 out of 418), a performance regression was fixed after 2 releases. Only rarely a regression

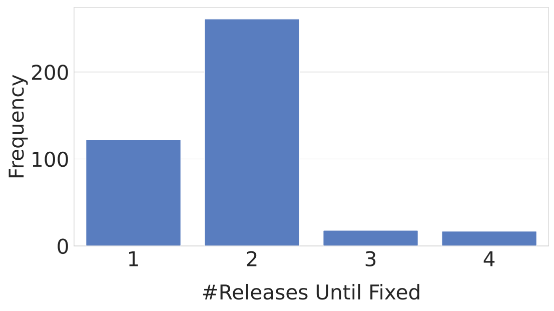


Figure 5.14: Overview of the number of releases until a regression was fixed.

was fixed after 3 or 4 releases. Such performance changes could be detected early on using a configuration-aware and workload-aware continuous performance testing pipeline.

Reported performance bugs In our exploratory study, we found several performance changes. Some of these performance changes were optimizations, others were regressions. Many of the identified performance changes have been fixed in the latest release 22.12; otherwise, the number of reported performance changes would be much higher. After the evaluation of the performance changes at the configuration level, we could also identify 3 persisting performance regressions and reported these performance changes to the developers[12]. Interestingly, one of the 3 performance regressions was only persisting in 2 out of 56 workloads; in the other workloads, this regression was fixed. The developers responded to our reports and started an investigation.

5.4.3 Threats to Validity

Construct Validity To select a reasonable set of workloads for our performance measurements, we executed preliminary performance measurements with over 1 800 workloads and selected afterwards 56 workloads from 56 different domains. However, given our setup, we could have missed important performance changes. Clearly, we could mitigate this by choosing a larger set of workloads. This, however, would have resulted in an even smaller configuration space or smaller amount of releases, due to limited resources. Moreover, we already tried to maximize the variation between workloads by using different domains (i.e., different types of planning problems for FastDownward) and selected workloads used in competitions [149]. This makes us confident that we identified the most important performance changes.

For the evaluation of our research questions, we rely on comparing our results at the configuration level and use it as a point of reference. The missing ground truth represents another threat. In theory, the best point of reference is a ground truth that contains all performance changes of FastDownward. This, however, is not feasible in practice since all possible code paths (i.e., control flow) have to be taken into account to get precise data. A white-box analysis typically needs a lot of resources in terms of processing power and time

12 Since we reported the performance regressions on the Discord server of FastDownward for further investigation by the developers, we cannot provide a link to the report.

for large subject systems such as FastDownward with thousands of lines of codes. So, we resort to use the performance data at the configuration level as point of reference, which represents a well established means to detect performance changes [71].

We selected the timeout for our preliminary performance measurements (i.e., 5 minutes) and for our overall performance measurements (i.e., 15 minutes) to correspond to a competition setting. The problem is that choosing other timeouts would change our results in this chapter. However, it is difficult to choose a reasonable timeout. In fact, we also tried a timeout of 24 hours and still observed some timeouts with some configurations in some releases and workloads. Developers of FastDownward suggested a timeout of 5 minutes. This way, we reduced the time needed for measurements, but also adapt the setup to real world scenarios in terms of planning competitions [149]. For our overall performance measurements, we increased the timeout from 5 minutes to 15 minutes to reduce the number of configurations where we had to impute a value.

The way how we treat a timeout in our measurement data is another threat. We imputed the value 1 800 seconds (i.e., 30 minutes—double the timeout) in all configurations that resulted in a timeout. The problem is that choosing another value than 1 800 for the imputation could change our results. For instance, selecting 900 seconds (i.e., 15 minutes, which is our timeout) would have been another candidate and would have resulted in less performance changes since a few configurations indicate performance values in some workloads and releases that are near to 15 minutes. For instance, a configuration that needs about 899 seconds in one release for a workload and runs into a timeout in the next release, would not be detected at the configuration level, as we demonstrate in the following example using our formula to identify performance changes at the configuration level from Equation 5.1:

$$|\mathcal{M}^{r_i,w_j}(c) - \mathcal{M}^{r_{i+1},w_j}(c)| > 2 \cdot \max(\mathrm{sd}^{r_i,w_j}(c), \mathrm{sd}^{r_{i+1},w_j}(c))$$

In this example, the configuration has a runtime of 899 in release r_i and runs into a timeout in release r_{i+1} (i.e., the imputed value will be 900). Further, we assume an average runtime of 100 seconds and a maximum standard deviation of 1%, which is well below our maximum standard deviation of 10%. This results in the value of 2 for $\max(\mathrm{sd}^{r_i,w_j}(c), \mathrm{sd}^{r_{i+1},w_j}(c))$. This results in the following: $|899 - 900| \ngtr 2 \cdot 2$. Clearly, we would miss such small configuration changes when using 900 for imputation. Since we do not know the real performance values of configurations that run into a timeout and we wanted to maximize the number of detected performance changes for our exploratory study, we used the double of the timeout as the imputed value.

Internal Validity Measurement noise is typically present in both, software and hardware [105]. Measurement noise, however, is a problem that threatens the reproducibility of our data. To counteract, we used identical hardware with a minimum Debian installation, which corresponds to the mitigations used in Chapter 3 and Chapter 4. We repeated our measurements 5 times except for the preliminary measurements, where we did not repeat our measurements. We repeated these measurements for configurations where the performance exceeded 0.1 seconds and 10% standard deviation. Note that we had to neglect repeating the measurements for configurations faster than 0.1 seconds since the performance noise was considerably higher in these measurements, which led to a higher variance (up to 80% standard deviation). The standard deviation, however, was considered in our analysis and, thus, is likely to neglect cases where performance noise is the reason for the performance change.

We used the variance inflation factor (VIF) analysis to reduce multicollinearity in our performance-influence models. The problem is that we did not follow community thresholds and removed only perfectly multicollinear terms, instead. Furthermore, we used metrics to decide whether a configuration on configuration level or an option/interaction on option level indicates a performance change. Other publications that focus on performance changes used different metrics. This represents an internal threat to validity since choosing other metrics would change the outcome of our analysis. We have already discussed the use of the metrics in Section 4.4.4 in more detail.

Our performance-modeling approach is based on multiple linear regression. Using multiple linear regression with feature forward selection poses another threat to internal validity. The problem is that using other machine-learning approaches or using multiple linear regression differently would produce other results. We selected multiple linear regression because it produces better interpretable models in terms of the additive structure of the models. But, always choosing the best candidate in the feature forward selection process might not represent the best strategy. Instead, choosing a worse performing candidate could result in much better performance models in a later iteration. In other words, our performance models could present only a local optimum instead of a global optimum.

For assessing the learning error in our performance-modeling approach, we use the mean absolute percentage error (MAPE). Again, choosing other metrics to assess the prediction error influences our results. It is important to note that one of the drawbacks of using MAPE for assessing the learning error is that the prediction is extremely high for values that are very cloes to zero [52]. As a reminder, we use the following formula to calculate the MAPE:

$$\text{MAPE} = \frac{1}{|\mathcal{C}|} \cdot \sum_{c \in \mathcal{C}} \frac{|\mathbb{P}(c) - \Pi(c)|}{\mathbb{P}(c)} \cdot 100$$

where $\mathbb{P}$ denotes the function that maps the configuration to its measured performance value and $\Pi(c)$ denotes the performance-influence model that maps the configuration to its predicted performance value. For instance, estimating 3.1 seconds for a configuration that takes about 0.003 seconds leads to the following error:

$$\epsilon = \frac{|3.1 - 0.003|}{0.003} \cdot 100 = 103\,230$$

In this example, we obtain a percentage error of 103 230%. This is an actual example from the workload visitallopt11strips/problem06full. In total, 2 out of 56 (i.e., visitallopt11strips/problem06full and miconic/s132) workloads indicate learning errors of over 10% because of this drawback. Although small values close to zero can be avoided by changing the measurement unit from seconds to miliseconds, we decided to maintain the measurement unit used by FastDownward for easier interpretation.

External Validity Although we demonstrated that our approach can identify performance changes at the option level for FastDownward in this chapter and also demonstrated in Chapter 4 that a similar approach can identify performance changes across configurations and releases for 12 different subject systems, we cannot state whether our approach will perform good on other subject systems. Since our approach is not designed specifically for FastDownward, it can be used also for other subject systems. But, there may be even more special cases such as the mandatory alternative groups presented in Section 5.3.3. For instance, the issue with mandatory alternative groups also applies for mandatory or groups. To the best of our knowledge, or groups are only rarely used in practice and do not appear in subject systems we have investigated in this thesis. Since our approach is not specifically designed for the subject system FastDownward, our analysis is also applicable on other subject systems that do not contain or groups.

5.5 Summary

Although performance changes have been extensively studied in the literature and in Chapter 4, prior work has not focused on investigating three different dimensions: configurability, software evolution, and workload variability. In this work, we addressed all three dimensions at one and devised a novel approach to identify performance changes using the information of all three dimensions. Our main focus was on assessing the limitations of a novel approach for identifying performance changes at option level. In an exploratory study, we analyzed performance changes of a single configurable software system over 49 configurations, 6 releases, and 56 workloads. Overall, we identified 7 464 performance changes in different configurations and workloads and 2 674 performance changes in different configuration options or interactions among them. Comparing these results, our approach obtains a precision of 88.2% while having a recall of 59.4%. While analyzing the precision and recall in-depth, we observed that the precision and recall could be increased by aligning the metrics to identify performance changes at the configuration level or the option level.

Our results emphasize the role of workloads for identifying performance changes in configurable software systems since the majority (92.4%) of performance changes was identified only in a subset of the workloads. Interestingly, we found one performance change that was identified only by 4 out of 56 workloads. Further, we cluster the workloads according to the performance changes each workload identifies and demonstrated how this clustering can be used to identify a suitable set of workloads for identifying performance changes. We also found that using only 2 different workloads already finds 38.7% of the performance changes, whereas increasing it to 10 different workloads finds 73.3% of the performance changes.

Another notable insight is that none of the performance regressions was fixed in the subsequent release, but, at least, 2 releases later. Using a testing pipeline that considers configurations and workloads alike to find such regressions early on could help in identifying or even avoiding performance regressions. Our performance analysis helped in uncovering 3 persisting regressions in FastDownward and, thus, demonstrates that it can be used to uncover performance regressions.

6 Concluding Remarks

Configurable software systems are widely established since they allow users to adapt the software to different application scenarios, user requirements, or hardware requirements. The flexibility of this adaptation comes with a price: Many software systems offer too many configuration options [152]. Due to this high number of configuration options it is unclear how the software system performs and, thus, which configuration exposes the best performance characteristics for a given environment. This unclarity is further aggravated by software evolution and workload variability, which are known to affect the performance of software systems.

6.1 Contributions of this Thesis

The contributions of this thesis are to extend and use performance modeling to (1) improve performance prediction and (2) to investigate how software evolution and workload variability affect the performance of configurable systems. We address contribution (1) by proposing a novel sampling approach to improve performance modeling. We address contribution (2) by investigating in an empirical study and an exploratory study the influence of performance while considering software evolution and workload variability, respectively.

Specifically, we made the following contributions:

1. We have proposed a novel sampling strategy, distance-based sampling, with the goal to overcome the disadvantages exposed by state-of-the-art sampling strategies. The idea of distance-based sampling is to diversify the configurations selected by a constraint solver by using a distance metric. Since only using a distance metric did not diversify the configurations enough, we expanded distance-based sampling to consider the frequency of configuration options. In an empirical comparison on 10 real-world configurable software systems, we have compared distance-based sampling to other sampling strategies. The evaluation has shown that distance-based sampling outperforms other sampling strategies and comes close to random sampling. Besides, we identified current limitations in terms of scalability while sampling on large configuration spaces, which leaves further room for improvement. Our evaluation provides important insights into the merits of using a distance-based metric for sampling configuration spaces.

2. In an empirical study, we have used performance modeling to assess how performance changes and investigated software evolution of up to 15 years in 12 real-world configurable software systems. To assess the practicability of using performance modeling for identifying performance changes, we have performed a meta-data analysis on the

change log and commit messages of these real-world configurable software systems. Our results indicate that most performance changes appear only in some specific configuration options. Further, we have found that most performance changes in general and, specifically the affected configuration options, are reported by developers in the change log or commit messages. Our empirical study provides important insights into the role of software evolution and configurability since it confirms prior beliefs that performance changes appear in only some configuration options. Our insights do have implications on other domains, such as transfer learning (i.e., the most relevant configuration options remain most relevant during software evolution). Our results demonstrate that using a performance testing pipeline during development helps in identifying performance changes early on.

3. In an exploratory study, we have investigated the role of workload variability, configurability, and software evolution together. To this end, we have investigated 49 configurations and 56 different workloads among 6 different releases of FastDownward. We found that 92.4% of the changes were identified only in a subset of the workloads. Further, we have used the exploratory study to assess how using performance modeling performs for finding performance changes. In our evaluation, we obtained a precision of 88.2% and a recall of 59.4%. A deeper inspection of these results revealed limitations of our approach and, on the base of that, suggested further improvements. Throughout this study, we found 3 persisting performance regressions, which we reported to the developers of FastDownward. This exploratory study helps in assessing whether performance modeling can be used to identify performance changes and confirms findings of other studies investigating the role of workload variability in presence of configurability.

Overall, we contributed to the field of performance modeling of configurable software systems by (1) proposing a novel sampling strategy designed to cover configuration spaces with regards to performance; (2) lifting how performance changes affect software configurability in practice; (3) demonstrating how performance modeling can be used to find performance changes while including workload variability. This way, we provide better means for software developers and users to identify performance changes in configurable software systems. To the best of our knowledge, we are the first to consider configurability, evolution, and workload variability of configurable software systems at once. Our contributions have implications to other domains, such as transfer learning.

6.2 Avenues of Future Work

Sampling Strategies Sampling is a problem not only specific to configurable software systems. Thus, a large variety of different sampling strategies have been proposed in the past. But despite the large variety of different sampling strategies, there is currently no guide on which sampling strategy should be used and in which cases. One prominent example for such sampling strategies is a graph-based algorithm for uniform sampling by Sharma et al. [133]. A promising step into the direction of providing a guide is undertaken by Acher et al. [2] by proposing a framework that combines many of the sampling strategies. Using this tool in an extensive study to compare this variety of sampling approaches would clarify the picture of which sampling strategy is most suitable for certain situations and, thus, is a promising avenue of future work.

Detection of Performance Changes Another avenue for future work is to improve the metrics for detecting performance changes even further. Although we have used performance modeling in an empirical and in an exploratory study, we conclude that there is still some room for improvement given the current precision and recall of our approach before applying the approach in the wild. The most important addition is to enhance our approach to support OR groups. Without the support of OR groups, our approach cannot be applied on any arbitrary configurable software system. Another addition to improve the outcome of our approach is to include the prediction error of the learned performance-influence models in the metrics for detecting performance changes.

Bibliography

[1] Mathieu Acher, Hugo Martin, Luc Lesoil, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Khelladi, Olivier Barais, and Juliana Pereira. "Feature Subset Selection for Learning Huge Configuration Spaces: The Case of Linux Kernel Size." In: *Proceedings of the International Systems and Software Product Line Conference* (*SPLC*). ACM, 2022, pp. 85–96.

[2] Mathieu Acher, Gilles Perrouin, and Maxime Cordy. "BURST: A Benchmarking Platform for Uniform Random Sampling Techniques." In: *International Systems and Software Product Line Conference* (*SPLC*). ACM, 2021, pp. 36–40.

[3] Juan Alcocer and Alexandre Bergel. "Tracking Down Performance Variation Against Source Code Evolution." In: *Proceedings of the Symposium on Dynamic Languages* (*DLS*). ACM, 2015, pp. 129–139.

[4] David Allen. "Mean Square Error of Prediction as a Criterion for Selecting Variables." In: *Technometrics* 13.3 (1971), pp. 469–475.

[5] Paul Allison. *Missing Data*. Sage Publications, 2001.

[6] David Andrews. "A Robust Method for Multiple Linear Regression." In: *Technometrics* 16.4 (1974), pp. 523–531.

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

[8] Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. "Exploring Feature Interactions in the Wild: The New Feature-Interaction Challenge." In: *Proceedings of the International Workshop on Feature-Oriented Software Development* (*FOSD*). ACM, 2013, pp. 1–8.

[9] Andrea Arcuri and Lionel Briand. "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering." In: *Proceedings of the IEEE/ACM International Conference on Software Engineering* (*ICSE*). IEEE / ACM, 2011, pp. 1–10.

[10] David Benavides, Sergio Segura, and Antonio Ruiz Cortés. "Automated Analysis of Feature Models 20 Years Later: A Literature Review." In: *Information Systems* 35.6 (2010), pp. 615–636.

[11] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. "A Study of Variability Models and Languages in the Systems Software Domain." In: *Transactions on Software Engineering* 39.12 (2013), pp. 1611–1640.

[12] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of Satisfiability*. Vol. 185. IOS press, 2009.

[13] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. *Handbook of Satisfiability - Second Edition*. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021.

[14] Markus Blatt and Peter Bastian. "The Iterative Solver Template Library." In: *Proceedings of the Workshop on State-Of-The-Art in Scientific and Parallel Computing (PARA)*. Springer. 2007, pp. 666–675.

[15] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. "WISE: Automated Test Generation for Worst-Case Complexity." In: *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE / ACM, 2009, pp. 463–473.

[16] Maria Calzarossa, Luisa Massari, and Daniele Tessera. "Workload Characterization: A Survey Revisited." In: *ACM Computing Surveys* 48.3 (2016), 48:1–48:43.

[17] Colin Cameron and Frank Windmeijer. "An R-Squared Measure of Goodness of Fit for Some Common Nonlinear Regression Models." In: *Journal of Econometrics* 77.2 (1997), pp. 329–342.

[18] Supratik Chakraborty, Daniel Fremont, Kuldeep Meel, Sanjit Seshia, and Moshe Vardi. "Distribution-Aware Sampling and Weighted Model Counting for SAT." In: *Proceedings of the Association for the Advancement of Artificial Intelligence (AAAI)*. AAAI Press, 2014, pp. 1722–1730.

[19] Supratik Chakraborty, Kuldeep Meel, and Moshe Vardi. "A Scalable and Nearly Uniform Generator of SAT Witnesses." In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*. Springer, 2013, pp. 608–623.

[20] Samprit Chatterjee and Jeffrey Simonoff. *Handbook of Regression Analysis*. Vol. 5. John Wiley & Sons, 2013.

[21] Jinfu Chen and Weiyi Shang. "An Exploratory Study of Performance Regression Introducing Code Changes." In: *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 341–352.

[22] Tsong Chen, Hing Leung, and Iengkei Mak. "Adaptive Random Testing." In: *Proceedings of the Asian Computing Science Conference (ASIAN)*. Springer. 2004, pp. 320–329.

[23] Zhifei Chen, Bihuan Chen, Lu Xiao, Xiao Wang, Lin Chen, Yang Liu, and Baowen Xu. "Speedoo: prioritizing performance optimization opportunities." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE / ACM, 2018, pp. 811–821.

[24] Jiezhu Cheng, Cuiyun Gao, and Zibin Zheng. "HINNPerf: Hierarchical Interaction Neural Network for Performance Prediction of Configurable Systems." In: *Transactions on Software Engineering and Methodology* 32.2 (2023), 46:1–46:30.

[25] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-Guided Abstraction Refinement." In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*. Vol. 1855. Lecture Notes in Computer Science. Springer, 2000, pp. 154–169.

[26] Diego Costa, Cor-Paul Bezemer, Philipp Leitner, and Artur Andrzejak. "What's Wrong with My Benchmark Results? Studying Bad Practices in JMH Benchmarks." In: *IEEE Transactions on Software Engineering (TSE)* 47.7 (2021), pp. 1452–1467.

[27] Johannes Dorn, Sven Apel, and Norbert Siegmund. "Mastering Uncertainty in Performance Estimations of Configurable Software Systems." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 684–696.

[28] Johannes Dorn, Sven Apel, and Norbert Siegmund. "Mastering Uncertainty in Performance Estimations of Configurable Software Systems." In: *Empirical Software Engineering* 28.2 (2023), p. 33.

[29] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. "Efficient Sampling of SAT Solutions for Testing." In: *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE / ACM, 2018, pp. 549–559.

[30] Dror Feitelson. "Workload Modeling for Performance Evaluation." In: *Performance Evaluation of Complex Systems: Techniques and Tools*. Springer, 2002, pp. 114–141.

[31] King Foo, Zhen Jiang, Bram Adams, Ahmed Hassan, Ying Zou, and Parminder Flora. "Mining Performance Regression Testing Repositories for Automated Performance Analysis." In: *Proceedings of the International Conference on Quality Software (QRS)*. IEEE, 2010, pp. 32–41.

[32] Hormozd Gahvari, Allison Baker, Martin Schulz, Ulrike Yang, Kirk Jordan, and William Gropp. "Modeling the Performance of an Algebraic Multigrid Cycle on HPC Platforms." In: *Proceedings of the International Conference on Supercomputing (ICSP)*. ACM, 2011, pp. 172–181.

[33] Andy Georges, Dries Buytaert, and Lieven Eeckhout. "Statistically Rigorous Java Performance Evaluation." In: *Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA)*. ACM, 2007, pp. 57–76.

[34] Vibhav Gogate and Rina Dechter. "A New Algorithm for Sampling CSP Solutions Uniformly at Random." In: *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*. Springer. 2006, pp. 711–715.

[35] Jingzhi Gong and Tao Chen. "Does Configuration Encoding Matter in Learning Software Performance? An Empirical Study on Encoding Schemes." In: *Proceedings of the International Conference on Mining Software Repositories (MSR)*. ACM, 2022, pp. 482–494.

[36] Alexander Grebhahn, Christian Kaltenecker, Christian Engwer, Norbert Siegmund, and Sven Apel. "Lightweight, Semi-Automatic Variability Extraction: A Case Study on Scientific Computing." In: *Empirical Software Engineering* 26.2 (2021), p. 23.

[37] Alexander Grebhahn, Carmen Rodrigo, Norbert Siegmund, Francisco Gaspar, and Sven Apel. "Performance-Influence Models of Multigrid Methods: A Case Study on Triangular Grids." In: *Concurrency and Computation Practice and Experience* 29.17 (2017).

[38] Alexander Grebhahn, Norbert Siegmund, and Sven Apel. "Predicting Performance of Software Configurations: There is no Silver Bullet." In: *Computing Research Repository* (2019). Available at https://arxiv.org/pdf/1911.12643.pdf.

[39] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. “Variability-Aware Performance Prediction: A Statistical Learning Approach.” In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 301–311.

[40] Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Atrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wasowski, and Huiqun Yu. “Data-Efficient Performance Learning for Configurable Systems.” In: *Empirical Software Engineering* 23.3 (2018), pp. 1826–1867.

[41] Huong Ha and Hongyu Zhang. “DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network.” In: *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE / ACM, 2019, pp. 1095–1106.

[42] 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.” In: *Empirical Software Engineering* 24.2 (2019), pp. 674–717.

[43] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. “Performance Debugging in the Large via Mining Millions of Stack Traces.” In: *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE / ACM, 2012, pp. 145–155.

[44] Xue Han and Tingting Yu. “An Empirical Study on Performance Bugs for Highly Configurable Software Systems.” In: *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE / ACM, 2016, 23:1–23:10.

[45] Christoph Heger, Jens Happe, and Roozbeh Farahbod. “Automated Root Cause Isolation of Performance Regressions During Software Development.” In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 2013, pp. 27–38.

[46] Malte Helmert. “The Fast Downward Planning System.” In: *Journal of Artificial Intelligence Resesearch* 26 (2006), pp. 191–246.

[47] Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. “Combining Multi-Objective Search and Constraint Solving for Configuring Large Software Product Lines.” In: *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE / ACM, 2015, pp. 517–528.

[48] Ruben Heradio, David Fernández-Amorós, José Galindo, David Benavides, and Don Batory. “Uniform and Scalable Sampling of Highly Configurable Systems.” In: *Empirical Software Engineering* 27.2 (2022), p. 44.

[49] Andreas M. Hinz, Sandi Klavzar, Uros Milutinovic, and Ciril Petr. *The Tower of Hanoi - Myths and Maths*. Birkhäuser, 2013.

[50] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. "Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study." In: *Empirical Software Engineering* 21.2 (2016), pp. 449–482.

[51] Karl Huppler. "The Art of Building a Good Benchmark." In: *TPC Technology Conference on Performance Evaluation & Benchmarking (TPCTC)*. Vol. 5895. Lecture Notes in Computer Science. Springer, 2009, pp. 18–30.

[52] Rob Hyndman and Anne Koehler. "Another Look at Measures of Forecast Accuracy." In: *International Journal of Forecasting* 22.4 (2006), pp. 679–688.

[53] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning*. Vol. 112. Springer, 2013.

[54] Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. "Transfer Learning for Performance Modeling of Configurable Systems: An Exploratory Analysis." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 497–508.

[55] Pooyan Jamshidi, Miguel Velez, Christian Kästner, and Norbert Siegmund. "Learning to Sample: Exploiting Similarities Across Environments to Learn Performance Models for Configurable Systems." In: *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2018, pp. 71–82.

[56] Zhen Jiang and Ahmed Hassan. "A Survey on Load Testing of Large-Scale Software Systems." In: *Transactions on Software Engineering* 41.11 (2015), pp. 1091–1118.

[57] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. "Understanding and Detecting Real-World Performance Bugs." In: *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2012, pp. 77–88.

[58] Fagereng Johansen, Øystein Haugen, and Franck Fleurey. "An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models." In: *Proceedings of the International Software Product Line Conference (SPLC)*. ACM. 2012, pp. 46–55.

[59] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, and Sven Apel. "The Interplay of Sampling and Machine Learning for Software Performance Prediction." In: *IEEE Software* 37.4 (2020), pp. 58–66.

[60] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. "Distance-Based Sampling of Software Configuration Spaces." In: *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE / ACM, 2019, pp. 1084–1094.

[61] Christian Kaltenecker, Stefan Mühlbauer, Alexander Grebhahn, Norbert Siegmund, and Sven Apel. "Performance Evolution of Configurable Software Systems: An Empirical Study." In: *Empirical Software Engineering* 28.6 (2023), 152:1–152:41.

[62] Maurice Kendall. "A New Measure of Rank Correlation." In: *Biometrika* 30.1/2 (1938), pp. 81–93.

[63] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. "TypeChef: Toward Type Checking #ifdef Variability in C." In: *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 2010, pp. 25–32.

[64] Chang Hwan Peter Kim, Christian Kästner, and Don S. Batory. "On the Modularity of Feature Interactions." In: *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 2008, pp. 23–34.

[65] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. "Is There a Mismatch Between Real-World Feature Models and Product-Line Research?" In: *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 291–302.

[66] Alexander Knüppel, Thomas Thüm, Carsten Pardylla, and Ina Schaefer. "Understanding Parameters of Deductive Verification: An Empirical Investigation of KeY." In: *Proceedings of the International Conference on Interactive Theorem Proving (ITP)*. Springer, 2018, pp. 342–361.

[67] Ugur Koc, Austin Mordahl, Shiyi Wei, Jeffrey Foster, and Adam Porter. "SATune: A Study-Driven Auto-Tuning Approach for Configurable Software Verification Tools." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 330–342.

[68] Sergiy S. Kolesnikov, Norbert Siegmund, Christian Kästner, Alexander Grebhahn, and Sven Apel. "Tradeoffs in Modeling Performance of Highly Configurable Software Systems." In: *Software and Systems Modeling.* 18.3 (2019), pp. 2265–2283.

[69] Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Sven Apel. "On the Relation of Control-flow and Performance Feature Interactions: A Case Study." In: *Empirical Software Engineering* 24.4 (2019), pp. 2410–2437.

[70] Lars Kotthoff. "Algorithm Selection for Combinatorial Search Problems: A Survey." In: *Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach*. Vol. 10101. Lecture Notes in Computer Science. Springer, 2016, pp. 149–190.

[71] Samuel Kounev, Klaus-Dieter Lange, and Jóakim von Kistowski. *Systems Benchmarking – For Scientists and Engineers*. Springer, 2020.

[72] Eugene Krause. *Taxicab Geometry: An Adventure in Non-Euclidean Geometry*. Courier Corporation, 1986.

[73] Rahul Krishna, Chong Tang, Kevin J. Sullivan, and Baishakhi Ray. "ConEx: Efficient Exploration of Big-Data System Configurations for Better Performance." In: *Transactions on Software Engineering* 48.3 (2022), pp. 893–909.

[74] William Kruskal and Allen Wallis. "Use of Ranks in One-Criterion Variance Analysis." In: *Journal of the American Statistical Association* 47.260 (1952), pp. 583–621.

[75] Max Kuhn and Kjell Johnson. *Applied Predictive Modeling*. Vol. 26. Springer, 2013.

[76] Donghun Lee, Sang Cha, and Arthur Lee. "A Performance Anomaly Detection and Analysis Framework for DBMS Development." In: *IEEE Transactions on Knowledge and Data Engineering* 24.8 (2012), pp. 1345–1360.

[77] Yu Lei, Raghu Kacker, Richard Kuhn, Vadim Okun, and James Lawrence. "IPOG/IPOG-D: Efficient Test Generation for Multi-Way Combinatorial Testing." In: *Software Testing, Verification and Reliability* 18.3 (2008), pp. 125–148.

[78] Philipp Leitner and Cor-Paul Bezemer. "An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects." In: *Proceedings of the ACM/SPEC on International Conference on Performance Engineering (ICPE)*. ACM, 2017, pp. 373–384.

[79] Luc Lesoil, Mathieu Acher, Arnaud Blouin, and Jean-Marc Jézéquel. "Input Sensitivity on the Performance of Configurable Systems an Empirical Study." In: *Journal of Systems and Software* (2023), p. 111671.

[80] Luc Lesoil, Hugo Martin, Mathieu Acher, Arnaud Blouin, and Jean-Marc Jézéquel. "Transferring Performance between Distinct Configurable Systems : A Case Study." In: *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2022, 10:1–10:6.

[81] Howard Levene. "Robust Tests for Equality of Variances." In: *Contributions to Probability and Statistics. Essays in Honor of Harold Hotelling*. Stanford University Press, 1961, pp. 279–292.

[82] Jia Liang, Vijay Ganesh, Krzysztof Czarnecki, and Venkatesh Raman. "SAT-Based Analysis of Large Real-World Feature Models is Easy." In: *Proceedings of the International Software Product Line Conference (SPLC)*. 2015, pp. 91–100.

[83] Lizhi Liao, Jinfu Chen, Heng Li, Yi Zeng, Weiyi Shang, Jianmei Guo, Catalin Sporea, Andrei Toma, and Sarah Sajedi. "Using Black-Box Performance Models to Detect Performance Regressions Under Varying Workloads: An Empirical Study." In: *Empirical Software Engineering* 25.5 (2020), pp. 4130–4160.

[84] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. "An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines." In: *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE / ACM, 2010, pp. 105–114.

[85] Jörg Liebig, Christian Kästner, and Sven Apel. "Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code." In: *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*. ACM, 2011, pp. 191–202.

[86] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. "Scalable Analysis of Variable Software." In: *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2013, pp. 81–91.

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

[88] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. "Evolution of the Linux Kernel Variability Model." In: *Proceedings of the International Systems and Software Product Line Conference (SPLC)*. Vol. 6287. Lecture Notes in Computer Science. Springer, 2010, pp. 136–150.

[89] Scott Lundberg and Su-In Lee. "A Unified Approach to Interpreting Model Predictions." In: *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*. 2017, pp. 4765–4774.

[90] Haroon Malik, Hadi Hemmati, and Ahmed Hassan. "Automatic Detection of Performance Deviations in the Load Testing of Large Scale Systems." In: *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE / ACM, 2013, pp. 1012–1021.

[91] Henry Mann and Donald Whitney. "On a Test of Whether One of Two Random Variables is Stochastically Larger than the Other." In: *Annals of Mathematical Statististics* 18 (1947), pp. 50–60.

[92] Dusica Marijan, Arnaud Gotlieb, Sagar Sen, and Aymeric Hervieu. "Practical Pairwise Testing for Software Product Lines." In: *Proceedings of the International Software Product Line Conference (SPLC)*. ACM. 2013, pp. 227–235.

[93] Hugo Martin, Mathieu Acher, Juliana Pereira, Luc Lesoil, Jean-Marc Jézéquel, and Djamel Khelladi. "Transfer Learning Across Variants and Versions: The Case of Linux Kernel Size." In: *Transactions on Software Engineering* 48.11 (2022), pp. 4274–4290.

[94] John McGregor. "Testing a Software Product Line." In: *Proceedings of the Pernambuco Summer School on Software Engineering (PSSE)*. Vol. 6153. Lecture Notes in Computer Science. Springer, 2007, pp. 104–140.

[95] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. "A Comparison of 10 Sampling Algorithms for Configurable Systems." In: *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE / ACM, 2016, pp. 643–654.

[96] Daniel Menascé, Virgílio Almeida, Rodrigo Fonseca, and Marco Mendes. "A Methodology for Workload Characterization of E-Commerce Sites." In: *Proceedings of the ACM Conference on Electronic Commerce (EC)*. ACM, 1999, pp. 119–128.

[97] Marcílio Mendonça, Moises Branco, and Donald Cowan. "S.P.L.O.T.: Software Product Lines Online Tools." In: *Proceedings of the Companion to the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2009, pp. 761–762.

[98] Tom Mens and Serge Demeyer, eds. *Software Evolution*. Springer, 2008.

[99] Nicholas Metropolis and Stanislaw Ulam. "The Monte Carlo Method." In: *Journal of the American statistical association* 44.247 (1949), pp. 335–341.

[100] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver." In: *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2008, pp. 337–340.

[101] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. "Accurate Modeling of Performance Histories for Evolving Software Systems." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, 2019, pp. 640–652.

[102] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. "Identifying Software Performance Changes Across Variants and Versions." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 611–622.

[103] Stefan Mühlbauer, Florian Sattler, Christian Kaltenecker, Johannes Dorn, Sven Apel, and Norbert Siegmund. "Analyzing the Impact of Workloads on Modeling the Performance of Configurable Software Systems." In: *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. to appear. IEEE / ACM, 2023.

[104] Raymond Myers. *Classical and Modern Regression with Applications*. Vol. 2. Duxbury press Belmont, 1990.

[105] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter Sweeney. "Producing Wrong Data Without Doing Anything Obviously Wrong!" In: *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2009, pp. 265–276.

[106] Arnaud De Myttenaere, Boris Golden, Bénédicte Le Grand, and Fabrice Rossi. "Mean Absolute Percentage Error for Regression Models." In: *Neurocomputing* 192 (2016), pp. 38–48.

[107] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. "Identifying Software Performance Changes Across Variants and Versions." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, 2020.

[108] Nico Nagelkerke. "A Note on a General Definition of the Coefficient of Determination." In: *Biometrika* 78.3 (1991), pp. 691–692.

[109] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. "Using Bad Learners to Find Good Configurations." In: *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 257–267.

[110] Subhash Narula and John Wellington. "Prediction, Linear Regression and the Minimum Sum of Relative Errors." In: *Technometrics* 19.2 (1977), pp. 185–190.

[111] Thanh Nguyen, Meiyappan Nagappan, Ahmed Hassan, Mohamed Nasser, and Parminder Flora. "An Industrial Case Study of Automatically Identifying Performance Regression-Causes." In: *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. ACM, 2014, pp. 232–241.

[112] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. "Finding Near-Optimal Configurations in Product Lines by Random Sampling." In: *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2017, pp. 61–71.

[113] Robert M O'brien. "A Caution Regarding Rules of Thumb for Variance Inflation Factors." In: *Quality & quantity* 41 (2007), pp. 673–690.

[114] Leonardo Passos and Krzysztof Czarnecki. "A Dataset of Feature Additions and Feature Removals from the Linux Kernel." In: *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. ACM, 2014, pp. 376–379.

[115] Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesús Padilla. "A Study of Feature Scattering in the Linux Kernel." In: *IEEE Transactions on Software Engineering (TSE)*. IEEE, 2018.

[116] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wasowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. "Coevolution of Variability Models and Related Software Artifacts - A Fresh Look at Evolution Patterns in the Linux Kernel." In: *Empirical Software Engineering* 21.4 (2016), pp. 1744–1793.

[117] Xin Peng, Yijun Yu, and Wenyun Zhao. "Analyzing Evolution of Variability in a Software Product Line: From Contexts and Requirements to Features." In: *Information & Software Technology* 53.7 (2011), pp. 707–721.

[118] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, and Jean-Marc Jézéquel. "Sampling Effect on Performance Prediction of Configurable Systems: A Case Study." In: *Proceedings of the ACM/SPEC on International Conference on Performance Engineering (ICPE)*. ACM, 2020.

[119] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. "Product Sampling for Product Lines: The Scalability Challenge." In: *Proceedings of the International Systems and Software Product Line Conference (SPLC)*. ACM, 2019, 14:1–14:6.

[120] Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. "Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?" In: *Proceedings of the International Conference on Software Testing, Validation and Verification, (ICST)*. IEEE, 2019, pp. 240–251.

[121] Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schröter, and Gunter Saake. "JavAdaptor - Flexible Runtime Updates of Java Applications." In: *Software: Practice and Experience* 43.2 (2013), pp. 153–185.

[122] David Reichelt and Stefan Kühne. "How to Detect Performance Changes in Software History: Performance Analysis of Software System Versions." In: *Proceedings of the International Conference on Performance Engineering (ICPE)*. ACM, 2018, pp. 183–188.

[123] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. "Variability-Aware Static Analysis at Scale: An Empirical Study." In: *Transactions on Software Engineering and Methodology* 27.4 (2018), 18:1–18:33.

[124] Marco Ribeiro, Sameer Singh, and Carlos Guestrin. ""Why Should I Trust You?": Explaining the Predictions of Any Classifier." In: *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining(SIGKDD)*. 2016, pp. 1135–1144.

[125] Pau Rodríguez, Miguel Bautista, Jordi Gonzàlez, and Sergio Escalera. "Beyond One-Hot Encoding: Lower Dimensional Target Embedding." In: *Image and Vision Computing* 75 (2018), pp. 21–31.

[126] Peter Rousseeuw. "Silhouettes: A Graphical Aid to the Interpretation and Validation of Cluster Analysis." In: *Journal of Computational and Applied Mathematics* 20 (1987), pp. 53–65.

[127] Per Runeson, Martin Höst, Austen Rainer, and Björn Regnell. *Case Study Research in Software Engineering - Guidelines and Examples*. Wiley, 2012.

[128] Andrea Saltelli. *Global Sensitivity Analysis: The Primer*. Wiley, 2008.

[129] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. "Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T)." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 342–352.

[130] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. "A Comparison of Decision Modeling Approaches in Product Lines." In: *Proceeddings of the International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM International Conference Proceedings Series. ACM, 2011, pp. 119–126.

[131] Ismael Seidel, Bruno de Moraes, Emilio Wuerges, and José Güntzel. "Quality Assessment of Subsampling Patterns for PEL Decimation Targeting High Definition Video." In: *Proceedings of the International Conference on Multimedia and Expo (ICME)*. IEEE, 2013, pp. 1–6.

[132] Christoph Seidl, Florian Heidenreich, and Uwe Aßmann. "Co-Evolution of Models and Feature Mapping in Software Product Lines." In: *Proceedings of the International Software Product Line Conference on (SPLC)*. ACM, 2012, p. 76.

[133] Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep Meel. "Knowledge Compilation meets Uniform Sampling." In: *Proceedings of the International Conference on Logic for Programming, Artificial Intelligence and Reasoning (ICLP)*. Vol. 57. EPiC Series in Computing. EasyChair, 2018, pp. 620–636.

[134] Steven She. "Feature Model Synthesis." PhD thesis. University of Waterloo, Canada, 2013.

[135] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. "The Variability Model of The Linux Kernel." In: *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. Vol. 37. ICB-Research Report. Universität Duisburg-Essen, 2010, pp. 45–51.

[136] Simon Sheather. *A Modern Approach to Regression with R*. Springer Science & Business Media, 2009.

[137] Galit Shmueli. "To Explain or to Predict?" In: *Statistical Science* 25.3 (2010), pp. 289–310.

[138] Norbert Siegmund. "Measuring and Predicting Non-Functional Properties of Customizable Programs." PhD thesis. Otto-von-Guericke-Universität Magdeburg, 2012.

[139] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. "Performance-Influence Models for Highly Configurable Systems." In: *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE / ACM, 2015, pp. 284–294.

[140] Norbert Siegmund, Sergiy Kolesnikov, Christian Kästner, Sven Apel, Don S. Batory, Marko Rosenmüller, and Gunter Saake. "Predicting Performance via Automated Feature-Interaction Detection." In: *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE / ACM, 2012, pp. 167–177.

[141] Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo Giarrusso, Sven Apel, and Sergiy Kolesnikov. "Scalable Prediction of Non-functional Properties in Software Product Lines: Footprint and Memory Consumption." In: *Information & Software Technology* 55.3 (2013), pp. 491–507.

[142] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. "SPL Conqueror: Toward Optimization of Non-Functional Properties in Software Product Lines." In: *Software Quality Journal* 20.3-4 (2012), pp. 487–517.

[143] Kevin Skadron, Margaret Martonosi, David August, Mark Hill, David Lilja, and Vijay Pai. "Challenges in Computer Architecture Evaluation." In: *Computer* 36.8 (2003), pp. 30–36.

[144] George Snedecor and Cochran William. *Statistical Methods*. Tech. rep. International Statistical Institute, 1989.

[145] Chico Sundermann, Kevin Feichtinger, Dominik Engelhardt, Rick Rabiser, and Thomas Thüm. "Yet Another Textual Variability Language?: A Community Effort Towards a Unified Language." In: *Proceedings of the International Systems and Software Product Line Conference (SPLC)*. ACM, 2021, pp. 136–147.

[146] Syncsort. *Assessing the Financial Impact of Downtime*. 2018.

[147] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. "Configuration Coverage in the Analysis of Large-Scale System Software." In: *Operating Systems Review* 45.3 (2011), pp. 10–14.

[148] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. "Towards Efficient Analysis of Variation in Time and Space." In: *Proceedings of the International Systems and Software Product Line Conference (SPLC)*. ACM, 2019, 69:1–69:8.

[149] Mauro Vallati, Lukás Chrpa, Marek Grzes, Thomas Leo McCluskey, Mark Roberts, and Scott Sanner. "The 2014 International Planning Competition: Progress and Trends." In: *AI Magazine* 36.3 (2015), pp. 90–98.

[150] Andras Vargha and Harold Delaney. "A Critique and Improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong." In: *Journal of Educational and Behavioral Statistics* 25.2 (2000), pp. 101–132.

[151] Elaine Weyuker and Filippos Vokolos. "Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study." In: *Transactions on Software Engineering* 26.12 (2000), pp. 1147–1156.

[152] 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: *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 307–319.

[153] Andy Yoo, Morris Jette, and Mark Grondona. "SLURM: Simple Linux Utility for Resource Management." In: *Proceedings of the International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*. Vol. 2862. Lecture Notes in Computer Science. Springer, 2003, pp. 44–60.

[154] Shahed Zaman, Bram Adams, and Ahmed Hassan. "Security Versus Performance Bugs: A Case Study on Firefox." In: *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, 2011, pp. 93–102.

[155] Shahed Zaman, Bram Adams, and Ahmed Hassan. "A Qualitative Study on Performance Bugs." In: *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. IEEE, June 2012, pp. 199–208.

[156] Neng-Fa Zhou and Jonathan Fruhman. "Toward a Dynamic Programming Solution for the 4-peg Tower of Hanoi Problem with Configurations." In: *Computing Research Repository* abs/1301.7673 (2013).

[157] Pieter van Zyl, Derrick Kourie, and Andrew Boake. "Comparing the Performance of Object Databases and ORM Tools." In: *Proceedings of the Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries (SAICSIT)*. South African Institute for Computer Scientists and Information Technologists, 2006, pp. 1–11.

A Appendix

In this chapter, we present further material for this thesis. In Section A.1, we list all constraints of the exemplary subject system Comp after one-hot encoding (see Section 2.1.5). The remainder is supplementary material for Chapter 5. In particular, we present the feature model from FastDownward in Figure A.1 and the exact recall and precision values from Section 5.4.1 in Table A.1.

A.1 Constraints after One-Hot Encoding

In the following, we present the constraints of the subject system Comp after the one-hot encoding:

$c(root)$

$c(\textsc{Encryption}) \Rightarrow c(root)$

$c(\textsc{Compression}) \Rightarrow c(root)$

$c(\textsc{CompressionLevel})$

$c(\textsc{CompressionLevel}) \Rightarrow c(root)$

$c(\textsc{CompressionLevel\_0}) \Rightarrow c(\textsc{CompressionLevel})$

$c(\textsc{CompressionLevel\_1}) \Rightarrow c(\textsc{CompressionLevel})$

$c(\textsc{CompressionLevel\_2}) \Rightarrow c(\textsc{CompressionLevel})$

$c(\textsc{Algorithm}) \Rightarrow c(\textsc{Compression})$

$c(\text{A1}) \Rightarrow c(\textsc{Algorithm})$

$c(\text{A2}) \Rightarrow c(\textsc{Algorithm})$

$c(\textsc{Compression}) \Rightarrow c(\textsc{Algorithm})$

$c(\textsc{Algorithm}) \Rightarrow (c(\text{A1}) \vee c(\text{A2}))$

$c(\text{A1}) \Rightarrow \neg c(\text{A2})$

$c(\text{A2}) \Rightarrow \neg c(\text{A1})$

$c(\textsc{CompressionLevel}) \Rightarrow (c(\textsc{CompressionLevel\_0}) \vee c(\textsc{CompressionLevel\_1}) \vee c(\textsc{CompressionLevel\_2}))$

$c(\textsc{CompressionLevel\_0}) \Rightarrow \neg c(\textsc{CompressionLevel\_1})$

$c(\textsc{CompressionLevel\_0}) \Rightarrow \neg c(\textsc{CompressionLevel\_2})$

$c(\textsc{CompressionLevel\_1}) \Rightarrow \neg c(\textsc{CompressionLevel\_0})$

$c(\textsc{CompressionLevel\_1}) \Rightarrow \neg c(\textsc{CompressionLevel\_2})$

$c(\textsc{CompressionLevel\_2}) \Rightarrow \neg c(\textsc{CompressionLevel\_0})$

$c(\textsc{CompressionLevel\_2}) \Rightarrow \neg c(\textsc{CompressionLevel\_1})$

$c(\textsc{Compression}) \Rightarrow (c(\textsc{CompressionLevel\_1}) \vee c(\textsc{CompressionLevel\_2}))$

$c(\textsc{CompressionLevel\_0}) \Rightarrow \neg c(\textsc{Compression})$

Table A.1: Precision and recall for each workload of FastDownward.

Workload	Precision	Recall
airport/p07airport2p2	92.31%	41.38%
barmanopt11strips/pfile01001	91.78%	68.79%
blocks/probBLOCKS100	96.83%	77.56%
datanetworkopt18strips/p13	80.00%	77.86%
depot/p10	87.80%	89.71%
driverlog/p08	100.00%	34.01%
elevatorsopt08strips/p24	100.00%	76.22%
elevatorsopt11strips/p20	95.24%	86.88%
floortileopt11strips/optp01002	83.33%	89.55%
freecell/prob45	93.33%	90.38%
gedopt14strips/d76	64.00%	10.34%
grid/prob02	82.35%	74.07%
gripper/prob07	89.47%	67.28%
hikingopt14strips/ptesting234	90.16%	72.33%
logistics00/probLOGISTICS71	98.00%	84.91%
logistics98/prob31	72.73%	54.22%
miconic/s132	92.59%	48.89%
movie/prob29	37.98%	100.00%
mprime/prob02	95.65%	86.55%
mystery/prob30	71.43%	64.66%
nomysteryopt11strips/p17	100.00%	40.94%
openstacksopt08strips/p12	75.00%	18.75%
openstacksopt11strips/p07	68.97%	20.41%
openstacksopt14strips/p20/3	64.00%	13.71%
openstacksstrips/p07	88.89%	74.62%
organicsynthesisopt18strips/p09	72.97%	53.12%
organicsynthesissplitopt18strips/p01	100.00%	50.00%
parcprinter08strips/p25	93.88%	43.23%
parcprinteropt11strips/p10	94.00%	46.20%
pathways/p04	77.36%	76.24%
pegsol08strips/p25	80.56%	59.06%
pegsolopt11strips/p15	91.18%	33.33%
pipesworldnotankage/p12net2b10g4	86.21%	20.49%
pipesworldtankage/p31net4b14g3t20	83.87%	35.57%
psrsmall/p48s101n5l3f30	85.71%	64.29%
rovers/p07	96.88%	80.27%
satellite/p06pfile6	85.07%	88.65%
scanalyzer08strips/p06	100.00%	72.67%
scanalyzeropt11strips/p12	98.53%	66.48%
snakeopt18strips/p04	92.86%	12.63%
sokobanopt08strips/p22	90.48%	80.77%

sokobanopt11strips/p17	98.39%	66.45%
storage/p14	87.80%	71.54%
termesopt18strips/p18	100.00%	64.57%
tetrisopt14strips/p024	87.88%	40.70%
tidybotopt11strips/p01	96.43%	43.08%
tpp/p06	94.23%	76.09%
transportopt08strips/p04	94.44%	71.32%
transportopt11strips/p06	92.86%	69.63%
transportopt14strips/p14	100.00%	69.50%
trucksstrips/p08	90.00%	37.34%
visitallopt11strips/problem06full	83.33%	24.39%
visitallopt14strips/p057	95.52%	92.57%
woodworkingopt08strips/p24	93.94%	47.10%
woodworkingopt11strips/p05	90.32%	46.43%
zenotravel/p11	100.00%	35.92%

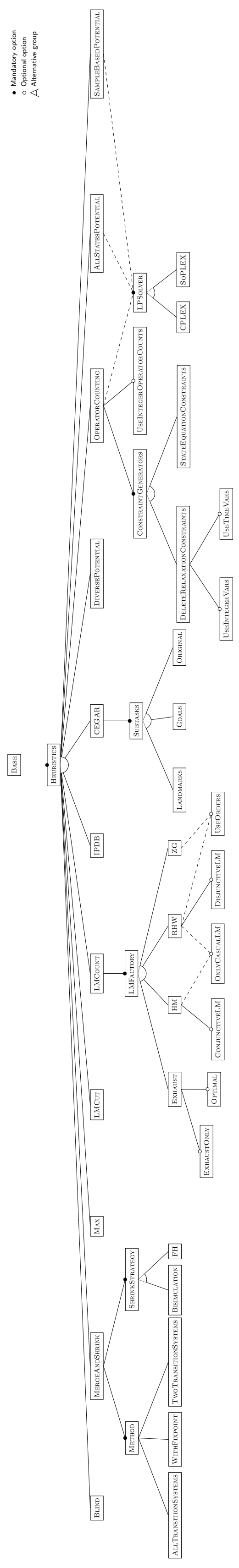


Figure A.1: Feature Model of FastDownward