Master's Thesis

THE BEST OF BOTH WORLDS: INTEGRATING WHITE-BOX WITH BLACK-BOX PERFORMANCE ANALYSIS

NICO BUCHHOLZ

March 1, 2023

Advisors:
Christian Kaltenecker Chair of Software Engineering
Florian Sattler Chair of Software Engineering

Examiners:
Prof. Dr. Sven Apel Chair of Software Engineering
Prof. Dr. Jan Reineke Real-Time and Embedded Systems Lab

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

UNIVERSITÄT DES SAARLANDES

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____

(Datum/Date) (Unterschrift/Signature)

ABSTRACT

For configurable software systems, the configuration of the software system can have an impact on its performance. However, the impact may change when the software evolves. During the evolution of a software system, the performance of some configurations might improve, while the performance of others could degrade. Additionally, the workload to be used with the system also has an impact on the performance.

While prior work exists that investigates each dimension, i. e., configurations, releases, and workloads, individually, there is a lack of research that examines multiple dimensions at the same time. Furthermore, existing publications mostly use either black-box or white-box performance analysis. However, it remains unclear if both approaches deliver comparable results and how well both approaches can be integrated.

In the thesis, we implement an approach that integrates white-box with black-box performance analysis. Using our approach, we conduct an empirical study to investigate the influence of configuration options and interactions on the performance in different releases with multiple workloads. We examine an artificial case study to demonstrate the general feasibility of our approach and, additionally, four real-world case studies to investigate how well an integrated approach can be applied to actual software systems. In our approach, we first use a black-box performance analysis to identify changes in the influence of configuration options or interactions. Then, we conduct a white-box performance analysis to perform a deeper investigation of the detected performance changes.

Our results indicate that, under optimal conditions, an integration approach works and provides valuable insights into the software system under investigation. For real-world case studies, we observe and investigate different sources of errors and problems when applying an integrated approach. Nevertheless, the integration approach provides insightful pointers for further manual investigation in many cases. We pinpoint areas that could profit from enhancement and provide pointers for subsequent research to improve black-box and white-box performance analysis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

VIF Variance Inflation Factor

HTN Hierarchical Task Network

GED Genome Edit Distances

NUMA Non-Uniform Memory Access

1 INTRODUCTION

In modern software development, many software systems are highly configurable. Due to the configurability, users can tailor the behavior and functionality of such software systems to their needs and to the environment in which the software system runs. However, determining the best configuration for the own use case is highly non-trivial. One reason for that is the fact that the configuration of a software system can have a huge impact on its performance [5, 15, 24]. Yet, it is often not clear how each configuration option influences the performance of the system. Additionally, multiple configuration options might interact with each other. Thus, the performance of a software system does not only depend on individual configuration options, but also on configuration option interactions. Furthermore, prior work [5, 18] has shown that the workload of a software system has a significant impact on the performance of the system as a whole, but also on the influence of individual configuration options and interactions. Hence, this is another factor that needs consideration when analyzing the performance of a software system. Moreover, we also need to acknowledge the evolution of software systems. Over time, developers publish new releases that might also change the performance of a software system [15]. Overall, this leads to three dimensions that are of particular relevance when investigating the performance of software systems: configurations, workloads, and releases.

To determine the impact of configuration options or option interactions on the performance, prior work introduces two contrary methods. First, black-box performance analysis only considers the system's behavior that is directly observable. In the context of performance investigations, we can execute the software system with a sample of configurations and measure the run-time for each configuration. From this measurement data, we can then draw conclusions about the impact of individual configuration options and option interactions, e. g., by applying machine-learning techniques. A different approach is white-box performance analysis. When conducting such an analysis, we have access to the internals, such as the source code, of the software system under investigation. By instrumenting the code, we can then measure the run-time of each code region and associate code regions to configuration options. This way, we can infer the impact of configuration options and option interactions on the performance.

When comparing the performance of software systems across dimensions, both approaches have their own advantages, but also their own problems. While conducting a black-box performance analysis is conceptually easy, it does not provide any information about the internals of the software system. In particular, this means that black-box performance analysis can determine whether there is a performance change and it can associate performance changes with configuration options or interactions. It is, however, not able to provide any insights about the internal changes in the subject system that caused the performance changes. In contrast, white-box performance analysis can supply hints about the cause of a performance change, e. g., by investigating changes in the code of the software system. However, a white-box performance analysis requires a lot of manual

effort and is generally complex to implement. Additionally, executing a software system while conducting a white-box performance analysis takes more time than usual due to the overhead introduced by the instrumentation of the code.

In this thesis, we introduce and evaluate an approach that combines the advantages of both approaches when comparing the performance across releases by integrating white-box with black-box performance analysis. In particular, we first conduct a black-box performance analysis on a sample of configurations with different workloads on multiple releases. Based on the results of the black-box analysis, we identify the configuration options and interactions, workloads, and release pairs that exhibit performance changes. Then, we further investigate these configuration options and interactions, workloads, and releases with a white-box performance analysis and analyze the resulting data to identify the cause of the performance changes. This way, we gather the additional information that a white-box performance analysis is able to deliver while we avoid investigating an infeasible amount of configurations with a white-box performance analysis.

To evaluate our approach, we examine an artificial case study to demonstrate the general feasibility of our approach and four real-world case studies to investigate the practical applicability of an integrated approach to actual software systems. Our results demonstrate the general usefulness of an integrated approach under optimal conditions. For the real-world case studies, we observe a limited utility of the integrated approach for more complex software systems, mainly due to missing capabilities of the used white-box performance analysis framework. Nevertheless, the integrated approach provides insightful pointers for further manual investigation in many cases. We highlight multiple limitations of both, black-box and white-box performance analysis, and provide pointers for potential improvements.

The remainder of this thesis is structured as follows. In Chapter 2, we introduce the concepts and terms that are essential to understand this thesis. Chapter 3 presents related publications and their relevance to our work. In Chapter 4, we outline the methodology that we use, i. e., our research questions and their operationalization. Afterwards, we present the evaluation of the research questions and possible threats to the validity of our results in Chapter 5. Finally, we conclude the thesis in Chapter 6 by summarizing the most important aspects of our work and providing pointers for future work.

2 BACKGROUND

In this chapter, we provide all relevant information that is required to understand this thesis. We start by describing three different dimensions along which we analyze software systems. First, we give an overview of configurable systems to introduce the dimension of configurations. Then, we describe the most important aspects of software evolution related to our topic. The last dimension we describe treats different workloads that systems can be confronted with. Afterwards, we illustrate different methods of performance analysis for software systems. We focus on black-box and white-box performance analysis and their differences.

2.1 CONFIGURABLE SYSTEMS

Many software systems offer multiple (configuration) options. A configuration option is a setting in a software system that can be adjusted by the user of the system and which changes the behavior of the system or adapts the system to the environment that it runs in. For example, a file compression tool might offer two configuration options, *Compression* and *Encryption*, that enable the compression or encryption of files, respectively.

Configuration options can also be put into a parent-child relationship. Semantically, we use this relationship to express that child options can only be active if the parent option is active. For instance, in addition to the *Compression* option, the tool could allow the user to decide which compression algorithm should be used. For this, we add a *CompressionAlgorithm* option as a child of the *Compression* option. The *CompressionAlgorithm* has again two children, e. g., *LZMA* and *LZMA*2, which represent the different algorithms. Applying our semantic, this means that the *CompressionAlgorithm* option can only be active if we enable *Compression* and we can only select *LZMA* or *LZMA*2 if we decide to manually choose the compression algorithm, i. e., we enable the *CompressionAlgorithm* option. By defining these relationships, we receive a directed tree structure representing the configuration options.

Configuration options are either mandatory or optional. If a configuration option is mandatory, it must always be enabled if its parent is enabled. If a mandatory configuration option does not have a parent, i. e., if it is the root of our tree, it must always be enabled. Otherwise, it is up to the user to enable the configuration option. In the context of our example, *Compression* could be a mandatory option since compression is the core task of a file compression tool. In contrast, *Encryption* could be an optional configuration option that the user can enable or disable. Similarly, we can specify that exactly one of the children of a configuration option must be active. Since it seems reasonable that only one compression algorithm can be used, either *LZMA* or *LZMA*2, but not both, must be active if the *CompressionAlgorithm* option is enabled. In such cases, we say that the options form an alternative group.

Generally, we differentiate between two different types of configuration options: run-time options and compile-time options. Run-time options are options that can be changed before

or during the execution of the system whereas compile-time options are options that must be set before compiling the program. In this thesis, we focus on run-time options.

Furthermore, configuration options are either binary or numeric. A binary configuration option is an option that can only be turned on or off. In our previous file compression example, all configuration options we defined so far would be binary. In contrast, a numeric configuration option can take a fixed number of numeric values in a fixed range. For the file compression example, a numeric option *MaxMemory_val* could specify the amount of main memory that the system is allowed to use. If we have 1 GiB of main memory available, a possible range for *MaxMemory_val* is $(0, 1073741824]$, specifying the value in bytes. As an additional option, we could have a binary option *MaxMemory* as a parent of *MaxMemory_val* that indicates whether the main memory that the system is allowed to use is limited or not.

We call a software system configurable if it provides configuration options. We denote the set of all configuration options of a system S by $\mathcal{O}_S$. In our compression tool example, we thus have

$$\mathcal{O}_S = \{Compression, CompressionAlgorithm, LZMA, LZMA2, Encryption, \\ MaxMemory, MaxMemory\_val\}.$$

A configuration of a configurable software system describes instances of all configuration options of the system, i. e., in a configuration, each option is assigned a value. We denote the configuration space, i. e., the set of all configurations of S, by $\mathcal{C}_S$. We formalize the concept of a configuration by closely sticking to a definition introduced by Siegmund et al. [24].

Definition 1. A *configuration* c for a *software* S is a total function

$$c : \mathcal{O}_S \to \{0, 1\}$$

assigning a binary value to every *configuration option*.

Applying this formal definition to our previous file compression tool example, we would, e. g., set $c(Encryption) = 1$ if encryption is enabled and $c(Encryption) = 0$ if it is disabled. While this formal definition seemingly only considers binary options, we can discretize numeric options to make this definition applicable. A numeric option can be modeled as multiple binary options by including a binary option for each of the values in the range of the numeric option and enabling the binary option with the desired value and disabling all others. We name the binary options by suffixing the name of the numeric option with the value of the option. For example, for *MaxMemory_val*, we generate the binary options $MaxMemory\_val\_1, \ldots, MaxMemory\_val\_1073741824$ where $c(MaxMemory\_val\_x) = 1$ if and only if *MaxMemory_val* has the value x. In the remainder of this thesis, we always discretize numeric options if not specified otherwise.

To formally model the configuration options of a system and their properties and relations, feature models can be used. A feature model is a model that contains all configuration options and their relations along with information about the options, e. g., whether the option is binary or numeric, or the range of allowed values in the case of a numeric feature. Depending on the intended use of the feature model, further information about the options, such as code locations, can be included.

Additionally, we can model constraints that configurations must fulfill. In the context of our file compression tool, we could, just for the sake of an example, say that if we use $LZMA2$, the memory must not be limited, i. e., $MaxMemory$ must not be active. Formally, we can define constraints as Boolean expressions.

Definition 2. A *constraint* b for a *software* S is a Boolean expression with variables x_i where $x_i \in \mathcal{O}_S$.

With these constraints, we can also formalize concepts like parent-child relationships and alternative groups by converting them to Boolean expressions. For instance, for our compression algorithm alternative group, we could use the expression

$$(LZMA \implies \neg LZMA2) \wedge (LZMA2 \implies \neg LZMA)$$

to specify that only one of both options can be active at the same time. We denote the set of all constraints for a software system S by $\mathcal{B}_S$. Using the concept of constraints, we can now formalize which configurations are valid.

Definition 3. A *configuration* $c \in \mathcal{C}_S$ for a *software* S is *valid* if it fulfills all $b \in \mathcal{B}_S$.

Since feature models are directed trees, it can ease understanding the models if we draw them. In Figure 2.1, we depict a visualization of an exemplary feature model based on the compression tool example that we previously described. We have a mandatory root configuration option that is always on. Then, we have the compression algorithm configuration option with two mutually exclusive child options and the encryption configuration option. Lastly, we have a binary configuration option to enable a memory limit. This binary option has a numeric child option that we use to specify the memory limit.

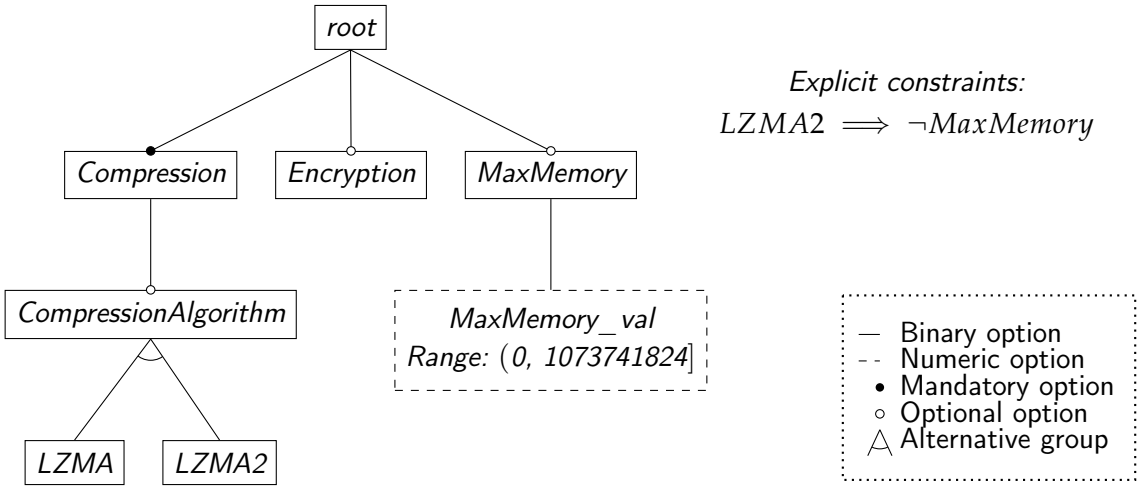


Figure 2.1: A visualization of an exemplary minimal feature model representing a compression tool

2.2 SOFTWARE EVOLUTION

Usually, the development of a software is a continuous process over a long time span. During this process, bugs are fixed, new functionality is added, or existing functionality

is enhanced. Since the code evolves over time, developers commonly make use of version control systems to track code changes over time. A certain state of the code base is called a revision. Each software has a finite amount of revisions. Hence, we can assign a unique numeric ID to each revision. Thus, we can formally define the set of all revisions as follows.

Definition 4. Let S be a *software* with $i \in \mathbb{N}$ *revisions*. $\mathcal{RV}_S = \{r_1, r_2, r_3, \ldots, r_i\}$ defines the set of all *revisions* of S.

Developers regularly declare some revisions as releases. Kaltenecker et al. [15] define releases as revisions that are expected to run stable, contain prominent changes, or highlight development milestones that have been reached. Revisions that are not releases usually represent an intermediate state of the software that might contain incomplete or faulty implementations of features, have not yet been fully tested, and are not expected to run without any problems. Hence, we only consider releases for the studies in this thesis. Formally, we define the set of releases as a subset of the set of revisions.

Definition 5. Let S be a *software* with *revisions* $\mathcal{RV}_S$. $\mathcal{R}_S \subseteq \mathcal{RV}_S$ defines the set of all *releases* of S.

Sometimes, releases are additionally tagged as "alpha" or "beta" [3]. While there are no fixed definitions for these tags, they commonly indicate that the tagged release is an early version of the release that is not thoroughly tested. Usually, "alpha" releases can still contain major bugs while "beta" releases are mainly expected to only come with some minor problems.

Releases almost always change the behavior of a system in some way. For example, the run-time of the system could decrease with a new release, e. g., due to code optimizations. In the context of configurable systems, releases also play an important role. New releases frequently introduce new configuration options or alter the implementation or behavior of existing ones. When looking at the development of a configurable system over time, we see that the set of configuration options and their properties might change. If we again consider our exemplary compression tool, a new release could, e. g., introduce a new compression algorithm that can be used. Similarly, a new release may improve and speed up an existing implementation of a compression algorithm. However, changes in a release might also, usually unintentionally, introduce, e. g., a run-time regression. In that case, the release slows down some configuration options.

2.3 WORKLOADS

The input provided to a software system can have an influence on the behavior and the non-functional properties of the system. In the context of this thesis, we call this input a workload. If we recall our compression tool example, the workload would be the file to be compressed. To formally describe the set of all workloads, we use the following definition.

Definition 6. Let S be a *software* that is executed with $i \in \mathbb{N}$ *workloads*.
$\mathcal{W}_S = \{w_1, w_2, w_3, \ldots, w_i\}$ defines the set of all *workloads* of S.

Workloads can have a lot of properties that heavily influence the system's behavior [5]. Regarding the compression tool, these properties could include the size of the file or the file

type. For instance, it seems reasonable that there is some correlation between the workload size and the run-time of the compression tool. Similarly, Powers [21] shows that the file type influences the compressibility of a file. For example, while files containing text often have a high compressibility, image files such as JPEG files have, on average, a lower compressibility since JPEG is already a compressed file format.

In the context of configurable software systems, the run-time of a system does not only depend on the configuration of the system, but also on the workload. In particular, the combination of configuration and workload can heavily influence the run-time or other non-functional properties [5]. While some configuration might be beneficial for the run-time of the system when executed with some workload A, it might be disadvantageous for the execution of the system with workload B.

2.4 PERFORMANCE ANALYSIS

Software systems can, e. g., be evaluated by measuring different non-functional properties. A commonly evaluated non-functional property of software systems is their performance. The term performance is frequently defined as the amount of work done per time unit. In the context of analyzing configurable software systems, we often examine the software with a fixed amount of work. If the amount of work is fixed, evaluating the amount of work done per time unit is equivalent to just measuring the time. Previous publications [5, 6, 15, 24] have shown that the configuration of a software system affects its performance. First, each configuration option itself has a certain effect on the performance of a system. In the context of our compression tool example, enabling encryption might increase the run-time of the tool. Furthermore, interactions between configuration options can also affect the performance. Depending on the compression algorithm that was used to compress the file, the tool might take more or less time to encrypt the file since the size of the compressed file can vary. Thus, the performance of a system is dependent on the configuration that is used. Considering all of the above, we can use the following definition for performance in the context of this thesis.

Definition 7. The *performance* p of a *configurable software* S under a *configuration* c is defined as the *run-time* of this software when executing it with this configuration c:

$$p_c(S) := \textit{run-time}_c(S)$$

To understand and improve the performance of a software system, it is helpful to analyze the performance of the system.

2.4.1 *Black-Box Performance Analysis*

One way to analyze the performance of a software system is to perform a black-box performance analysis. In this approach, we consider the software system to be a black box, i. e., we only consider the behavior that is observable from the outside. We do not know anything about the internals, such as the code, of the software system.

When performing a black-box performance analysis, we measure the performance of a sample of configurations of the system in a stable environment. It is then possible to analyze

these measurements and to draw some conclusions from them. One established way to perform the analysis is to use performance-influence models.

Performance-Influence Models

A performance-influence model is a mathematical model introduced by Siegmund et al. [24] that predicts the expected performance of a configurable software system given a configuration. Formally, a performance-influence model can be defined as follows [24].

Definition 8. Let S be a *configurable* software system and let C_S be the set of all *configuration options* of S. A *performance-influence model* of S is a total function Π_S of the form

$$\Pi_S(c) = \beta_0 + \sum_{i \in C_S} \phi_i(c(i)) + \sum_{i..j \in C_S} \Phi_{i..j}(c(i)..c(j)),$$

that, given a configuration c, returns the expected performance of S when executing S with c. β_0 denotes the base value, $\sum_{i \in C_S} \phi_i(c(i))$ denotes the influence of the configuration options and $\sum_{i..j \in C_S} \Phi_{i..j}(c(i)..c(j))$ denotes the influence of the configuration option interactions.

Going back to our compression tool example, an exemplary performance-influence model could look as follows.

$$\begin{aligned}\Pi_S(c) = 1 &+ 2 \cdot c(Compression) + 3 \cdot c(Encryption) \\ &+ 1 \cdot c(Encryption) \cdot c(LZMA) - 1 \cdot c(Encryption) \cdot c(LZMA2)\end{aligned}$$

This performance-influence model tells us that our compression tool has a base value of one second that is always needed to execute the system, regardless of the configuration. If we enable compression or encryption, two or three seconds are added to the run-time of our system, respectively. If we compress the data with $LZMA$ and encrypt it, the run-time increases by one second. Contrary, if we use $LZMA2$ for compression and encrypt the data, we deduct one second. A possible explanation for this would be that $LZMA2$ has a higher compression rate than $LZMA$, decreasing the size of the file that needs to be encrypted.

When creating a performance-influence model for a real software system, the coefficients β_0, ϕ_i, and $\Phi_{i..j}$ are determined by using a multiple linear regression approach with feature forward selection on the measurement data obtained by the performance measurements. In our research, we use the tool SPL Conqueror[1] which implements this approach to learn the coefficients.

As described above, by analyzing these coefficients, it is possible to see the influence of each configuration option or configuration option interaction on the performance of the system. While this can already deliver useful information about, e.g., possible bottlenecks of a software, the analysis can be further enhanced by comparing performance-influence models across multiple dimensions. As discussed in the previous two sections, the performance of a system does not only depend on the configuration, but also on the releases and workloads that are used. Hence, it can be useful to consider these two dimensions when working with performance-influence models.

1 SPL Conqueror. Available online at https://github.com/se-sic/SPLConqueror; visited on October 13th, 2022.

Considering our compression tool example, we could have the following performance-influence model for our tool at release i.

$$\Pi_{S_i}(c) = 1 + 2 \cdot c(Compression) + \mathbf{3} \cdot c(Encryption)$$

In release $i+1$, an encryption algorithm that is more secure, but slower could be implemented. Thus, encryption would take more time and, hence, the influence on the performance of our system changes. This is reflected in our performance-influence model by adjusting the coefficient of the *Encryption* configuration option.

$$\Pi_{S_i}(c) = 1 + 2 \cdot c(Compression) + \mathbf{5} \cdot c(Encryption)$$

Similarly, performance improvements might be made or performance regressions might be unintentionally introduced. Consequently, analyzing performance-influence models can help to evaluate whether the performance has improved as expected or to identify potential performance bugs. Kaltenecker et al. [15] perform such an analysis and compare changes of the coefficients of performance-influence models across multiple releases of the same software. A more detailed description of their findings can be found in Chapter 3.

Similar analyses can be performed for different workloads. Dincher [5] analyzes performance-influence models for the same software system across multiple workloads. Again, Chapter 3 contains an overview of the conducted work.

Multicollinearity

An important aspect to consider when comparing performance-influence models is their interpretability and comparability. If we, e. g., compare performance-influence models of the same software across releases, it is important that we know whether changes in the coefficients of the model are caused by actual changes in the software or if they are caused by some other factors. One factor that can cause such changes in the coefficients of a linear model without changes in the software is multicollinearity. In the context of linear regression, we speak of multicollinearity if one of the feature variables is (almost) equal to a linear combination of other feature variables [2]. If we have an exact equality, we speak of perfect multicollinearity. If perfect multicollinearity is present in the feature variables, the coefficients determined by the linear regression are no longer unique [23]. Since our performance-influence models are linear models, we have to take this into account.

Consider the following two performance-influence models for our compression tool example.

$$\Pi_S(c) = 1 + 3 \cdot c(LZMA2) + 1 \cdot c(Compression) \cdot c(LZMA2)$$
$$\Pi'_S(c) = 1 + 1 \cdot c(LZMA2) + 3 \cdot c(Compression) \cdot c(LZMA2)$$

If we again take a look at our feature model in Figure 2.1, we see that $LZMA2$ is a child of *Compression*. Hence, if $c(LZMA2) = 1$, then we also know that $c(Compression) = 1$. Thus, we can conclude that $c(LZMA2) = c(Compression) \cdot c(LZMA2)$ and, consequently, Π_S=Π'_S. Taking these thoughts into account, we can form infinitely many performance-influence models that predict the same values, but have different coefficients. To solve the issue, we can simply remove one of the terms $c(LZMA2)$ and $c(Compression) \cdot c(LZMA2)$ from the

model. If we do this, the coefficients are uniquely determined and, hence, our model is unique but still has the same predictive power.

The example above can be easily detected manually when inspecting the feature model. However, there can be other non-trivial cases where we have a similar situation. This holds true especially if more than two or three features are involved in such a dependency. Hence, an automated solution is desirable to detect multicollinearity. One way to do this is by performing a Variance Inflation Factor (VIF) analysis [14] as proposed by Dorn, Apel, and Siegmund [6], Kaltenecker et al. [15], and Dincher [5]. The VIF can be used as a measure of multicollinearity between multiple feature variables. A VIF of 1 indicates no multicollinearity at all while a VIF of ∞ indicates perfect multicollinearity. When performing the VIF analysis, we iteratively add feature variables representing configuration options to our model and calculate the VIF. Whenever we encounter a VIF of ∞, we remove the most recently added feature variable since it can already be represented by other feature variables in the model. This way, we end up with a model that does not contain perfect multicollinearity. Similar to Kaltenecker et al. [15], we only consider perfect multicollinearity since we always expect a certain extent of multicollinearity when dealing with configuration options.

2.4.2 *White-Box Performance Analysis*

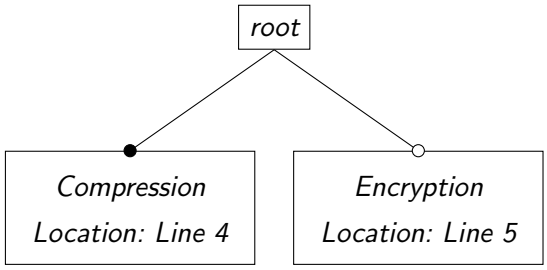
By definition, black-box performance analysis does not require any information about the internals of the software system that should be analyzed. Hence, it also does not provide a lot of details about the internals of a software system. It can only make statements about behavior that can be observed from the outside. To close this gap, white-box performance analysis can be used. A white-box approach does not only consider the system's observable behavior but also its internal information, such as the source code. For example, one way of performing a white-box analysis is to instrument the code of the software system with measurement instructions. This permits an exact analysis of which code was executed when and how long it took to execute the code. In combination with the feature model of the software, it is then possible to track the execution of configuration options by assigning configuration options to code locations. This enables us to perform similar analyses as with the black-box approach while additionally taking the information about the internals into account. For example, it is possible to compare changes in the performance of certain code regions instead of the software system as a whole.

As an example, recall our compression tool example and only consider the *Compression* and *Encryption* configuration options. For simplicity, assume that each variable declaration takes place on a separate line such that line information is sufficient to uniquely identify variables. If we annotate the feature model with code locations, a simplified version of the source code and the feature model could abstractly look like Figure 2.2.

We assign the Boolean variable `compression` to our *Compression* configuration option and `encryption` to our *Encryption* configuration option. By analyzing the control flow that depends on the selected variables, the analysis can deduct that the lines six and seven are related to the *Compression* option and the lines eight and nine deal with the *Encryption* option. We can then instrument the code with measurement instructions. As a result, we receive a binary that tracks the execution of each configuration option. Consequently, we

```
1   def main():
2       file_path = Path(sys.argv[1])
3       data = read_from_file(file_path)
4       compression = bool(sys.argv[2])
5       encryption = bool(sys.argv[3])
6       if(compression):
7           data = compress(data)
8       if(encryption):
9           data = encrypt(data)
10      write_to_file(data,
11          file_path / ".out")
```

(a) Abstract source code



(b) Annotated feature model

Figure 2.2: A simplified and abstract source code and feature model example for a compression tool

can see which configuration option was active at which point in time. An exemplary result when enabling both configuration options is shown in Figure 2.3.

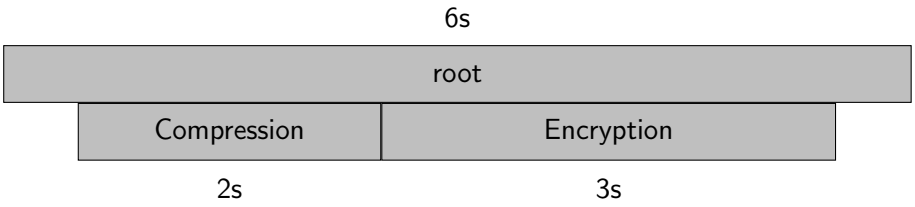


Figure 2.3: An exemplary result of a white-box analysis of a compression tool

We can see that the compression tool took six seconds in total to execute. The *Compression* configuration option was active for two seconds whereas the *Encryption* configuration option was active for three seconds. Additionally, we see that one second is not labeled with any configuration option. This is due to the code locations that were not assigned to any configuration options. In our example, this includes the parsing of the command line arguments, the reading from the file and the writing to the file. So, this time is by default attributed to the *root* feature.

A concrete implementation of such an approach has been introduced by Velez et al. [25] with ConfigCrusher. ConfigCrusher first utilizes a static data-flow analysis to detect configuration options that have no impact on the performance and to detect which configuration options do not interact with each other. This reduces the number of configurations to be measured. For the actual analysis, it instruments the code at control-flow statements, such as `if-then-else` or `while`, that relate to configuration options. With this instrumentation, ConfigCrusher divides the code into regions. ConfigCrusher then measures the performance of the regions by executing the selected configurations with the instrumented code.

In this thesis, we use the VaRA-Tool-Suite[2] and the underlying VaRA framework for our white-box measurements. VaRA is a white-box analysis framework that enables feature-

2 VaRA-Tool-Suite. Available online at https://github.com/se-sic/VaRA-Tool-Suite; visited on November 27th, 2022.

targeted analyses. On a technical level, VaRA divides the code into regions similar to ConfigCrusher. Then, it assigns configuration options to the code regions. When executing the software, VaRA records timestamps t for the configuration option every time a code region r is entered or exited. By calculating the difference $\Delta_t := t_r^{exit} - t_r^{enter}$ of the exit timestamp and the enter timestamp of one execution of a code region, we receive the execution time of the respective execution of the code region. Executions of code regions can also be nested inside each other. We consider the execution of a region r to be nested inside the execution of another region r' if $t_{r'}^{enter} \leq t_r^{enter} < t_r^{exit} \leq t_{r'}^{exit}$. Figure 2.4 depicts a visualization of a nesting where y is nested inside x. Note that we do not consider the nesting inside *root* since it is not a real configuration option, but a default fallback and, thus, every execution is nested inside the execution of *root*.

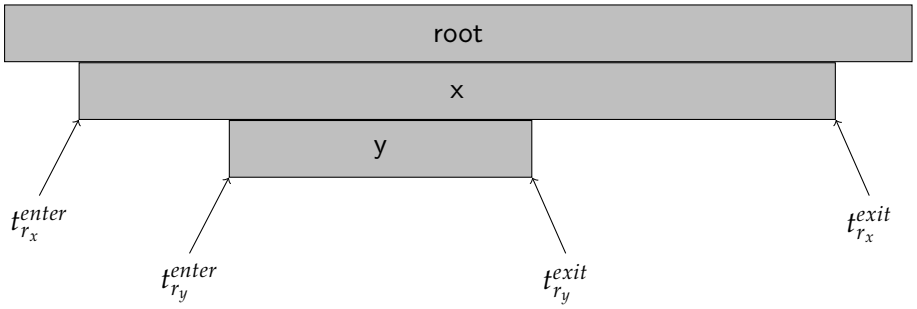


Figure 2.4: An example for nested executions of regions

If the execution of a region r is not nested inside any other region except for *root*, we attribute the execution time of the region to the configuration option that it is assigned to. If the execution of a region r_y that belongs to a configuration option y is nested inside the execution of other regions $r_{x_1}, \ldots, r_{x_i}$ that are assigned to different configuration options $x_1, \ldots, x_i$, we attribute the execution time of r_y to the interaction between $r_{x_1}, \ldots, r_{x_i}$ and r_y, i. e., to the term $x_1 \cdot \ldots \cdot x_i \cdot y$.

Considering this, it is then possible to determine the white-box performance $p_w^{\square,r}(t)$ of a term t by summing up all execution times attributed to this term as described above.

While a white-box approach is able to close some gaps of a black-box approach, it comes with its own disadvantages. Performing a white-box performance analysis is much more costly per configuration to be measured in comparison to a black-box approach. First, the execution of the program under investigation is slowed down due to the measurement instructions that are used to trace the execution. Thus, it is often infeasible to measure a large number of configurations with a white-box approach. Furthermore, a lot of manual effort is required to assign features to code locations. This can become even more complex if code locations change between releases.

3 RELATED WORK

This chapter presents publications that are related to the topic of this thesis. We provide a general overview of the related studies and describe their relevance to this thesis. Additionally, we outline the differences between each publication and this thesis.

Since the majority of related work focuses either on black-box performance analysis or on white-box performance analysis, this chapter is subdivided into two sections. First, we present literature that relates to the black-box part of this thesis. Then, we outline related work that is related to the white-box aspects of our work.

3.1 BLACK-BOX PERFORMANCE ANALYSIS

Most publications either exclusively deal with one dimension or at least focus on one of the dimensions. In the following sections, we first give an overview of related work that investigates the influence of configurations on the performance of software systems. Afterwards, we present literature that focuses on the evolution of software and its performance over time. In the third section, we introduce a related thesis that examines the influence of workloads on the performance of a software system.

3.1.1 *Configurations*

The relation between the performance of a software system and its configurations plays a key role in this thesis. There exists a multitude of publications [6, 10, 11, 24, 26] about the performance of configurable systems. In the following two subsections, we present a selection of these publications that are of particular relevance to this thesis.

Performance-Influence Models for Highly Configurable Systems

One of the most fundamental papers that this thesis is based on has been published by Siegmund et al. [24]. As already briefly mentioned in Chapter 2, Siegmund et al. [24] introduce the notion of performance-influence models to model the influence of configuration options and configuration option interactions on the performance of a software system. In contrast to, for example, Ha and Zhang [10] who use neural networks to create similar models, Siegmund et al. [24] rely on linear regression. In particular, they propose and implement a multiple linear regression approach with feature forward selection to incrementally create performance-influence models from measurement data. Due to this, the models can be read and interpreted by humans. This can be a huge advantage when analyzing the performance of software systems because the linear models make it easy to directly see from the model which configuration options or interactions slow down or speed up the execution. In addition, the approach enables an easy comparison of performance-influence models. Their approach supports binary as well as numeric configuration options. Moreover, they propose

multiple sampling approaches to select a meaningful subset of configurations from the configuration space $\mathcal{C}_S$.

In their evaluation, Siegmund et al. [24] investigate the correctness and accuracy of their models and the feasibility of their approach in practice. Their findings show that the approach yields promising results for the investigated systems.

However, in contrast to this thesis, Siegmund et al. [24] only consider each performance-influence model individually and do not compare performance-influence models to each other. Nevertheless, the performance-influence models that they introduce play a key role in the black-box performance analysis approach that we use to identify configurations, releases, and workloads that are worth investigating further with a white-box performance analysis.

An Empirical Study on Performance Bugs for Highly Configurable Software Systems

While Siegmund et al. [24] first introduced performance-influence models, they are not the only researchers studying the influence of configurations on the performance of software systems. Han and Yu [11] conduct a study on performance bugs for highly configurable software systems. In particular, they examine the relations between the configuration of a system and the influence of performance bugs.

In their evaluation, Han and Yu [11] claim that in their case studies, the majority of the performance bugs they examined are configuration-related. This emphasizes the necessity to consider configurations when analyzing the performance of a software system. They also state that all performance bugs in their case studies exclusively appeared with valid configurations, probably due to the early detection of invalid configurations by the software systems under investigation. This hints that restricting performance analysis to valid configurations is a reasonable limitation. Another interesting claim they make is that the majority of configuration-dependent performance bugs is caused by a small subset of configuration options. This result supports the plausibility of our approach where we select few configuration options to investigate with a white-box performance analysis based on the results of a black-box performance analysis. Finally, they also state that it is more complex to fix configuration-dependent performance bugs than to fix configuration-independent performance bugs. This indicates that the additional insights provided by a white-box performance analysis compared to a pure black-box performance analysis approach are actually beneficial in the development process.

Generally, their key findings support the plausibility of our approach. However, Han and Yu [11] do not study the effects of configurations along the dimensions of releases in workloads. Furthermore, a possible limitation to the generalizability of their findings is the limited number of case studies they investigated, making it unclear how their findings can be transferred to other software systems.

3.1.2 *Software Evolution*

The performance of software systems frequently changes over time. Few publications exist that systematically evaluate the evolution of the performance of software systems in the

context of configurable software systems. In the following subsections, we provide an overview of the most relevant papers in this area that this thesis is based on.

An Exploratory Study of Performance Regression Introducing Code Changes

Chen and Shang [4] investigate the prevalence of performance regressions between different revisions of software systems. Specifically, they conduct performance measurements for two case studies with ten and five releases, respectively, and compare them across revisions. Afterwards, they also manually analyze issue reports and change logs to identify the cause of the performance regression.

In their evaluation, Chen and Shang [4] state that a large number of revisions introduces performance regressions. According to their results, most of the performance regressions occur as the result of attempts to fix other bugs. Interestingly, performance regressions were sometimes introduced when trying to improve the performance of other functionalities. Both of these facts hint that performance regressions are easy to miss and often happen without the knowledge of the developer. This reinforces the need for sophisticated analysis tools that can help in spotting performance regressions and in identifying their root cause. Our work aims at establishing a first pointer for such a toolchain. Additionally, in contrast to our work, Chen and Shang [4] do not consider the impact of different configurations on the performance of the software system.

Performance Evolution of Configurable Software Systems: An Empirical Study

Kaltenecker et al. [15] conduct an empirical study that investigates the performance of software systems across multiple releases while also considering different configurations. They compare the influence of configurations of a whole and of configuration options and configuration option interactions on the performance of a software system between different releases. For this, they conduct experiments on 12 configurable software systems. For each software system, they investigate multiple releases of the software system. In total, they analyze 190 different releases.

In their results, Kaltenecker et al. [15] claim that almost every release they examine introduces a performance change in some configuration in comparison to the previous release. Mostly, only few configurations are affected by performance changes. In the majority of cases, they are able to track down performance changes to individual configuration options or configuration option interactions. Additionally, they analyze the metadata, such as commit messages and change logs, of the releases that exhibit performance changes. Using their analysis, they are able to confirm many of the changes related to configuration options or interactions.

Generally, the publication of Kaltenecker et al. [15] shows that performance changes between releases appear frequently and often depend on the configuration or even configuration options and interactions. This provides a strong indicator that the influence of configurations and configuration options is worth investigating when analyzing the performance of software systems. However, Kaltenecker et al. [15] do not consider workloads in their research, so it remains unclear which role workloads play in the performance changes across releases. This is one of the gaps that we try to close with this thesis.

3.1.3 *Workloads*

The workload used to execute a software system can heavily influence its performance. However, the influence might be of different strength depending on the configuration. In the following subsection, we present related work that investigates the impact of workload on the performance in the context of configurable software systems.

The Impact of Workloads on Performance of Configurable Software Systems

Dincher [5] examines the performance of configurable software systems if executed with different workloads. He studies the changes of the performance of configurations and of the influence of configuration options and interactions on the performance when using different workloads. For this, Dincher [5] investigates two case studies with ten and seven workloads, respectively.

He reports that when comparing workloads, the majority of configurations exhibits a performance change. Additionally, his results also show that the influence of some configuration options and interactions is heavily dependent on the workload. In many cases, also the ranking of configuration options and interactions changed significantly between workloads when ordering the options and interactions by their influence.

While Dincher [5] only investigated two case studies, his results still hint at the importance of workloads when performing performance measurements. The workload can substantially impact the performance of configurations and the influence of configuration options and interactions. Thus, it seems reasonable to consider the dimension of workloads for our investigations. In contrast to this thesis, Dincher [5] does not deal with releases, which differentiates his work from ours.

3.2 WHITE-BOX PERFORMANCE ANALYSIS

In the context of configurable systems, few publications deal with white-box performance analysis. In the following sections, we present two papers closely related to the topic of this thesis and describe how they relate to our work.

Performance Evolution Blueprint: Understanding the Impact of Software Evolution on Performance

Alcocer et al. [1] propose a white-box-based approach to measure and visualize the difference in performance across different dimensions. They introduce a code execution profiler that can handle the dimensions of workloads and revisions. Their tool reports performance regressions between two revisions or highlights the performance difference between two workloads. In their publication, Alcocer et al. [1] apply their approach to one case study and illustrate how it helped them tracking down performance regressions. However, a thorough evaluation of other case studies is missing, making it unclear whether their results can be transferred to other software systems.

Additionally, their approach behaves similarly to well-known code profilers. While it can collect information about different pieces of code, it is not aware of the concept of

configurations or configuration options. Hence, the approach introduced by Alcocer et al. [1] is not suited to automatically track down performance regressions to configuration options or to predict the impact of configuration options on the performance. This is problematic since, especially for large software systems, it is tedious and non-trivial to keep track of which code belongs to which configuration options. In the approach that we evaluate in this thesis, we close this gap by associating code locations with configuration options.

ConfigCrusher: Towards White-Box Performance Analysis for Configurable Systems

As already briefly mentioned in Chapter 2, Velez et al. [25] propose and evaluate a white-box performance analysis approach called ConfigCrusher. In contrast to most other publications, ConfigCrusher is targeted towards configurable systems. In particular, ConfigCrusher supports associating code regions with configuration options. This allows a detailed analysis of the impact of configuration options on the performance of a software system. From their gathered information, they build performance-influence models similar to the models that Siegmund et al. [24] use in their black-box approach.

Velez et al. [25] conduct experiments on ten case studies to investigate the effectiveness and practicality of their approach. Generally, their evaluation shows that the approach achieves comparable or even better results in terms of accuracy compared to a black-box approach. However, they also find that their white-box approach significantly decreases the performance of a software system due to the instrumentation of the code. Although they are able to reduce the overhead by optimizing the instructions, this still emphasizes one of the biggest disadvantages of a white-box approach in comparison to a black-box approach.

The results of Velez et al. [25] hint that an integration of black- and white-box performance analysis as done in this thesis could yield promising results. Velez et al. [25] do, however, not consider the dimensions of releases and workloads, leaving it unclear whether performance regressions can be reliably detected by such an approach.

4 METHODOLOGY

In this chapter, we introduce the methodology for our evaluation. First, we present the research questions that we investigate in our work. Next, we describe our measurement setup for the experiments that we conduct to evaluate the research questions. Afterwards, we provide an overview of our case studies. Finally, we lay out how we evaluate the research questions based on the data we obtained from our experiments.

4.1 RESEARCH QUESTIONS

We integrate white-box with black-box performance analysis to take advantage of both approaches. For example, both approaches individually can yield false positives or false negatives. By integrating both approaches, we identify flaws in both approaches by comparing and verifying their results. We choose our research questions in a way such that they give a strong indicator of whether the integrated approach produces promising results or not. Furthermore, we assess whether our approach indeed provides benefits over a pure black-box performance analysis approach. We investigate our integrated approach along the dimensions of configurations, releases, and workloads. Hence, we also incorporate these dimensions into our research questions. The dimension of configurations can be explored on multiple levels. First, there is the configuration level which means comparing configurations as a whole across other dimensions. Secondly, the option level can be investigated. In the option level, single configuration options are compared across dimensions. Since our integrated approach targets at providing detailed insights into the influence and changes in the influence of configuration options, we focus on the option level in our research questions.

Performance Changes Between Consecutive Releases for Different Workloads

The first research question that we investigate is targeted towards the black-box part of our approach and consists of two sub-questions. The initial step of our integrated approach is to perform a black-box performance analysis on the software system under investigation to identify the configuration options and configuration option interactions whose influence on the performance changes over time. First, we are interested in the number of configuration options and interactions whose influence changes between consecutive releases. Consequently, we formulate the first sub-question that we evaluate as follows.

> **RQ1.1:** How frequent are changes of the performance influence of individual configuration options and interactions among them?

As outlined in Chapter 2, the performance influence of configuration options and option interactions might be dependent on the workload. Hence, we evaluate the research question for multiple workloads.

We are not only interested in the number of configuration options and configuration option interactions whose influence changes, but also in how much they change. This allows us to identify the most severe changes. Thus, we formulate the second sub-question similar to the first one, but focus on the strength instead of the frequency of changes.

> **RQ1.2:** How strong are changes of the performance influence of individual configuration options and interactions among them?

Since the strength of changes is also dependent on the workload, we evaluate the research question for multiple workloads, analogously to RQ1.1.

Fraction of Performance Changes Confirmable by White-Box Approach

In the second step of our integrated approach, we take the configuration options we identified by the black-box approach as potentially interesting to investigate and perform a white-box performance analysis with them to gain more insights on the performance changes. Before we can take a closer look at the additional insights that the white-box performance analysis can deliver, we first need to investigate whether the performance changes we identified by the black-box approach can be confirmed by the white-box performance analysis. This means that we need to examine whether the same performance changes are detected by the white-box approach, leading us to the following formulation of the second research question.

> **RQ2:** What fraction of performance changes identified by our black-box approach can be confirmed by our white-box approach?

Identification of Configuration-Dependent Code Responsible for Performance Changes

Since we aim at gaining additional insights by integrating the white-box performance analysis with the black-box performance analysis, we also need to evaluate whether we actually achieve this goal. The additional insights that the white-box performance analysis can deliver are information about the code locations that are responsible for a performance change and that depend on the configuration that was used to execute the system. Hence, we need to identify the fraction of cases in which the white-box performance analysis is able to deliver this information. We pose the following research question.

RQ3: In what fraction of cases is our white-box approach able to identify the configuration-dependent code responsible for a performance change identified by our black-box approach?

4.2 EXPERIMENT DESIGN

This section provides an overview of the general experiment design that we use for all case studies. In particular, we describe the general measurement setup that we use.

We perform our measurements on machines that are organized in a cluster. For our experiments, we use two different types of machines. To guarantee comparable results, all measurements for a case study are performed on machines of the same type. All machines of the same type have identical hardware.

There are 14 machines of the first type. Each of these machines is equipped with an *Intel Core i5-4590*. The processor has four cores and four threads. The base frequency is 3.3 GHz and turbo boost is disabled. Additionally, the machines each have 16 GB of RAM. As an operating system, Debian 11 is used. In the remaining part of the thesis, we call the machines of this type the *i5-machines*.

The machines of the second type each run with two *Intel Xeon E5-2630 v4* CPUs with ten cores, 20 threads, and a base frequency of 2.2 GHz. Turbo boost is enabled and the maximum turbo boost frequency is 3.1 GHz. We call these machines the *Xeon-machines*. To minimize measurement noise due to Non-Uniform Memory Access (NUMA), we ensure that our case studies are executed only on one of both CPUs and that they exclusively use the main memory associated with that CPU. Each *Xeon-machine* has 256 GB of RAM. In total, there are 20 *Xeon-machines*. The operating system of the *Xeon-machines* is Debian 11.

In the cluster, we use a workload manager to distribute the measurements to the machines. To minimize side effects and, thus, measurement noise, we configure the workload manager in a way that ensures that the measurements are executed exclusively on each machine. This guarantees that no other processes are interfering with our measurements. Although all machines have access to a shared network drive, we copy the relevant binaries and workloads to the local drive of the machine before performing measurements. This eliminates fluctuations in the network connection as a possible cause of measurement noise.

For our black-box measurements, we repeat each measurement five times. If we observe a relative standard deviation larger than 5% between the five measurements, we repeat the measurements until the relative standard deviation is below 5%. To calculate the relative standard deviation, we divide the absolute standard deviation by the mean performance across all measurements. Through these repetitions, we make sure to reliably detect and eliminate high measurement noise. We perform the same procedure for our white-box measurements. The white-box measurements for a case study are executed on the same type of machines as the black-box measurements for the respective case study.

4.3 CASE STUDIES

To evaluate our research questions, we investigate one case study specifically crafted for this thesis and four real-world case studies. In the following sections, we briefly introduce each of the case studies.

4.3.1 *CompEnc*

To demonstrate the theoretical capabilities of our approach without being restricted by the natural complexity of real-world systems, we manually craft a case study that is loosely based on our compression tool example introduced in Chapter 2. For the remainder of this thesis, we call this exemplary tool CompEnc. CompEnc is written in C++.

CompEnc has two optional binary options, *Compression* and *Encryption*. Additionally, it has a mandatory binary option *Iteration* that has a numeric option *Iterations_val* as a child. This option specifies the compression iterations that CompEnc performs. Figure 4.1 depicts the resulting feature model of CompEnc.

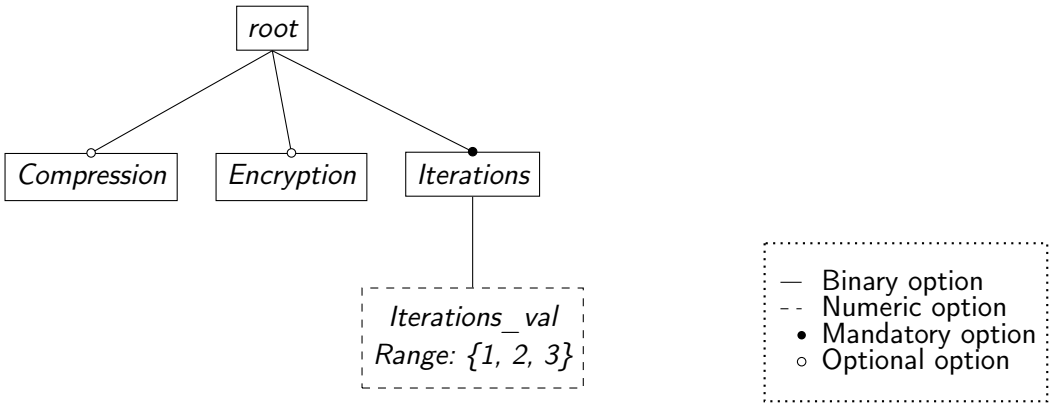


Figure 4.1: The feature model of CompEnc

In the code, we emulate computations by simply sleeping for a specified amount of time. We have different parts of the code to emulate the different configuration options and interactions between these configuration options. In particular, we have statements that emulate compressing the data and statements emulating the encryption of data. Additionally, we have statements that emulate the compression iterations by increasing the compression time and by adding a pre-processing stage depending on the iterations. Listing 4.1 contains an abstract representation of the core part of CompEnc in its initial release (v1).

We create four different releases where we introduce performance regressions by simply adjusting the sleep time. In the second release (v2), we introduce a regression related to the *Encryption* option by changing the sleep time from 5 to 10. The third release (v3) contains a performance regression related to the *Iterations_val* option, introduced by changing the sleep time in the pre-processing phase from *Iterations* to $2 \cdot Iterations$. In the last release (v4), we add a performance regression related to the interaction between *Compression* and *Iterations_val*. We increase the sleep time in the additional compression iterations from *Iterations* to $2 \cdot Iterations$.

Listing 4.1: Abstract representation of the core part of the source code of CompEnc (v1)

```
1   void sendPackage(PackageData Data) {
2     if (Iterations) {
3       // Pre-process
4       sleep_for_secs(Iterations);
5     }
6
7     if (UseCompression) {
8       // Compress
9       Data.Data = Data.Data.substr(0, Data.Data.size() / 2);
10      sleep_for_secs(3);
11
12      if (Iterations) {
13         // Additional compression iterations
14        sleep_for_secs(Iterations);
15      }
16    }
17
18    if (UseEncryption) {
19      // Encryption
20      reverse(Data.Data.begin(), Data.Data.end());
21      sleep_for_secs(5);
22    }
23
24    // Sending
25    sleep_for_secs(2);
26    send(&Data);
27
28    puts(Data.Data.c_str());
29  }
```

Since we knowingly introduce the performance changes, we do not consider different workloads for CompEnc because the main reason to look at different workloads is to identify performance changes that might only be prevalent in some workloads.

For our measurements, we select all configuration options and generate all possible valid configurations, resulting in 12 configurations in total. We measure each configuration in each release. We perform the measurements on the *i5-machines*.

4.3.2 *PicoSAT*

PicoSAT[1] is a SAT solver written in C that provides proof and trace capabilities. PicoSAT achieved good results in some SAT competitions[2] and is relatively small with only a few

1 PicoSAT. Available online at http://fmv.jku.at/picosat/; visited on November 23rd, 2022.
2 The International SAT Competition Web Page. Available online at http://www.satcompetition.org/; visited on November 23rd, 2022.

thousand lines of code. For these two reasons, we selected it as our first real-world case study.

For our measurements, we use two releases. We measure version 951 and version 965. Version 951 was released in 2012 and version 965 was released in 2016. We select version 965 since it is the most recent available release. Version 951 is the oldest available release that provides a similar set of configuration options in comparison to version 965.

As workloads, we select four different SAT problems from SAT competitions from different domains. This selection is a subset of workloads that Dincher [5] used as workloads for another SAT solver, Clasp, and that seemed to have different impact on performance. We decide to use a subset of the workloads selected by Dincher [5] since the other workloads took more than three hours to complete for many configurations with PicoSAT, making it infeasible to perform measurements for these workloads for all configurations and all releases. In particular, we use the following workloads.

abw-n-bcsstk07.mtx-w44.cnf
: This workload by Fazekas et al. [7] contains a SAT problem describing the feasibility of the antibandwidth problem which is a max-min optimization problem on graphs.

traffic_kkb_unknown.cnf
: traffic_kkb_unknown.cnf is a SAT problem that models a traffic situation to be solved [5].

unsat_h_instances_childsnack_p12.hddl_1.cnf
: This workload contains a Hierarchical Task Network (HTN) planning problem and has been provided by Froleyks [8].

unsat_h_instances_childsnack_p11.hddl_1.cnf
: UNSAT_H_instances_childsnack_p11.hddl_1.cnf describes another HTN planning problem by Froleyks [8].

Regarding the dimension of configurations, we select all configuration options that are present in both versions and that do not lead to instant termination of PicoSAT such as, for example, `--version` which simply prints the current version. Kolesnikov et al. [16] show that frequently, covering interactions of up to two configuration options is sufficient to model most of the variability in performance. Hence, we perform a pairwise sampling approach [24] to generate configurations covering all valid pairs of configuration options. This way, we hope to achieve a reasonable trade-off between the number of configurations considered and the measurement time. In total, we cover 16 configuration options resulting in 182 configurations.

We measure each configuration with both releases and all five workloads. Since PicoSAT requires a high amount of main memory in some configurations, we perform the measurements on the *Xeon-machines*.

4.3.3 *Clasp*

Clasp[3] is an answer set solver that supports normal and disjunctive logic. As such, it can be used as a SAT solver similar to picoSAT. Clasp is written in C++. It achieved good results in some SAT competitions, making it a viable candidate for performance analysis.

We measure two releases of Clasp. The first release we consider is release v3.3.4 which was published in 2018. The second release is release v3.3.9 which was published in 2022. Similar to the PicoSAT case study, we select the most recent release and the oldest-possible release. All releases prior to v3.3.4 do not compile without errors with recent versions of well-known compilers which is why we select v3.3.4.

Similar to PicoSAT, we use a subset of the SAT problems that Dincher [5] used to investigate Clasp. We select seven workloads from the set of workloads that Dincher [5] investigated. We do not use the remaining three workloads since these lead to infeasible run-times in some configurations and releases. In particular, we use the same workloads as for PicoSAT and additionally the following three workloads.

sat_h_instances_childsnack_p08.hddl_2.cnf
: This workload contains a satisfiable HTN planning problem provided by Froleyks [8].

unsat_p_opt_snake_p06.pddl_30.cnf
: In this workload, Froleyks [8] model an unsatisfiable classical planning problem.

sat_p_opt_snake_p10.pddl_27.cnf
: This workload is a classical planning problem that is satisfiable and, again, provided by Froleyks [8].

Clasp offers a huge number of configuration options. To be able to perform measurements in a feasible amount of time, we pre-select a subset of configuration options that we expect to influence the performance. We base our selection on the results of Dincher [5]. With the same reasoning as before, we perform pairwise sampling to generate configurations from the set of configuration options. Totally, we consider 33 configuration options resulting in 91 configurations.

We measure each configuration with both releases and all seven workloads on the *Xeon-machines*.

4.3.4 *XZ*

XZ is a general-purpose file compression and decompression tool that is part of the XZ Utils[4]. The XZ Utils are written in C. They are the successor to LZMA Utils[5] and still support the legacy LZMA format.

We consider three releases of XZ. The first release is v5.2.3 which has been released in 2016. We choose it because it is the oldest version of XZ of the current v5.2 release branch that does not exhibit any memory leaks on our measurement systems. Additionally, we

3 Clasp - A conflict-driven nogood learning answer set solver. Available online at https://potassco.org/clasp/; visited on November 23rd, 2022.
4 XZ Utils. Available online at https://tukaani.org/xz/; visited on November 23rd, 2022.
5 LZMA Utils. Available online at https://tukaani.org/lzma/; visited on November 26th, 2022.

measure v5.2.6 which was released in 2022 and was the newest version of the current release branch at the time of our measurements. The third release we evaluate is v5.3.3alpha which was the most recent version of the current development branch at the time of our measurements.

Regarding the dimension of workloads, we measure the following two workloads that we expected to differ in many aspects.

ENWIK9
: In ENWIK9[6], the first ten billion bytes of the English Wikipedia pages in the state of March 3rd, 2006 are dumped into a text file. The workload has a size of 1 GB and prior experiments by us showed that this workload is very well compressible, shrinking up to 80% in size.

DAVIS 2016
: The DAVIS 2016[7] workload is a set of annotated images. The images are in JPEG format and are available in resolutions of 480p and 1080p. The workload has a size of 1.8 GB. Prior experiments indicated a bad compressibility, providing only a size loss of a few megabytes. Originally, the DAVIS 2016 dataset has been created by Perazzi et al. [20] to be used for training machine learning systems to identify objects on images.

XZ offers a wide range of configuration options. We pre-select configuration options that we expect to influence the performance based on their description in the manual of XZ. With a pairwise sampling approach, we generate configurations from the configuration options. In total, we measure 79 configuration options resulting in 713 configurations per release and workload. We split up our measurements into two parts. First, we measure the performance of XZ when compressing the workload. Then, we measure the performance when decompressing the just compressed file again. All measurements are performed on the *Xeon-machines*.

4.3.5 *Fast Downward*

FAST DOWNWARD[8] is a classical planning system that is written in C++. Since the measurement data for FAST DOWNWARD has already been present before the start of this thesis, the measurements were not time-critical. Thus, FAST DOWNWARD is the case study that covers the widest range of releases and workloads.

We measure nine releases from 2016 to 2020. Until June 2019, the FAST DOWNWARD development team did not label any revisions as releases. Hence, we contacted the FAST DOWNWARD development team and they selected suitable revisions that could have been declared as releases in a six-month cycle. From June 2019 on, we use the official releases.

For each release, we measure 19 different workloads. We base our workload selection on the official benchmarks[9] provided by the FAST DOWNWARD development team. For brevity

6 enwik9. Available online at http://mattmahoney.net/dc/textdata; visited on November 26th, 2022.
7 DAVIS: Densely Annotated VIdeo Segmentation. Available online at https://davischallenge.org/davis2016/browse.html; visited on November 26th, 2022.
8 Fast Downward. Available online at https://www.fast-downward.org/; visited on November 23rd, 2022.
9 Fast Downward Benchmarks. Available online at https://github.com/aibasel/downward-benchmarks; visited on November 26th, 2022.

reasons, we do not describe every workload in detail, but we group them into different types of workloads that we outline in the following.

DATA_NETWORK_P05
: This workload belongs to the data network planning domain. The goal is to find an optimal plan for the communication between multiple servers which is constrained by server properties such as, e. g., main memory.

SCANALYZER_P11
: The workload describes an instance of a planning problem that deals with greenhouse logistic management.

SOKOBAN_P13, SOKOBAN_P17, SOKOBAN_OPT08_P04 AND SOKOBAN_OPT08_P08
: The workloads describe instances of Sokoban games. Sokoban is a puzzle game.

TRANSPORT_P04, TRANSPORT_P08 AND TRANSPORT_OPT08_P04
: These workloads deal with satisfiability and optimization problems in the transport domain, i. e., how to (optimally) transport goods from one place to the other with certain constraints.

TERMES_P17
: The TERMES_P17 workload models problems the Harvard TERMES robot has to solve. The TERMES robot[10] is a robot that can move around and build structures from blocks.

AGRICOLA_P02
: This workload models the board game Agricola which consists of a farm with multiple workers.

HIKING_PTESTING225, HIKING_PTESTING225 AND HIKING_PTESTING244
: These three workloads belong to the hiking domain which deals with planning the preparation and execution of a hiking trip with several constraints.

ELEVATORS_P22
: The ELEVATORS_P22 workload models the problem of controlling multiple elevators.

GED_D28 AND GED_D43
: These workloads belong to the domain of Genome Edit Distances (GED). In the GED domain, the planning tool has to find operations that transform one genome into another and are minimal with regard to cost.

VISITALL_OPT11_P05 AND VISITALL_OPT14_P056
: In these two workloads, the goal is to find an optimal plan to visit all cells in a grid under some given constraints.

FAST DOWNWARD offers a large number of configuration options. In cooperation with the development team, we pre-select configuration options that are of particular relevance or are expected to have a significant impact on the performance. From these configuration options, we generate configurations. In total, we measure 60 configuration options leading

10 TERMES. Available online at https://wyss.harvard.edu/media-post/termes/; visited on November 26th, 2022

to 412 configurations per release and workload. The measurements are performed on the *Xeon-machines*. In contrast to the other case studies, the measurements for FAST DOWNWARD were performed without specific NUMA control. This is due to the fact that the measurements for FAST DOWNWARD were performed before the beginning of this thesis and would have taken too much time to repeat. However, since we made sure that the standard deviation of the measurements is below 5%, we still expect the data to be valid.

4.4 OPERATIONALIZATION

In this section, we outline the operationalization of each research question. We describe which metrics we use to evaluate the research questions and how we apply them.

RQ1: Performance Changes Between Consecutive Releases for Different Workloads

To answer RQ1, we measure all configurations of the respective case study as described before with our black-box approach for each release and workload that we consider. Then, we learn a performance-influence model based on the measurements for all configurations. We learn one performance-influence model per release and workload that we consider. For the software system S with releases $\mathcal{R}_S$ and workloads $\mathcal{W}_S$, we thus receive $|\mathcal{R}_S| \cdot |\mathcal{W}_S|$ performance-influence models. As we have seen before, each performance-influence model contains configuration options and interactions as terms. We denote the set of all terms in a performance-influence model for a software system S with workload w and release r by $\mathcal{T}_{S,w}^{\blacksquare,r}$ where the black square $\blacksquare$ indicates that we are referring to the black-box data. To guarantee the comparability of performance-influence models, we have to make sure that all models contain the same configuration options and interactions. Hence, we first take the iteratively learned performance-influence models for each release and workload and collect all configuration options and interactions from them. Formally, we build the set $\mathcal{T}_S^{\blacksquare} := \bigcup_{r \in \mathcal{R}_S,\ w \in \mathcal{W}_S} \mathcal{T}_{S,w}^{\blacksquare,r}$. Then, we apply a VIF analysis to detect and prevent multicollinearity. Based on the results of this analysis, we remove all options and interactions causing perfect multicollinearity. This way, we receive a new set $\mathcal{T}_S^{\blacksquare\prime} \subseteq \mathcal{T}_S^{\blacksquare}$. Finally, we re-learn performance-influence models for each release and workload, but force the terms in the model to be the ones contained in $\mathcal{T}_S^{\blacksquare\prime}$. By doing this, all performance-influence models for S include the same terms and only the coefficients of the terms differ.

For a release r and a workload w, we denote the performance-influence model by $\Pi_{S,w}^r$. Similarly, we denote the coefficients for a term t in $\Pi_{S,w}^r$ by $\beta_{S,w}^r(t)$. In the following, we usually omit S from all indices for better readability if the software system S for which the model applies is clear from the context or irrelevant. In this case, we denote performance-influence models simply by Π_w^r.

In both sub-questions, we use the performance-influence models that we have learned before. We compare the coefficients of each term in the performance-influence models of consecutive releases, separately for each workload. The difference between the two coefficients corresponds to a performance change. To prevent mistaking measurement noise for performance changes, we first need to determine which performance changes we consider. In both sub-questions, we only consider performance changes whose difference

in the coefficients of the performance-influence model exceeds twice the maximum of the two mean standard deviations of all configurations of the respective workload for both releases. Additionally, we exclude tiny performance changes by only considering performance changes larger than 100ms. Formally, we express this as

$$|\beta_w^{r_i}(t) - \beta_w^{r_{i+1}}(t)| > |2 \cdot max(\overline{sd_w^{r_i}}, \overline{sd_w^{r_{i+1}}})| \wedge |\beta_w^{r_i}(t) - \beta_w^{r_{i+1}}(t)| > 0.1s$$

where $\overline{sd_w}^r$ denotes the mean standard deviation of all measurements for workload w and release r. We define the set of all terms that are involved in a performance change for workload w and releases r_i and r_{i+1} by

$$\mathcal{T}_{w,rel}^{\blacksquare,r_i,r_{i+1}} := \{t \mid t \in \mathcal{T}^{\blacksquare\prime} \wedge |\beta_w^{r_i}(t) - \beta_w^{r_{i+1}}(t)| > |2 \cdot max(\overline{sd_w^{r_i}}, \overline{sd_w^{r_{i+1}}})| \wedge |\beta_w^{r_i}(t) - \beta_w^{r_{i+1}}(t)| > 0.1s\}.$$

RQ1.1: Frequency of Performance Changes Between Consecutive Releases for Different Workloads

In RQ1.1, we are interested in the frequency of performance changes between consecutive releases for different workloads. Hence, we set the number of configuration options and interactions that were involved in a performance change in relation to the total number of configuration options. For each workload and pair of consecutive releases, we determine the fraction of terms that were involved in a performance change by computing

$$f_w^{r_i,r_{i+1}} := \frac{|\mathcal{T}_{w,rel}^{\blacksquare,r_i,r_{i+1}}|}{|\mathcal{T}^{\blacksquare\prime}|}.$$

Additionally, we measure the mean fraction of terms involved in a performance change across all workloads by computing

$$F^{r_i,r_{i+1}} := \frac{\sum_{w \in \mathcal{W}} f_w^{r_i,r_{i+1}}}{|\mathcal{W}|}.$$

since we have $|\mathcal{W}|$ workloads. Similarly, we aggregate the data across all releases and workloads by computing

$$F := \frac{\sum_{i=1}^{|\mathcal{R}|-1} \sum_{w \in \mathcal{W}} f_w^{r_i,r_{i+1}}}{(|\mathcal{R}| - 1) \cdot |\mathcal{W}|}.$$

Note that we always compare pairs of consecutive releases and since we have $|\mathcal{R}|$ releases, we have $|\mathcal{R}| - 1$ such pairs.

Based on the metrics defined above, we are able to conclude the frequency of performance changes influenced by configuration options and interactions for each case study which answers RQ1.1.

RQ1.2: Strength of Performance Changes Between Consecutive Releases for Different Workloads

In RQ1.2, we consider the strength of the performance changes between consecutive releases for different workloads. Thus, we restrict our evaluations to those configuration options and

interactions that were identified to contribute to performance changes. For a performance change of a term t with workload w and releases r_i and r_{i+1}, we define the strength of the performance change by the quotient of the difference in the coefficient and the mean of the mean performances of both releases for the respective workload. Formally, we express this as

$$\delta_w^{r_i,r_{i+1}}(t) := \frac{|\beta_w^{r_i}(t) - \beta_w^{r_{i+1}}(t)|}{\frac{1}{2} \cdot (\overline{p_w^{\blacksquare,r_i}} + \overline{p_w^{\blacksquare,r_{i+1}}})}$$

where $\overline{p_w^{\blacksquare,r}}$ denotes the mean performance for workload w and release r across all configurations based on the black-box measurements.

With this definition, we can determine the mean strength of the performance changes for a workload and pair of consecutive releases by

$$\Delta_w^{r_i,r_{i+1}} := \begin{cases} \dfrac{\sum_{t \in \mathcal{T}_{w,rel}^{\blacksquare,r_i,r_{i+1}}} \delta_w^{r_i,r_{i+1}}(t)}{|\mathcal{T}_{w,rel}^{\blacksquare,r_i,r_{i+1}}|}, & \text{if } |\mathcal{T}_{w,rel}^{\blacksquare,r_i,r_{i+1}}| > 0, \\ 0, & \text{otherwise.} \end{cases}$$

Similar to RQ1.1, we also determine the mean strength across all workloads by

$$\Delta^{r_i,r_{i+1}} := \frac{\sum_{w \in \mathcal{W}} \Delta_w^{r_i,r_{i+1}}}{|\mathcal{W}|}$$

and across all releases and workloads by

$$\Delta := \frac{\sum_{i=1}^{|\mathcal{R}|-1} \sum_{w \in \mathcal{W}} \Delta_w^{r_i,r_{i+1}}}{(|\mathcal{R}| - 1) \cdot |\mathcal{W}|}.$$

These metrics provide an indicator of the strength of performance changes of individual configuration options and interactions for each case study which results in an answer to RQ1.2.

RQ2: Fraction of Performance Changes Confirmable by White-Box Approach

In RQ2, we are interested in the relation between performance changes identified by the black-box approach and by the white-box approach. In particular, we want to know whether the white-box approach is able to detect the performance changes that the black-box approach identified. Therefore, the performance changes of the black-box approach identified in RQ1 serve as a basis for this research question. Based on the configuration options and interactions that were detected to contribute to performance changes, we generate minimal configurations in which these configuration options or interactions are active. The term minimal configuration refers to a valid configuration in which a minimal number of configuration options is enabled. Depending on the feature model of the software system, we either have one or multiple minimal configurations for a configuration option or interaction.

After generating the minimal configurations, we perform white-box measurements for these configurations with all releases and workloads on the same systems as the black-box measurements.

As briefly described in Chapter 2, the white-box measurements deliver information about the amount of time each configuration option and interaction contributed to the total run-time of a software system. Hence, if we see a performance change in the black-box measurements for a configuration option or interaction, we should, intuitively, be able to see that a similar change is attributed to the same configuration option or interaction in the white-box measurements. This research question aims at investigating to what extent this intuitive statement holds true.

To answer RQ2, we first determine the white-box performance $p_w^{\square,r}(t)$ for each term t as described in Chapter 2. Then, we perform a similar approach as in RQ1. We investigate the white-box measurement data for each workload and identify performance changes between consecutive releases for configuration options or interactions. As before, we make sure to filter out measurement noise by only considering performance changes larger than twice the mean standard deviation $\overline{sd_w^{\square,r}}$ across all minimal configurations that we generated for the respective configuration option or interaction. Furthermore, we again filter out tiny performance changes by only considering performance changes above 100ms. We formally express this similar to as in RQ1 as

$$|p_w^{\square,r_i}(t) - p_w^{\square,r_{i+1}}(t)| > |2 \cdot max(\overline{sd_w^{\square,r_i}}, \overline{sd_w^{\square,r_{i+1}}})| \wedge |p_w^{\square,r_i}(t) - p_w^{\square,r_{i+1}}(t)| > 0.1s.$$

Then, we create the set $\mathcal{T}_{w,rel}^{\square,r_i,r_{i+1}}$ that contains all terms that exhibit a relevant performance change for workload w and releases r_i and r_{i+1}. The white square $\square$ indicates that we refer to the white-box data.

For our evaluation, we now compute the fraction of black-box performance changes that were confirmed by the white-box approach. Formally, we determine this by the formula

$$c_w^{r_i,r_{i+1}} := \begin{cases} \dfrac{|\mathcal{T}_{w,rel}^{\blacksquare,r_i,r_{i+1}} \cap \mathcal{T}_{w,rel}^{\square,r_i,r_{i+1}}|}{|\mathcal{T}_{w,rel}^{\blacksquare,r_i,r_{i+1}}|}, & \text{if } |\mathcal{T}_{w,rel}^{\blacksquare,r_i,r_{i+1}}| > 0, \\ 1, & \text{otherwise.} \end{cases}$$

Based on this, we then again aggregate the results across all workloads

$$C^{r_i,r_{i+1}} := \frac{\sum_{w \in \mathcal{W}} c_w^{r_i,r_{i+1}}}{|\mathcal{W}|}$$

and across all workloads and release pairs

$$C := \frac{\sum_{i=1}^{|\mathcal{R}|-1} \sum_{w \in \mathcal{W}} c_w^{r_i,r_{i+1}}}{(|\mathcal{R}|-1) \cdot |\mathcal{W}|}.$$

Since VaRA uses a static data flow analysis, it can happen that the white-box measurements do not correctly track the execution of all configuration options, e. g., if the variables associated with a configuration option are stored in a dynamic object. Hence, for a term t

that exhibits a performance change between two releases r_1 and r_2 in the black-box data, but not in the white-box data, we differentiate whether the white-box data actually does not show any difference or if the white-box data is incomplete. We consider the white-box data for t to be incomplete if in at least one of r_1 and r_2, the white-box data reports an execution time of 0 for t. Based on the terms that the black-box analysis reports to exhibit a performance change, we calculate the fraction of terms that exhibit a measurement problem on the white-box data. Let $\mathcal{T}_{w,rel}^{\square,r_i,r_{i+1}}$ be the set of terms for which the white-box data is incomplete, formally expressed as $\mathcal{T}_{w,rel,incomplete}^{\square,r_i,r_{i+1}} := \{t \mid t \in \mathcal{T}_{w,rel}^{\blacksquare,r_i,r_{i+1}} \wedge (p_w^{\square,r_i}(t) = 0 \vee p_w^{\square,r_{i+1}}(t) = 0)\}$. Then, we compute

$$err_w^{r_i,r_{i+1}} := \begin{cases} \dfrac{|\mathcal{T}_{w,rel,incomplete}^{\square,r_i,r_{i+1}}|}{|\mathcal{T}_{w,rel}^{\blacksquare,r_i,r_{i+1}} \setminus \mathcal{T}_{w,rel}^{\square,r_i,r_{i+1}}|}, & \text{if } |\mathcal{T}_{w,rel}^{\blacksquare,r_i,r_{i+1}} \setminus \mathcal{T}_{w,rel}^{\square,r_i,r_{i+1}}| > 0, \\ 0, & \text{otherwise} \end{cases}$$

to determine the fraction of terms that are not correctly measured by the white-box analysis in relation to the terms whose performance changes have not been confirmed by the white-box analysis.

We then, again, aggregate the results across all workloads

$$Err^{r_i,r_{i+1}} := \frac{\sum_{w \in \mathcal{W}} err_w^{r_i,r_{i+1}}}{|\mathcal{W}|}$$

and across all workloads and release pairs

$$Err := \frac{\sum_{i=1}^{|\mathcal{R}|-1} \sum_{w \in \mathcal{W}} err_w^{r_i,r_{i+1}}}{(|\mathcal{R}|-1) \cdot |\mathcal{W}|}.$$

Additionally, we investigate whether the performance changes detected by the white-box analysis are of similar strength as the performance changes that the black-box analysis reports. While, at a first glance, it might seem intuitive to use our strength measure $\delta_w^{r_i,r_{i+1}}(t)$ that we introduced in RQ1.2, using the measure $\delta_w^{r_i,r_{i+1}}(t)$ on the white-box data would not be comparable to the black-box data. $\delta_w^{r_i,r_{i+1}}(t)$ puts the performance change in relation to the average performance across all configurations. Since we only measure the minimal configurations for a term t on the white-box data, we can only consider the average performance across the minimal configurations. Therefore, we define

$$\delta_{w,min}^{\blacksquare,r_i,r_{i+1}}(t) := \frac{|\beta_w^{r_i}(t) - \beta_w^{r_{i+1}}(t)|}{\frac{1}{2} \cdot (\overline{p_{w,min}^{\blacksquare,r_i}} + \overline{p_{w,min}^{\blacksquare,r_{i+1}}})}$$

where $\overline{p_{w,min}^{\blacksquare,r}}$ refers to the mean performance for workload w and release r across the minimal configurations for t based on the black-box measurements. Analogously, we define

$$\delta_{w,min}^{\square,r_i,r_{i+1}}(t) := \frac{|p_w^{\square,r_i}(t) - p_w^{\square,r_{i+1}}(t)|}{\frac{1}{2} \cdot (\overline{p_w^{\square,r_i}} + \overline{p_w^{\square,r_{i+1}}})}$$

where $\overline{p_w^{\square,r}}$ denotes the mean performance for workload w and release r across the minimal configurations for t determined by the white-box measurements. If

$$|\delta_{w,min}^{\blacksquare,r_i,r_{i+1}}(t) - \delta_{w,min}^{\square,r_i,r_{i+1}}(t)| \leq 0.05,$$

we declare the performance change as *similar*. Otherwise, we declare it as *different*. We group the performance changes into the categories and determine the fraction $s_w^{r_i,r_{i+1}}$ of *similar* performance changes. As before, we aggregate the results across workloads ($S^{r_i,r_{i+1}}$) and across releases and workloads (S).

This data indicates whether the white-box approach is able to find the same performance changes as the black-box approach and if they are of similar strength as in the black-box approach. This way, we answer RQ2.

RQ3: Identification of Configuration-Dependent Code Responsible for Performance Changes

In RQ3, we are interested in the fraction of cases in which our white-box approach is able to identify the configuration-dependent code responsible for a performance change identified by our black-box approach. For the performance changes identified by the black-box analysis (RQ1) that were not identified by the white-box analysis (RQ2), it is trivial that in these cases, the white-box approach is not able to identify the code that is responsible for a performance change. Hence, in the following, we restrict our investigation to the performance changes that the white-box analysis successfully identified.

As described before, the VaRA framework that we use generates a mapping of configuration options to code regions. Let $\mathcal{FR} = \{FR_1, \ldots, FR_i\}$ be the set of regions in the code that VaRA creates as described in Chapter 2. Then, we have a mapping $\mathcal{O} \to \mathcal{P}(\mathcal{FR})$ that maps configuration options to regions, where $\mathcal{P}(\mathcal{FR})$ refers to the power set of $\mathcal{FR}$. Additionally, VaRA creates a mapping $\mathcal{FR} \to \mathcal{L}$ where $\mathcal{L}$ abstractly describes the set of all code locations in S. Hence, we have a mapping $\mathcal{O} \to \mathcal{P}(\mathcal{L})$ by transitivity.

Using the timestamps that VaRA records during the execution of a software, we can extract the execution times of each region as done in RQ2. To investigate a performance change of a term t between two releases r_j and r_{j+1}, we could now, intuitively, look at all regions that belong to t that exhibit a performance change and compare the execution times of each region assigned to t in r_j and r_{j+1}. This way, we would be able to identify the code locations related to a performance change. However, it is not possible to reliably create an exact mapping between the regions $\mathcal{FR}_j$ in r_j and the regions $\mathcal{FR}_{j+1}$ in r_{j+1}. Thus, the comparison of execution times of regions between r_j and r_{j+1} is not trivial. This is due to the fact that the region IDs are depending on the software as a whole. Hence, changes anywhere in the software that, e. g., introduce a new region, can, in the worst case, change all region IDs.

To mitigate this problem, we assign additional, independent IDs to regions relative to the function they are placed in. For example, assume we have a function `foobar` that contains three regions with region IDs 1234, 1235, and 1236. Then, we assign them the function-relative ID $\texttt{foobar}_1$, $\texttt{foobar}_2$, and $\texttt{foobar}_3$ in the order of their appearance in the function. Afterwards, we compare $\texttt{foobar}_1$ in r_j and $\texttt{foobar}_1$ in r_{j+1}, $\texttt{foobar}_2$ in r_j and $\texttt{foobar}_2$ in r_{j+1}, and $\texttt{foobar}_3$ in r_j and $\texttt{foobar}_3$ in r_{j+1}. Note that the approach described above only works if in both releases, the number of option-related regions within `foobar` is the same. If the

number of regions within `foobar` changes, we manually investigate all the regions in `foobar`. In particular, we manually assign the same function-relative IDs to regions that are identical in both releases with regard to the code they are assigned to. For the new or removed regions, we assign additional function-relative IDs and attribute a performance of 0 to this region in the release in which the region does not exist.

Then, for each release r_i and r_{i+1} and workload w, we look at each term t in $\mathcal{T}_{w,rel}^{\blacksquare,r_i,r_{i+1}} \cap \mathcal{T}_{w,rel}^{\square,r_i,r_{i+1}}$ and collect the (function-relative) region IDs associated with t. Formally, we collect them in a set $\mathcal{FR}_w^{r_i,r_{i+1}}(t)$. For each region f in $\mathcal{FR}_w^{r_i,r_{i+1}}(t)$ and configuration c, we sum up all execution times of f to get the total performance $tp_{c,w}^r$ of f in r when executing c. Afterwards, we build the mean performance $\overline{tp_w^r}$ over all minimal configurations associated with t. We compare this mean performance for r_i and r_{i+1}. If the difference between the performance of a region in both releases exceeds twice the maximum of the mean standard deviation $\overline{sd_{tp,w}^r}$ of the region performance across all minimal configurations that we generated for t, we consider the performance change to be relevant. We formally express this as

$$|\overline{tp_w^{r_i}} - \overline{tp_w^{r_i}}| > |2 \cdot max(\overline{sd_{tp,w}^{r_i}}, \overline{sd_{tp,w}^{r_{i+1}}})|.$$

If we identify a relevant performance change for a region between two releases, we manually investigate the code location that is assigned to the region for changes. During our manual investigation, we compare the code for the affected region for both releases. If the code for a region that exhibited a performance change is identical in both releases, we consider the location responsible for the change in the performance of the region as *not identified*. Otherwise, we consider the location as *identified*. Since we manually examine the code as a last step, we can also detect if our function-relative approach fails, e. g., if two code regions simply switch places and, thus, swap their function-relative IDs. We consider the code location responsible for the change in the performance of t as *identified*, if we identify a code location responsible for the change in the performance of f for any $f \in \mathcal{FR}_w^{r_i,r_{i+1}}(t)$.

In all cases, we group all terms affected by a performance change on the black-box side into *location identified* and *location not identified* by the white-box analysis based on our description above. Then, we compute the fraction $l_w^{r_i,r_{i+1}}$ of the performance changes whose cause could be identified and whose cause could not be identified. Formally, we define

$$l_w^{r_i,r_{i+1}} := \begin{cases} \frac{\{t \in \mathcal{T}_{w,rel}^{\blacksquare,r_i,r_{i+1}} \cap \mathcal{T}_{w,rel}^{\square,r_i,r_{i+1}} \mid \text{code location for } t \text{ identified}\}}{|\mathcal{T}_{w,rel}^{\blacksquare,r_i,r_{i+1}}|}, & \text{if } |\mathcal{T}_{w,rel}^{\blacksquare,r_i,r_{i+1}}| > 0, \\ 1, & \text{otherwise.} \end{cases}$$

Note that this definition implies that $l_w^{r_i,r_{i+1}} \leq c_w^{r_i,r_{i+1}}$ for all $w \in \mathcal{W}$ and $r_i, r_{i+1} \in \mathcal{R}$ since we can only identify the configuration-dependent code responsible for a performance change if the white-box analysis confirms the performance change.

Again, we aggregate the data across workloads

$$L^{r_i,r_{i+1}} := \frac{\sum_{l \in \mathcal{W}} l_w^{r_i,r_{i+1}}}{|\mathcal{W}|}$$

and across all workloads and release pairs

$$L := \frac{\sum_{i=1}^{|\mathcal{R}|-1} \sum_{w \in \mathcal{W}} l_w^{r_i, r_{i+1}}}{(|\mathcal{R}| - 1) \cdot |\mathcal{W}|}.$$

5 EVALUATION

This chapter presents the evaluation of our case studies. First, we evaluate each case study as described in Chapter 4. Then, we discuss our results, elaborate on particular findings that emerged during the evaluation and answer our research questions. Finally, we present possible threats to the validity of our results and how we address these threats.

5.1 RESULTS

In this section, we provide an overview of the results for each case study and the overall results across all case studies.

5.1.1 *Performance Changes Between Consecutive Releases for Different Workloads*

As presented in Chapter 4, we investigate the performance changes between consecutive releases for different workloads that the black-box analysis reports in our first research question.

5.1.1.1 *Frequency of Performance Changes*

In the first sub-question, we investigate the frequency of performance changes.

> **RQ1.1:** How frequent are changes of the performance influence of individual configuration options and interactions among them?

CompEnc

For our artificial case study CompEnc, we consider four releases. In each release, we measure six configuration options resulting in twelve configurations.

Figure 5.1 depicts and compares the coefficients of the performance-influence models of CompEnc for all releases. The performance-influence models contain seven terms where five terms (71.4%) are associated with individual configuration options and two terms (28.6%) represent interactions of configuration options. The most influential terms are *Encryption* in v2, v3, and v4 and *root*, *Compression*, and *Encryption* in v1.

Investigating the terms in the model, we note that *Iterations_val_3* does not appear at all in the performance-influence model. This is due to the VIF analysis that we conduct to prevent multicollinearity as described in Chapter 2 to guarantee the comparability of the performance-influence models. Since *Iterations_val_1*, *Iterations_val_2*, and *Iterations_val_3* form a mandatory alternative group, one of them is selected as the default and, thus, omitted from the performance-influence models. In this case, *Iterations_val_3* was

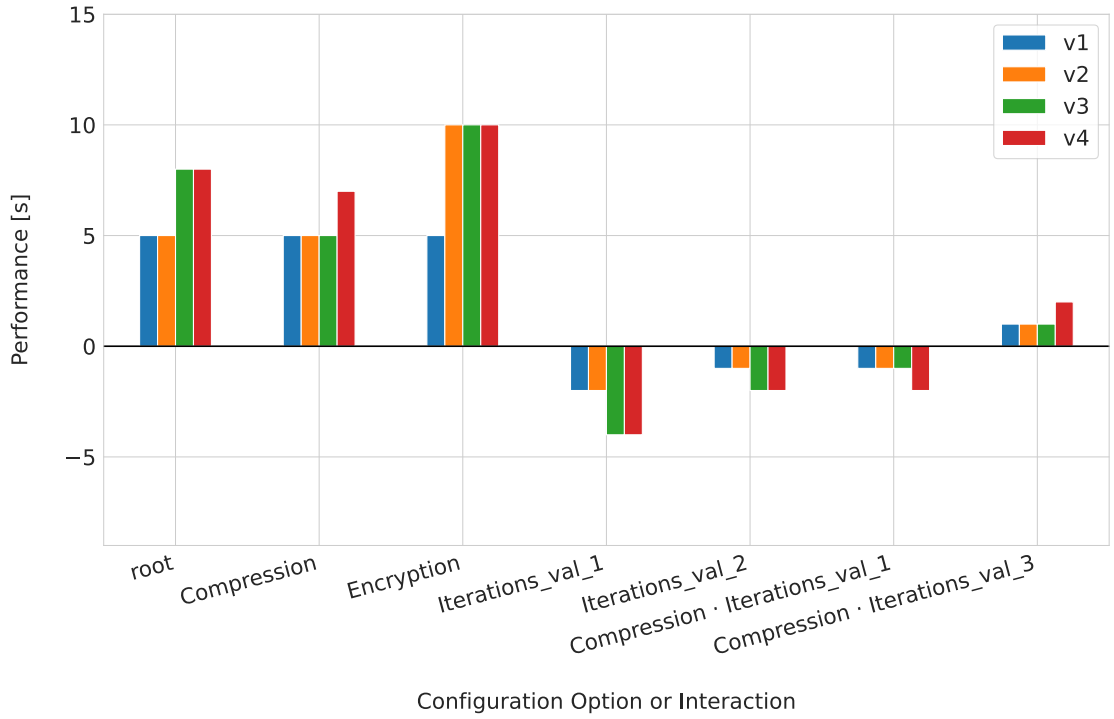


Figure 5.1: Coefficients of the performance-influence models of CompEnc comparing all releases

selected as the default value and, thus, its influence is part of the *root* configuration option. If we recall the source code of CompEnc depicted in Listing 4.1, we see that, in the initial release, we expect an execution time of two seconds not related to any configuration option and three seconds related to *Iterations_val*_3. Summing these two values up, this explains the value of 5 for *root*. Keeping the inclusion of *Iterations_val*_3 in mind, it also becomes clear why the coefficients of *Iterations_val*_1 and *Iterations_val*_2 are negative. Analogously, *Compression* · *Iterations_val*_2 is selected as the default for the interaction between *Compression* and *Iterations_val* and, hence, does not appear in the performance-influence models. Therefore, *Compression* · *Iterations_val*_2 is part of the *Compression* option. The same reasoning as before explains why the values for *Compression* · *Iterations_val*_1 and *Compression* · *Iterations_val*_3 are not what we would intuitively expect.

Table 5.1 shows the metrics for this research question that we defined in Chapter 4. Aggregating $f_w^{r_i,r_{i+1}}$ across all releases, we receive an average fraction of performance changes of approximately $F = 33.3\%$. Since CompEnc is an artificial case study to demonstrate the general process, we do not further elaborate on these numbers for this case study. Instead, we qualitatively analyze the results in the following paragraph.

Between the first two releases (v1 and v2), we introduce a performance regression of the *Encryption* configuration option. By looking at Figure 5.1, we see that the regression is visible in the performance-influence models since the coefficient of *Encryption* increases from 5 to 10. Between the second (v2) and the third (v3) release, we add a regression to the *Iterations_val* option that is as large as the value of *Iterations_val*. Recalling that *Iterations_val*_3 is part of *root*, we identify the performance change for *Iterations_val*_3 in

Table 5.1: Metrics related to CompEnc for RQ1.1

Release 1	Release 2	Workload	$f_w^{r_i,r_{i+1}}$
v1	v2	none	14.3%
v2	v3	none	42.9%
v3	v4	none	42.9%

the coefficient of *root*. Analogously, the coefficients of *Iterations_val*_1 and *Iterations_val*_2 decrease even further due to the increase of the coefficient of the *root* option. In the last release (v4), we include a performance change affecting the interaction between *Compression* and *Iterations_val* that is as large as the value of *Iterations_val*. As before, the performance models reflect the performance change by including an increase of the coefficient of *Compression* and, consequently, a change in the coefficients of *Compression* · *Iterations_val*_1 and *Compression* · *Iterations_val*_3.

Overall, we see that all performance regressions that we introduce are detected by the black-box approach. However, some of them are not attributed to the term that we would expect them to be.

PicoSAT

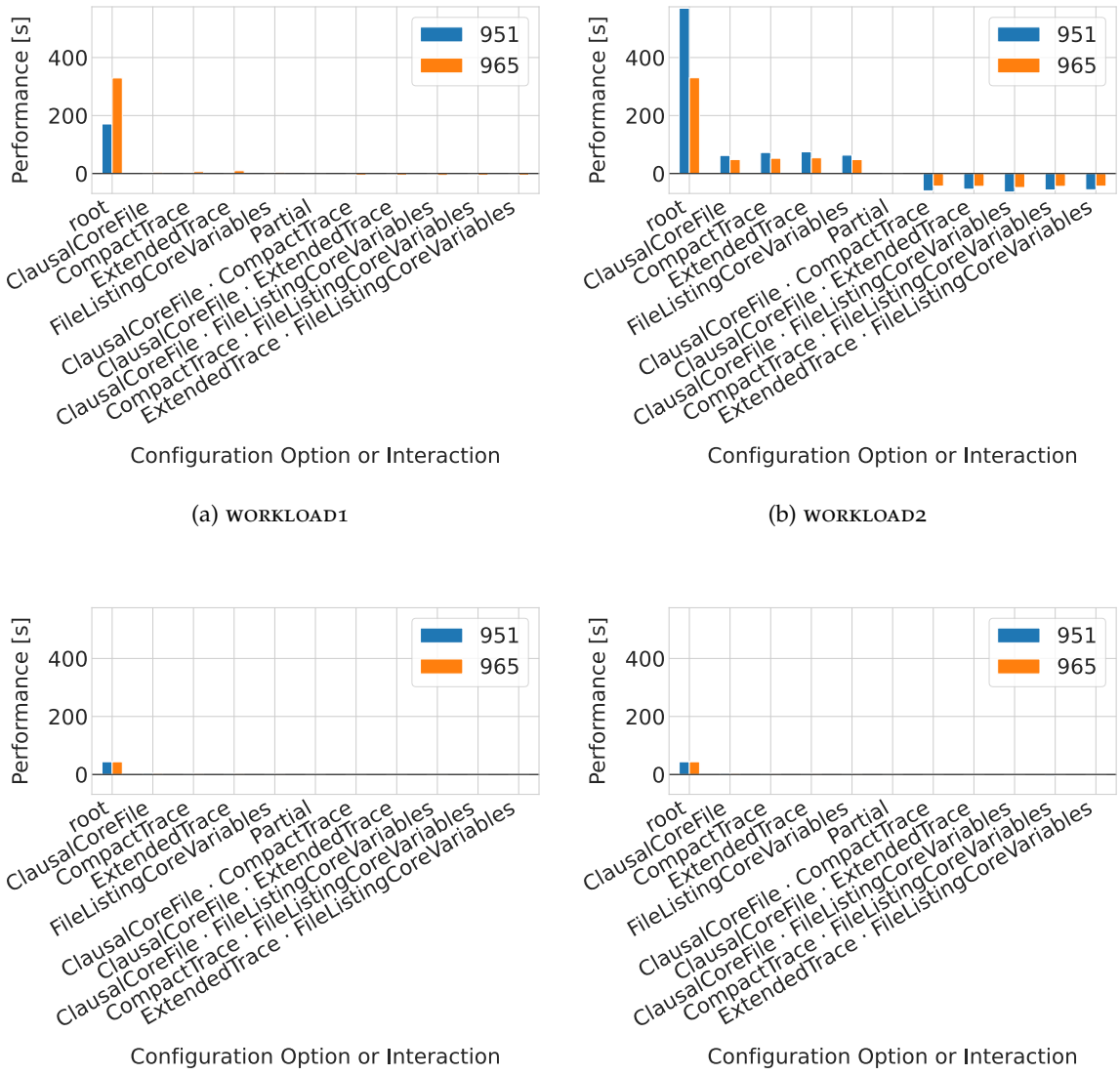
As described in Chapter 4, we measure 16 configuration options resulting in 182 configurations with four different workloads in PicoSAT and we compare two releases. For better readability, we assign IDs to each workload and reference the workloads by ID in the following. Table 5.2 contains the mapping from workloads to IDs.

Table 5.2: Workload IDs for PicoSAT

Workload	ID
abw-N-bcsstk07.mtx-w44.cnf	workload1
traffic_kkb_unknown.cnf	workload2
UNSAT_H_instances_childsnack_p11.hddl_1.cnf	workload3
UNSAT_H_instances_childsnack_p12.hddl_1.cnf	workload4

In Figure 5.2, we plot the coefficients of the performance-influence models for all workloads and compare both releases. The performance-influence models for PicoSAT contain eleven terms. Six terms (54.5%) correspond to individual configuration options whereas five terms (45.5%) correspond to interactions of configuration options. The most influential configuration option in all performance-influence models is the *root* option.

We note that nine out of eleven (81.8%) terms in the performance-influence models are related to generating traces or proofs for the satisfiability of the workload. When looking at all configuration options, only five out of 16 configuration options (31.3%) deal with tracing and proofs. Therefore, we observe that tracing- and proof-related configuration options are overrepresented in the performance-influence models.



(a) WORKLOAD1

(b) WORKLOAD2

(c) WORKLOAD3

(d) WORKLOAD4

Figure 5.2: Coefficients of the performance-influence models of PICOSAT for all workloads comparing all releases

Table 5.3 contains the metrics for PicoSAT for this research question. We observe performance changes affecting at least one configuration option or interaction for three out of four (75%) workload and release pair combinations. On average, $F = 47.7\%$ of all configuration options and interactions across all workloads and releases are affected by a performance change. Regarding both the presence and frequency of performance changes, we observe a strong dependency on the workload. For WORKLOAD1 and WORKLOAD2, ten (90.9%) configuration options or interactions exhibit a performance change. In contrast, for WORKLOAD4 we see only a single (9.1%) relevant change and even no relevant changes at all (0%) for WORKLOAD3.

Table 5.3: Metrics related to PicoSAT for RQ1.1

Release 1	Release 2	Workload	$f_w^{r_i,r_{i+1}}$
951	965	WORKLOAD1	90.9%
951	965	WORKLOAD2	90.9%
951	965	WORKLOAD3	0.0%
951	965	WORKLOAD4	9.1%

The configuration options and interactions that participate in a performance change are identical for WORKLOAD1 and WORKLOAD2. However, we note that for all affected terms, the influence increases for WORKLOAD1, but decreases for WORKLOAD2. The only term that participates in a performance change for WORKLOAD4 also exhibits a change in WORKLOAD1 and WORKLOAD2, i. e., we see that $\mathcal{T}^{\blacksquare,951,965}_{\text{WORKLOAD4},rel} \subset \mathcal{T}^{\blacksquare,951,965}_{\text{WORKLOAD1},rel} = \mathcal{T}^{\blacksquare,951,965}_{\text{WORKLOAD2},rel}$.

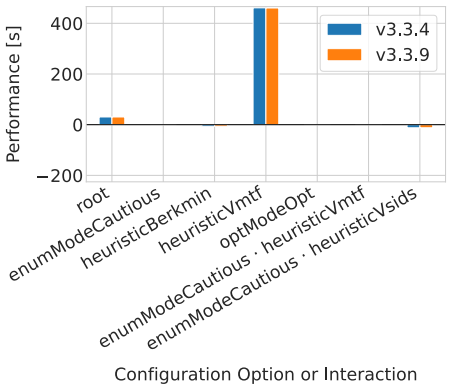
Clasp

For Clasp, we measure seven workloads in two releases. We consider 33 configuration options which result in 91 configurations. As before, we assign IDs to workloads for better readability. The mapping is shown in Table 5.4.

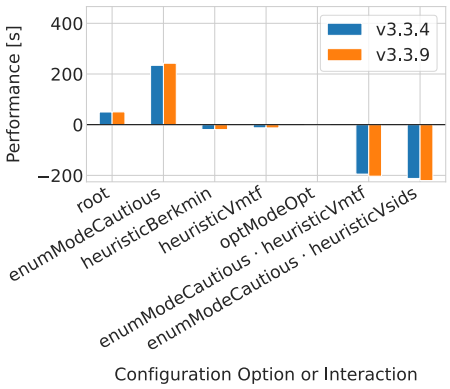
Table 5.4: Workload IDs for Clasp

Workload	ID
ABW-N-BCSSTK07.MTX-W44.CNF	WORKLOAD1
SAT_H_INSTANCES_CHILDSNACK_P08.HDDL_2.CNF	WORKLOAD2
SAT_P_OPT_SNAKE_P10.PDDL_27.CNF	WORKLOAD3
TRAFFIC_KKB_UNKNOWN.CNF	WORKLOAD4
UNSAT_H_INSTANCES_CHILDSNACK_P11.HDDL_1.CNF	WORKLOAD5
UNSAT_H_INSTANCES_CHILDSNACK_P12.HDDL_1.CNF	WORKLOAD6
UNSAT_P_OPT_SNAKE_P06.PDDL_30.CNF	WORKLOAD7

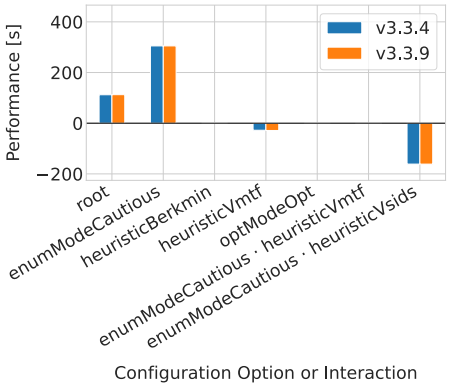
We plot the coefficients of the performance-influence models for Clasp per workload in Figure 5.3. We omit the performance-influence model for WORKLOAD5 because it is



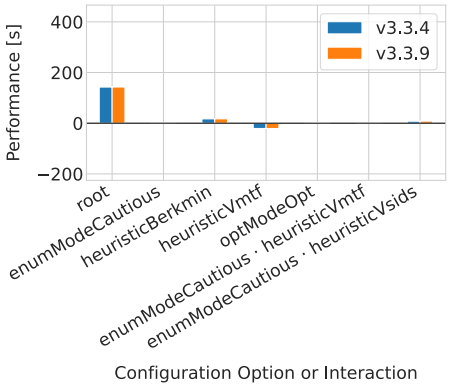
(a) WORKLOAD1



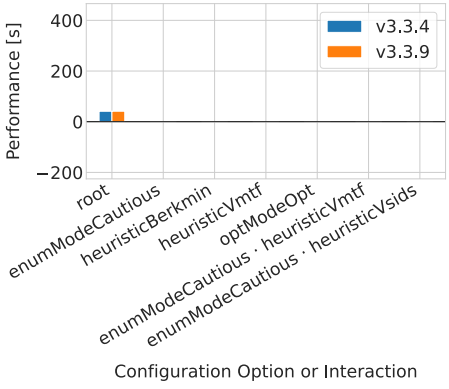
(b) WORKLOAD2



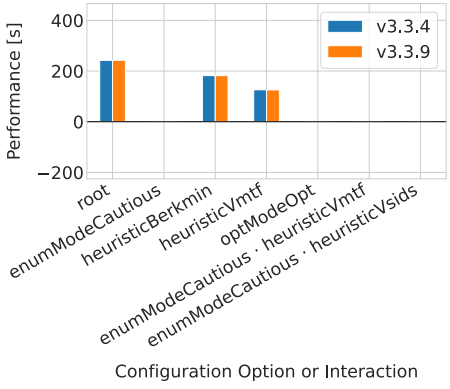
(c) WORKLOAD3



(d) WORKLOAD4



(e) WORKLOAD6



(f) WORKLOAD7

Figure 5.3: Coefficients of the performance-influence models of CLASP for six workloads comparing all releases

very similar to the model of WORKLOAD6. The performance-influence models for CLASP contain seven terms. Five (71.4%) terms correspond to individual configuration options and two (28.6%) terms correspond to interactions of configuration options. Depending on the workload, different terms are the most influential terms. For WORKLOAD1, the most influential term is *heuristicVmtf* which specifies the usage of the VMTF [17] heuristic. *enumModeCautious* is the most influential term for WORKLOAD2 and WORKLOAD3 and selects a cautious enumeration algorithm. For the remaining workloads, *root* has the strongest influence.

Looking at all models, we see that the main source for variation in the performance is related to the heuristic that is used and to the enumeration mode. Furthermore, interactions between the heuristic and the enumeration mode also influence the performance for some workloads. Apart from the heuristic and the enumeration algorithm, the only other configuration option included in the performance-influence models is *optModeOpt* which selects an optimization algorithm. However, its influence is minor with at most 0.2*s* which is less than 0.5% in relation to the average performance.

Additionally, we note that depending on the workload, different heuristics have a completely different impact on the performance. For example, when comparing WORKLOAD4 and WORKLOAD7, we see that using *heuristicVmtf* decreases the expected run-time for WORKLOAD4, but increases it for WORKLOAD7. We make similar observations when comparing other workloads. Additionally, it is noteworthy that for WORKLOAD5 and WORKLOAD6, the configuration almost does not matter at all since around 99% of the performance influence is attributed to the *root* feature. In contrast, for WORKLOAD1 we observe that the influence of *heuristicVmtf* is around 460*s* (approximately 90% in relation to the average performance) and therefore considerably larger than the influence of all other configuration options, including *root*. We make similar observations for the enumeration mode, e. g., when comparing WORKLOAD1 and WORKLOAD2 or WORKLOAD3.

In Table 5.5, we list the metrics for CLASP for this research question. We observe performance changes affecting at least one configuration option or interaction for two out of seven (28.6%) release pair and workload combinations. On average, $F = 10.2\%$ of all configuration options are affected by a performance change across all workloads and releases. WORKLOAD1, WORKLOAD3, WORKLOAD4, WORKLOAD6, and WORKLOAD7 do not exhibit any performance change. For WORKLOAD2, three (42.9%) terms exhibit a performance change. In contrast, for WORKLOAD5 we observe a performance change for two (28.6%) terms. The interaction $enumModeCautious \cdot heuristicVmtf$ participates in a performance change for both workloads. Additionally, the influence of *heuristicVmtf* changes for WORKLOAD5. In contrast, *enumModeCautious* and the interaction $enumModeCautious \cdot heuristicVsids$ exhibit a performance change for WORKLOAD2. Therefore, not only the number of affected configuration options and interactions depends on the workload, but also the affected terms themselves.

XZ

For XZ, we consider three releases and two workloads. For each release and workload, we consider 79 configuration options resulting in 713 configurations. We investigate the compression and the decompression process separately.

Table 5.5: Metrics related to Clasp for RQ1.1

Release 1	Release 2	Workload	$f_w^{r_i,r_{i+1}}$
v3.3.4	v3.3.9	WORKLOAD1	0.0%
v3.3.4	v3.3.9	WORKLOAD2	42.9%
v3.3.4	v3.3.9	WORKLOAD3	0.0%
v3.3.4	v3.3.9	WORKLOAD4	0.0%
v3.3.4	v3.3.9	WORKLOAD5	28.6%
v3.3.4	v3.3.9	WORKLOAD6	0.0%
v3.3.4	v3.3.9	WORKLOAD7	0.0%

COMPRESSION. For the compression process, the performance-influence models for XZ contain 59 terms. Out of these 59 terms, 22 (37.3%) correspond to individual configuration options and 37 (62.7%) correspond to interactions of configuration options. The most influential configuration option in all performance-influence models that affect the compression process is the *root* option. We do not observe any peculiarities in the performance-influence models and the performance changes we observe are very weak (see RQ1.2) which is why we omit the plotting of the coefficients for the compression process.

Table 5.6: Metrics related to the compression process of XZ for RQ1.1

Release 1	Release 2	Workload	$f_w^{r_i,r_{i+1}}$
v5.2.3	v5.2.6	DAVIS 2016	11.9%
v5.2.3	v5.2.6	ENWIK9	5.1%
v5.2.6	v5.3.3alpha	DAVIS 2016	10.2%
v5.2.6	v5.3.3alpha	ENWIK9	3.4%

Table 5.6 lists the metrics for the compression process of XZ. We observe performance changes affecting at least one configuration option in all workload and release pair combinations. We note an average fraction of affected configuration options and interactions of $F = 7.6\%$. Between v5.2.3 and v5.2.6, $F^{v5.2.3,v5.2.6} = 8.5\%$ of all terms participate in a performance change, on average. For the DAVIS 2016 workload, seven out of 59 (11.9%) terms exhibit a change in their influence whereas the influence of three (5.1%) terms changes for ENWIK9. We note that the sets of affected terms for both workloads are disjunct. All changes for DAVIS 2016 relate to the compression level (*Preset_val*) and interactions of the compression level with the compression format (*xz*, *auto*, and *raw*). In contrast, for ENWIK9 we also see a change associated with an interaction between the compression level and the compression format, but also changes in the influence of the *XtrmCPU* option and the selection of the block size. The *XtrmCPU* option tries to improve the compression ratio by using more CPU time. When comparing v5.2.6 and v5.3.3alpha, the influence of $F^{v5.2.6,v5.3.3alpha} = 6.8\%$ of all terms changes, on average. Similar to the previous release pair, we observe a change in six out of 59 (10.2%) terms for DAVIS 2016 and a change in

two (3.4%) terms for ENWIK9. Again, the sets of affected terms are disjunct. For ENWIK9, both changes are related to the interaction between the compression level and *XtrmCPU*. In contrast, changes for DAVIS 2016 affect *root* and interactions between the compression format and the maximum number of threads (*Threads_val*).

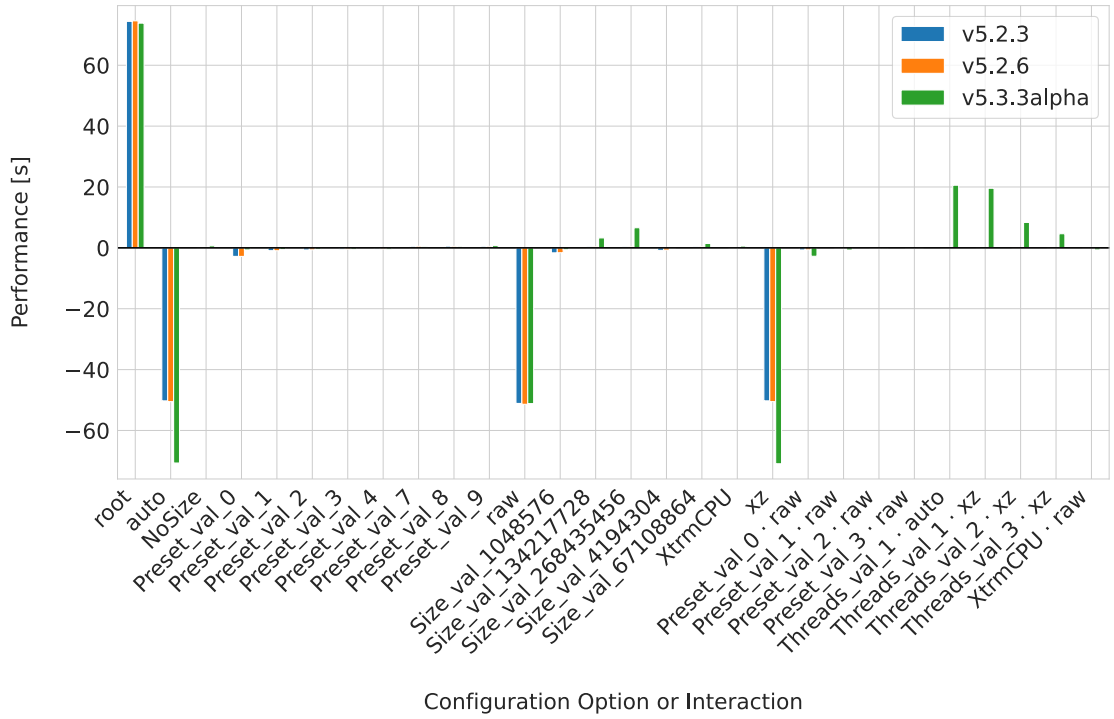
DECOMPRESSION. The performance-influence models for the decompression process of XZ contain 28 terms. 19 (67.9%) terms correspond to individual configuration options whereas nine (32.1%) terms correspond to interactions of configuration options. In all performance-influence models, the most influential term corresponds to the *root* option.

Figure 5.4 depicts the coefficients of the performance-influence models for the decompression phase for both workloads with both release pairs. We identify six categories of configuration options that are included in the performance-influence models. First, as always, the *root* feature is part of the performance-influence models. Then, the compression formats *xz*, *raw*, and *auto* influence the performance. Additionally, the compression level represented by the *Preset_val* options has an impact on the performance. Furthermore, the block size values (*Size_val*) and the number of threads (*Threads_val*) are included in the performance-influence models. Lastly, the *XtrmCPU* option impacts the performance. We also see interactions between the compression level and the compression format, the compression format and the number of threads, and between *XtrmCPU* and the compression format.

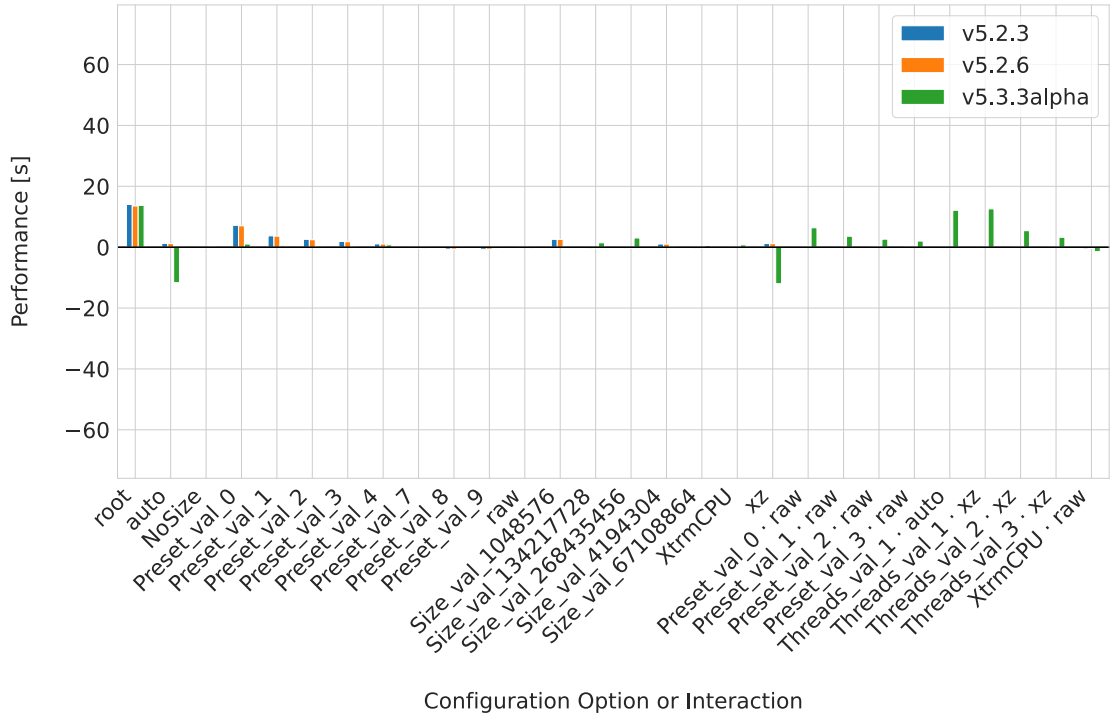
In Table 5.7, we list the metrics for all workloads and release pairs. All workload and release pair combinations exhibit a performance change affecting the influence of at least one configuration option or interaction. On average, $F = 44.6\%$ of all configuration options participate in a performance change of the decompression process. However, between v5.2.3 and v5.2.6 the performance-influence of only $F^{v5.2.3,v5.2.6} = 10.7\%$ of options and interactions changes while between v5.2.6 and v5.3.3alpha the performance of $F^{v5.2.6,v5.3.3alpha} = 78.6\%$ of options and interactions changes. Again, we note a dependency on the workload. For the first release pair, performance changes affect three terms (10.7%) in both workloads, although the sets of affected terms are, again, disjunct. However, all changes except for one are associated with the compression level and the compression format. We additionally see a change in *root* for ENWIK9. For the second release pair, again both workloads are affected by changes. However, only 19 (67.9%) terms change for DAVIS 2016 while 25 (89.3%) terms change for ENWIK9. We note that, ignoring the *root* configuration option, $\mathcal{T}^{\blacksquare,v5.2.6,v5.3.3alpha}_{\text{DAVIS 2016},rel} \subset \mathcal{T}^{\blacksquare,v5.2.6,v5.3.3alpha}_{\text{ENWIK9},rel}$, i. e., all terms (except for *root*) that exhibit a change for DAVIS 2016 also exhibit a change in ENWIK9. The terms that exhibit a change in both workloads either determine the compression format, the compression level, the number of threads, and the block size or interactions between number of threads and compression format or compression level and compression format. For the ENWIK9 workload, additional values for the compression level and interactions with it are affected.

Fast Downward

We measure nine releases with 19 workloads for FAST DOWNWARD. For each workload and release, we consider 60 configuration options leading to 412 configurations. Again, we assign IDs to workloads for better readability and show the mapping in Table 5.8.



(a) DAVIS 2016



(b) ENWIK9

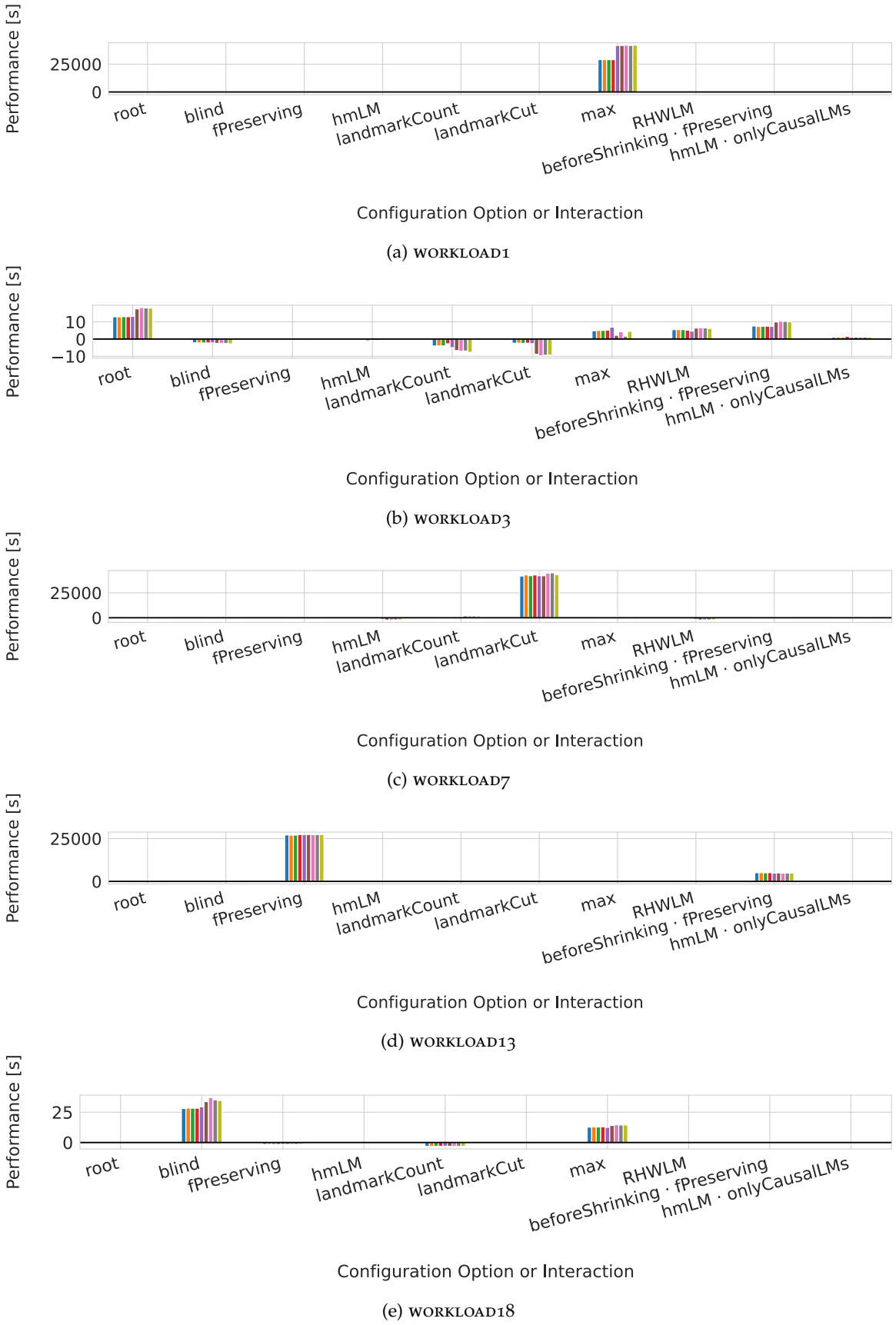
Figure 5.4: Coefficients of the performance-influence models of XZ for the decompression phase for both workloads comparing all releases

Table 5.7: Metrics related to the decompression process of XZ for RQ1.1

Release 1	Release 2	Workload	$f_w^{r_i,r_{i+1}}$
v5.2.3	v5.2.6	DAVIS 2016	10.7%
v5.2.3	v5.2.6	ENWIK9	10.7%
v5.2.6	v5.3.3alpha	DAVIS 2016	67.9%
v5.2.6	v5.3.3alpha	ENWIK9	89.3%

Table 5.8: Workload IDs for Fast Downward

Workload	ID	Workload	ID
AGRICOLA_P02	WORKLOAD1	SOKOBAN_OPT08_P08	WORKLOAD11
DATA_NETWORK_P05	WORKLOAD2	SOKOBAN_P13	WORKLOAD12
ELEVATORS_P22	WORKLOAD3	SOKOBAN_P17	WORKLOAD13
GED_D28	WORKLOAD4	TERMES_P17	WORKLOAD14
GED_D43	WORKLOAD5	TRANSPORT_OPT08_P04	WORKLOAD15
HIKING_PTESTING225	WORKLOAD6	TRANSPORT_P04	WORKLOAD16
HIKING_PTESTING226	WORKLOAD7	TRANSPORT_P08	WORKLOAD17
HIKING_PTESTING44	WORKLOAD8	VISITALL_OPT11_P05	WORKLOAD18
SCANALYZER_P11	WORKLOAD9	VISITALL_OPT11_P056	WORKLOAD19
SOKOBAN_OPT08_P04	WORKLOAD10		



(a) WORKLOAD1

(b) WORKLOAD3

(c) WORKLOAD7

(d) WORKLOAD13

(e) WORKLOAD18

Figure 5.5: Coefficients of an excerpt of the performance-influence models of FAST DOWNWARD for five workloads comparing all releases where each bar for a term corresponds to a release and releases increase from left (July 2016) to right (June 2020)

The performance-influence models for FAST DOWNWARD contain 131 terms where 29 (22.1%) terms correspond to individual configuration options and 102 (77.9%) correspond to interactions between configuration options. Due to the large number of terms, we only plot the coefficients of the ten most influential terms in Figure 5.5. By most influential terms, we refer to the terms which have the highest absolute coefficients. Furthermore, we do not plot the coefficients for all workloads due to the high number of workloads. We group the workloads by their most influential option and plot the performance-influence model with the highest coefficient associated with the most influential term. For WORKLOAD1 and WORKLOAD9, *max* is the most influential option. It specifies the usage of the h^{max} [12] heuristic. With WORKLOAD3, *root* is the most influential option. When using WORKLOAD18 and WORKLOAD19, the most influential term is associated with the *blind* configuration option which enables blind search instead of heuristic search. For WORKLOAD2 and for WORKLOAD10 to WORKLOAD13, the most influential option is *fPreserving* which selects an f-preserving shrink strategy [13]. For the remaining workloads, *lmCut* is the most influential term which specifies usage of the *landmarkCut* [12] heuristic.

As we can see, the influence of each option is highly dependent on the workload. On average, the most influential option is responsible for 30.8% (WORKLOAD3) to 98.3% (WORKLOAD9) of variability in the performance, showing that for all workloads, a single configuration option is responsible for a large part of the variability.

In Table 5.9, we show the aggregated metrics over all workloads for each release pair for FAST DOWNWARD. Due to the number of combinations, we omit the detailed listing of the metrics for all release pair and workload combinations. We note performance changes affecting at least one configuration option or interaction for 150 of 152 (98.7%) workload and release pair combinations. Aggregating the results across all release pairs, we observe that, on average, the influence of $F = 19.9\%$ of all configuration options and interactions changes between releases. Excluding the workload and release pair combinations where no performance changes were present, the number of affected terms ranges from one out of 131 (0.8%) terms for WORKLOAD19 to 78 out of 131 (59.5%) terms for WORKLOAD15. Again, we observe a strong dependency on the workload for the frequency of performance changes. Analogously to the fraction of affected terms, the affected terms themselves also depend on the release pair and the workload. However, we observe that some configuration options change more frequently than others. Five out of 131 (3.8%) terms participate in a change in more than 50% of all workload and release pair combinations. These terms are *max* (121 changes), *landmarkCut* (105 changes), *landmarkCount* (95 changes), *blind* (90 changes), and *canonicalPDB* (78 changes). We note that all of these terms are related to the selection of a heuristic.

Summary

Considering all real-world case studies, we see that the average fraction of configuration options and interactions that contribute to a performance change ranges from 0.8% for the compression phase of XZ to 47.7% for PICOSAT. Aggregating the results for all case studies, we observe an average fraction of 24.2% of configuration options and interactions that participate in a performance change. Looking at the individual results for workloads and release pairs, the fraction ranges from 0% (e. g., for CLASP v3.3.4 and v3.3.9 for WORKLOAD3,

Table 5.9: Metrics related to FAST DOWNWARD for RQ1.1 aggregated over all workloads

Release 1	Release 2	$F^{r_i,r_{i+1}}$
2016_07	2017_01	20.5%
2017_01	2017_07	20.6%
2017_07	2018_01	21.7%
2018_01	2018_07	31.6%
2018_07	2019_01	17.2%
2019_01	2019_06	23.5%
2019_06	2019_12	10.8%
2019_12	2020_06	13.2%

WORKLOAD4, WORKLOAD5, and WORKLOAD7) to 90.9% (e. g., for PICOSAT 951 and 965 for WORKLOAD1 and WORKLOAD2).

Overall, we see that is very rare that the influence of no configuration option or interaction changes between two releases. In all case studies, performance changes occur in each release pair for at least one workload. Additionally, all case studies show that performance changes heavily depend on the workload that is used. Not only does the fraction of affected terms change between workloads, but also the affected terms themselves change. For some case studies, we even see that the sets of affected terms are almost disjunct between some workloads.

5.1.1.2 *Strength of Performance Changes*

The second sub-question deals with the strength of the performance changes that we identified in RQ1.1.

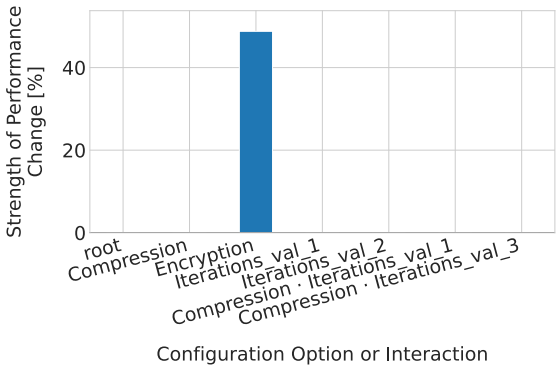
> **RQ1.2:** How strong are changes of the performance influence of individual configuration options and interactions among them?

CompEnc

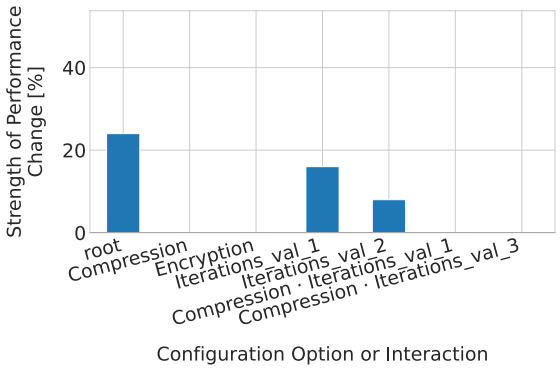
In Figure 5.6, we depict the strength of the performance changes for each term included in the performance-influence models of COMPENC.

If we aggregate $\Delta_w^{r_i,r_{i+1}}$ across all releases, we see an average strength of $\Delta = 24.8\%$. However, analogous to RQ1.1, the reported strength of a configuration option between individual release pairs differs from the actual strength if the option is part of an alternative group. This applies to all changes related to the number of compression iterations (*Iterations_val*). The changes for the *Encryption* configuration option align with our expectations.

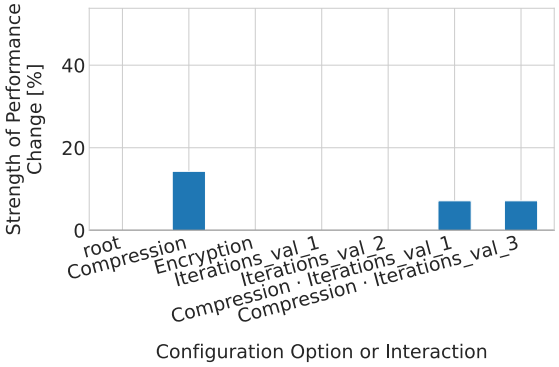
Table 5.10 contains the metrics for this research question according to our definition in Chapter 4. Analogously to RQ1.1, we do not elaborate on these numbers further for COMPENC due to its artificiality.



(a) V1 AND V2



(b) V2 AND V3



(c) V3 AND V4

Figure 5.6: Strength of performance changes of CompEnc for all release pairs

Table 5.10: Metrics related to CompEnc for RQ1.2

Release 1	Release 2	Workload	$\Delta_w^{r_i,r_{i+1}}$
V1	V2	none	48.8%
V2	V3	none	16.0%
V3	V4	none	9.5%

PicoSAT

In Figure 5.7, we plot the performance changes for each option in all workloads that exhibit any performance change. Note that we only display the performance changes that exceed the relevance threshold. All others are drawn with a strength of 0. The most notable changes in the performance influence are related to the *root* configuration option. For WORKLOAD1 and WORKLOAD2, we observe a performance change of $\delta^{\blacksquare,951,965}_{workload_1}(root) = 62.8\%$ and $\delta^{\blacksquare,951,965}_{workload_2}(root) = 49.4\%$, respectively. This is not necessarily surprising since *root* is also by far the most influential configuration option in both releases. Interestingly, the performance change of *root* is below the relevance threshold for WORKLOAD3 and WORKLOAD4. For all other relevant configuration options and interactions, the changes are between 2% and 5% for WORKLOAD1 and WORKLOAD2. Looking at the data for WORKLOAD4, the only relevant performance change has a strength of 2.5%.

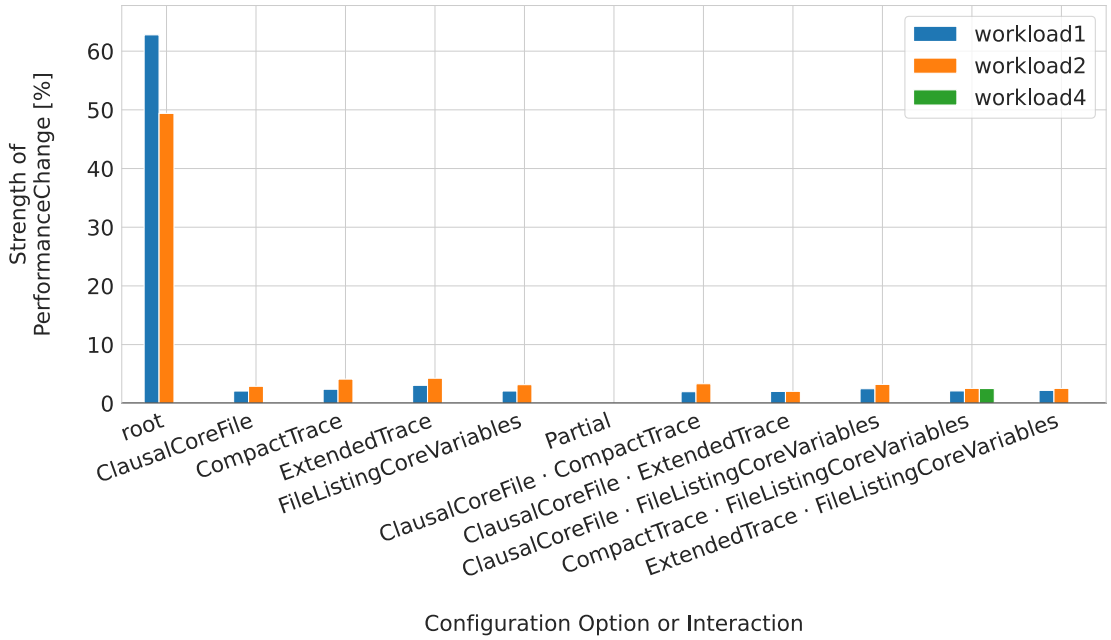


Figure 5.7: Strength of performance changes of PICOSAT comparing all workloads that exhibit any performance change

Aggregating the data, we observe an average strength of performance changes of $\Delta = 4.6\%$ across workloads and releases. In Table 5.11, we show the results for all releases and workloads.

We see that the average strength of the performance changes differs between workloads. WORKLOAD1 and WORKLOAD2 have changes of similar strength with 8.3% and 7.8%, respectively. In contrast, WORKLOAD3 and WORKLOAD4 have weaker performance changes with an average strength of 0% and 2.5%, respectively. Considering the individual terms, we observe a similar distribution of strengths for WORKLOAD1 and WORKLOAD2. Except for *root*, WORKLOAD2 exhibits stronger changes for all terms in comparison to WORKLOAD1. For

Table 5.11: Metrics related to PicoSAT for RQ1.2

Release 1	Release 2	Workload	$\Delta_w^{r_i,r_{i+1}}$
951	965	WORKLOAD1	8.3%
951	965	WORKLOAD2	7.8%
951	965	WORKLOAD3	0.0%
951	965	WORKLOAD4	2.5%

WORKLOAD4, the only affected term is *CompactTrace* · *FileListingCoreVariables* for which the strength is nearly identical to the strength of the term for WORKLOAD2.

Clasp

In Figure 5.8, we plot the strength of the performance changes for each configuration option and interaction and for the two workloads that exhibit any performance changes. We see that, generally, WORKLOAD2 exhibits stronger performance changes than WORKLOAD5. Additionally, we note that all changes for WORKLOAD2 are of similar strength, ranging from 9.1% to 9.3% for *enumModeCautious*, *enumModeCautious* · *heuristicVmtf*, and *enumModeCautious* · *heuristicVsids*. Similarly, the changes for WORKLOAD5 are of similar strength with 1.2% and 1.1% for *heuristicVmtf* and *enumModeCautious* · *heuristicVmtf*, respectively.

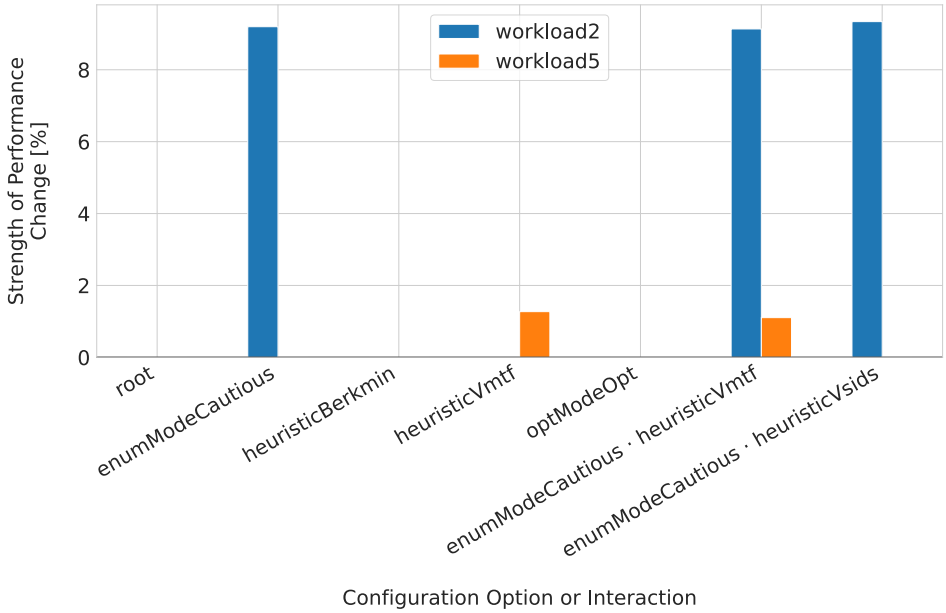


Figure 5.8: Strength of performance changes of Clasp comparing all workloads that exhibit any performance change

We observe an average change of $\Delta = 1.5\%$ in the influence of any configuration option or interaction across all workloads. In Table 5.12, we list the computation results for each

workload for the metric we defined for this research question. We note that WORKLOAD2 exhibits stronger performance changes in comparison to WORKLOAD6 which only exhibits a minor performance change.

Table 5.12: Metrics related to CLASP for RQ1.2

Release 1	Release 2	Workload	$\Delta_w^{r_i,r_{i+1}}$
v3.3.4	v3.3.9	WORKLOAD1	0.0%
v3.3.4	v3.3.9	WORKLOAD2	9.2%
v3.3.4	v3.3.9	WORKLOAD3	0.0%
v3.3.4	v3.3.9	WORKLOAD4	0.0%
v3.3.4	v3.3.9	WORKLOAD5	1.2%
v3.3.4	v3.3.9	WORKLOAD6	0.0%
v3.3.4	v3.3.9	WORKLOAD7	0.0%

Again, we observe a strong dependency of the strength of performance changes on the workload.

XZ

COMPRESSION. For the compression phase of XZ, we observe very weak performance changes that only barely exceed the relevance threshold. Therefore, we omit plotting the strengths of the performance changes. The option that exhibits the strongest change is the *root* option for which we observe a performance change of $\delta^{\blacksquare,v5.2.6,v5.3.3alpha}_{\text{DAVIS 2016}}(root) = 1.7\%$ for the DAVIS 2016 workload in the first release pair. For ENWIK9 or the second release pair, we observe no performance change at all for *root*.

Aggregating the data, we observe an average strength of performance changes of $\Delta = 1.0\%$ across workloads and releases. In Table 5.13, we show the results for all releases and workloads aggregated over all terms. The changes are of strength 0.8% to 1.2%. Looking at all terms individually, the strength of changes ranges from $\delta^{\blacksquare,v5.2.3,v5.2.6}_{\text{DAVIS 2016}}(Preset\_val\_9 \cdot raw) = 0.6\%$ to $\delta^{\blacksquare,v5.2.6,v5.3.3alpha}_{\text{DAVIS 2016}}(root) = 1.7\%$, i. e., there is only little variety in the strength of changes for the compression phase of XZ.

Table 5.13: Metrics related to the compression process of XZ for RQ1.2

Release 1	Release 2	Workload	$\Delta_w^{r_i,r_{i+1}}$
v5.2.3	v5.2.6	DAVIS 2016	1.2%
v5.2.3	v5.2.6	ENWIK9	0.8%
v5.2.6	v5.3.3alpha	DAVIS 2016	1.2%
v5.2.6	v5.3.3alpha	ENWIK9	0.9%

DECOMPRESSION. For the decompression process, we see more and stronger changes. In Figure 5.9, we plot the strength of the performance changes for each configuration option and interaction for both workloads and both release pairs.

Table 5.14 contains the metrics for the decompression phase of XZ for this research question. On average, we observe a change of $\Delta = 15.3\%$ in the influence of configuration options or interactions across both release pairs and workloads. With $\Delta^{v5.2.3,v5.2.6} = 1.4\%$ and $\Delta^{v5.2.6,v5.3.3alpha} = 29.2\%$ we see that the changes between v5.2.3 and v5.2.6 are much weaker than between v5.2.6 and v5.3.3alpha. We also note that some terms exhibit much stronger changes than others. The strongest change is associated with the *xz* configuration option with $\delta^{\blacksquare,v5.2.6,v5.3.3alpha}_{\text{ENWIK9}}(xz) = 105.5\%$. The weakest change affects compression level two with $\delta^{\blacksquare,v5.2.3,v5.2.6}_{\text{ENWIK9}}(Preset\_val\_2) = 0.8\%$.

Table 5.14: Metrics related to the decompression process of XZ for RQ1.2

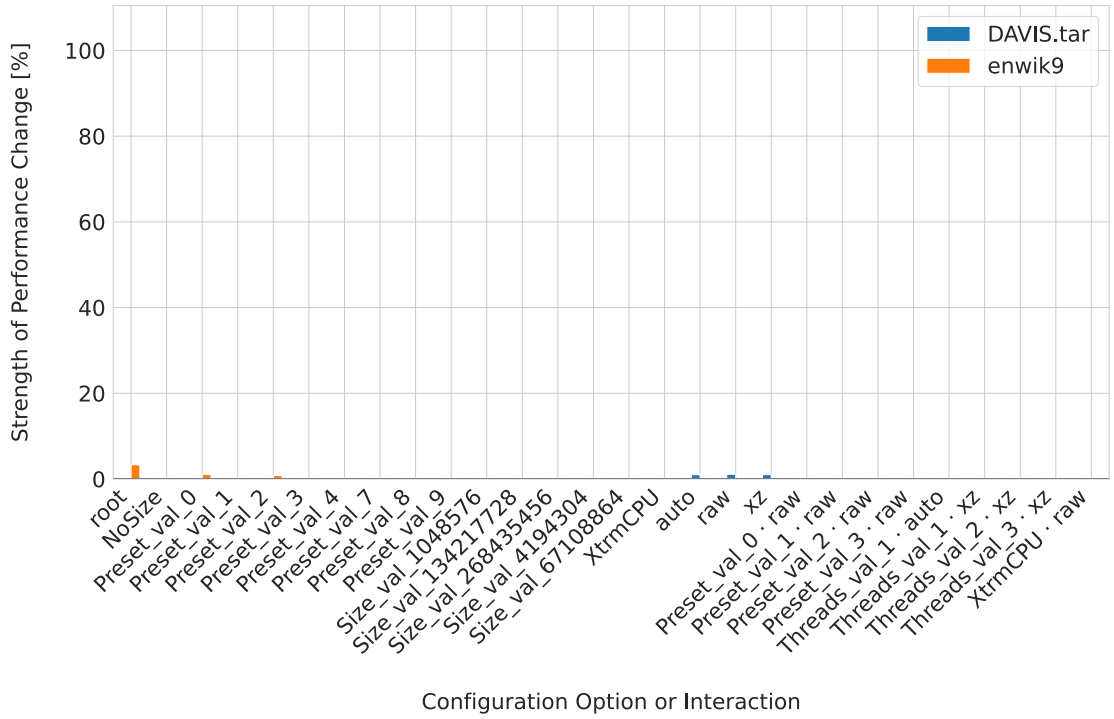
Release 1	Release 2	Workload	$\Delta^{r_i,r_{i+1}}_w$
v5.2.3	v5.2.6	DAVIS 2016	1.0%
v5.2.3	v5.2.6	ENWIK9	1.7%
v5.2.6	v5.3.3alpha	DAVIS 2016	26.8%
v5.2.6	v5.3.3alpha	ENWIK9	31.6%

Generally, we see a wide variety in the strength of performance changes. While we still see a dependency on the workload, the distribution of the strengths across the terms is similar for both workloads. Recalling that the sets of affected terms for v5.2.6 and v5.3.3.alpha are very similar, we note that we see stronger changes for ENWIK9 for all terms except for *root* and *Size_val_*268435456.

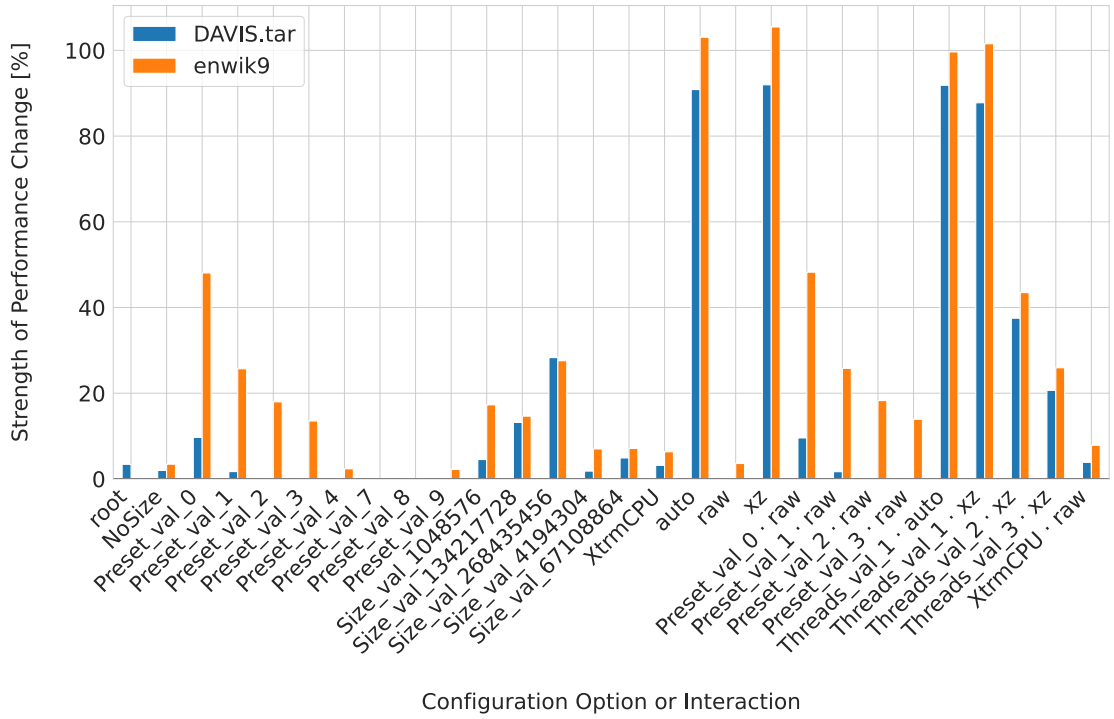
Fast Downward

In Figure 5.10, we plot the strength of performance changes of a selection of release pairs. We display the two release pairs that contain the terms with the strongest changes. As for the first subquestion, we only plot the ten most influential terms.

We see that the strength of the performance change heavily depends on the workload. In comparison to the other case studies, the performance changes for FAST DOWNWARD are much stronger. For example, in both plots, we see that the *max* configuration option exhibits the strongest change for WORKLOAD9 with changes of strength 4 228.5% and 2 182.2% between 2018_07 and 2019_01 and between 2017_07 and 2018_01, respectively. While changes of such strength seem very extreme, we need to keep in mind that we calculate the strength of a performance change in relation to the average performance of that release pair and workload. Some configuration options or interactions have a very high influence that exceeds the average performance by far. For example, for WORKLOAD9 and release 2018_07, enabling the *max* configuration option increases the run-time by more than 10 000*s* while the average run-time is only 48*s*. Therefore, changes in the influence of *max* that are small in relation to the influence of *max* are still huge in relation to the average performance. Nevertheless, changes that are small in relation to the influence of *max* still result in a change of the

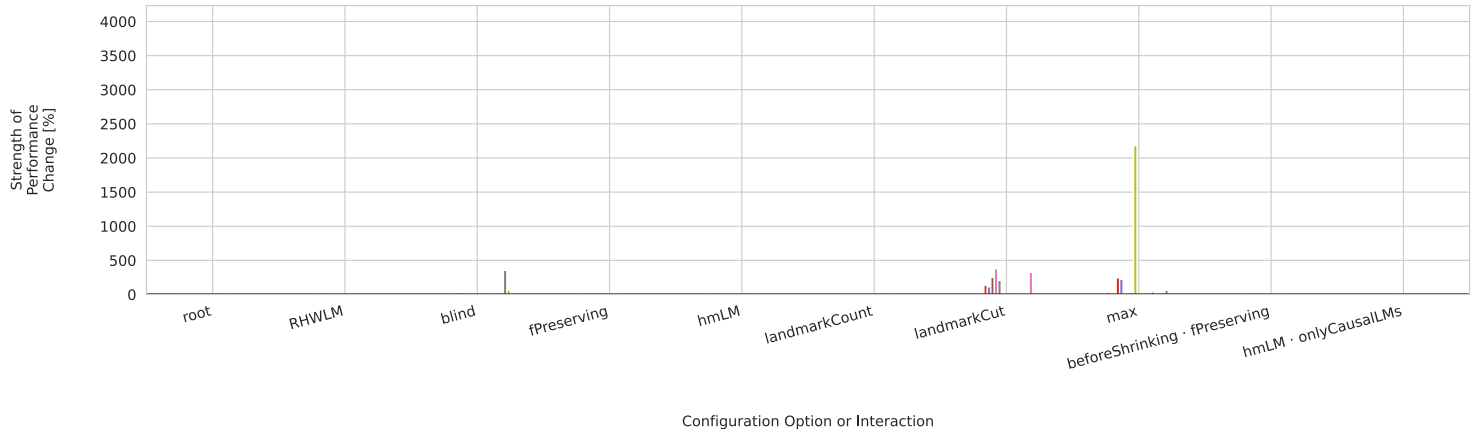


(a) v5.2.3 and v5.2.6

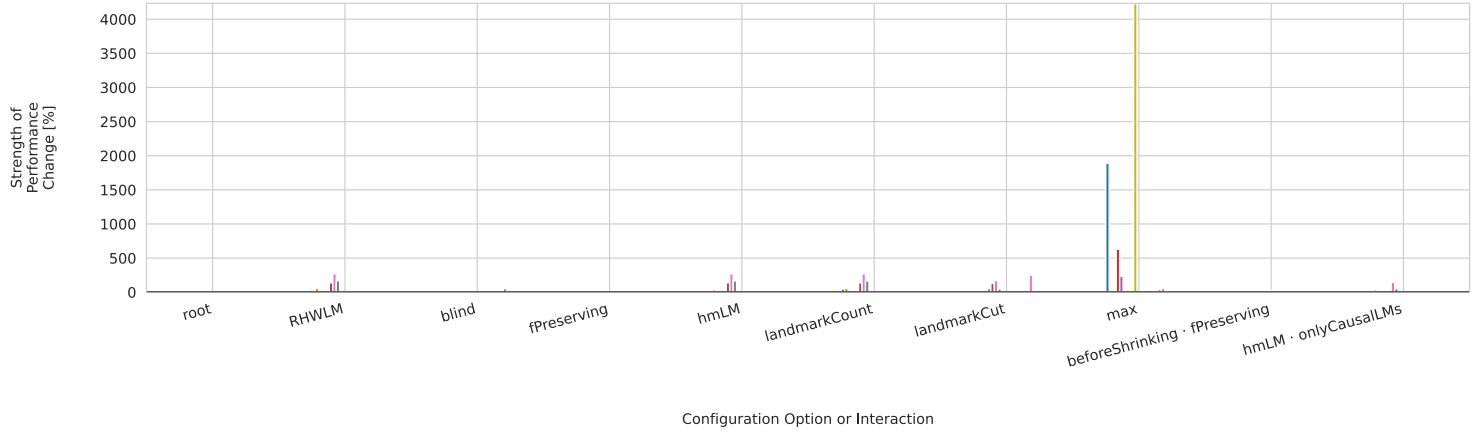


(b) v5.2.6 and v5.3.3alpha

Figure 5.9: Strength of performance changes for the decompression process of XZ comparing all workloads



(a) 2017_07 and 2018_01



(b) 2018_07 and 2019_01

Figure 5.10: Strength of excerpt of performance changes of Fast Downward for two release pairs comparing all workloads where each bar for a term corresponds to a workload and workloads are ordered by their ID

run-time of multiple minutes. The strength of 4 228.5% corresponds to an absolute change of 1 190.3*s* if the *max* configuration option is active. Across all workloads and releases, we observe that 156 out of 3 961 (3.9%) relevant performance changes have a strength of more than 100% and 12 (0.3%) even have a strength that exceeds 1 000%.

On average, we observe a strength of $\Delta = 22.8\%$ for performance changes across all workloads and releases. Looking at release pairs and workloads individually, we observe a range from 1.1% to 389.4% if aggregating the strength across all terms. Table 5.15 contains the aggregated results across workloads. We see that some releases introduce stronger performance changes than others. For example, between 2019_12 and 2020_06, performance changes are, in comparison, weak with an average strength of 9.9% while we see stronger changes of 43.7% between 2018_07 and 2019_01.

Table 5.15: Metrics related to Fast Downward for RQ1.2

Release 1	Release 2	$\Delta^{r_i,r_{i+1}}$
2016_07	2017_01	15.2%
2017_01	2017_07	17.7%
2017_07	2018_01	24.1%
2018_01	2018_07	37.3%
2018_07	2019_01	43.7%
2019_01	2019_06	24.8%
2019_06	2019_12	11.2%
2019_12	2020_06	9.0%

Generally, we again see a strong dependency on workloads and major differences between releases.

Summary

Looking at all real-world case studies, the average strength of changes in the influence of configuration options and interactions ranges from 1.0% for the compression phase of XZ to 22.8% for Fast Downward. Overall, across all real-world case studies, we observe a strength of 8.9%. Considering the individual results for workloads and release pairs and ignoring the cases where no performance changes were present, the strength ranges from 0.8% for the compression phase of XZ (v5.2.3 and v5.2.6 with the enwik9 workload) to 389.4% for Fast Downward (2018_07 and 2019_01 with workload9). Looking at the strength of relevant performance changes of individual terms, we even see a range from less than 0.1% (e. g., the *fPreserving* option for Fast Downward in workload1 and releases 2017_01 and 2017_07) to 4 228.5% (*max* option for Fast Downward in workload9 and releases 2018_07 and 2019_01).

Generally, we observe a wide range in the strength of performance changes between two releases. Some releases introduce very weak changes of less than 2% while others introduce considerably strong changes of up to 43.7%, aggregated across all workloads. Analogously to the first subquestion, we see a strong dependency on the workload that is used.

On average across all case studies, 9.1% of all terms in a performance-influence model are affected by a performance change that is stronger than 10%.

5.1.2 *Fraction of Performance Changes Confirmable by White-Box Approach*

In RQ2, we investigate the fraction of performance changes that are reported by the black-box analysis and can be confirmed by the white-box analysis.

> **RQ2:** What fraction of performance changes identified by our black-box approach can be confirmed by our white-box approach?

CompEnc

For CompEnc, the white-box analysis correctly detects the performance regression of the *Encryption* option introduced between the first two releases (v1 and v2). Furthermore, the white-box approach detects the performance regression between v2 and v3 related to the numeric *Iterations_val* option and, in contrast to the black-box approach, correctly attributes it to *Iterations_val*_1 and *Iterations_val*_2 with the expected strength. Note that we do not check *Iterations_val*_3 with the white-box approach according to our operationalization since we only consider the performance changes that the black-box approach reports. When comparing v3 and v4, we see that the white-box analysis correctly identifies the performance changes in *Compression* · *Iterations_val*_1 and *Compression* · *Iterations_val*_3 with the correct strength. Again, this behavior differs from the black-box approach which is able to detect the performance change, but attributes it to a different configuration option due to VIF countermeasures. For the same reason, the white-box analysis does not report the performance changes in the *root* and the *Compression* configuration options that the black-box analysis falsely reports due to the VIF countermeasures.

Table 5.16: Metrics related to CompEnc for RQ2

Release 1	Release 2	Workload	$c_w^{r_i,r_{i+1}}$	$s_w^{r_i,r_{i+1}}$	$err_w^{r_i,r_{i+1}}$
v1	v2	none	100.0%	100.0%	0.0%
v2	v3	none	66.7%	0.0%	0.0%
v3	v4	none	66.7%	50.0%	0.0%

Based on the observations above, we receive the values listed in Table 5.16 to answer RQ2. The fraction of confirmed performance changes ranges from $c_{none}^{v2,v3} = c_{none}^{v3,v4} = 66.7\%$ to $c_{none}^{v1,v2} = 100\%$, leading to an aggregated value of $C = 77.8\%$. We observe differences in the strength of the performance changes on the white-box side in comparison to the strengths detected on the black-box side. The similarity ranges from $s_{none}^{v2,v3} = 0\%$ to $s_{none}^{v1,v2} = 100\%$. On average, we have $S = 50\%$, i. e., half of the performance changes on the black-box side are similar to the performance changes on the white-box side in terms of strength. Figure 5.11 depicts the distribution of the absolute difference in the strength of performance changes

across all workloads and releases when comparing the performance changes reported by the black-box and white-box performance analysis, respectively.

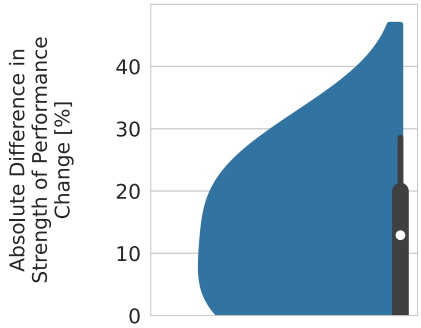


Figure 5.11: Distribution of the absolute difference in the strength of performance changes between black-box and white-box performance analysis for CompEnc

Since the white-box analysis correctly measures all configuration options and interactions, $err_w^{r_i,r_{i+1}}$ is 0% for all release pairs and, thus, $Err = 0\%$.

PicoSAT

For WORKLOAD1 and WORKLOAD2, the black-box analysis reports changes for *root* and nine proof- and tracing-related configuration options and interactions. For both workloads, the white-box performance analysis confirms five ($c_{\text{WORKLOAD1}}^{951,965} = c_{\text{WORKLOAD2}}^{951,965} = 50\%$) out of these ten performance changes. In particular, a closer look reveals that the white-box analysis confirms the change for *root* and all changes related to individual configuration options while it does not confirm any change related to interactions between configuration options. Four ($s_{\text{WORKLOAD1}}^{951,965} = s_{\text{WORKLOAD2}}^{951,965} = 80\%$) out of five changes are of similar strength according to our definition when comparing them to the changes that the black-box analysis reports. The only strength that deviates between black-box and white-box analysis is associated with the *root* option. For WORKLOAD2, the white-box analysis reports a weaker change for *root* (44.6%) in comparison to the black-box analysis (54.4%). In contrast, the white-box analysis reports a stronger change for *root* (145.8%) for WORKLOAD1 when comparing the changes to the black-box analysis (63.4%). For all terms associated with performance changes that the white-box analysis does not confirm, the white-box analysis reports a performance of 0 in both releases, i. e., it does not record any data for the interaction. Therefore, $err_w^{951,965} = 100\%$ for all workloads. For WORKLOAD4, the black-box performance analysis reports only a single change. This change is associated with the interaction $CompactTrace \cdot FileListingCoreVariables$ which creates a compact trace file and writes a list of all core variables to a file. The white-box analysis does not confirm this change ($c_{\text{WORKLOAD4}}^{951,965} = 0\%$). Consequently, $s_{\text{WORKLOAD4}}^{951,965} = 0\%$ and, for the same reason as for the other workloads, $err_{\text{WORKLOAD4}}^{951,965} = 100\%$.

In Table 5.17, we list an overview of all metrics related to PicoSAT for this research question. We see a range of 0% to 50% of confirmed performance changes with an average of $C = 33.3\%$. Looking at the average similarity score of $S = 53.3\%$, we note that most performance changes are of similar strength when comparing black-box and white-box analysis. All performance changes that the white-box analysis does not confirm relate to

interactions between configuration options and, as stated above, the white-box analysis does not track the performance of these interactions. Hence, $Err = 100\%$.

Table 5.17: Metrics related to PicoSAT for RQ2

Release 1	Release 2	Workload	$c_w^{r_i,r_{i+1}}$	$s_w^{r_i,r_{i+1}}$	$err_w^{r_i,r_{i+1}}$
951	965	WORKLOAD1	50.0%	80.0%	100.0%
951	965	WORKLOAD2	50.0%	80.0%	100.0%
951	965	WORKLOAD4	0.0%	0.0%	100.0%

Figure 5.12 depicts the distribution of the absolute difference in the strength of the performance changes when comparing black-box and white-box performance analysis. The difference ranges from 0.3% to 82.1%.

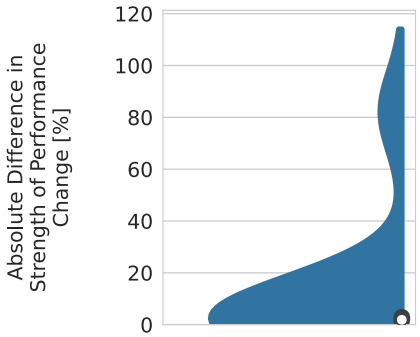


Figure 5.12: Distribution of the absolute difference in the strength of performance changes between black-box and white-box performance analysis for PicoSAT

In summary, we see that most confirmed performance changes are of similar strength, but, across all workloads, only 33.3% of all performance changes are confirmed. In all cases where the white-box analysis does not confirm the change, the affected terms are interactions between two configuration options and the white-box data does not contain any performance data for these interactions.

Clasp

For Clasp, the white-box performance analysis does not confirm any of the performance changes reported by the black-box performance analysis. The black-box performance analysis reported changes related to the enumeration mode and to the heuristic that is used. The white-box analysis is not able to track the performance of the configuration options and interactions related to the enumeration mode. In contrast, the white-box analysis tracks the performance of the heuristics, but does not report any relevant changes when comparing both releases. Table 5.18 contains the metrics we defined to answer this research question for Clasp. Since the white-box performance analysis is not able to confirm any performance change, we note that $c_w^{v3.3.4,v3.3.9} = 0\%$ for both workloads that exhibit a performance change on the black-box side and, thus, $C = 0\%$. Consequently, this yields that $s_w^{v3.3.4,v3.3.9} = 0\%$ for both workloads and, hence, $S = 0\%$. Since all changes for WORKLOAD2 are related to the

enumeration mode which the white-box analysis is not able to track, we have an error rate of $err^{v3.3.4,v3.3.9}_{\text{WORKLOAD2}} = 100\%$ for WORKLOAD2. For WORKLOAD5, one change reported by the black-box analysis is related to heuristics and one change is related to an interaction between the heuristic and the enumeration mode. Therefore, one out of two terms ($err^{v3.3.4,v3.3.9}_{\text{WORKLOAD5}} = 50\%$) can be tracked by the white-box analysis. On average, this yields $Err = 75\%$.

Table 5.18: Metrics related to CLASP for RQ2

Release 1	Release 2	Workload	$c_w^{r_i,r_{i+1}}$	$s_w^{r_i,r_{i+1}}$	$err_w^{r_i,r_{i+1}}$
v3.3.4	v3.3.9	WORKLOAD2	0.0%	0.0%	100.0%
v3.3.4	v3.3.9	WORKLOAD5	0.0%	0.0%	50.0%

Since no performance change can be confirmed by the white-box analysis, we refrain from plotting the differences in the strength of performance changes reported by the black-box and the white-box analysis.

XZ

As for the other research questions, we investigate the compression and the decompression process separately.

COMPRESSION. For the compression phase of XZ, the white-box performance analysis does not confirm any of the performance changes that the black-box analysis reports. Hence, $c_w^{r_i,r_{i+1}} = 0\%$ for all workloads and releases and, thus, $C = 0\%$. Similarly, this results in $S = 0\%$. Except for the *root* option, the white-box analysis does not report any performance data for the configuration options and interactions affected by the performance changes reported by the black-box analysis. The only workload and release combination that is affected by a performance change of *root* is DAVIS 2016 between v5.2.6 and v5.3.3alpha. Consequently, we see that $err_w^{r_i,r_{i+1}}$ ranges from $err^{v5.2.6,v5.3.3alpha}_{\text{DAVIS 2016}} = 83.3\%$ to $err_w^{r_i,r_{i+1}} = 100\%$ for all remaining workloads and releases. This yields an average of $Err = 95.8\%$.

In Table 5.19, we list the metrics we defined to answer this research question for the compression process of XZ.

Table 5.19: Metrics related to the compression process of XZ for RQ2

Release 1	Release 2	Workload	$c_w^{r_i,r_{i+1}}$	$s_w^{r_i,r_{i+1}}$	$err_w^{r_i,r_{i+1}}$
v5.2.3	v5.2.6	DAVIS 2016	0.0%	0.0%	100.0%
v5.2.3	v5.2.6	ENWIK9	0.0%	0.0%	100.0%
v5.2.6	v5.3.3alpha	DAVIS 2016	0.0%	0.0%	83.3%
v5.2.6	v5.3.3alpha	ENWIK9	0.0%	0.0%	100.0%

Due to the absence of confirmed performance changes, we omit the plots showing the differences in the strength of performance changes when comparing black-box and white-box analysis.

DECOMPRESSION. Analogously to the compression phase of XZ, the white-box analysis does not report any performance data for the configuration options and interactions affected by the performance changes reported by the black-box analysis, except for *root*. However, in contrast to the compression process, the fraction of confirmed performance changes ranges from $c^{v5.2.3,v5.2.6}_{\text{DAVIS 2016}} = c^{v5.2.6,v5.3.3alpha}_{\text{ENWIK9}} = 0\%$ to $c^{v5.2.3,v5.2.6}_{\text{ENWIK9}} = 33.3\%$. This yields $C^{v5.2.3,v5.2.6} = 16.7\%$ and $C^{v5.2.6,v5.3.3alpha} = 2.6\%$ and, thus, an aggregated value of $C = 9.6\%$. In particular, the white-box analysis confirms all performance changes related to the *root* option that the black-box analysis reports. All confirmed performance changes are of similar strength according to our definition when comparing the results of black-box and white-box analysis. Therefore, we see a range from $s^{v5.2.3,v5.2.6}_{\text{DAVIS 2016}} = s^{v5.2.6,v5.3.3alpha}_{\text{ENWIK9}} = 0\%$ to $s^{v5.2.3,v5.2.6}_{\text{ENWIK9}} = s^{v5.2.6,v5.3.3alpha}_{\text{DAVIS 2016}} = 100\%$. Hence, we note that that $S^{v5.2.3,v5.2.6} = S^{v5.2.6,v5.3.3alpha} = S = 50\%$. Due to the absence of performance data for all terms related to non-confirmed performance changes, we observe an error rate of $err^{r_i,r_{i+1}}_w = 100\%$ for all workload and release pairs and, thus, $Err = 100\%$.

Table 5.20 lists the metrics we defined to answer this research question for the decompression process of XZ.

Table 5.20: Metrics related to the decompression process of XZ for RQ2

Release 1	Release 2	Workload	$c^{r_i,r_{i+1}}_w$	$s^{r_i,r_{i+1}}_w$	$err^{r_i,r_{i+1}}_w$
v5.2.3	v5.2.6	DAVIS 2016	0.0%	0.0%	100.0%
v5.2.3	v5.2.6	ENWIK9	33.3%	100.0%	100.0%
v5.2.6	v5.3.3alpha	DAVIS 2016	5.3%	100.0%	100.0%
v5.2.6	v5.3.3alpha	ENWIK9	0.0%	0.0%	100.0%

Figure 5.13 depicts the distribution of the absolute difference in the strength of performance changes between black-box and white-box performance analysis for the decompression process. We note that all differences are small, ranging from 2.9% to 3.6%.

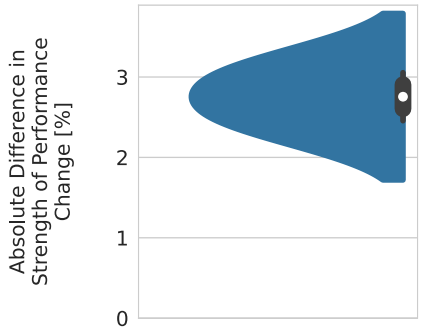


Figure 5.13: Distribution of the absolute difference in the strength of performance changes between black-box and white-box performance analysis for the decompression process of XZ

Generally, we observe that only few performance changes can be confirmed by the white-box analysis for XZ, but if they are confirmed, the changes are of similar strength. All changes that cannot be confirmed are caused by an absence of white-box performance data for the affected terms.

Fast Downward

Unfortunately, at the time of our measurements, a bug in the VaRA framework hindered a successful compilation of Fast Downward with the VaRA compiler. Since using the VaRA compiler is a pre-requisite for the white-box performance analysis, this prevented us from performing the white-box analysis for Fast Downward. Therefore, we omit the evaluation of this research question for this case study. We note that this bug has been fixed[1] in the meantime. However, due to the time constraints imposed by a thesis, we were not able to perform the missing measurements after the release of the fix.

Summary

Taking all real-world case studies into account, we observe an average fraction of 12.7% of performance changes reported by the black-box performance analysis that the white-box analysis confirms. We note few differences between the case studies, ranging from 0% for Clasp and the compression process of XZ to 33.3% for PicoSAT. Across all case studies, the strengths of performance changes reported by the white-box and the black-box analysis are similar in 26.1% of all cases. On average, in 90.1% of all cases, the missing confirmation of performance changes by the white-box performance analysis is caused by a lack of performance data associated with the affected terms. Looking at individual workload and release pairs, up to 50% (PicoSAT with workload1 and workload2 between 951 and 965) of all performance changes are confirmed by the white-box performance analysis whereas up to 80% of all changes are of similar strength in comparison to the black-box performance analysis.

5.1.3 *Identification of Configuration-Dependent Code Responsible for Performance Changes*

In RQ3, we investigate whether the white-box analysis is able to identify the root cause of a performance change, i. e., if it can localize the configuration-dependent code that changed.

> **RQ3:** In what fraction of cases is our white-box approach able to identify the configuration-dependent code responsible for a performance change identified by our black-box approach?

CompEnc

We observe that we can identify the configuration-dependent code that causes the performance changes for all performance changes confirmed by the white-box analysis. For CompEnc, these are the lines of code in which we changed the sleep time. Table 5.21 lists the results of the metrics we defined to answer the research question. Note that the white-box performance analysis does not confirm all changes reported by the black-box performance analysis. Hence, although we can identify the configuration-dependent code responsible for the performance changes confirmed by the white-box analysis, we can only identify the

1 "Introduces unlimited stack wrapper for phasar using experiments". Available online at https://github.com/se-sic/VaRA-Tool-Suite/pull/742; visited on February 15th, 2023.

responsible code locations for $L = 77.8\%$ of all changes reported by the black-box analysis. The fraction ranges from $l_{none}^{v2,v3} = l_{none}^{v3,v4} = 66.7\%$ to $l_{none}^{v1,v2} = 100\%$.

Table 5.21: Metrics related to CompEnc for RQ3

Release 1	Release 2	Workload	$l_w^{r_i,r_{i+1}}$
v1	v2	none	100.0%
v2	v3	none	66.7%
v3	v4	none	66.7%

PicoSAT

The white-box analysis confirms the performance changes related to *root* and to all individual configuration options. We identify changes in the code associated with *root* between both releases. Analogously, for all individual configuration options, we identify changes in at least one code region associated with the respective configuration option. Consequently, we are able to identify the configuration-dependent code that causes the performance changes reported by the black-box analysis in all cases in which the white-box analysis confirms the performance change. Hence, $l_w^{r_i,r_{i+1}} = c_w^{r_i,r_{i+1}}$ for all workloads and release pairs. On average, we identify the responsible code in $L = 33.3\%$ of all cases.

Table 5.22 lists the detailed evaluation results for this research question for PicoSAT.

Table 5.22: Metrics related to PicoSAT for RQ3

Release 1	Release 2	Workload	$l_w^{r_i,r_{i+1}}$
951	965	WORKLOAD1	50.0%
951	965	WORKLOAD2	50.0%
951	965	WORKLOAD4	0.0%

Clasp

Since the white-box analysis does not confirm any performance changes reported by the black-box performance analysis for Clasp, we are not able to identify the configuration-dependent code responsible for the performance changes. Hence, it is clear that $L = 0\%$. Therefore, we omit the detailed listing of metrics for all workloads and release pairs for this research question for Clasp.

XZ

As before, we separately evaluate the compression and the decompression process.

COMPRESSION. Due to the lack of confirmation of any performance changes reported by the black-box analysis, the white-box performance analysis is also not able to identify the configuration-dependent code responsible for the performance changes. Hence, $L = 0\%$. Consequently, we refrain from listing the metrics for all workload and release pairs for this research question for the compression process of XZ.

DECOMPRESSION. As outlined in the previous research question, the white-box performance analysis confirms all performance changes related to the *root* configuration option. In all cases, we identify changes in the code associated with *root* between the release pairs. Hence, we identify the configuration-dependent code responsible for a performance change in all cases in which the white-box performance analysis confirms the performance changes reported by the black-box analysis. Therefore, $l_w^{r_i,r_{i+1}} = c_w^{r_i,r_{i+1}}$ for all workloads and release pairs. On average, this yields $L^{v5.2.3,v5.2.6} = 16.7\%$ and $L^{v5.2.6,v5.3.3alpha} = 2.6\%$ and, thus, $L = 9.6\%$.

In Table 5.23, we list the results for this research question for the decompression process of XZ.

Table 5.23: Metrics related to the decompression process of XZ for RQ3

Release 1	Release 2	Workload	$l_w^{r_i,r_{i+1}}$
v5.2.3	v5.2.6	DAVIS 2016	0.0%
v5.2.3	v5.2.6	ENWIK9	33.3%
v5.2.6	v5.3.3alpha	DAVIS 2016	5.3%
v5.2.6	v5.3.3alpha	ENWIK9	0%

Fast Downward

For the same reason as for the previous research question, we have to omit the evaluation of this research question for FAST DOWNWARD.

Summary

Across all real-world case studies, we can identify the configuration-dependent code responsible for a performance change reported by the black-box analysis in 12.7% of all cases. In all other cases, the white-box analysis does not confirm the performance changes (see RQ2) and, consequently, is not able to identify the configuration-dependent code.

5.2 DISCUSSION

In this section, we discuss noteworthy insights that we obtained during the evaluation of our research questions.

5.2.1 *Performance Changes Between Consecutive Releases for Different Workloads*

In the following paragraphs, we discuss some insights for each case study related to both sub-questions of the first research question.

COMPENC. For our artificial case study, the black-box performance analysis is able to detect all performance changes that we introduced. However, some changes are not attributed to the correct term due to multicollinearity which cannot fully be avoided. While this still delivers some helpful insights, the reported changes must be interpreted carefully to avoid drawing wrong conclusions due to multicollinearity.

PICOSAT. For PicoSAT, we witness a very strong dependency of performance changes on the workload. If we look at WORKLOAD1, 90% of all configuration options and interactions participate in a performance change. In contrast, not a single option changes for WORKLOAD3. The major difference between these two workloads is their domain. WORKLOAD1 models the feasibility of the antibandwidth problem which is a max-min optimization problem on graphs. WORKLOAD3 models a problem in the planning domain. Furthermore, the average run-time is considerably different for both workloads. While WORKLOAD1 has an average run-time of 171.4*s*, WORKLOAD3 runs only 46.2*s*, on average. We also note that the performance changes do not necessarily align with the expectations. For example, WORKLOAD3 does not exhibit any performance changes, but performance changes are present for WORKLOAD4. Both workloads are different instances of the same problem domain. Therefore, we would have expected a very similar behavior for these two workloads. Yet, we also note that only a single term changes for WORKLOAD4 and that the change is minor with a strength of only 2.5%. Hence, it is questionable whether there is an actual performance change or if the change is an artifact of other circumstances, e. g., the model error that is introduced by the linear regression. Although we make sure to filter out changes introduced by measurement noise as described in Chapter 4, such cases cannot be fully excluded. We also observe that WORKLOAD1 and WORKLOAD2 show a very similar behavior although they also stem from different domains since WORKLOAD2 models a traffic situation to be solved.

CLASP. Inspecting the performance-influence models for Clasp in general, we note that different workloads profit from different heuristics. For example, workloads from the planning domain (WORKLOAD2, WORKLOAD3, WORKLOAD5, and WORKLOAD6) seem to profit from the VMTF heuristic whereas WORKLOAD1 (antibandwith problem) seems to perform poorly with that heuristic. Similar observations apply to the enumeration mode, although we do not observe a clear mapping to problem domains here. Similar to PicoSAT, we also observe a dependency of performance changes on the workload. Only two out of seven workloads exhibit any performance change. Both workloads are from the planning domain. Workloads from other domains do not exhibit any performance changes. However, we again note that even similar workloads do not necessarily behave similarly. For example, WORKLOAD5 and WORKLOAD6 are different instances on the same problem space, but a performance change is only present for WORKLOAD5. We also note that the general structure of their performance-influence models is almost identical. The changes for WORKLOAD5 are again small with a strength of only 1.2%. Therefore, the performance changes could again

be an artifact of, e. g., the model error. The changes that affect WORKLOAD2 are stronger with 9.2% and, thus, more likely to actually originate from code changes.

XZ. Since there are only very few (7.6%) and weak (1.0%) changes for the compression process of XZ, we focus on the decompression phase in our discussion. For the decompression process, we observe very few (10.7%) and weak (1.4%) changes between v5.2.3 and v5.2.6, while we observe more (78.6%) and stronger (29.2%) changes between v5.2.6 and v5.3.3alpha. Eleven out of the 15 strongest changes between v5.2.6 and v5.3.3alpha are related to the compression format or interactions between the compression format and the number of threads. We need to keep in mind that due to the multicollinearity countermeasures, the influence of the default value of the number of threads in interaction with the compression format is part of the influence of the compression format. In our case, this means that *xz* and *auto* are implicitly equivalent to $xz \cdot Threads\_val\_4$ and $auto \cdot Threads\_val\_4$, respectively. Therefore, the black-box performance analysis hints at threading-related changes between v5.2.6 and v5.3.3alpha. An investigation of the change log between both versions reveals that v5.3.3alpha introduced multi-threaded decompression for the *xz* format which was not supported before, confirming the hint. An investigation of the manual of *xz* also revealed that in our measurement setup, the *auto* configuration option is implicitly equivalent to the *xz* configuration option which is why we also observe a similar behavior for both options. Regarding the dimension of workloads, both workloads exhibit a similar behavior in terms of performance changes although they are different types of workloads. ENWIK9 consists 1 GB of English text whereas DAVIS 2016 contains 1.8 GB of image data. Another interesting observation we make is that for the compression process, DAVIS 2016 has a better average performance (254.1*s*) than ENWIK9 (296.7*s*). In contrast, ENWIK9 performs better (14.8*s*), on average, than DAVIS 2016 (23.3*s*) when decompressing the data. This might also be related to the workload type.

FAST DOWNWARD. In Chapter 4, we divided the workloads for FAST DOWNWARD into different groups, depending on the problem type. We note that within each group, the general structure of the performance-influence models is mostly consistent. For FAST DOWNWARD, we make similar observations as for CLASP. Different workloads profit from different heuristics. For example, the run-time of the AGRICOLA group heavily increases by more than 25 000*s* when using the h^{max} heuristic. In contrast, the h^{max} heuristic has much less influence (less than 1 200*s*) for workloads from the HIKING_PTESTING group. Contrary, the *landmarkCut* heuristic has an influence of up to 45 000*s* for the HIKING_PTESTING group, but of less than 1*s* for the AGRICOLA group. We make similar observations for all workload groups with different types of heuristics. Nevertheless, we notice that also within a workload group, performance changes are of different strengths. For example, between releases 2016_07 and 2017_01, the influence of the *landmarkCut* heuristic has changes of strength 566.2% for WORKLOAD17, but only 21.9% and 14.0% for WORKLOAD15 and WORKLOAD16, respectively, although all three belong to the TRANSPORT group. Generally, we note that is hard to predict performance changes, even within the same workload group.

SUMMARY. Across all case studies, we note that performance changes are heavily dependent on the workload. Different types of workloads might profit from different configuration

options such as, e. g., heuristics. However, even for workloads of the same type, it is not always possible to predict performance changes. While they often exhibit similar behavior, we also note some outliers for each case study. The reasons for that are not clear based on the data provided by the black-box performance analysis.

Therefore, we formulate the answer to our first research question as follows.

Answer to RQ1.1: Every release of every case study exhibits at least one change in the influence of a configuration option or interaction on the performance for at least one workload. In the majority of cases, the influence of multiple configuration options or interactions changes. The workload has a considerable impact on the presence of performance changes.

Answer to RQ1.2: The strength of performance changes vastly varies between case studies and releases. The average strength across all workloads and releases does not exceed 9%, but ranges from 1.0% to 22.8%. The strength of performance changes is strongly dependent on the workload. Most performance changes are of moderate strength with only 9.1% of all performance changes exceeding a strength of 10%.

5.2.2 *Fraction of Performance Changes Confirmable by White-Box Approach*

For each case study, we discuss insights about the second research question in the following paragraphs.

COMPENC. For CompEnc, the white-box performance analysis detects all performance changes that actually exist. As we already noted in the discussion of the first research question, the black-box performance analysis attributes some changes to the wrong configuration options and interactions due to multicollinearity in the performance-influence models. For the same reason, the strength of the reported performance changes deviates from the strength of the actual performance changes. The white-box performance data for CompEnc does not suffer from these flaws. It reports all performance changes correctly and with the correct strength. The comparison with the black-box data stresses the necessity to be careful when interpreting the black-box data since multicollinearity cannot fully be avoided. We also note that lower values of $c_w^{r_i,r_{i+1}}$ do not necessarily imply a shortcoming of the white-box analysis, but merely indicate a deviation from the black-box results. This, in turn, hints at a need for a deeper manual analysis since we cannot automatically determine which, if any, of both approaches reports the correct results.

Generally, we note that the white-box analysis delivers reliable results for this case study.

PICOSAT. For PicoSAT, the white-box performance analysis confirms all performance changes associated with individual configuration options, but does not confirm any performance change related to an interaction between configuration options. An explanation for this could be the difference in the semantics of interactions between black-box and white-box performance analysis. In the black-box performance analysis, the influence of

an interaction describes all changes in the performance that occur when all configuration options involved in the interaction are enabled. In contrast, the white-box analysis requires explicit interaction or nesting of the configuration options in the code to count it as an interaction. For example, enabling an additional configuration option may increase the data to be processed by the application and, thus, indirectly influence the performance of other configuration options. If this dependency is not possible to detect by a static code analysis, the white-box analysis does not consider this dependency as an interaction. The black-box performance analysis detects this as an interaction. A manual investigation of the code of PicoSAT reveals that all proof- and trace-related configuration options share a part of the code. In particular, this part of the code is only executed once, independent of how many of these configuration options are active. Recalling the performance-influence models from the first research question, we see that the coefficients associated with all interactions of two configuration options are negative. Due to the shared code, the influence of both configuration options involved in the interaction does not fully add up. Instead, we need to deduct the influence of the shared code that is included in the influence of both configuration options, but that only runs once if both are enabled. This results in the negative coefficients of the black-box performance-influence models. This also explains why we do not see the performance change in these configuration option interactions in our white-box data since our definition does not consider these kinds of interactions. Therefore, the performance of these interactions is not tracked in the white-box data and, thus, we cannot detect the performance changes associated with these interactions.

We observe similar strengths of performance changes when comparing black-box and white-box data for all configuration options except *root* which reinforces the black-box results. The reason for the difference in the strength for *root* is not clear, but could result from the measurement overhead introduced by the white-box performance analysis.

CLASP. In contrast to the black-box analysis, the white-box analysis does not report any performance changes for Clasp. We see two explanations for this. First, the white-box analysis is currently not able to track the performance of the code depending on the enumeration mode. This is due to the fact that the enumeration mode in Clasp is encoded by the usage of an `enum` which is, as of this writing, not yet fully supported[2] by the VaRA framework that we use. Therefore, the only performance change in a configuration option or interaction that the white-box analysis could have detected is associated with *heuristicVmtf*. The white-box performance analysis is able to measure the performance of *heuristicVmtf*, but it does not report a relevant change for it. However, as we already pointed out in the discussion of the first research question, the performance change for *heuristicVmtf* reported by the black-box performance analysis is very weak and only barely exceeds the relevance threshold. Therefore, it is questionable whether the performance change actually exists or if it is merely an artifact of measurement noise or the error introduced by linear regression. Another factor that supports this assumption is that a manual investigation reveals that the heuristics-related code has not been changed between both releases and,

2 This is a known issue in VaRA's issue queue: "Support Feature enums with more precision". Available online at https://github.com/se-sic/VaRA/issues/918; visited on February 9th, 2023.

additionally, the change log of CLASP[3] also does not mention any heuristic-related changes. Hence, the absence of a performance change in the white-box data may be reasonable and hints at a falsely reported performance change. While we cannot reliably determine which approach is right here, the integrated approach still provides additional insights by indicating performance changes that require further investigation.

XZ. For both, the compression and the decompression process, the white-box analysis does not confirm any performance changes reported by the black-box analysis, except for changes related to *root*. In more than 90% of all cases, this was due to the absence of white-box performance data associated with the corresponding configuration option or interaction. A closer manual investigation reveals multiple reasons for this. Considering the compression and the decompression process at the same time, 26 out of 68 terms affected by performance changes across all workloads and release pairs involve the compression level (*Preset_val*). XZ stores multiple configuration options in a single integer and retrieves the individual configuration options by using bit masks. The compression level is one of these configuration options. This means that it would be necessary to trace the influence of configuration options on a bit level. However, the white-box analysis tool VARA that we use does not support this. Hence, VARA is not able to identify the code that depends on the compression level. The same explanation applies to the *XtrmCPU* configuration option which participates in seven performance changes. The compression format, which participates in 33 performance changes, is encoded by using an `enum`. As already outlined in the discussion of CLASP, this is currently not supported by VARA.

While the white-box analysis confirms the changes related to *root*, we need to consider that *root* contains the influence of all configuration options that the white-box analysis is not able to track. Hence, the performance associated with *root* must be treated with care when comparing it to the influence of *root* in the black-box data. Nevertheless, the changes in *root* are similar to the changes reported by the black-box performance analysis.

FAST DOWNWARD. As outlined in the description of the results, it was not possible to gather white-box performance analysis data for FAST DOWNWARD. While this prevents us from gaining additional insights, it highlights a weakness of the white-box performance analysis. A white-box approach cannot be universally applied to all software systems. While much effort is put into making the white-box analysis as generally applicable as possible, examples like this show that potential pitfalls still exist. This is a disadvantage in comparison to the black-box performance analysis that only requires a working binary compiled with an arbitrary compiler to analyze the performance.

SUMMARY Under optimal conditions, the white-box analysis confirms all performance changes reported by the black-box analysis that actually exist and points out weaknesses in the data reported by the black-box analysis. For one real-world case study (PICOSAT) the white-box analysis confirms half of the performance changes. The missing confirmation for the other half is due to the difference between black-box and white-box performance analysis in the semantics of interactions. For CLASP, the white-box analysis contradicts a

3 clasp: Changes. Available online at https://github.com/potassco/clasp/blob/master/CHANGES; visited on February 16th, 2023.

performance change reported by the black-box analysis. The contradiction is backed up by a manual investigation of the related code and the change log. However, for the other real-world case studies, we also observe some flaws in the white-box performance analysis. In some cases, the white-box analysis is not able to track the performance impact of features, especially if they use specific programming constructs like bit masks or `enums`. Generally, this emphasizes the inherent complexity of white-box performance analysis and shows that a universal approach is hard to implement. Nevertheless, we see multiple instances in which the white-box analysis is able to provide additional information in comparison to a pure black-box approach.

We answer RQ2 as follows.

> **Answer to RQ2:** The minority (12.7%) of performance changes reported by the black-box performance analysis can be confirmed by the white-box performance analysis. In the majority of cases (90.1%), this is due to the absence of white-box performance data related to the affected terms.

5.2.3 *Identification of Configuration-Dependent Code Responsible for Performance Changes*

In the following paragraphs, we report insights about the identification of configuration-dependent code responsible for performance changes.

COMPENC. For CompEnc, the white-box performance analysis correctly identifies the configuration-dependent code responsible for each confirmed performance change. Thereby, we gain additional insights in comparison to a pure black-box performance analysis which only tells us about the presence, but not the cause of a performance change. With the additional information, we are able to investigate the performance changes more closely to, ideally, better understand their cause and impact.

PICOSAT. The white-box analysis identifies the configuration-dependent code for all confirmed performance changes for PicoSAT. We note two things. First, one of the confirmed performance changes is associated with the *root* configuration option. Since *root* includes the impact of every piece of code that the white-box analysis does not associate to a configuration option, many lines of code are associated with *root*. The performance of all of these lines is tracked in a single region. Therefore, in the case of *root*, the identification of the responsible code region is only of limited use since we do not know which exact changes in the code caused the performance changes. The second observation we make is that the code changes in the code regions associated with the confirmed configuration options are seemingly minor. In particular, an additional manager object is added as an argument to all method calls. Hence, it is questionable whether these changes are indeed responsible for the performance changes. It seems likely that, instead, an implicit data flow dependency is responsible for the performance changes since the proof- and tracing-related code depends on the data provided by the SAT solving process itself. Nevertheless, the

code locations identified by the white-box performance analysis provide pointers for further manual investigation.

CLASP. Since the white-box performance analysis is not able to confirm any changes reported by the black-box performance analysis, we do not gain any information about the configuration-dependent code responsible for performance changes. Therefore, we omit a further discussion of this research question for CLASP.

XZ. All performance changes that the white-box analysis confirms are associated with the *root* configuration option. For the same reasoning as for PICOSAT, the identification of the configuration-dependent code is only of limited use in this case since we do not have fine-granular information about the performance of *root*.

FAST DOWNWARD. Due to the absence of white-box performance data for FAST DOWNWARD, we omit a further discussion for this case study.

SUMMARY. Similar to the second research question, we gain useful additional insights from the white-box performance analysis under optimal conditions. However, the insights are limited when considering real-world case studies. While we are able to identify the configuration-dependent code responsible for the performance change in all cases in which the white-box performance analysis confirms the black-box results, we need to keep two things in mind. First, some changes are associated with the *root* option for which the location data is not fine-granular. Additionally, we cannot tell if the code changes are actually responsible for the performance change. Our way of identifying configuration-dependent code responsible for a performance change only implies correlation, but not necessarily causality. Nevertheless, the identified code locations provide pointers for further manual investigation.

We formulate the answer to RQ3 as follows.

Answer to RQ3: In all cases in which the white-box performance analysis confirms the performance changes reported by the black-box performance analysis, we are able to identify the configuration-dependent code responsible for the performance change. Overall, we can identify the responsible code in 12.7% of all cases.

5.3 THREATS TO VALIDITY

In this section, we discuss possible threats to the validity of our results. Furthermore, we describe how we mitigate these threats. First, we discuss threats to the internal validity of our results, i. e., potential sources of error in our measurement data and evaluation. Afterwards, we discuss the external validity, i. e., the generalizability of our results.

5.3.1 *Internal Validity*

Reasons for measurement noise are manifold and can be caused by software as well as hardware [19]. To mitigate these effects, for each subject software system under investigation, we used identical hardware for all of our measurements related to this system. Additionally, we kept the software environment for all measurements stable by using an identical and minimal installation of Debian 11 on all machines. For the *Xeon* machines, we made sure to use NUMA control such that the case studies are executed only on one of both CPUs and that they exclusively use the main memory associated with that CPU. This way, we eliminate potential sources of randomness. By using a workload manager to distribute our measurements to machines, we ensured that each measurement is executed exclusively on a machine without the interference of any other software. Before each measurement, we executed a warm-up phase for the CPU by operating it at full capacity for five seconds. This way, we mitigated the probability of the current state of the CPU influencing our measurements. This is of particular importance for measurements that terminate within a very short time frame. To ensure that our precautions actually lead to reliable results, we repeated each measurement five times and verified that the relative standard deviation of the measurements is below 5%. If this was not the case, we repeated the measurements until the relative standard deviation dropped below 5%.

Another possible source of inaccuracy of our results is caused by the fact that we used a pairwise sampling approach for three of our case studies to generate configurations for our measurements. Using a different sampling approach could yield different results. However, due to the immense size of the configuration space of some software systems and the limited time frame of a thesis, we had to opt for a sampling approach. We decided to use a pairwise sampling approach since related work [9, 16] shows that covering interactions of up to two configuration options covers most of the variability of the performance of a software system.

We only examined configuration options that were present in all releases that we considered for the software system under investigation. This way, we ensure that we can reliably compare the performance-influence models of our approach in the black-box part of our approach, leading to a higher internal validity. We manually ensured that no newly introduced configuration option was enabled by default so that we ruled out the possibility of such a new configuration option indirectly influencing our models.

The decision on using SPL Conqueror to create performance-influence models poses another threat to the internal validity of our results. As described in Chapter 2, SPL Conqueror implements multiple linear regression with feature-forward selection to generate performance-influence models. Using a different approach could lead to completely different models which would possibly yield different results. However, using multiple linear regression ensures that, in comparison to, e. g., neural networks, the models can be easily interpreted by humans. Furthermore, our linear models guarantee that we can compare them across releases and workloads due to their structure staying constant and that we can also interpret their changes. Additionally, multiple publications have successfully applied this approach in practice. For example, Grebhahn et al. [9] have learned performance-influence models for a software system by using this approach. They handed the performance-influence models to domain experts for verification and the experts reported good results. Furthermore, multiple publications [15, 16, 24], show that this approach yields performance-

influence models that cover the majority of the influence of configuration options and interactions on the performance with high accuracy.

Another problem related to the usage of multiple linear regression with feature-forward selection is multicollinearity as outlined in Chapter 2. Multicollinearity hinders the comparison of performance-influence models. We applied a VIF analysis to detect and eliminate terms from our performance-influence models that cause perfect multicollinearity. By removing these terms from the models, we reduce multicollinearity and, hence, improve the comparability of performance-influence models across releases and workloads.

The overhead introduced by the instrumentation of the code for the white-box analysis poses another threat to the internal validity of our results. While we perform very little computation at run-time, adding overhead cannot fully be avoided. Furthermore, we decided to aggregate the performance data for each feature region and region interaction at run-time. Otherwise, we would have produced up to 200 GB of measurement data per execution, leading to 1 TB of measurement data per configuration due to the five repetitions we perform. This would have made an evaluation of the measurement data infeasible on a large scale. We had to make a trade-off between keeping the overhead small and ensuring the feasibility of evaluating the data. On average, we observe an overhead of 38.8% when comparing the run-time of the instrumented code with the original code. However, to answer our research questions, we never directly compare the performance of instrumented code with original code. We identify performance changes within the instrumented code and the original code, respectively, and only compare the relative values. This way, we mitigate the influence of the overhead. Nevertheless, we cannot fully exclude that the overhead has an influence on our results.

Another threat to the validity of our results is posed by our decision on using the VaRA framework for our white-box performance analysis. As we have seen in the evaluation of our results, the VaRA framework does not work flawlessly under all circumstances. However, few reliable white-box performance analysis tools are available and prior publications [22] have successfully used the VaRA framework for similar purposes. Therefore, we opted for this framework.

In addition to the threats stated above, the metrics that we used pose a threat to the internal validity of our results. Using different metrics could have yielded different results. However, due to the absence of a ground truth, except for our artificial case study, we needed to establish an automated way to detect performance changes. Related publications [15] utilize similar metrics. For our white-box performance analysis, we compared the results against the results of the black-box performance analysis. Hence, we use the black-box performance analysis as a baseline for the white-box performance analysis which is in line with the general integrated approach that we explore. Furthermore, we manually investigated the performance changes and examined whether they are related to a code change in our last research question, increasing the plausibility of the results.

5.3.2 *External Validity*

To achieve a high generalizability of our results, we investigated five case studies in our thesis. In total, we examined one artificial case study and four real-world case studies from three different domains, namely compression, planning, and solving satisfiability problems.

All of the subject systems are written in either C or C++ which can be an influential factor in a white-box analysis. However, in its current state, the VaRA framework that we used does not support any other programming languages which limits the generalizability of our results to software written in other languages.

For each system, we measured 91 to 713 configurations with two to 20 workloads and two to nine releases. By using releases that have been developed over many years, we tried to cover as many performance changes as possible. Basing our selection of workloads on results from related work such as [5] or on the general properties of the workloads, we aimed at identifying a broad range of performance changes that depend on workloads. Note that one of the goals of our work was to identify flaws in the black-box and the white-box approach. Therefore, we decided to cover a wide range of subject systems. Since the white-box performance analysis needs to be carefully adapted and prepared for each subject system and each release of the system, we sacrificed some external validity with regard to the range of releases and workloads covered for some external validity with regard to the number of subject systems.

Our work aimed at a first exploration of the integration of white-box with black-box performance analysis which, to the best of our knowledge, has not yet been covered by prior work. Considering the limited time scope of a thesis, we decided to not focus on covering too many subjects in each dimension and rather aimed at increasing the internal validity of our results.

6 CONCLUDING REMARKS

In this chapter, we summarize the insights that the results of this thesis provide and, finally, we outline potential points for further improvements or investigations that subsequent research could be based on.

6.1 CONCLUSION

In most cases, performance analysis approaches view the subject system as a black box. However, there also exist approaches that conduct a performance analysis in a white-box fashion. In this thesis, we implemented an approach that integrates both approaches. In an empirical study, we first conducted a black-box performance analysis to identify potentially interesting configuration options and interactions. Based on this data, we determined the frequency and strength of performance changes between releases with multiple workloads. Then, we used the VaRA framework to conduct a white-box analysis to more closely investigate the configuration options and interactions that are affected by a performance change according to the black-box performance analysis. We examined the fraction of cases in which the white-box performance analysis confirms the black-box results. Furthermore, we investigated whether we are able to retrieve information about the configuration-dependent code responsible for a performance change from the white-box analysis data. We used an artificial case study to demonstrate the general feasibility of our approach and evaluated four real-world case studies to investigate the applicability of an integrated approach to real-world case studies.

The results of our black-box performance analysis indicate that changes in the influence of individual configuration options or configuration options are quite frequent. Often, performance changes are prevalent only for some workloads or affect different configuration options and interactions depending on the workload. Under optimal circumstances, the white-box analysis confirms the performance changes reported by the black-box performance analysis and is also able to uncover some shortcomings of the black-box analysis. For example, the performance changes reported by the black-box analysis are sometimes attributed to the wrong configuration options and interactions due to multicollinearity that cannot fully be avoided. Moreover, in the evaluation of real-world case studies, we uncovered some weaknesses of the white-box analysis. For instance, the white-box analysis is not able to track the performance of all configuration options. This is due to, e. g., unsupported programming language constructs or to bit-level encoding of configuration options. In the real-world case studies, this affected a large portion of the configuration options that exhibited a performance change according to the black-box performance analysis. Therefore, the white-box analysis confirmed only few performance changes. Consequently, the configuration-dependent code responsible for the performance change was also only located in few instances. Nevertheless, in many cases, the results of the white-box performance

analysis provide insightful pointers that indicate a possible starting point for further manual investigation.

Overall, our results show that an integrated approach delivers reliable and insightful results under optimal conditions, but requires further work to become an "out of the box" solution for real-world scenarios.

6.2 FUTURE WORK

During our investigations, we discovered shortcomings of both, the black-box and the white-box performance analysis. These shortcomings also lead to potential problems in our integrated approach. In the following, we outline possibilities for future work based on the identified flaws and their implications for our integrated approach.

For the black-box performance analysis, we note that multicollinearity can still hinder the interpretation of the results despite countermeasures being in place. Future research could further investigate the general prevalence of this problem and evaluate different approaches to mitigate this flaw. Further research can also be conducted on tackling this problem in the integrated approach. Currently, we generate configurations to investigate with the white-box performance analysis based on the results reported by the black-box analysis. Thus, if the black-box analysis attributes a performance change to the wrong configuration option or interaction, we do not even consider the actually affected configuration option or interaction in the white-box analysis. Approaches should be investigated that mitigate this problem without requiring to measure all configurations with a white-box approach.

Similarly, we observed differences in the semantics of interactions when comparing black-box and white-box performance analysis that resulted in unexpected results in some cases. Further research could be conducted to develop a solution that unifies the semantics between both approaches.

Since configuration-aware white-box performance analysis is a relatively new field of study, only few research has been conducted and applications in real-world situations are still rare. As a consequence, problems and bugs are still prevalent in its implementation. Finding one-size-fits-all solutions for these problems is a difficult task due to the inherent complexity of real-world software systems. Hence, future work could investigate and evaluate different approaches to mitigate the existing problems. For example, bit-level tracking of configuration options is still an open problem that hinders proper measuring of some configuration options. Similarly, using `enums` to model configuration options is a common pattern in software systems, but not yet supported. Adding support for this could be another possibility for further extensions of the white-box performance analysis.

While the white-box analysis does not support all programming language constructs, it still supports a wide range of them. In the case studies we investigated, many performance changes that we observed were related to configuration options that currently cannot be reliably tracked by the white-box analysis. However, our case studies are only very few instances of a vast number of software systems. The shortcomings we encountered might not apply to other case studies. It could be insightful to replicate our study with additional software systems.

Finally, it could be interesting to re-evaluate the case studies that we investigated after further improvements for the black-box and the white-box performance analysis are in place.

A

APPENDIX

A.1 CONTENT OF ACCOMPANYING ZIP FILE

The digital version of this thesis is accompanied by a supplementary ZIP file. In the following, we outline the content of this ZIP file.

Figure A.1 depicts the directory structure of the supplementary ZIP file. The Scripts directory contains all scripts that we used for the evaluation of our research questions and are based on the scripts used by Kaltenecker et al. [15]. The Data directory contains one directory per case study. Each case study directory contains three directories: Performance, Results, and Plots. The Performance directory contains all performance data that we gathered for the respective case study. In particular, we include the feature model, all performance-influence models (black-box) for the case study, the feature and region performance data (white-box) for the case study, and the configurations that we used for our black-box and the white-box measurements, respectively. The Results directory contains all computed metrics that we defined for each research question. In the Plots directory, we include all plots that we generated for the respective case studies, including the plots we omitted in the written composition of our evaluation.

Our contributions to the VaRA framework reside in the GitHub repositories of the VaRA-Tool-Suite[1], VaRA[2] and the vara-llvm-project[3].

1 GitHub repository of the VaRA-Tool-Suite. Available online at https://github.com/se-sic/VaRA-Tool-Suite; visited on February 16th, 2023.

2 GitHub repository of VaRA. Available online at https://github.com/se-sic/VaRA; visited on February 16th, 2023.

3 GitHub repository of the vara-llvm-project. Available online at https://github.com/se-sic/vara-llvm-project; visited on February 16th, 2023.

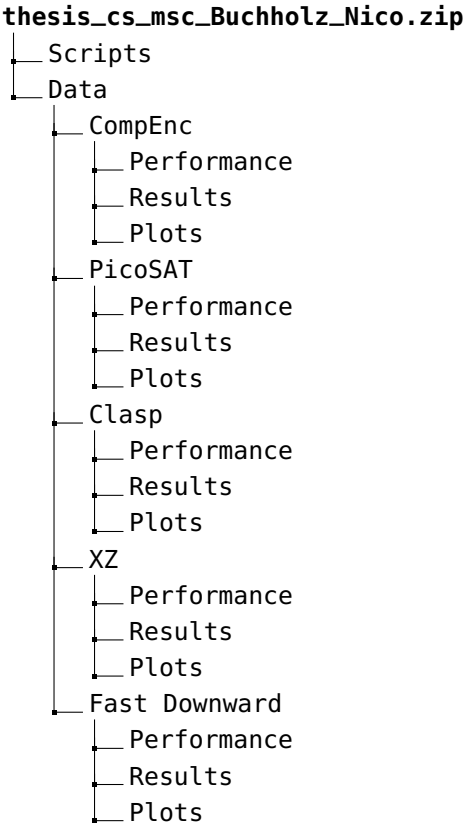


Figure A.1: Directory structure of the supplementary ZIP file

BIBLIOGRAPHY

[1] Juan Pablo Sandoval Alcocer, Alexandre Bergel, Stéphane Ducasse, and Marcus Denker. "Performance Evolution Blueprint: Understanding the Impact of Software Evolution on Performance." In: *Proceedings of the Working Conference on Software Visualization, VISSOFT*. IEEE Computer Society, 2013, pp. 1–9.

[2] Vlasta Bahovec. "Multicollinearity." In: *International Encyclopedia of Statistical Science*. Springer, 2011, pp. 869–870.

[3] Ron Burback. "Software Engineering Methodology: The WaterSluice." Dissertation. Stanford University, 1998.

[4] 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 Computer Society, 2017, pp. 341–352.

[5] Alexander Dincher. "The Impact of Workloads on Performance of Configurable Software Systems." Bachelor's Thesis. Saarland University, 2021.

[6] 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 Computer Society, 2020, pp. 684–696.

[7] Katalin Fazekas, Markus Sinnl, Armin Biere, and Sophie Parragh. "Duplex Encoding of Antibandwidth Feasibility Formulas Submitted to the SAT Competition 2020." In: *Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions*. Department of Computer Science, University of Helsinki, 2020, pp. 81–82.

[8] Nils Froleyks. "Planning Track Benchmarks." In: *Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions*. Department of Computer Science, University of Helsinki, 2020, p. 64.

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

[10] Huong Ha and Hongyu Zhang. "DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network." In: *Proceedings of the International Conference on Software Engineering, ICSE*. IEEE Computer Society, 2019, pp. 1095–1106.

[11] 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*. ACM, 2016, 23:1–23:10.

[12] Malte Helmert and Carmel Domshlak. "Landmarks, Critical Paths and Abstractions: What's the Difference Anyway?" In: *Proceedings of the International Conference on Automated Planning and Scheduling, ICAPS*. AAAI, 2009, pp. 162–169.

[13] Malte Helmert, Patrik Haslum, and Jörg Hoffmann. "Flexible Abstraction Heuristics for Optimal Sequential Planning." In: *Proceedings of the International Conference on Automated Planning and Scheduling, ICAPS*. AAAI, 2007, pp. 176–183.

[14] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. "Linear Regression." In: *An Introduction to Statistical Learning with Applications in R*. Springer, 2021, pp. 59–128.

[15] Christian Kaltenecker, Alexander Grebhahn, Stefan Mühlbauer, Norbert Siegmund, and Sven Apel. *Performance Evolution of Configurable Software Systems: An Empirical Study*. Submitted to: *Empirical Software Engineering, EMSE*. 2023.

[16] 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, SoSyM* 18.3 (2019), pp. 2265–2283.

[17] Ryan Lawrence. "Efficient Algorithms for Clause-Learning SAT Solvers." Master's Thesis. Simon Fraser University, 2002.

[18] 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 International Conference on Software Engineering, ICSE*. IEEE Computer Society, 2023.

[19] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. 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.

[20] Federico Perazzi, Jordi Pont-Tuset, Brian McWilliams, Luc Van Gool, Markus H. Gross, and Alexander Sorkine-Hornung. "A Benchmark Dataset and Evaluation Methodology for Video Object Segmentation." In: *Proceedings of the Conference on Computer Vision and Pattern Recognition, CVPR*. IEEE Computer Society, 2016, pp. 724–732.

[21] Carson Powers. "Estimating File Compressibility Using File Extensions." Master's Thesis. University of Minnesota, 2009.

[22] Florian Sattler. "A Variability-Aware Feature-Region Analyzer in LLVM." Master's Thesis. University of Passau, 2017.

[23] Ashish Sen and Muni Srivastava. "Multicollinearity." In: *Regression Analysis: Theory, Methods, and Applications*. Springer, 1990, pp. 218–232.

[24] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. "Performance-Influence Models for Highly Configurable Systems." In: *Proceedings of the Joint Meeting on Foundations of Software Engineering, ESEC/FSE*. ACM, 2015, pp. 284–294.

[25] Miguel Velez, Pooyan Jamshidi, Florian Sattler, Norbert Siegmund, Sven Apel, and Christian Kästner. "ConfigCrusher: Towards White-Box Performance Analysis for Configurable Systems." In: *Automated Software Engineering, ASE* 27.3 (2020), pp. 265–300.

[26] Yi Zhang, Jianmei Guo, Eric Blais, and Krzysztof Czarnecki. "Performance Prediction of Configurable Software Systems by Fourier Learning (T)." In: *Proceedings of the International Conference on Automated Software Engineering, ASE*. IEEE Computer Society, 2015, pp. 365–373.