University of Passau

Department of Informatics and Mathematics

Master's Thesis

Evolution of Performance Influences in Configurable Systems

Author:

Johannes Hasreiter

June 17, 2019

Advisors:

Prof. Dr.-Ing. Sven Apel

Chair of Software Engineering I

Christian Kaltenecker

Chair of Software Engineering I

Alexander Grebhahn

Chair of Software Engineering I

Hasreiter, Johannes:
Evolution of Performance Influences in Configurable Systems
Master's Thesis, University of Passau, 2019.

Abstract

During the evolution of a software system, the performance often changes between consecutive revisions of the system due to code modifications. Although some of these changes are anticipated by the developer of the system (e.g., by using newer versions of an underlying library), others are introduced unintentionally (e.g., by causing a regression). Identifying and removing these accidentally introduced performance bugs is crucial for user experience. However, identifying these performance bugs in configurable software system is non-trivial because these bugs might only have an influence on the performance of a small number of configurations of the system.

In this work, we aim at identifying the revisions where performance changes are introduced from different perspectives and relate them to the documentation of the system via the change log and the commit messages. Overall, we consider 3 open-source software systems and analyze their evolution of performance across various releases. In more detail, we determined the performance of the configurations of the systems and performance-influence models for the different releases that enable us to determine the performance contribution and the relevance of individual configuration options and interactions among them, causing the performance changes of the configurations.

Contents

List of Figures

List of Tables

1. Introduction

Most of today's software systems are highly configurable and provide a vast number of possibilities to configure the behavior of the underlying program. Configuration options can modify not only the functionality, the results, and the way of computation, but also non-functional properties like the execution time can be affected. For system administrators or end-users, it might be of interest to find performance-optimal configurations for their desired use case.

Over time, performance of software underlies changes, which can for example be cause by the creators in the ongoing development. While producing new releases the developers might introduce performance bugs and, thus, it would be very helpful for them to put down those performance bugs to specific configuration options.

In this exploratory work, we aim at analyzing the evolution of performance in configurable software systems over time. Whenever refering to performance in this work, we mean the execution time of the corresponding program. By learning trends which might appear in the regarded time span, we want to provide generalizable insights in the performance changes which configurable software systems underlie. The basic approach of this work is to increase the level of granularity stepwise in the upcoming research questions. Beginning with (1) the overall performance of a release, we further investigate (2) the configurations of each release, and investigate it in even mor detail by identifying (3) the influence of individual options or interactions among them.

Siegmund et al. [SGAK15] provided so-called performance-influence models which can be used to determine the influences of individual configuration options or interactions among configuration options on the overall performance of a configurable software system. These performance-influence models, for example, can be used as a human understandable way to identify performance-intensive configuration options or which configuration options do not affect the performance at all. As a result, performance-influence models are a useful concept to get the most out of the configuration space regarding performance.

It is not sufficient for highly configurable software systems, to only measure the performance of a few specific configurations, for predicting the performance evolution of the entire system. Therefore, we will conduct different case studies where we consider a high number of individual configurations and make the use of performance-influence models, to gain insights into the performance changes appearing over time. By using computed performance-influence models, we want to trace back performance changes between different software releases to individual configuration options or interactions among them. Finally, we compare our results with the documentation provided by the software developers to detect possible congruence and to identify whether the developers are aware of introduced performance changes. The results of our case studies are meant to help the understanding of performance evolution in highly configurable software systems which might lead to different toolsets helping software developers to prevent or identify performance bugs.

For this exploratory research, we selected the following research questions, which we consider appropriate for investigating the desired results explained above. The questions will incrementally increase the level of granularity.

RQ1: How frequent and how strong are performance changes between consecutive releases in configurable software systems?

RQ2: What is the fraction of software configurations that are affected by a performance change between consecutive releases in configurable software systems?

RQ3.1: How frequent and how strong are performance changes between consecutive releases influenced by individual configuration options and interactions among them?

RQ3.2: Which configuration options influence the overall performance of all configurations the most and how does this change over time?

RQ4: Are performance changes between consecutive releases documented in the version control system?

This thesis is structured as follows:

At first, in Chapter 2, we provide relevant background information which is necessary to understand the concepts and terminology used in this work. Therefore, we will explain the basics of configurable software and variability modeling followed by the functionality of performance-influence models. The chapter is concluded by a summary of version control systems.

The research questions above are explained in detail in Chapter 3, as well as the operationalization, which describes how we structure the investigation of those questions.

The results of the proposed research questions and investigation methods are presented in Chapter 4. We will discuss the results for each research question directly after we showed the outcomes for each case study.

Chapter 5 is used to discuss the internal and external validity of this thesis and its results.

In Chapter 6, we present releated work which deal with performance in highly configurable software systems, the evolution of performance and the evolution of software in general.

Finally, Chapter 7 concludes this work and provides ideas for future work.

2. Background

In the following chapter, we describe all relevant background knowledge which is necessary to understand and follow the concepts which will be introduced in this work. First, we introduce variability models in Section 2.1 followed by the explanation of performance-influence models in Section 2.2. The chapter will be concluded with a short outline on version control systems in Section 2.3.

2.1 Variability Models in Configurable Software

Most common big software systems provide a variety of configuration options which may interact with each other to a certain degree. Configuration options are run-time or compile-time parameters which are provided to the user by the software developers. The configuration options enable the user to change the behavior, the desired computation outcome and other paramters of the program. On the one hand, compile-time options define parameters, which are already known at compile time and can be set statically. On the other hand, run-time options can be chosen while executing the program [ABKS13].

Not all arbitrary combinations of configuration options are valid. For instance, there can be configuration options which mutually exclude each other. Exemplarily, a program could provide two compression algorithms, where exactly one has to be selected. This example and other types of relationships between configuration options precisely define, which configuration option selections are valid and which ones are not. There are two types of constraints that are important for this work. The first one defines, which options are mandatory and which ones are optional. The other crucial concept is the alternative group, which states, that precisely one of the contained options has to be selected at a time [ABKS13].

Throughout this work, we use the following formal terminology when addressing variability models. Let O be the set of all configuration options and C the set of all configurations. A configuration $c \in C$ is modeled as a function $c : O \to \mathbb{R}$ which assigns a selected value to every option. A binary option is represented by $c(o) = 1$

if the corresponding option is selected ($o = \text{True}, o \in O$) and $c(o) = 0$ otherwise ($o = \text{False}, o \in O$) [SGAK15].

To get an overview of all configuration options, there exist different approaches to formalize the way of representing configuration options with so-called variability models. Variability models describe all possible configuration options and relationships among them in software systems. Each of the approaches has other applications, benefits, and disadvantages. In this work, we use the graphical representation consisting of variability diagrams which are introduced in the following.

Variability Diagrams

If the primary goal is displaying all possible configuration options and their dependencies, variability diagrams are much more appealing. Variability diagrams are a graphical representation of variability models, which use hierarchical tree structures. Each node of the tree represents a configuration option in the corresponding model. A parent-child relation involves that the child feature is only selectable if the parent feature also is selected. Parent nodes define more general concepts, whereas child nodes specialize those abstractions. Several graphical notations illustrate additional constraints, like for instance the information if an option is mandatory or optional [ABKS13].

The graphical representation of mandatory or optional configuration options is illustrated by a small circle on the child node, which is filled when the option is mandatory (●) and empty otherwise (○). There are also ways of representing additional constraints concerning disjunctive combinations of configuration options. The edges between a parent option and the child options are connected with an empty arc if it concerns an alternative group, where only one option can be selected at a time.

In Figure 2.1, we examplary show the variability diagram of a hyptothetic tool for compression. It contains one alternative group where one is forced to select exactly one compression algorithm (lzma, gzip, or lzo). Furthermore, there is an optional configuration option, which can be used for splitting the compressed archive into multiple files.

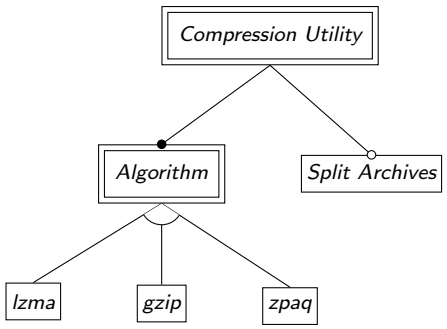

Figure 2.1: Visual representation of a hypothetic compression tool. It consists of a mandatory alternative group, where one compression algorithm has to be chosen and an optional configuration option for splitting archives.

2.2 Performance-Influence Models

When facing big software systems, there is a vast number of configuration options which must be considered by the user. The Linux Kernel is one example with about 10,000 configuration options which lead to billions of different possible configurations [LvRK+13]. The configuration options have a huge influence as well on used algorithms, output quality, and functionality as on performance indicators like execution time or energy consumption. Often, configurable software systems allow the combination of the offered configuration options. As a result, it is hard for users to identify the configuration which suits their use case the best. Here, not only the desired output is a relevant factor, but also the performance is of interest for the user. For example, one could be interested in the optimal configuration regarding the performance, and since often, not all configurations can be measured, heuristics have to be used. In contrast to the performance-optimal case, overwhelmed by the massive variety of configuration options, users often stick to the default configuration or change only certain specific options. To face the stated problems and to find a configuration which performs best, performance-influence models can be used [SGAK15]. Their functionality will be described in the following.

Terminology

The primary goal of performance-influence models is to give a hint on the individual influences of specific configuration options and interactions among them on the overall performance of the software system. The individual influences and potential interactions between multiple options find their way in a formula which is easily understandable. Given the terminology which was introduced in Section 2.1, a performance-influence model is a function $\Pi : C \to \mathbb{R}$ that uses a configuration as input and states the performance of the used configuration. Here, the performance can be any measurable property which leads to interval-scaled data, e.g., execution time or energy consumption.

An example performance-influence model for the previously introduced compression utility example with the options lzma (L), gzip(G), zpaq(Z), and split (S) could look like the following:

$$\Pi(c) = \underbrace{100}_{\text{Basic influence}} \overbrace{-10 \cdot c(L)}^{\text{Influence of L}} + 20 \cdot c(G) + 50 \cdot c(Z) + 2 \cdot c(S) + \overbrace{0.5 \cdot c(Z) \cdot c(S)}^{\text{Influence of interaction of Z and S}}$$

It is essential to understand that there are parts, which describe both the influence of an individual option and the influence of performance interactions. The influence of the option lzma (L) is described by the term $-10 \cdot c(L)$. An interaction can be found for the options zpaq (Z), and split (S) when looking at the term $0.5 \cdot c(Z) \cdot c(S)$. Furthermore, there is a basic influence, which is always part of the term, no matter what configuration was inserted (coefficient 100).

In general, performance-influence models have the following form:

$$\Pi(c) = \beta_0 + \sum_{i \in O} \phi(c(i)) + \sum_{i..j \in O} \Phi_{i..j}(c(i)..c(j)) \tag{2.1}$$

β_0 describes a constant base performance which is shared by all configurations. ϕ_i represents the individual influence of option i, whereas $\Phi_{i,j}$ denotes the influence of the interaction between options i and j. Consequently, $\sum_{i \in O} \phi(c(i))$ describes the sum of the influences of all individual options and $\sum_{i..j \in O} \Phi_{i..j}(c(i)..c(j))$ refers to the sum of the influences of all interactions among all options.

2.2.1 Sampling Configuration Spaces

The computation algorithm for performance-influence models is based on measurements which have to be taken for the considered software system. A measurement consists of a configuration and the corresponding performance results. Since the configuration space for big software systems can be huge and the number of configurations increases exponentially with the number of configuration options, it is not possible to measure all configurations in the configuration space. Therefore, some strategies for identifying relevant configurations exist. Sampling strategies come with the observation that individual influences and interactions among a small number of configuration options are more relevant compared to higher order interactions [KSK+19]. Among others, the following sampling approaches can be used for binary options [SGAK15]:

- ***Option-Wise Sampling (OW):***
  Option-wise sampling aims at minimizing the number of configurations while having each option selected at least once. Furthermore, it tries to minimize the number of selected options per configuration. The goal of this strategy is to identify the influence of individual configuration options with the absence of interactions and this way, interactions between options can be minimized in the resulting configurations. For each binary option, a configuration is searched with as many options disabled as possible under the constraint $c(o) = 1$ and the given constraints among the options. Valid configurations $c \in C$ can be identified by using a constraint-satisfaction problem solver. This step then has to be repeated for all options. Additionally, the strategy adds one configuration with all configuration options deselected, as far as possible.

- ***Pair-Wise Sampling (PW):***
  Pair-wise sampling again tries to minimize the number of configurations and the options per configurations. In contrast to option-wise sampling, for each pair of options $p \in O$ and $q \in O$, a configuration is calculated with as many options disabled as possible under the constraints $c(p) = 1 \wedge c(q) = 1$. By proceeding like this, a minimal set of configurations including all two-way interactions can be determined.

- ***Random Sampling:***
  Random sampling, as the name suggests, randomly chooses configurations from the configuration space. Of course, all constraints from the variability

model have to be fulfilled. In this work, random samples are created by computing all valid configurations of the entire configuration space and, afterwards, picking random examples thereout. The reason for using random sampling is the fact, that we cannot know the behavior of the individual configuraion options in advance. Therefore, random sampling provides a evenly distributed selection of configurations of the entire configuration space. Random sampling is an appropriate sampling strategie for huge configuration spaces, like it was also used in [LvRK+13].

2.2.2 Learning Performance-Influence Models

Performance-influence models for software systems with a formula of the form of Equation 2.1 can be derived by using an incremental learning algorithm. This algorithm uses a set of measured configurations and the variability model of the software system as input. The procedure incrementally computes relevant options and the corresponding influence on the overall performance. In each round of the process, a configuration option or an option interaction is added to the concluding performance-influence model until no more relevant improvements can be achieved [SGAK15].

To calculate relevant options for the model, candidate options are considered. These options potentially can be added to the model if the algorithm decides that a particular option has a significant influence on the performance of the system. Initially, the candidate options contain all single options that are provided by the variability model of the software system.

For every candidate, a model is computed using multivariable linear regression. This computation results in a term like $20 \cdot c(E)$ (given the example from above). With the use of these terms, the algorithm tries to predict the measured performance values and calculates an error rate for each candidate. The candidate with the smallest error, then, will be added to the overall model. After that, new candidates can be added. The selection of the new candidates will be described later. This process is repeated until the error rate decreases only marginal or a threshold for the expected accuracy is reached [SGAK15] [KSK+19].

Candidate Selection

Due to the enormous number of theoretically possible option interactions, some heuristics have to be considered when adding candidate features in the algorithm described above. In this work only one heuristic is used, namely **hierarchical interactions**.

It was found that often only those interactions are relevant where also the individual options had an impact on the overall influence. As a result, after a candidate was selected in one round of the algorithm, only the interactions with that particular option are added to the possible candidates [SGAK15].

2.3 Version Control System

All common software projects these days use some source code management system which helps the developers to keep track of all changes in the program code and supports parallel developement. In this work, we will investigate different configurable software systems. The search for appropriate candidates for our case studies required the software systems to use Git as their version control system.

Requiring Git as the version control system, is not a huge constraint, because it is one of the most used version control systems these days and a lot of open source projects use it to handle their code base and allow contributors distributed around the world to collaborate. Among other things, it allows the users to assign messages to their code changes when uploading them to the server (commit messages). These messages usually are used to inform what exactly is intended by the corresponding code changes (e.g., *"Fixed performance bug for compression algorithm lzma".*). Apart from that, specific software versions can be tagged. Tagging is commonly done for major software releases. Git enables the user to check out older releases of a software system and get the commit messages between those versions with a simple API [GDT]. In this work, we will use the tags to identify releases which are relevant for our experiments and use the API to retrieve all commit messages between two specific releases.

3. Methodology

In this chapter, we formalize the investigation of performance evolution in configurable software systems. First of all, we precisely define the previously outlined research questions in Section 3.1. Afterwards, we explain how we assess to answer the research questions in Section 3.2.

3.1 Research Questions

In this exploratory research, we investigate the behavior of performance in configurable software systems over time. For the performance analysis, we consider the performance from different granularity levels. Firstly, we will have a look at entire releases of configurable systems by investigating the overall performance of releases. Afterwards, individual configurations will be investigated. The details will be analyzed when we deal with the performance of individual configuration options or option interactions. At last, we try to validate our findings by comparing them to the documentation provided by the case study program's owners. In this section, each research question will be explained in detail.

RQ1: How frequent and how strong are performance changes between consecutive releases in configurable software systems?

With the first research question, we want to observe, if there are performance changes between consecutive releases of configurable software systems. Here, the focus lies on whole releases and not on individual configurations or configuration options. Both the frequency of performance changes between releases and the magnitude of those changes are of interest.

RQ2: What is the fraction of software configurations that are affected by a performance change between consecutive releases in configurable software systems?

We assume that the detected performance changes in the first research question are caused by individual configuration options, which are not present in all possible

configurations of the software system. Therefore, we want to determine the number of configurations, which are concerned with performance changes regarding consecutive releases. For example, we can imagine, that only a few configurations cause the overall performance to change due to specific configuration options which are selected.

RQ3.1: How frequent and how strong are performance changes between consecutive releases influenced by individual configuration options and interactions among them?

In this research question, we aim at identifying wheter single configuration options or interactions among them cause the potentially detected performance changes between consecutive releases. One example we consider as possible is a performance bug which was introduced between to releases and only impacts one specific configuration option because the corresponding change in the program code was related to that particular configuration option.

RQ3.2: Which configuration options influence the overall performance of all configurations the most and how does this change over time?

We want to find out, if these configuration options or option interactions have relevance on the overall performance changes we consider in the first research question. One can imagine an example, where an individual configuration option has a high impact on the overall performance of configurations where this configuration option is selected. However, if there are only a few configurations in the configuration space, where this configuration option is selected, the regarded behavior may not be of much interest. Therefore, we want to use a combined metric for deciding, if a configuration option both has high influence and appears in most of the configurations. Additionally, we want to explore the described approach over time and learn how the relevance of configuration options or option interactions changes between releases.

RQ4: Are performance changes between consecutive releases documented in the version control system?

With this research question, we want to check if the results of the previous work align with the documentation of the configurable software systems. Eventually, the previously mentioned performance bugs are documented in the changelogs or commit messages between two releases. If we can find corresponding text passages which fit the results of the previous research questions, our methods and models can be validated.

Experimental Dependencies

For the construction of our experiments, we consider the following empirical variables which are valid for all our research questions.

Independent Variables Since we consider different case studies for answering the research questions, we count the used configurable software systems as independent variables. These configurable software systems are modeled with different variability models and have different configuration and release spaces. This is the reason why these parameters are independent variables, too.

Dependent Variables All our research questions deal with the performance of configurable software or its configurations. Furthermore, we aim at investigating the influences of individual configuration options by using the approach of performance-influence models. Therefore, we consider both the execution time and the performance-influence models as dependent variables of our studies.

Confounding Variables The experiments heavily depend on the choice of the used sampling strategies, because they are used to select specific configurations which the execution time is measured for. Also, the hardware for the performance measurements could affect our results. To keep the experiment effort moderate, those two mentioned confounding variables are fixed.

3.2 Operationalization

In the following section, we will explain our approach of investigating, evaluating and answering the previously defined research questions. In Section 3.2.1, we will describe the experimental setup consisting of our case studies and the performance measurement design. Afterwards, our evaluation strategies are explained in Section 3.2.2.

3.2.1 Experimental Setup

Our experiments consist of a high number of individual performance measurements of different configurable software systems. In this section, we provide an overview of those case studies in Section 3.2.1.1, followed by the configuration sampling strategies in Section 3.2.1.2, and the measurement setup in Section 3.2.1.3.

3.2.1.1 Case Studies

In the first place, it is necessary to analyze the chosen software systems. In the following section, we aim at describing their uses, the reason for choosing them, how they are structured, and what exactly gets measured. We selected three software systems of two different fields. Two of them are compression utilities, and the other one is a video encoding software.

While selecting potential candidates for our case studies, we only considered software systems which use Git as their version control system. Having Git as a base for retrieving different revisions of a software system is convenient, because of its tagging features. The tags assigned to specific commits were used to identify major releases of the regarded software system. There are several reasons for only using releases in our case studies. Usually, releases have a robust state which has a minimum of

bugs. Furthermore, there might be some states between two releases which do not compile at all because of some critical software errors.

For each software system, we defined a variability model which contains configuration options that are valid in all releases. Although new functionality and, thus, new configuration options are added to software systems in newer releases, we excluded these configuration options to maintain comparability between releases. Additionally, to keep the measurement effort reasonible, we only focused on configuration options that might have an influence on the output quality or the execution time. To this end, we used the provided documentation of the configuration options. There usually are options in programs, which modify the output in the logs or are used for choosing a filename for the output. We exculed such options, because we assume that these configuration options do not influence the resulting performance.

In the following sections, we discuss the different used software systems separately.

lrzip

lrzip is a compression utility with high performance for larger files. If the files which have to be compressed, are bigger than 10-50 MB and a sufficient amount of RAM is available, the execution time is the best [Kol]. The tool provides different compression algorithms which differ in execution time and the quality of the output. There are fast options with a low compression rate (e.g., *lzo*), meaning that the resulting files are bigger, and slow options with a high compression rate (e.g., *zpaq*) [Kol].

Measurement Space. In this case study, we considered all releases which were available in the Git repository, namely all tagged releases between *v0.45* and *v0.631*. These releases span across a period between March 30th 2010 and October 21st 2016. In total, we analyzed 38 releaes with 187 configurations each.

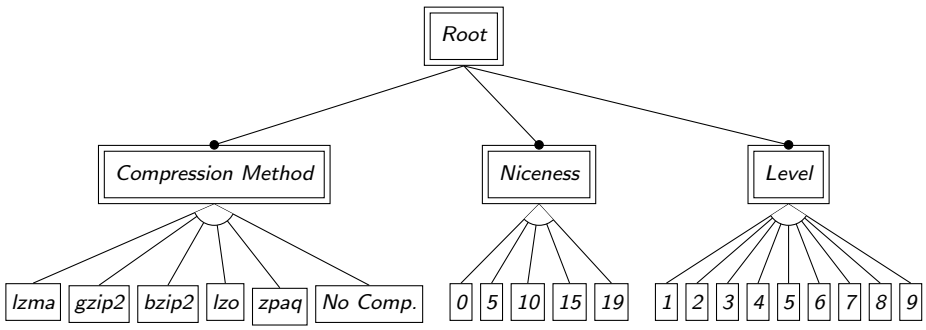

Figure 3.1: The variability model of *lrzip* which consists of three mandatory alternative groups (*Compression Method*, *Niceness*, and *Level*).

Variability Model. The variability model of lrzip generally consists of three different alternative groups. These are the used compression method, a level number,

and a niceness value. The graphical representation of the model is shown in Figure 3.1. All those options are runtime options which are available in all regarded software releases.

The user has to choose one method which shall be used for compression. According to the documentation of lrzip, these modes differ in the output quality and the execution time. The alternative group *Compression Method* summarizes the user's choices in this concern and additionally provides the option *No Backend Compression* which deactivates compression completely.

The configuration options *Level* (values between 0 and 9) and *Niceness* (values between 0 and 19) are numeric options which we converted to alternative groups consisting of binary options. The configuration option *Level* is used to set the compression level. The compression level is strongly related with the memory consumption of the program. The configuration option *Niceness* can be used to set the priority scheduling for the lrzip backup or decompression. To keep the configuration space in an manageable size, we did not consider all possible numeric values for the alternative group *Niceness*. Therefore, we just used niceness values between 0 and 19 with step size 5.

GNU XZ

The tool *GNU XZ* or *XZ Utils* is a free general-purpose data compression software with a high compression ratio. It has a similar command line interface to gzip and provides its file format for compressed data ".xz". The software system is the successor of *LZMA Utils*, thus the file format ".lzma" is also supported [Proa].

Measurement Space. The Git repository of *GNU XZ* provides 14 tagged releases with version numbers between *v5.0.0* and *v5.2.4*. There are also some alpha or beta releases, which we did not consider in our studies. According to the git repository, the regarded revisions were released in the period between October 23rd 2010 and April 29th 2018 [Proa]. In total, we measured 14 releases with 179 configurations respectively.

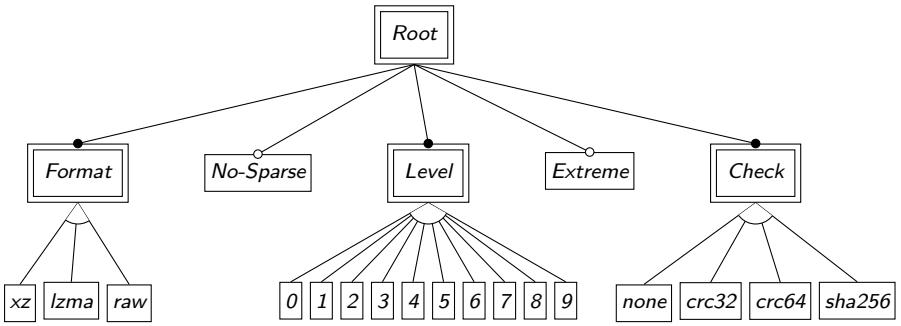

Figure 3.2: The variability model of *GNU XZ* consisting of two optional configuration options *No-Sparse* and *Extreme*, and three mandatory alternative groups *Format*, *Level* and *Check*.

Variability Model. GNU XZ offers three big mandatory alternative groups in its variability model. Mandatory in this case means that there are defaults values for each group if the user does not specifically select options. The most important one is the group *Format*, which the user can use for controlling the output file format. The quality of the compression can be set by varying the *Level* value between 0 and 9. Lower levels result in a fast compression with a lower compression ratio, whereas higher levels take more time but offer a higher compression ratio. With higher levels, also the memory consumption rises. The third alternative group is called *Check* and deals with the integrity checks, which are calculated for the uncompressed data and stored in the compressed file. There are three different checks available, namely *crc32*, *crc64*, and *sha256*. Besides, the user is allowed to disable integrity checks with the option *none* completely.

Additionally, there are two optional configuration options, which can be selected by the user. *No-Sparse* disables the creation of sparse files and *Extreme* uses a slower variant for the selected compression level to get a better compression ratio. Figure 3.2 shows the visual representation of the described variability model.

libvpx VP8

VP8 is a video codec for compressed video files which is supported by the WebM file format. It is mainly used in web applications and therefore is present in today's typical web browsers. The library *libvpx* contains the reference implementation of the video encoding format *VP8* and is developed by the WebM Project [Prod].

Measurement Space. For our case studies, we used the versions between *v0.9.6* and *v1.7.0* which are the major releases provided by the Git repository. There are also older releases in the repository we did not consider, because of there utterly different structure. The used releases were published in the repository between 4th March 2011 and 24th January 2018 [Prob]. This case study contains 11 releases, whereas each release will be measured in 488 different configurations.

Variability Model. During our studies of the documentation provided by the WebM project for the video codec VP8, we came across the variability model shown in Figure 3.3.

First of all, there is an alternative group affecting the encoding quality and speed. The contained options are *best*, *good* and *rt* (real-time). *best* gives the best quality output but is the slowest of the mentioned options. *good* is a trade-off between quality and execution time. When choosing the *real-time* option, the encoder auto adjusts the trade-off between quality and speed with a CPU utilization target in mind [Proc].

The video encoder provides specific options to influence the bitrate of the output. These are *CBR* (constant bitrate) and *VBR* (variable bitrate). The *constant bitrate* option aims at keeping the bitrate stable between buffering constraints. *VBR* distributes the bits between different frames or sections to get a maximum of output

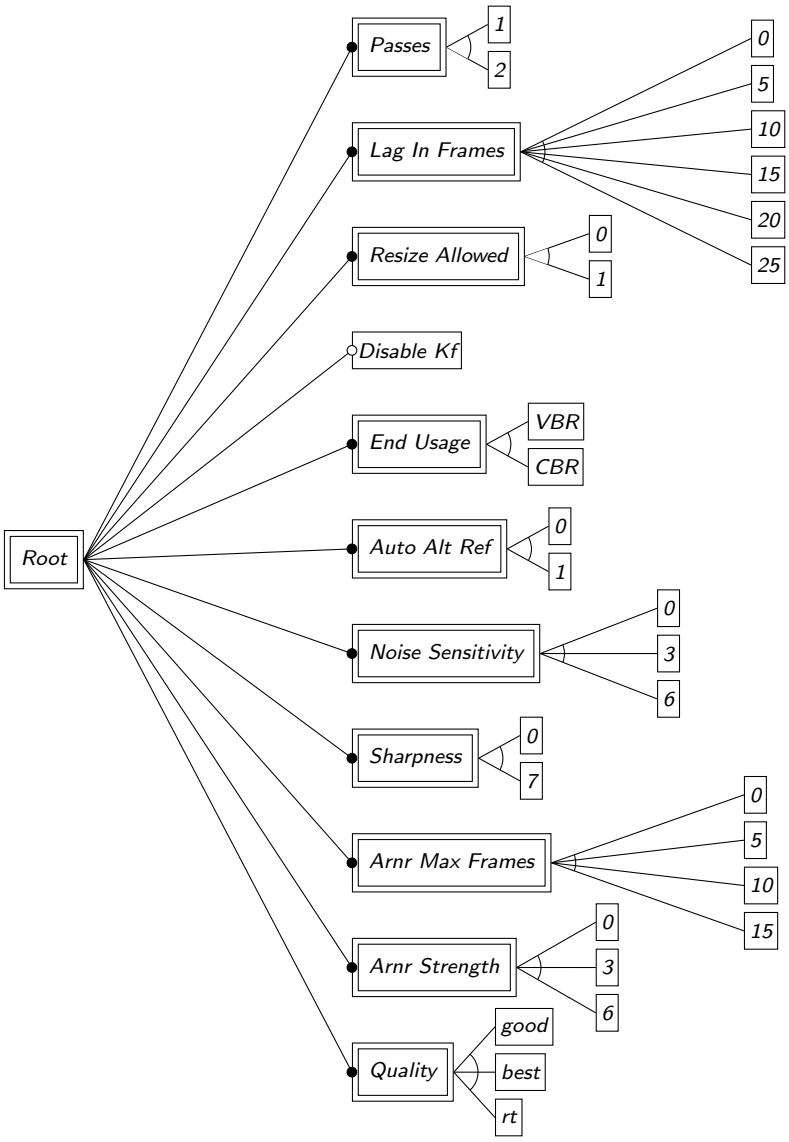

Figure 3.3: The variability model of *libvpx VP8* consisting of one optional configuration option *Disable Kf* and ten mandatory alternative groups.

quality. For tough sections, more bits are allocated to prevent a drastic quality drop [Proc].

VP8 provides both one-pass and two-pass encoding. When choosing the one-pass option, the passes the video only one time and processes it directly. The two-pass filter does two separate walks through the data. The first pass is used to collect statistical data about the frames which then can be used to better allocate bits between different frames or sections of the video. Afterwards, the second pass processes the video given the data from the first pass. Two-pass encoding usually delivers better quality but cannot be used for some use cases like live streams [Proc].

Apart from the previously mentioned configuration options, there are a few other ones, which may influence the output quality and the execution time, according to the documentation. These configuration options need a deeper understanding of video encoding in general, and thus, we want to mention them for the sake of completeness. *Auto Alt Ref* enables the *Alternate Reference Frame* mode which can substantially improve quality in many situations, but is only available for two-pass encodes. *Arnr Max Frames* and *Arnr Maxstrength* also deal with the mentioned *Alternate Reference Frame* mode. *Sharpness* and *Noise-Sensitivity* address internal filters which are applied during the encoding process. VP8 supports temporal and spatial resampling which change the scale of certain frames and can be controlled by the parameter *Resize-Allowed*. The keyframe placement can be disabled by the option *Disable Kf* which might also affect the overall performance of the encoder [Proc].

3.2.1.2 Configuration Sampling

An essential step before measuring the performance of the selected case studies is to identify the set of configurations which shall be considered. Therefore, [SGAK15] provided different sampling strategies, which were described in Section 2.2.1. Since we only used binary configuration options in our case studies, the sampling strategies confine to mainly two of them. We considered option-wise and pair-wise sampling due to their simple way of constructing different valid configurations and their deterministic behavior. After applying the two mentioned approaches, we added the same number of random, but valid samples, which are not already covered by the previous strategies. By going that way, we do not specialize in specific configuration spaces. The software collection SPL Conqueror [Pas] provides algorithms which were utilized to calculate sets of configurations according to the previously mentioned strategies for each case study respectively [Sie].

3.2.1.3 Performance Measurements

After sampling a set of configurations, each case study can be measured accordingly. To record the execution times of the regarded software systems, suitable workloads and measurement settings have to be identified.

Workloads

Since we consider two kinds of software systems, we found two corresponding workloads. There is a need for the workloads to have a certain minimal size because we want to reduce the ratio of noise and basic influence in our measurements.

For the compression tools *GNU XZ* and *lrzip* we used files of the *Canterbury Corpus* [UoC] which is a well-known and often used workload for compression utilities. We combined two of the offered corpora, namely *The Canterbury Corpus* (4 times) and *The Large Corpus* (6 times). This way, we achieve diversity in the contained files and the necessary size. The compression tools in our case studies are not able to compress multiple files at once. Therefore, we packaged the corpora in an uncompressed tar archive which then can be picked up by the programs.

For the video encoding tool *libvpx VP8*, we used the movie *Big Buck Bunny* [Fou]. This video is often used as a benchmark for all sorts of video related tests. Only the first minute of the film with a resolution of 1080p was used to measure the performance of the video encoder to keep the execution time in an acceptable range.

Measurement Settings

Each configuration was measured three times with the workloads discussed above. In addition to the configuration options which are part of the sampled configurations, we added options to control the input and output filenames as well as the logs. This was necessary, to automate the measurement process. For the tool *lrzip* we additionally had to provide the option "-T 0" (until *v0.552*) or "-T" (from *v0.560*), to prevent the program from doing a compressibility test before the real compression, because this test would distort the measurement results.

After measuring, the standard deviation of the three recorded execution times was computed. We repeated all measurements of configurations with a standard deviation of more than 10%. The resulting execution time value in milliseconds which will be used for evaluation is the mean value of the three measurements.

Measurement Hardware

All our performance measurements were performed on workstation PCs which all have the exact same hardware. The computers are equipped with 16GiB RAM and the Intel Core i5-4590 with 4 corse and a base frequency of 3.3GHz.

3.2.2 Evaluation

The following section will provide a detailed explanation of our approaches which are used to answer our research questions, which we defined in Section 3.1. For each research question, there will be a passage which explains our corresponding experiments.

3.2.2.1 RQ1: Overall Execution Time Evaluation

Our first research question deals with performance changes between consecutive releases in general. This means we don't want to analyze the different configurations separately. Instead, we try to get an overall picture of the execution time trend over time.

Nonetheless, our approach for identifying trends across different releases of the software systems relies on all considered configurations. It does not make sense only to

investigate one particular configuration of the program because that specific configuration most certainly does not reflect the execution time behavior of other configurations, where utterly different configuration options are selected. Therefore, we try to take all measured configurations into consideration by averaging the measured execution times per release, respectively. The following formula is used to calculate the average execution time AVG per release $r \in R$ where R is the set of all releases:

$$AVG_r = \frac{\sum_{c \in C} t_{c_r}}{|C|},$$

whereas C is the set of all measured configurations and t_{c_r} depicts the execution time of configuration $c \in C$ in release $r \in R$.

After computing the average execution time per release, we can compute the change rate CR between the releases r and $r+1$ as follows:

$$CR_{r,r+1} = \frac{AVG_{r+1} - AVG_r}{AVG_r}$$

If the mean execution time between two consecutive releases changes more than 20 percent ($|CR_{r,r+1}| > 20\%$), we recognize this as a *substantial* performance change. Since the maximum standard deviation between measurements was set to 10 percent during the measurement stage, choosing twice the standard deviation for comparing two different releases is a conservative estimation. In the worst case, both releases were measured with a standard deviation of 10 percent in opposite directions. If the mean execution time across all configurations exceeds double the standard deviation, we can be certain that we deal with a real performance change. We call releases with a *substantial* performance change *outstanding releases*.

After identifying releases which underlay performance changes, we additionally are able to make statements about the strength of the performance changes by again investigating the change rates between the mean execution times across all configurations of consecutive releases. Obviously, larger change rates between the mean execution time across all configurations indicate more significant performance changes between successive releases.

3.2.2.2 RQ2: Configuration Execution Time Evaluation

In contrast to the first research question, which deals with releases as such, the second research question points at specific configuration inside the investigated releases. It is possible, that a release with an relevant performance trend found with *RQ1* only has substantial performance changes in certain configurations. For example, a high change rate of mean execution times can be produced by only one configuration, which has a vast performance difference while the other configurations do not show much change. It is also possible that all configurations show a small but significant performance change, which again results in a performance change rate, which is of interest for *RQ1*.

To address the stated behavior and to increase granularity, we focus on configurations in particular with this research question. Again, we observe the change rates between

consecutive releases, but for individual configurations in this case. Since the same set of configurations is measured for every release, we can compare the exact same configuration in two consecutive releases and compute the corresponding change rate CR with the following formula:

$$CR_{c_r,c_{r+1}} = \frac{t_{c_{r+1}} - t_{c_r}}{t_{c_r}}$$

For the same reasons as in Section 3.2.2.1, we consider the performance change rate as substantial, if it exceeds 20 percent.

By counting the configurations with *substantial* changes ($|CR_{c_{r-1},c_r}| > 20\%$), we can state if only a few configurations contributed to the overall performance difference or if more configurations are affected by a performance change. The corresponding configuration count CC_r is defined as:

$$CC_r = |\{c \mid c \in C, |CR_{c_{r-1},c_r}| > 20\%\}|$$

We can calculate the fraction of configurations that are affected by a performance change by computing the ratio between configurations with a *substantial* change and all measured configurations.

Furthermore, we aim at identifying releases, where the fraction of configurations with *substantial* performance changes is very high. Therefore, we compute the mean μ and the standard deviation σ of the amounts of configurations with *substantial* changes (CC_r). We call a release an *outstanding release* regarding this research question if $CC_r > \mu + 2 \cdot \sigma$.

3.2.2.3 RQ3: Performance-Influence Model Evaluation

After evaluating *RQ2*, we are able to tell, if specific configurations are affected by substantial performance changes. Since a configuration is defined by the selected configuration options, we can again increase the level of detail in our investigation when looking at individual configuration options of option interactions. If only specific configurations show a substantial performance change, it might be possible, that a particular configuration option or option interaction causes that behavior. We will use the approach of performance-influence models to analyze the described situation.

Performance-Influence Models

After measuring the execution times for all sampled configurations, we are able to compute performance-influence models per release of the corresponding software system. Therefore, the approach by *Siegmund et al.* [SGAK15] which is described in Section 2.2 can be applied. SPL Conqueror [Pas] provides an implementation for the learning algorithm which takes the performance measurements as an input and determines performance-influence models for the particular software revision.

When learning a performance-influence model using the incremental algorithm described by *Siegmund et al.* [SGAK15], the procedure decides which configuration

options or option interactions are the most promising ones for predicting the measured values (see Section 2.2.2). This is the reason for some options being part of the resulting performance-influence model and some options which are absent. The same holds for option interactions. As a result, the performance-influence models of the different measured releases of the software systems used in our case studies may not include the same terms. Although, one calculated performance-influence model is a good estimation for the corresponding software revision, the performance-influence models of different software revisions might be not comparable, because their terms consider different configuration options.

Nevertheless, the resulting performance-influence models are a good indicator for the maximum degree of interaction which influences the overall performance significantly, because the incremental learning algorithm always chooses the best candidate out of the candidate set. Here, the algorithm might favor an interaction over an individual option, if the interaction has a higher influence. Thus, if an interaction of a higher degree is relevant for the model, it is chosen by the incremental approach. We use the highest degree of interaction found in the results of the incremental approach in the following new approach.

Fitting a Performance-Influence Model. Due to the lack of comparability when using the incremental approach, we suggest a new method of learning performance-influence models. The main problem of the approach from the previous section is that we cannot affect the choice of terms and thus, ending in different selected terms in the models from different software revisions. Our new approach is to specify the terms which we want to have in the model beforehand. As a result, the performance-influence models of all measured software revisions have the same set of terms which leads to improved comparability. SPL Conqueror provides the option to pass a specific set of terms which then will be part of the calculated performance-influence model.

The choice of terms which will be handed over to SPL Conqueror is crucial. To avoid the chance of missing specific relevant terms, we decided to choose all possible terms up to a certain degree of interaction. First of all, this means that we use all single options including the root option. Additionally, we take all possible combinations of individual options to construct terms for the interactions. These combinations have as many members as the degree of interaction we want to model. The degree of interaction describes, how many invidivual options are interacting with each other. We use the previously described incremental model to decide, which degree of interaction is the highest we wish to create terms for.

Shifting in Alternative Groups. The approach of choosing all possible terms up to a certain degree of interaction results in a problem which again might worsen the comparability of different performance-influence models. If there are mandatory alternative groups in the variability model, the algorithm can ”shift” influences between the alternative group, the root option, and between different elements of the alternative group. The following example illustrates the problem by providing

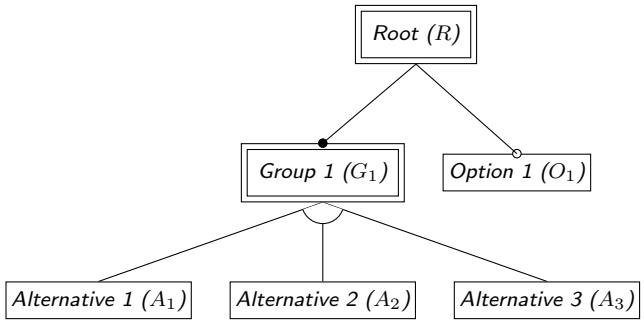

Figure 3.4: Exemplary variability model for the shifting issue regarding mandatory alternative groups.

two different performance-influence models which are equivalent. The underlying variability model is shown in Figure 3.4.

$$\Pi_1(c) = \overbrace{100}^{\beta_R} + \beta_{G_1} * c(G_1) + \overbrace{500}^{\beta_{G_1}} * c(A_1) + \overbrace{400}^{\beta_{G_2}} * c(A_2) + \overbrace{300}^{\beta_{G_3}} * c(A_3) + \beta_{O_1} * c(O_1)$$

$$\Pi_2(c) = \underbrace{200}_{\beta_R} + \beta_{G_1} * c(G_1) + \underbrace{400}_{\beta_{G_1}} * c(A_1) + \underbrace{300}_{\beta_{G_2}} * c(A_2) + \underbrace{200}_{\beta_{G_3}} * c(A_3) + \beta_{O_1} * c(O_1)$$

The reason for $\Pi_1(c)$ and $\Pi_2(c)$ being equivalent is the fact that both lead to the same predictions for all configurations. Since the alternative group is mandatory, exactly one of the three alternatives (A_{1-3}) has to be selected. As a concequence, any random number can be "shifted" between the coefficients of the group ($\beta_{G_{1-3}}$) and the coefficient of the root option (β_R). In the provided example an influence value of 100 is shifted from the alternative group to the root option. The same form of shifting can happen between the alternatives of the alternative group (A_{1-3}) and the parent of the alternative group (G_1).

We overcome this issue by removing some specific entries from the set of terms which we fit a performance-influence model for. By deleting the parent of all alternative groups and one random option for each mandatory alternative group in the set of requested terms, we avoid the described issue. We also need to remove all interactions wich contain the removed terms. Figure 3.5 shows a variability model with the removed terms for the previously given example.

The illustrated approach comes with one disadvantage. By randomly removing one option per mandatory alternative group, the learning algorithm is no longer able to calculate the influence on the performance for that particular option. However, the influences of the removed terms are part of the root option in that case. This behavior is equal for all resulting performance-influence models of all revisions and thus, leads to improved comparability.

To prevent the described artifact of shifting between alternative groups, we withdrew specific terms for each case study, which are listed in Table 3.1.

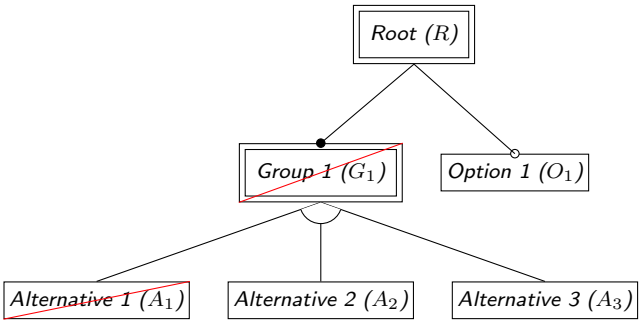

Figure 3.5: Removing terms to avoid shifting regarding mandatory alternative groups.

Table 3.1: Withdrawn terms for fitting a performance-influence model for the case studies *lrzip*, *GNU XZ*, and *libvpx VP8*. For numeric configuration options, the name of the option is given with the withdrawn value in parentheses.

lrzip	**GNU XZ**	**libvpx VP8**
No Backend Compression	Raw	Passes (2)
Level (9)	Level (9)	Lag In Frames (25)
Niceness (19)	crc64	Resize Allowed (1)
		CBR
		Auto Alt Ref (1)
		Noise Sensitivity (6)
		Sharpness (7)
		Arnr Max Frames (15)
		Arnr Strength (6)
		Rt

Intermediate Models It is possible that not all configurations of all releases of the conducted case studies can be executed successfully. In the case of failing measurements due to software bugs, there is the need of intermediate performance-influence models in our evaluation. If a release contains configurations, which can not be measured, the performance-influence model can only be learned with fewer measured configurations as in the other releases with only successful configurations. In that case, we calculate additional performance-influence models for the neighboring releases, where the the problematic configurations are excluded. We use these intermediate models to compare the release to the one with failing configurations.

If there are multiple neighboring releases which have failing configurations, we take the intersection of successful configurations to calculate intermediate performance-influence models which then can be used for comparison approaches.

Evaluation of the Performance-Influence Models

The goal of *RQ3* is to identify specific configuration options or option interactions which are affected by performance changes between consecutive releases. After cal-

culating performance-influence models for every release as described above, we can compare the computed results.

RQ3.1: Identifying affected configuration options. Similar to the investigation of single configurations in *RQ2*, we want to compare the calculated coefficients $\beta_o, o \in T$, whereas T is the set of all possible configuration options and option interactions. Respective change rates CR can be computed as follows:

$$CR_{\beta_{o_r},\beta_{o_{r+1}}} = \frac{\beta_{o_{r+1}} - \beta_{o_r}}{\beta_{o_r}}$$

If the change rate surpasses 20 percent ($|CR_{\beta_{o_{r-1}},\beta_{o_r}}| > 20\%$), we regard the corresponding configuration option or option interaction to have a *substantial* change rate. The arguments for selecting 20 percent as lower bound are the same as previously. Again, we can count the configuration options and option interactions which have a substantial ($> 20\%$) change rate in their coefficients between two consecutive releases and make statements on how many of them are involved in the overall performance change. The corresponding option count OC_r is defined as:

$$OC_r = |\{o \mid o \in T, |CR_{\beta_{o_{r-1}},\beta_{o_r}}| > 20\%\}|$$

Similar to *RQ2*, we call a release an *outstanding release* regarding this research question, if $OC_r > \mu + 2 \cdot \sigma$. Here, μ is the mean and σ is the standard deviation of the amounts of configuration options or option interactions with *substantial* changes (OC_r).

RQ3.2: Relevance Ranking of the Configuration Options. We assume that not all configuration options or option interactions have the same degree of relevance concerning the overall performance of a release across all configurations, which was of interest in *RQ1*. To make statements about the relevance of configuration options, we come up with the following metric, which allows us to rank the configuration options and option interactions according to their relevance.

We define the Relevance Factor (RF) for a specific configuration option or option interaction $o \in T$ for a specific release $r \in R$ as:

$$RF_r(o) = norm(\beta_{o_r}) * f(o),$$

whereas $norm(\beta_{o_r})$ is the normalized coefficient of the configuration option or option interaction o in the performance-influence model of release r and $f(o)$ is the frequency of configurations with o selected. Normalizing, in this case, means that the maximum coefficient in a performance-influence model is set to 1 and all other coefficients are scaled accordingly to a value between 0 and 1. Since the frequency $f(o)$ also has to be a value between 0 and 1, the resulting relevance factor must be between 0 and 1.

The computed relevance factors allow us to compare the relevance of all considered configuration options and option interactions for a specific release. Configuration options or option interactions with a larger relevance factor are regarded as more influential on the overarching performance of the release.

The relevance factors of a configuration option or option interaction can be compared between the releases. Similar to the previous research questions, we calculate the mean μ and the standard deviation σ of all relevance factors of all releases ($RF_r(o)$). Whenever $RF_r(o) > \mu+2\cdot\sigma$, for an arbitrary release $r \in R$, we call the configuration option or option interaction $o \in T$ a *dominating configuration option* or *dominating option interaction*.

3.2.2.4 RQ4: Documentation Analysis

RQ4 aims at confirming the results of the previous research questions. If we can find matches between the documentation and our calculated results, it strengthens our statements. Configuration options and option interactions with a substantial coefficient change rate seem to underly program code changes between the corresponding releases. The developers of the regarded software systems provide documentation in the form of changelogs for every release and commit messages in the version control system Git. If program code was changed, which is only used by a specific configuration option, there is the chance that exactly this change is documented either in the changelogs or in a commit message. We, therefore, try to find appearances of names of configuration options in the documentation between two releases, where our computed results also identified a substantial change in the corresponding coefficient.

To find the matches between documentation and our results, we implemented the following search. Firstly, this investigation searches the names of configuration options in the changelogs and in the commit messages. For every appearance, we can tell, between which two consecutive releases the corresponding code change was implemented. We then try to find those appearances in the respective performance-influence models, meaning that the coefficient of the configuration option changed substantially in the two respective consecutive releases.

We define the set of options o, which appeared between the releases r and $r+1$ in the commit messages or in the changelog respectively as:

$$T_{commit_r} = \{o | o \in O \text{ found in commit messages between } r \text{ and } r-1\}$$

$$T_{changelog_r} = \{o | o \in O \text{ found in changelog between } r \text{ and } r-1\}$$

The corresponding term counts TC_{commit_r} and $TC_{changelog_r}$ are defined as follows:

$$TC_{commit_r} = |T_{commit_r}|$$

$$TC_{changelog_r} = |T_{changelog_r}|$$

The sets of matches between the occurences in the documentation (commit messages or changelogs) and the computed performance-influence models are denoted as:

$$M_{commit_r} = \{o | o \in O, o \in T_{commit_r}, CR_{\beta_{o_{r-1}},\beta_{o_r}} > 0.2\}$$

$$M_{changelog_r} = \{o | o \in O, o \in T_{changelog_r}, CR_{\beta_{o_{r-1}},\beta_{o_r}} > 0.2\}$$

Respectively, the amounts of matches then are:

$$MC_{commit_r} = |M_{commit_r}|$$

$$MC_{changelog_r} = |M_{changelog_r}|$$

We do not assume to find all the term names in the texts as they are called in our model. For example, we do not believe to find the word 'level1' in the documentation texts. However, it might very well be possible to find the word 'level'. This is the reason, why create dictionaries, which contain words to search for, when dealing with specific configuration options (see Section A.1). Whenever talking about option interactions, we try to find all names of all configuration options of the interaction in the corresponding text passage of the documentation.

For every case study, we compute the match ratios MR_{commit} and $MR_{changelog}$. The ratios describe, which fraction of the occurences of option names in the documentation resulted in a match with the corresponding performance-influence model.

$$MR_{commit} = \frac{\sum_{r \in R} TC_{commit_r}}{\sum_{r \in R} MC_{commit_r}}$$

$$MR_{changelog} = \frac{\sum_{r \in R} TC_{changelog_r}}{\sum_{r \in R} MC_{changelog_r}}$$

4. Results

Next, we describe the results of the research questions for the case studies. After presenting some basic information of the executed performance measurements in Section 4.1 there will be a section for every research question which presents the results according to our operationalization described in Section 3.2. All following sections will contain the results for all case studies, we worked on. The sections for the research questions will be concluded with a discussion of the results.

4.1 Experiment Results

The performance measurement experiments described in Section 3.2.1 resulted in a vast number of execution time values for all combinations of releases and configurations. Table 4.1 summarizes some key figures, which provide a rough overview of the results for each case study. It contains information about the investigated measurement space and the maximum and minimum measured execution time values per case study. Indeed, all presented execution time values are the average of three independent measurements of the same configuration in the same release. As already stated in Section 3.2.1.3, the maximum standard deviation of the three measurements was limited to 10%.

lrzip

As shown in Table 4.1, the mean measured execution time values for lrzip range between *810.71ms* and *127,579.33ms*. The fastest value was accomplished in the most recent release (*v0.631*), whereas the slowest measured execution time can be found in an older release (*v0.5.2*), which however is not the oldest release in this case study.

Two subsequent releases contained configurations which could not successfully be measured. When executing those configurations, the program was interrupted due to a software bug. In release *v0.550*, 45 configurations could not be measured successfully, in release *v0.551*, 28 configurations lead to this problem. The 28 configurations

Table 4.1: Basic information about measured experiment results of the three case studies *lrzip*, *GNU XZ* and *libvpx*. There are metrics about the measurement space, the minimum, and the maximum measured execution times.

Metric	**lrzip**	**GNU XZ**	**libvpx VP8**
Number of Releases	38	14	11
Number of Config.	186	178	488
Measured Variants	7,068	2,492	5,368
Max. Time [ms]	127,579.33	30,399	120,322.33
Release	*v0.5.2*	*v5.2.3*	*v1.4.0*
Min. Time [ms]	810.71	14,151	930.66
Release	*v0.631*	*v5.0.0*	*v0.9.6*

of the second release are a strict subset of the 45 failing configurations in the preceding release. We found no obvious similarity in the selected features of the failing configurations which may have caused the issue.

GNU XZ

The 2492 individual measurements for *GNU XZ* resulted in a range of measured execution times between *14,151ms* and *30,399ms*. The slowest measured value was found in release *v5.2.3*, which is the second youngest release we investigated. The fastest performance was measured in release v5.0.0, which is the oldest release in this study. The stated values can be found in Table 4.1.

libvpx VP8

For the 11 investigated releases of *libvpx VP8* we measured 5368 individual performance values, as shown in Table 4.1. The fastest performance was found in release *v0.9.6* with a value of *930.66ms*. The slowest performance was measured in release *v1.4.0* with a mean execution time value of *120,322.33ms*. Release *v0.9.6* is the oldest release in our investigation, where as release *v1.4.0* is situated in the middle of our rlease list.

4.2 RQ1: Overall Execution Time Evaluation

The first research question (Section 3.1) deals with performance changes between two consecutive releases and overall trends we can find over time. Our approach for addressing this issue is to investigate the mean execution time across all configurations per release as described in Section 3.2.2.1. Therefore, we search for pairs of consecutive releases, where the mean execution time across all configurations changed more than 20%. We call those changes *substantial*.

In Figure 4.1, we show the mean execution time value per release across all configurations AVG_R for all case studies. The blue line indicates the trend over time of the mean execution time while the grey area illustrates a channel of 20% above

and below the average execution time. Whenever the blue line exceeds the grey area, a *substantial* performance change between the two regarded releases took place ($CR_{R,R+1} > 20\%$). In case of *lrzip*, this happens exactly three times (*v0.45 - v0.46*, *v0.520 - v0.530*, *v0.543 - v0.544*). For *GNU XZ*, we did not detect this event at all and for *libvpx VP8*, we found one occurrence of a *substantial* performance change (*v1.2.0 - v1.3.0*).

As stated in Section 3.2.2.1, we want to make statements about the strength of performance changes between the identified consecutive releases with *substantial* changes. In Table 4.2, we show the exact change rates of the previously determined releases with change rates above 20%. The height of the percentage value indicates the strength of the corresponding performance change from one release to the other. For the case study *lrzip*, we can see two performance improvements of around *57%* and one performance decline of *38%*. For *GNU XZ*, there was no substantial performance change at all. *libvpx VP8* showed one performance enhancement by *58%*. All in all, we found three *outstanding* releases for *lrzip* and one for *libvpx VP8*.

Table 4.2: Change rates of *substantial* mean execution time changes for all case studies.

Case Study	**Releases**	$\mathbf{CR_{R,R+1}}$
lrzip	v0.45 - v0.46	-56.5%
	v0.520 - v0.530	-56.8%
	v0.543 - v0.544	+37.5%
GNU XZ	-	-
libvpx	v1.2.0 - v1.3.0	-58.3%

Discussion

The results shown above indicate that we are able to make statements on the overall performance trend between different releases of configurable software systems. Therefore, we did not only take specific configurations into account but had a look at a wide variety of different configurations of the configurable software systems. By averaging the execution times across all configurations for one release, we get a first impression of how the performance of that particular release behaves in general. If we had only measured a few specific configurations, we would not have been able to make statements on the overall release. We found only a few *substantial* changes in our case studies, whereas most of those changes appeared in older releases of the case studies. The pairs of consecutive releases, we did not consider having a *substantial* performance change may have specific configurations which underly performance inprovements or reductions, which we then cannot detect when calculating the mean value. Also, the releases with *substantial* performance changes may have many configurations which did not change much regarding the execution time and the same time this release may contain only one specific configuration which has a massive performance in- or decreasement. This then can be sufficient to classify the whole release as *substantial*.

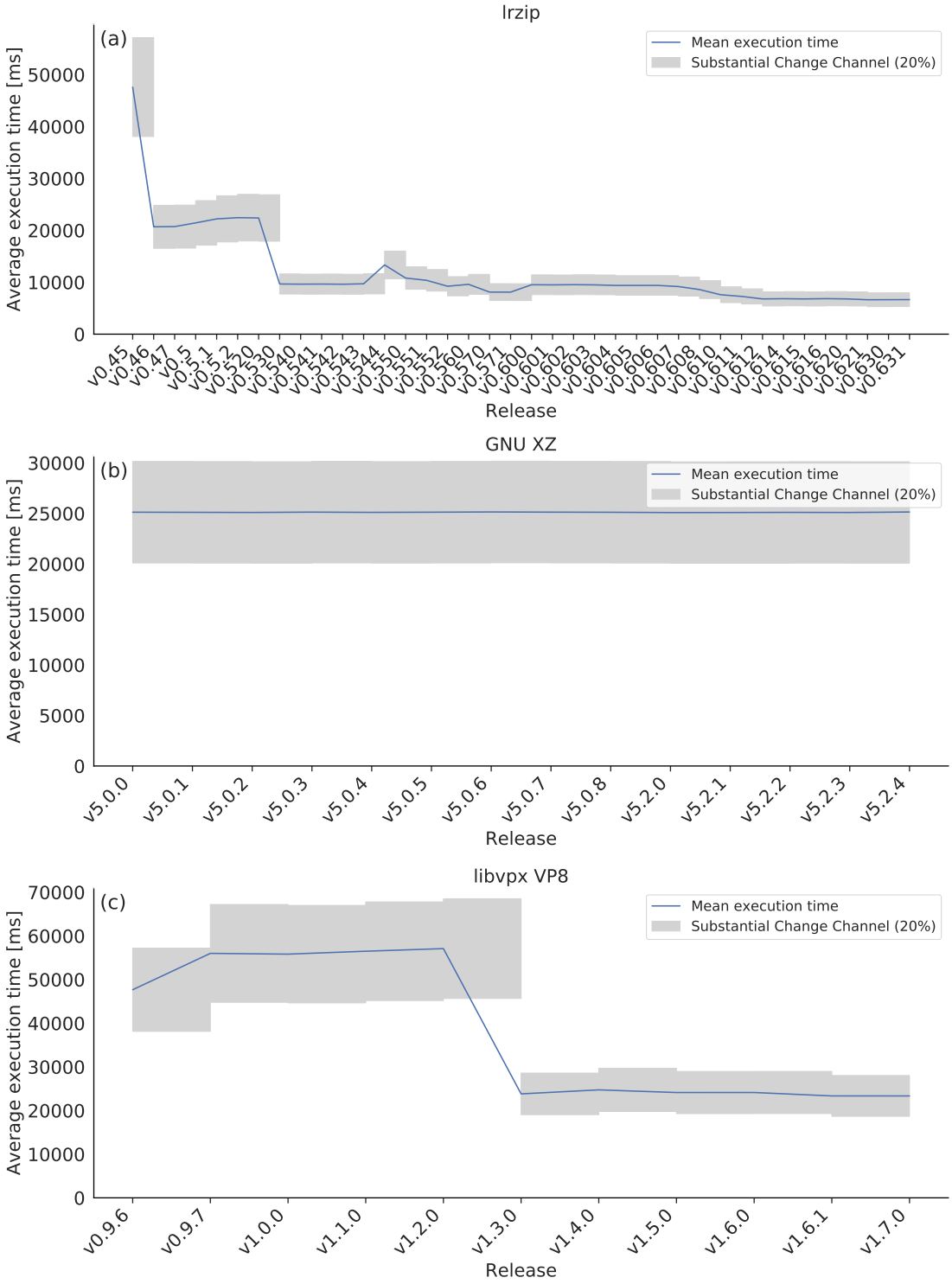

Figure 4.1: Average Execution times per release (AVG_r) across all configurations for case study (a) *lrzip*, (b) *GNU XZ*, (c) *libvpx VP8*. The mean execution time across all considered configurations is illustrated by the blue line respectively. The grey area sketches the *substantial* change channel of 20% around the mean values.

Summary: We found three *outstanding* releases for *lrzip* with the change rates −56.5%, −56.8%, and +37.5%, and one for *libvpx VP8* (−58.3%). The case study *GNU XZ* had no substantial changes in the mean execution time.

4.3 RQ2: Configuration Execution Time Evaluation

In contrast to *RQ1*, we now will take a more in-depth look into the individual configurations of each case study in the following. We want to know if there are specific configurations with performance changes and how strong these performance changes are. Therefore, we will divide this section into three parts for the three different case studies in this work.

lrzip

A first impression of the behavior of all configurations in all measured releases can be gathered in Figure 4.2. Each row represents one release, each column stands for one configuration. Therefore, each cell of the heatmap describes the average of three measured execution times for the corresponding configuration and revision. Low execution times are illustrated with a light yellow color (■), high execution times with a dark red color (■). The failing configurations in the releases *v0.550* and *v0.551* are represented by the black colored cells (■).

When examining the heatmap from Figure 4.2, we can identify the big range of execution time values which was already stated in Table 4.1. The three *substantial* performance changes between consecutive releases we detected in Section 4.2 can also be found in this graph. The first *substantial* performance change was identified between release *v0.45* and *v0.46* and on closer inspection, can be seen in the heatmap. The cells with a deep red color for release *v0.45* (e.g., around the configuration *112*) change to a brighter tone in the row for release *v0.46*. Most configurations change there color to a brighter (faster) tone when comparing the rows for release *v0.520* and *v0.530*. Between release *v0.543* and *v0.544*, the tone gets a little bit darker again, which indicates an overall performance decline. Hence, all *substantial* performance changes declared in Table 4.2 can be seen in the heatmap.

On further inspection of the heatmap's columns in Figure 4.2, we can see configurations or groups of configurations which have nearly the same color throughout several releases or throughout the whole regarded time span. This means that these configurations have a similar execution time throughout the corresponding releases. The configurations inside one release can have considerable differences in execution times. The fastest configurations can be found in the magnitude of *1,000ms*, whereas the slowest configurations in the same release can have execution times in the magnitude of *100,000ms*. The slow configurations are slower by a factor of around 100.

We calculated the change rates ($CR_{c_r,cr+1}$) in execution times for every configuration across subsequent releases. The maximum increase in execution time was found between release *v0.621* and v0.630, where one configuration performed about 124% slower in the younger release. The highest decrease in execution time was

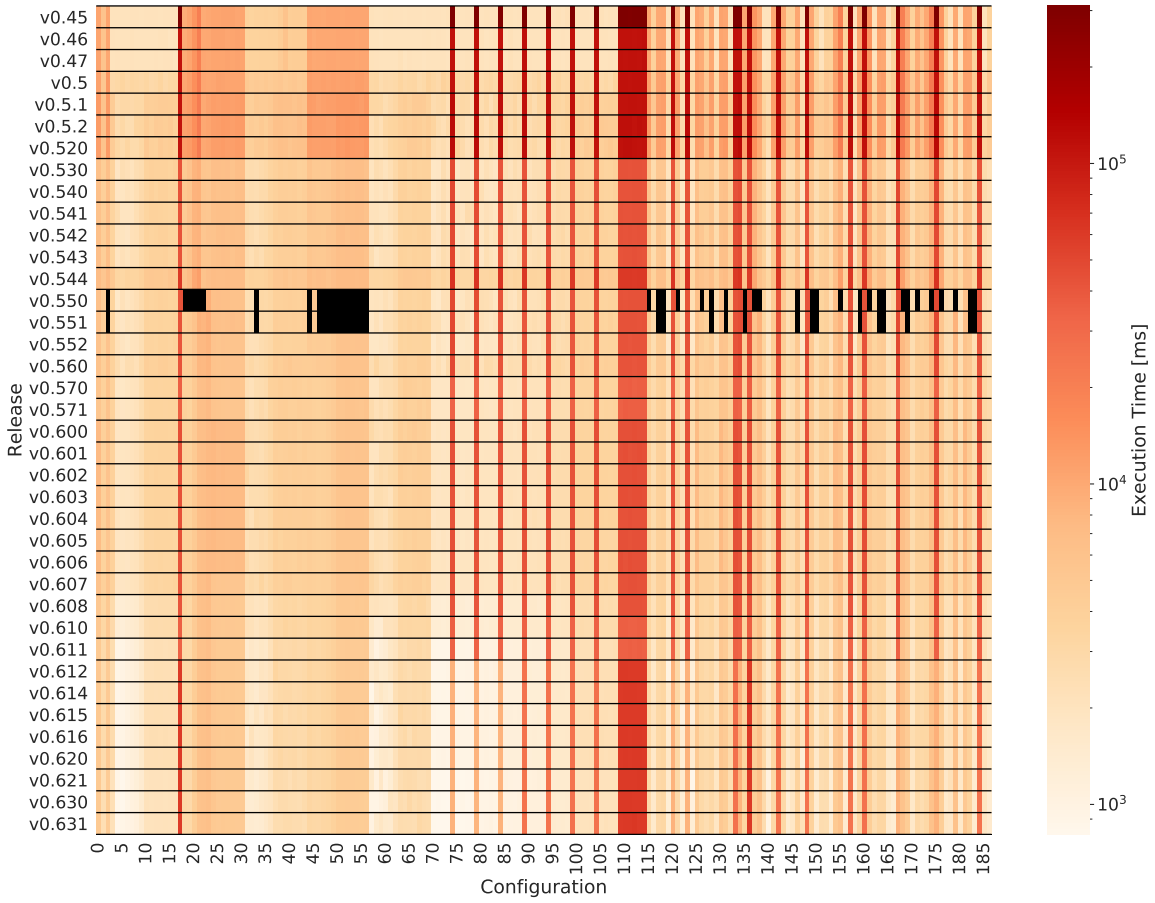

Figure 4.2: The execution time measurement values for different releases of *lrzip*. The y-axis consists of all measured releases, the x-axis of all measured configurations. Each cell is colored according to the measured mean execution time value. The legend on the right provides the colors for the corresponding time values in milliseconds. Failed measurements are colored black (■).

found between release *v0.611* and *v0.612*. The affected configuration's execution time reduced by approximately 75%. Every release contains both configurations which performed better and configurations which performed worse compared to the preceding release.

The heatmap in Figure 4.3 illustrates the previously described changes in execution times. Like in the previous heatmap, rows stand for different releases, columns for different configurations. Each cell represents the change rate of the respective configurations comparing the release of the row and the release of the previous row. A white colored cell means that the configuration had approximately the same execution time in the two compared releases. Red cells (■) describe configurations, which performed slower compared to the previous release. In contrast, blue cells (■) illustrate better performing configurations. The first row (release *v0.45*) contains only white cells because there is no preceding release to compare with.

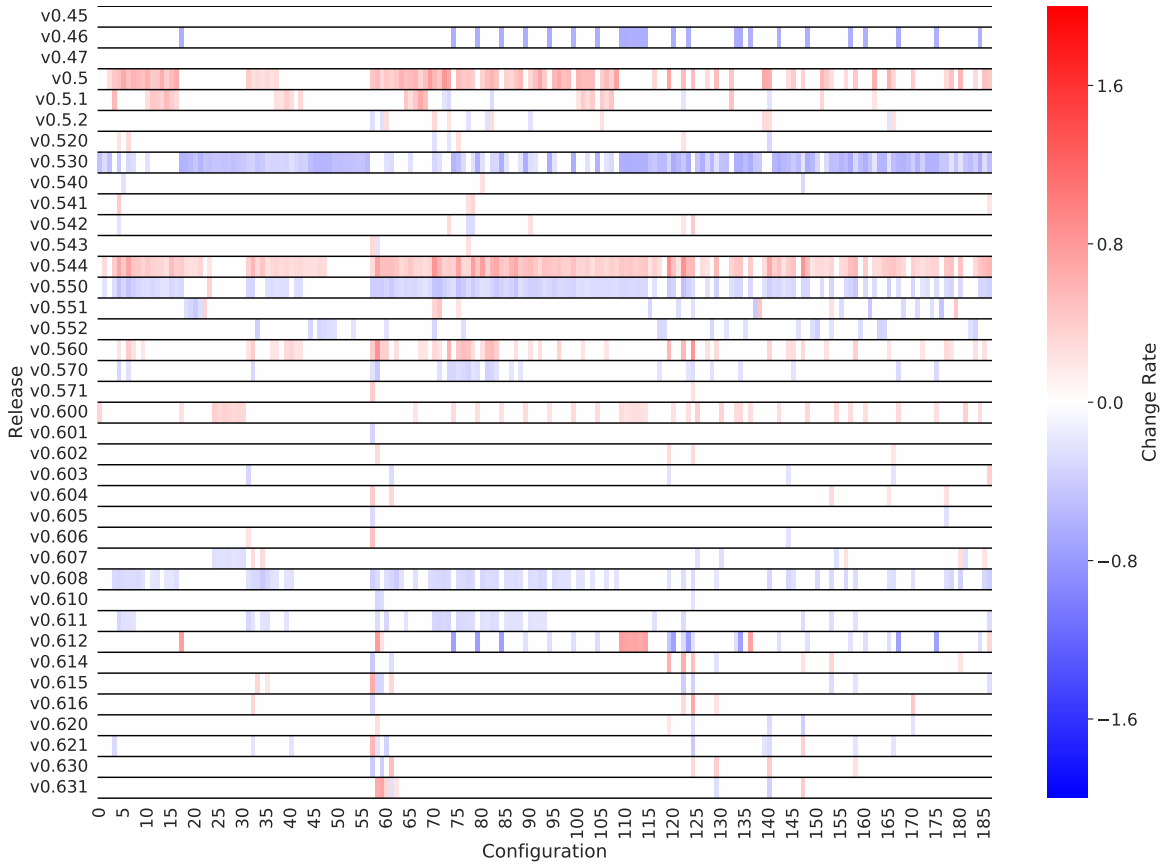

Figure 4.3: Change rates of configuration execution times comparing subsequent releases of *lrzip*. The y-axis consists of all measured releases, the x-axis of all measured configurations. Each cell is colored according to the calculated change rate regarding the preceding release. The legend on the right provides the colors for the corresponding change rates. Blue color tones (■) indicate performance improvements, whereas red colors (■) stand for slower performance.

Since there were some configurations in two releases, we could not measure due to a software bug, there is an intermediate step for calculating the change rates in those cases. Therefore, we took the last working release for that particular configuration into consideration to compute the corresponding change rate.

We used the previously calculated change rates between subsequent releases to find configurations which have a *substantial* change rate between two specific releases. As described in Section 3.2.2.2, we consider the change rate as a *substantial* deviation if the absolute value of the change rate for a particular configuration for two subsequent releases is higher than 20%. Exemplarily, Figure 4.3 compares the measured execution times of the releases *v0.560* and *v0.570*. One point in the plot stands for one configuration and describes the particular measured execution times for the two compared releases. If the dot is red, the two execution times have a *high* deviation.

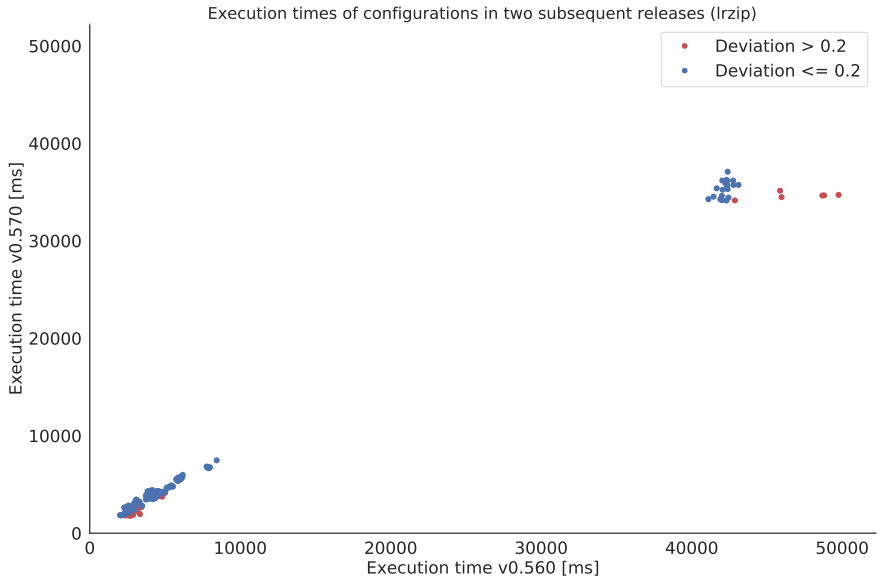

Figure 4.4: Comparing configuration execution times of releases *v0.560* (x-axis) and *v0.570* (y-axis) for case study *lrzip*. Each dot illustrates the execution times of one configuration in both consecutive releases. Red dots indicate a *substantial* performance change for the particular configuration.

Configurations, which have no *high* deviations can be found near the diagonal of the graph and are colored blue.

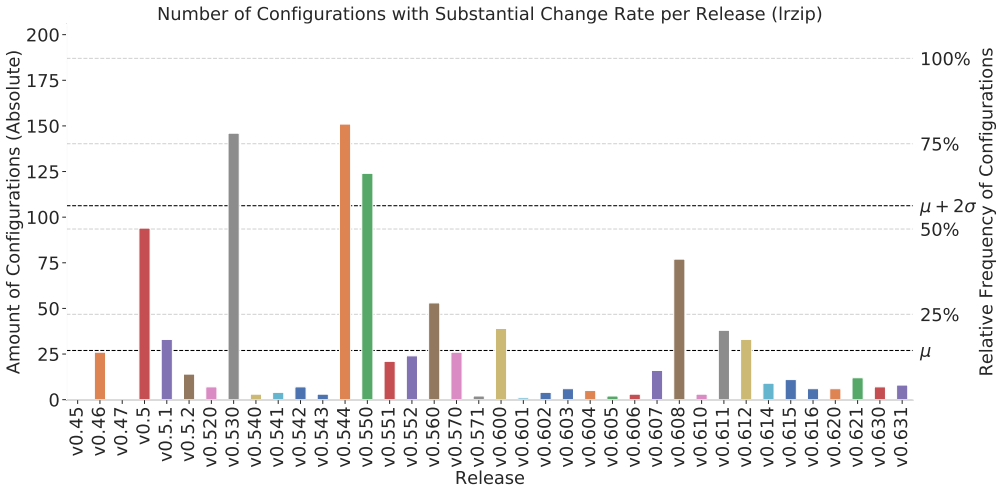

Figure 4.5: Number of configurations with *substantial* performance change per release for case study *lrzip*. There are two dashed black lines which illustrate (1) the mean value of amounts μ and (2) the threshold for *outstanding* releases $\mu + 2 \cdot \sigma$.

For every measured release, we counted the number of configurations which have a *substantial* performance change compared to the preceding release (CC_r). Figure 4.5

shows the amount of those configurations with *substantial* change. The dashed lines in the plot depict the relative amount of the overall measured configurations. There are some releases which exceed half or even three-quarters of the total amount of configurations (e.g., *v0.530*, *v0.544*). The plot additionally shows the mean μ and the threshold $\mu + 2 \cdot \sigma$, which was described in Section 3.2.2.2. According to the threshold, there are three *outstanding* releases for *lrzip* regarding this research question.

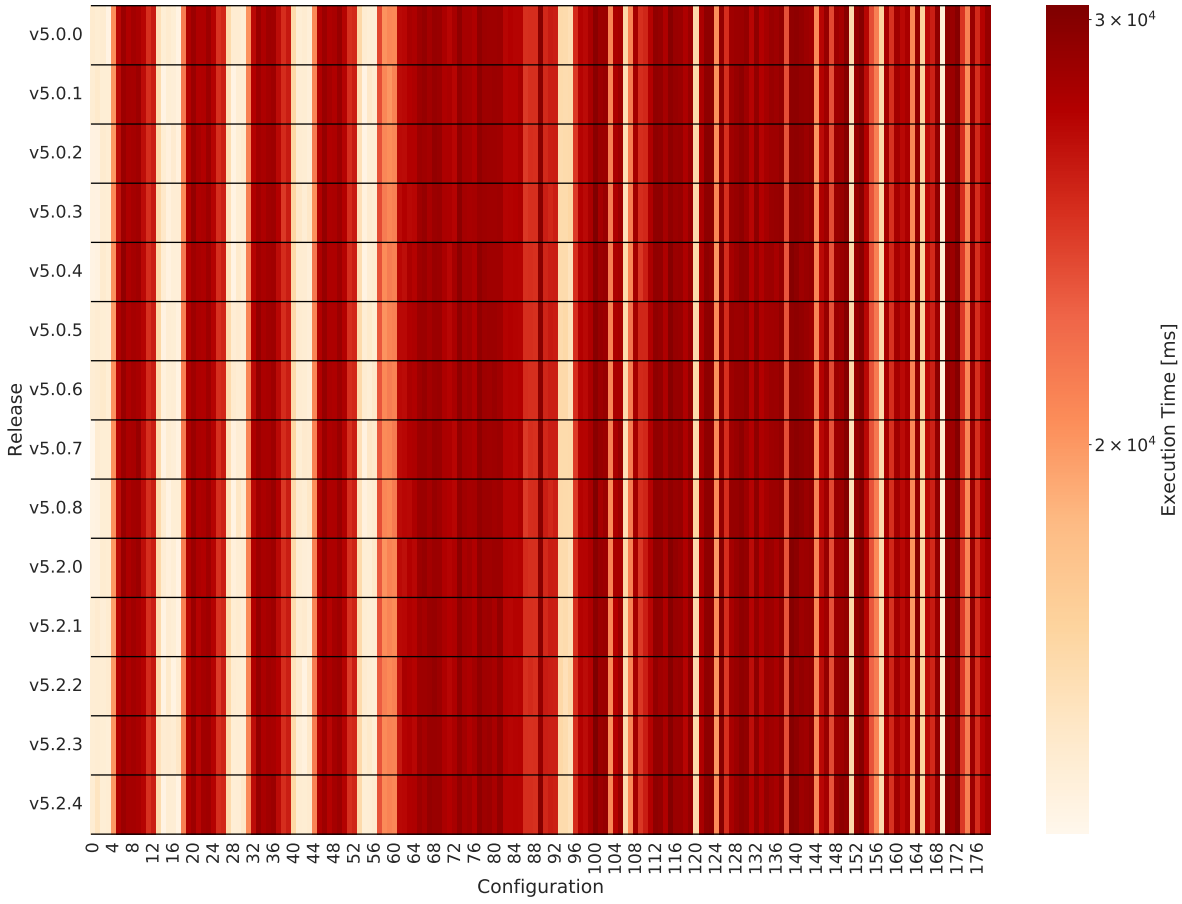

Figure 4.6: The execution time measurement values for different releases of *GNU XZ*. The y-axis consists of all measured releases, the x-axis of all measured configurations. Each cell is colored according to the measured mean execution time value. The legend on the right provides the colors for the corresponding time values in milliseconds.

GNU XZ

Similar to the previous case study, Figure 4.6 shows the pure execution time values of all measurements for *GNU XZ*. Analogous to *lrzip* the configurations differ in their execution time values. This time, they range between around *15,000ms* and *30,000ms* (see also Table 4.1). Again, columns have the same color tone throughout

the entire time span of the case study. In the heatmap, we cannot detect specific releases which stand out.

The change rates of the execution times for single configurations between subsequent releases range between -7% (from release *v5.2.3* to release *v5.2.4*) and +6% (from release *v5.0.3* to release *v5.0.4*). The heatmap in Figure 4.7 shows all change rates of all configurations for all releases. Similar to the previous case study, the first release in the first row is colored white completely because there is no release to compare with. The heatmap already indicates that we did not find configurations with *substantial* performance changes. As a result, we also did not find any *outstanding* releases for *GNU XZ* regarding this research question.

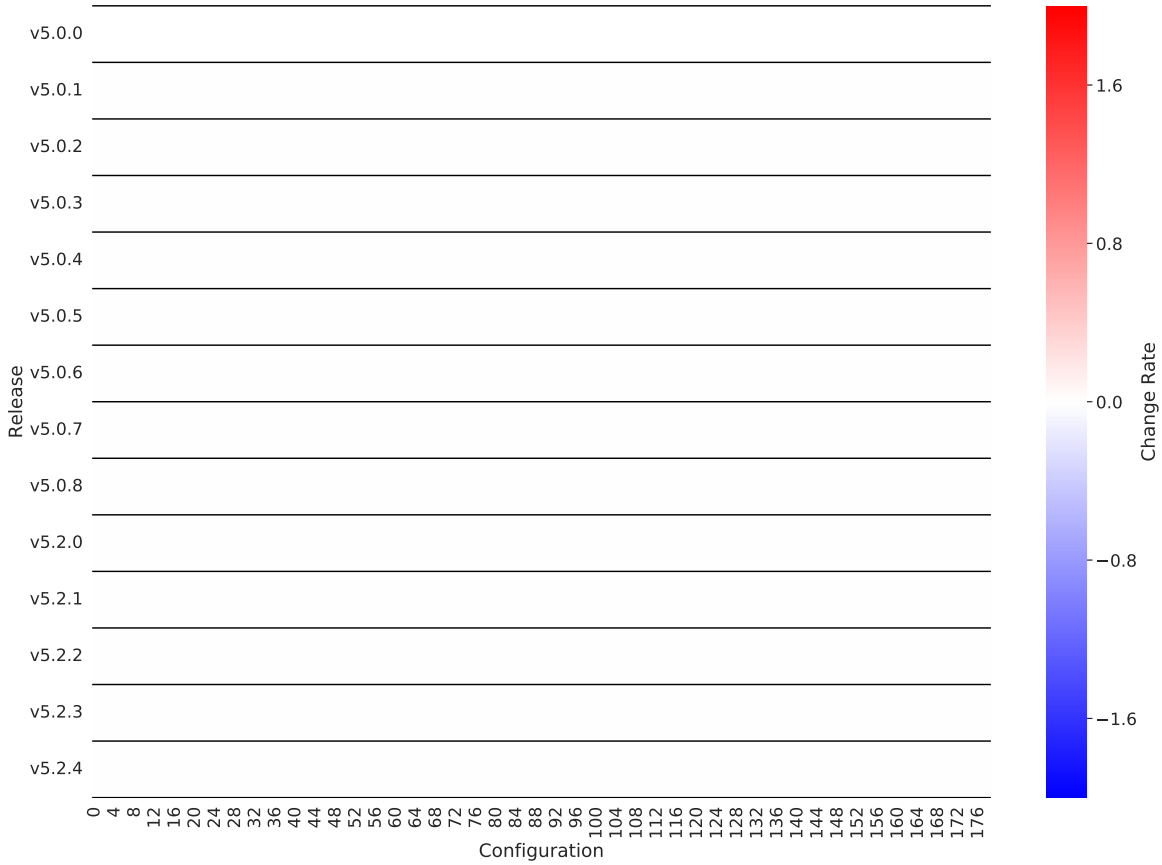

Figure 4.7: Change rates of configuration execution times comparing subsequent releases of *GNU XZ*. The y-axis consists of all measured releases, the x-axis of all measured configurations. Each cell is colored according to the calculated change rate regarding the preceding release. The legend on the right provides the colors for the corresponding change rates. Blue color tones (■) indicate performance improvements, whereas red colors (■) stand for slower performance.

libvpx VP8

Finally, we present the results for *RQ2* for our third case study *libvpx VP8*. Figure 4.8 sketches the execution times for all configurations of all considered releases.

The heatmap has the same structure and color schemes as the equivalent illustrations of the previous case studies. There is one noticeable leap in the execution times between release *v1.2.0* and *v1.3.0* which affects nearly all configurations. This perfectly matches the *substantial* mean performance change across all configurations found for the first research question (see Table 4.2 and Figure 4.1). The execution times of all configurations in all releases lie in a range between around *1,000ms* and *120,000ms* which is roughly a difference by the factor 100.

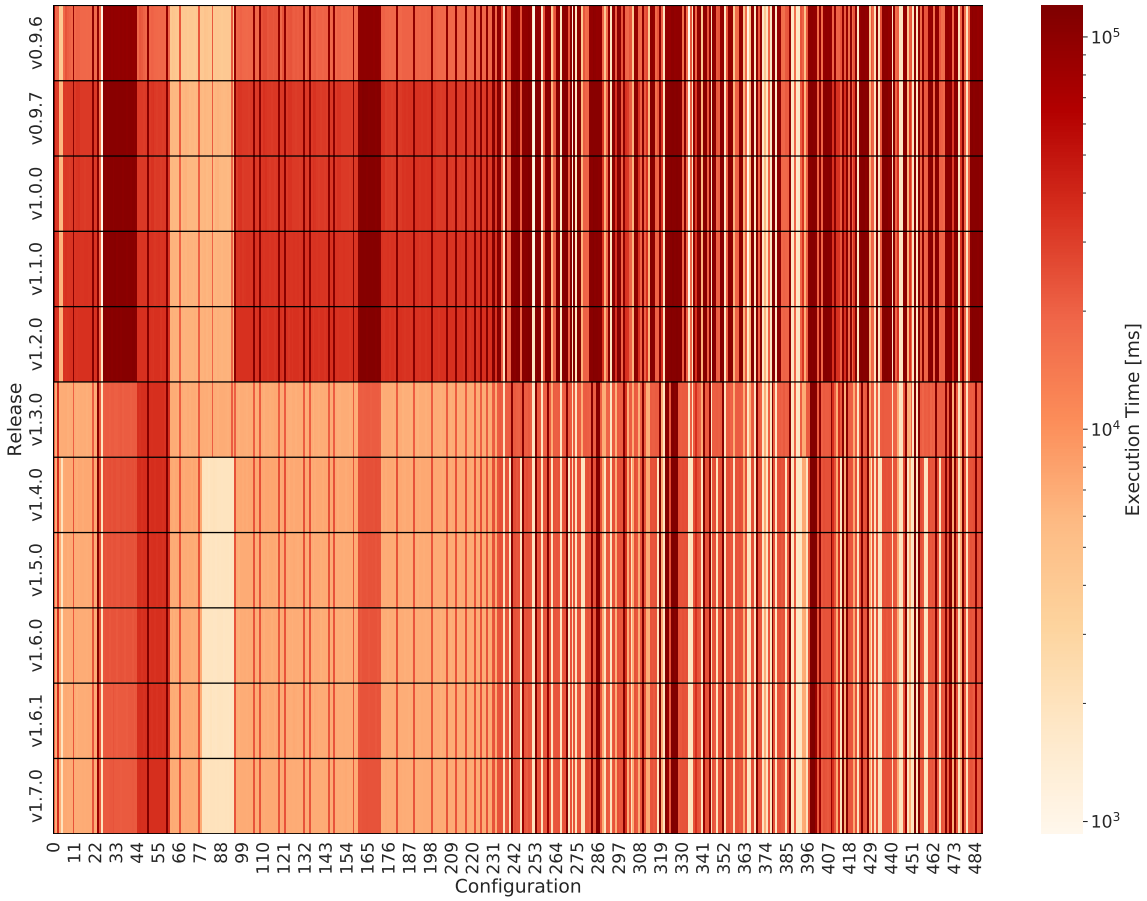

Figure 4.8: The execution time measurement values for different releases of *libvpx VP8*. The y-axis consists of all measured releases, the x-axis of all measured configurations. Each cell is colored according to the measured mean execution time value. The legend on the right provides the colors for the corresponding time values in milliseconds.

In Figure 4.8, we again can observe configurations or groups of configurations, which have the same color throughout multiple releases. Especially after the noticeable color change between *v1.2.0* and *v1.3.0*, we can notice that groups of configurations behave the same after the color change. For example, there is a group of configurations (*162 - 171*) which has a very high execution time in the first releases (dark red color). From release *v1.3.0* on, they have a faster execution time (lighter red color) but they still are clearly identifiable as a group afterwards.

As already described in Section 4.3, we computed the change rates in execution times for every configuration for subsequent releases. The heatmap in Figure 4.9 displays the results for those calculations. There are two releases which stand out (*v0.9.7* and *v1.3.0*). Both show a significant amount of configurations with a performance change of more than 50% compared to the preceding release. Release *v0.9.7* has a lot of configurations which performed slower than in the previous release. In contrast, release *v1.3.0* has performance improvements in lots of configurations. Release *v1.3.0* was already identified for having a *substantial* performance change across all configurations. In Figure 4.1, we can see an increase of the overall mean performance for release *v0.9.7*. We didn't classify this rise as *substantial* because it was lower than 20%.

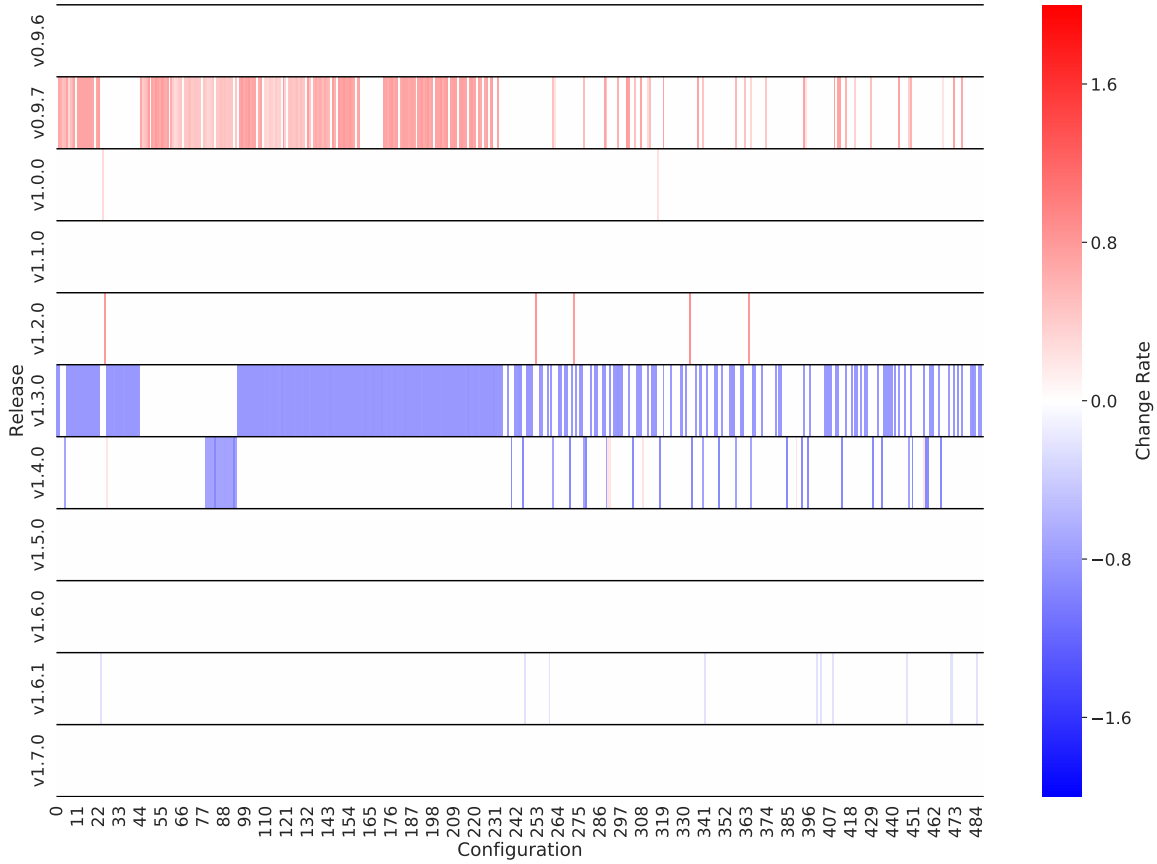

Figure 4.9: Change rates of configuration execution times comparing subsequent releases of *libvpx VP8*. The y-axis consists of all measured releases, the x-axis of all measured configurations. Each cell is colored according to the calculated change rate regarding the preceding release. The legend on the right provides the colors for the corresponding change rates. Blue color tones (■) indicate performance improvements, whereas red colors (■) stand for slower performance.

Counting the configurations per release whose change rates exceed 20% leads to the results in Figure 4.10. As already hinted above, the two releases *v0.9.7* and *v1.3.0*

are the ones which have the highest amount of configurations with *substantial* change rates. For those two releases, the sum of the configurations with *substantial* change ranges around 50% of all measured configurations. However, we again calculated the mean μ and the threshold $\mu + 2 \cdot \sigma$, which is only exceeded by the release *v1.3.0*. As a result, we only found one *outstanding* release for *libvpx* regarding this research question.

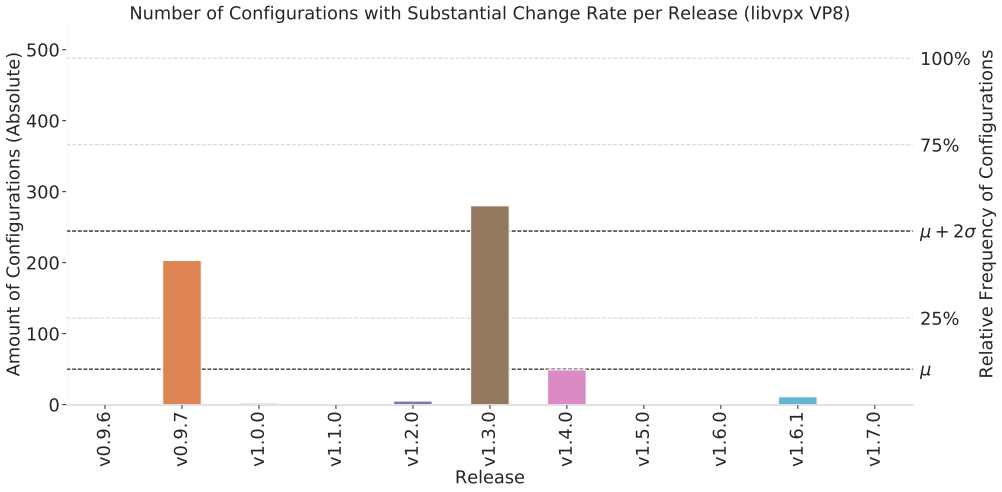

Figure 4.10: Number of configurations with *substantial* performance change per release for case study *libvpx VP8*. There are two dashed black lines which illustrate (1) the mean value of amounts μ and (2) the threshold for *outstanding* releases $\mu + 2 \cdot \sigma$.

Discussion

The second research question aimed at breaking down the overall performance changes found in *RQ1* into a more sophisticated sight on individual configurations. We saw that the execution times for the configurations in one release might differ by a huge factor. In two case studies, the slowest configuration was 100 times slower than the fastest one. Therefore, it makes sense to investigate the individual configurations for configurable software systems and not just to look at one particular configuration or averaging the execution times across all configurations. For example, it might be the case that the average execution time across all configurations gets faster from one release to the next one, but one user might be interested in the one configuration which performs slower compared to the previous release.

We also concluded that the results of *RQ1* and *RQ2* can be congruent, meaning that the same releases show peculiarities which match. Some releases of the investigated case studies have a *substantial* performance change and at the same time a high fraction of configurations which in itself have *substantial* changes. Furthermore, it is possible to have releases which did not attract our attention in *RQ1* but had many configurations with *substantial* changes. Of course, it is possible that those configurations compensate each other when calculating the mean value for the complete release. In contrast, we can have a release with a *substantial* performance change

across all configurations, but this change is only caused by one or a few individual configurations which had a considerable performance change from one release to the next one.

After seeing the results for *RQ2*, especially groups of configurations which behave in the same manner throughout the entire release time span, we assume that individual configuration options cause the previously described effects. The configurations which form a group of the same behavior may have a similar set of selected configuration options which then influence the execution times in the same way for the affected configurations. This is the reason why we will go a step further and have a more in-depth look inside the measured configurations and the respectively selected configuration options.

Summary: The case studies *lrzip* and *libvpx VP8* show configurations with a *substantial* change ($> 20\%$) in the execution time between two consecutive releases. We did not find such behavior for *GNU XZ*. Some results align with the ones from *RQ1* and others provide new insights concerning individual configurations. We found three *outstanding* releases for *lrzip* and one for *libvpx VP8* regarding this research question.

4.4 RQ3: Learning Performance-Influence Models

Answering *RQ3.1* and *RQ3.2* require the computation of performance-influence models as we stated in Section 3.2.2.3. We described in Section 3.2.2.3 that we want to fit a performance-influence model to all possible term combinations up to a certain degree of interaction. We used the incremental learning approach for performance-influence models to get the maximum degree of option interactions which we consider relevant.

Table 4.3 shows our decisions for the maximum degree of interaction and the resulting number of terms as well as some metrics about the different error rates, we came across while learning the performance-influence models. It should be noted that we got very small error rates for *lrzip* in the magnitude of 10^{-14} and relatively high error rates for *libvpx VP8* in the magnitude of 10^{-1}. *GNU XZ* ranges in between with error rates around 10^{-4}.

Table 4.3: Basic information about learned performance-influence models for the case studies *lrzip*, *GNU XZ*, and *libvpx VP8*.

Metric	**lrzip**	**GNU XZ**	**libvpx VP8**
Max. Degree of Interaction	3	3	4
Number of Terms	270	267	3897
Min. Error Rate	9.4×10^{-15}	2.2×10^{-5}	1.6×10^{-2}
Release	v0.570	v5.2.3	v1.3.0
Max. Error Rate	7.0×10^{-14}	4.5×10^{-4}	0.27
Release	v0.45	v5.0.8	v1.0.0
Avg. Error Rate	1.6×10^{-14}	2.2×10^{-4}	0.16

4.4.1 RQ3.1: Identifying affected configuration options

After computing the performance-influence models as described in Section 4.4, we can present our results for *RQ3.1*. In Section 3.2.2.3, we explained our approach for identifying and counting terms with *substantial* change in the corresponding coefficient in the performance-influence model.

Since we fitted the performance-influence models to a high number of terms (e.g., 3,897 for *libvpx VP8*, see Table 4.3), there are naturally many terms which have a coefficient of 0 or near 0. There is a high probability for very small terms near 0 for finding *substantial* changes, because there is no need for a drastic absolute change in the influence coefficient for a resulting deviation greater than 20 percent. This is the reason why we set all coefficients which have a value smaller than 1,000 (1 second) to zero. This way, we prevent small coefficients from distorting the results.

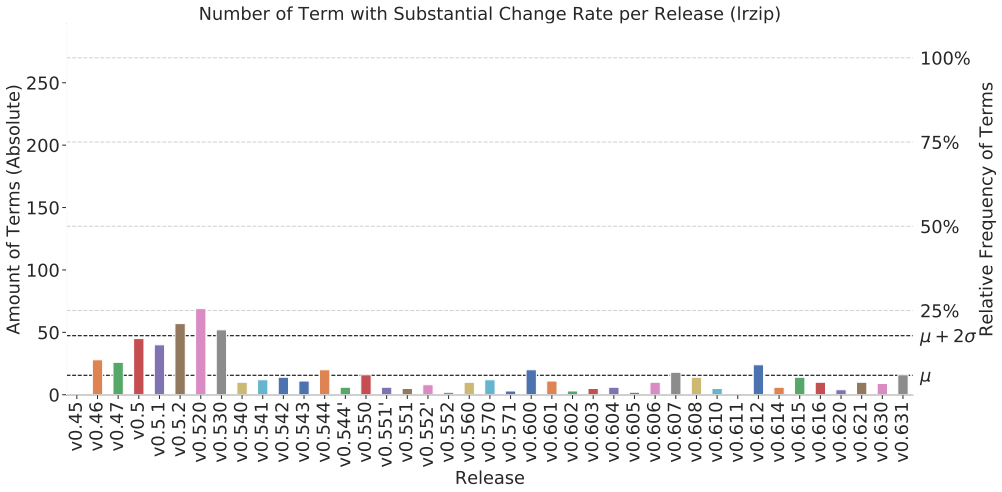

Figure 4.11: Number of terms with *substantial* change in influence per release for case study *lrzip*. There are two dashed black lines, which illustrate (1) the mean value of amounts μ and (2) the threshold for *outstanding* releases $\mu + 2 \cdot \sigma$.

In Figure 4.11, Figure 4.12, and Figure 4.13, we illustrate the number of terms with *substantial* change in the coefficient value for the corresponding case studies. The plots show that almost every release of every case study has terms with *substantial* changes. The plots also show black dashed lines for the mean amount μ and the threshold for *outstanding* releases $\mu + 2 \cdot \sigma$. Releases with a single quote in the name represent intermediate performance-influence models which were necessary because of certain configurations which we could not measure because of software bugs.

For the case study *lrzip*, the highest fraction of terms with *substantial* changes is around 25%, and we found three *outstanding* releases (*v0.5.2*, *v0.520*, *v0.530*). Release *v0.530* already was identified as an *outstanding* release regarding *RQ1* and *RQ2*. *GNU XZ* hast roughly the same number of terms with *substantial* changes in every release. There is no *outstanding* release as it was not in the previous investigations, too. The plot for case study *libvpx VP8* shows three releases with higher amounts of terms with *substantial*. According to the threshold $\mu + 2 \cdot \sigma$,

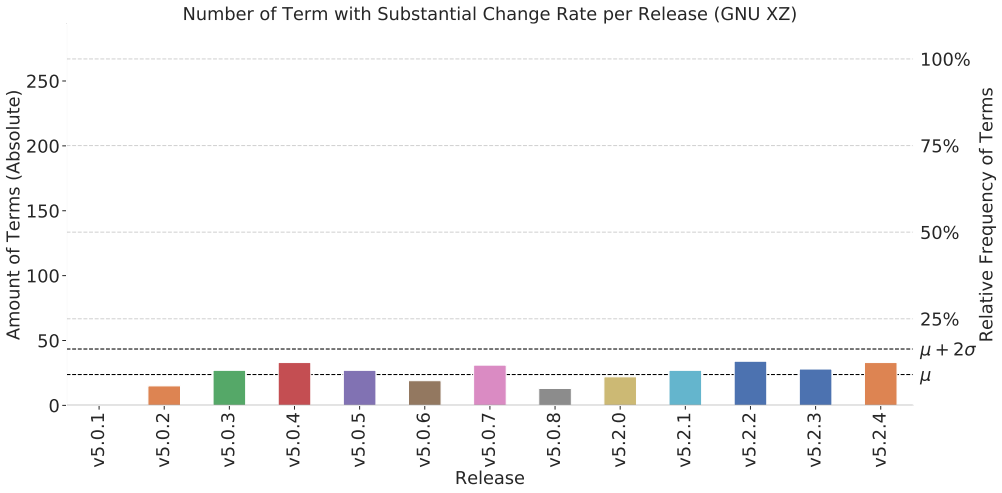

Figure 4.12: Number of terms with *substantial* change in influence per release for case study *GNU XZ*. There are two dashed black lines, which illustrate (1) the mean value of amounts μ and (2) the threshold for *outstanding* releases $\mu + 2 \cdot \sigma$.

release *v1.3.0* is an outstanding release regarding this research question. It has about 75% of terms with *substantial* change which also matches the previous results for *outstanding* releases (see Figure 4.1). Also, *v0.9.7* as a higher number of terms with *substantial* changes in coefficients. This is similar behavior to what we saw in Figure 4.1. The higher number of *substantial* changes for release *1.4.0* cannot be retrieved in the previous investigations. However, *v0.9.7* and *v1.4.0* are no *outstanding* release according to our threshold.

Discussion

We showed that we were able to fit performance-influence models to a given set of predefined terms for the different releases of our three case studies which afterwards, can be compared. It is striking that we found terms with *substantial* change for almost every release in our investigation. This can partly be explained by the high number of terms we considered in this approach. Since the computed performance-influence models are based on the measured execution times which have a natural fluctuation, it is obvious that also the performance-influence models must have some fluctuation. Especially for small numbers, the randomized learning process leads to different coefficients which then exceed our threshold for *substantial* changes (20%). Two of our case (*lrzip* and *libvpx VP8*) studies showed *outstanding* releases according to our threshold. Although *GNU XZ* didn't show *outstanding* releases, we are able to make statements about specific terms where the influence coefficient changed significantly.

Based on the results for this research question, we want to make statements about which terms influence the performance of our measurement space the most and how this changes over time. Therefore, we will have a more specific look at the relevance of the terms with *substantial* changes in the regarded configuration space in the next research question.

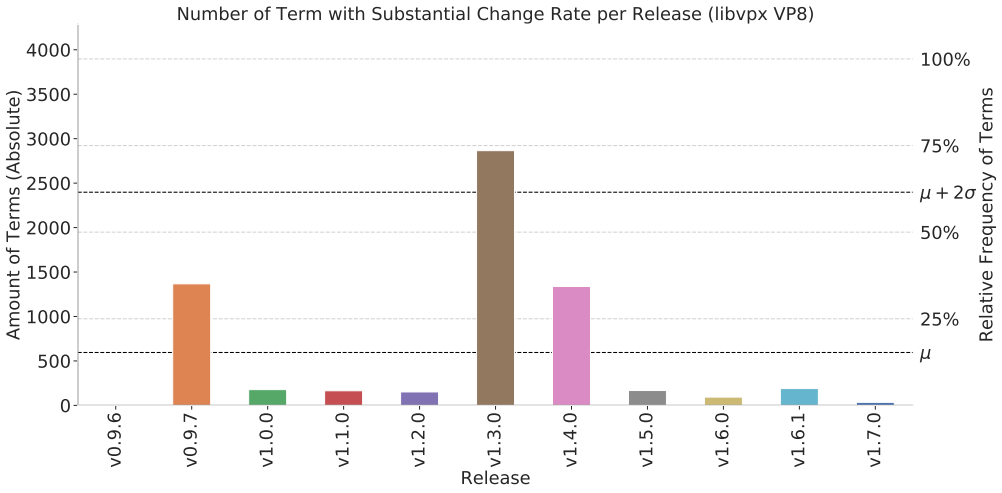

Figure 4.13: Number of terms with *substantial* change in influence per release for case study *libvpx VP8*. There are two dashed black lines, which illustrate (1) the mean value of amounts μ and (2) the threshold for *outstanding* releases $\mu + 2 \cdot \sigma$.

Summary: We found configuration options or option interactions whose influence coefficients of the computed performance-influence models showed *substantial* change rates. Some of those occurences strengthen the results of the previous research questions, the others provide new insights regarding individual configuration options and option interactions. We found three *outstanding* releases for *lrzip* and one for *libvpx VP8* regarding this research question.

4.4.2 RQ3.2: Relevance Ranking of the Configuration Options

The results for *RQ3.1* did not point to specific configuration options or option interactions which had a *substantial* change in the coefficient and at the same time is a relevant term in the configuration space we took into consideration when measuring the execution times. Therefore, we provided a new metric in Section 4.4 to make statements about the relevance of specific configuration options or option interactions. For every term in every release of our case studies, we calculated the relevance factor which we proposed in Section 4.4. Afterwards, we were able to have a look at the terms with the highest relevance factors per release. We considered the top 5 relevance factors, respectively. Figure 4.14, Figure 4.15, and Figure 4.16 show the corresponding trend over time for the respective case study, as well as the overall mean relevance factor μ and the threshold for *dominating* configuration options or option interactions $\mu + 2 \cdot \sigma$.

In every case study, there are some terms which dominate the ranking throughout the entire time span of the considered releases. In *lrzip* for example, the terms *zpaq* and *Root* are the most present ones. For *GNU XZ* only the term *Root* and for *libvpx* the terms *vbr*, *Resize Allowed 0*, *Passes 1* and *Root* have the highest ranking values over time. According to the threshold $\mu + 2 \cdot \sigma$, we found two *dominating* configuration options or option interactions for *lrzip*, one for *GNU XZ* and 101 for

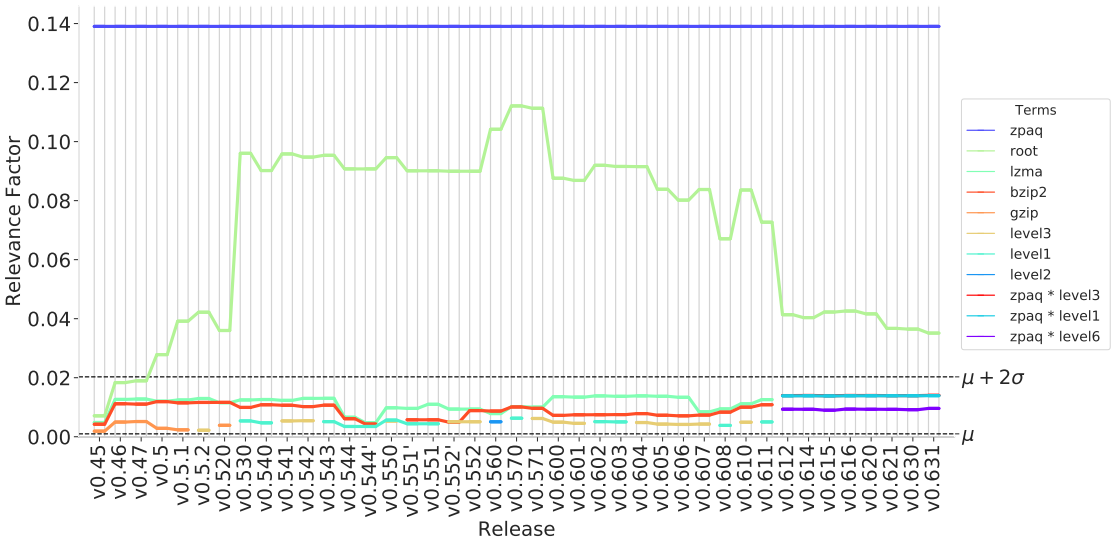

Figure 4.14: Trend over time of the top five terms according to their relevance factor for case study *lrzip*. The y-axis describes the height of the relevance factor, the x-axis contains all releases. For every release, there is a small section in the graph with five thick lines for the five terms with the highest relevance factors. Terms appearing in subsequent releases are connected with a dashed line. There are two dashed black lines, which illustrate (1) the mean value of amounts μ and (2) the threshold for *outstanding* releases $\mu + 2 \cdot \sigma$.

libvpx VP8 (not all visible in the plot). The terms with smaller relevance factors fluctuate in a small range and often switch places with other terms with similar relevance factors. For the case studies *lrzip* and *GNU XZ*, we can see that the only top-ranking term stays at the same level throughout all releases. For *lrzip* the term *root* has a high range of different relevance factors compared to the other terms. Most of the terms in all regarded case studies which rank in the top 5 consist only of one configuration options. There are only some terms for *lrzip* and *libvpx VP8* which represent an interaction between multiple configuration options. The highest degree of interaction we could find in this approach is a second level interactions. Release *v1.3.0* in the case study *libvpx VP8* shows a drop of the relevance factor for most of the terms. For the case study *lrzip*, we can see an increase big increase in the relevance factor for the option *root* between release *v0.520* and release *v0.530*. These to observations match the previous investigations of *outstanding* releases in the previous research questions.

Discussion

The results of *RQ3.2* clearly show that we are able to make profound statements on the relevance of different terms of configurable software. We saw that the relevance changes over time between releases and that there are a few terms which dominate the results of the shown approach.

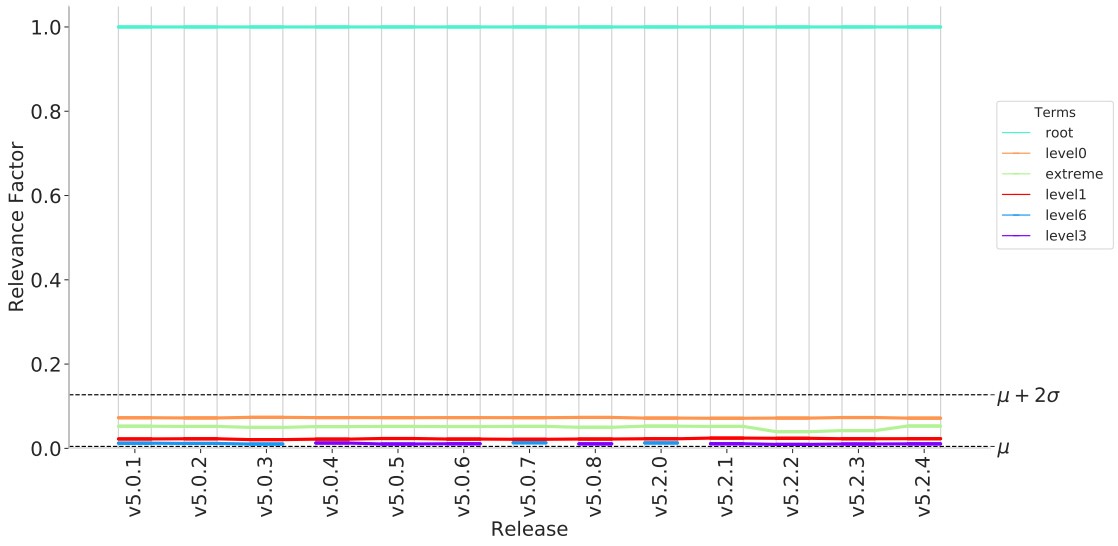

Figure 4.15: Trend over time of the top five terms according to their relevance factor for case study *GNU XZ*. The y-axis describes the height of the relevance factor, the x-axis contains all releases. For every release, there is a small section in the graph with five thick lines for the five terms with the highest relevance factors. Terms appearing in subsequent releases are connected with a dashed line. There are two dashed black lines, which illustrate (1) the mean value of amounts μ and (2) the threshold for *outstanding* releases $\mu + 2 \cdot \sigma$.

For two case studies (*lrzip* and *GNU XZ*), we saw that there is only one term which has the highest rank throughout the complete time span we considered in our investigations. For those two terms, the relevance factor stays exactly the same throughout all releases. The reason for this behavior is the fact, that we normalized the influence coefficients per release. This results in the normalized coefficient to always have the value 1 because the corresponding term with the high ranking has the biggest coefficient regarding the performance-influence model of a specific release. The frequency for each term is the same for all releases because we always considered the same configurations. For *GNU XZ* the term with the highest ranking (*Root*) has the value 1. This is caused by the fact that additionally to the previously described behavior regarding the normalized influence coefficients, the term *Root* has the frequency 1. The artificial configuration option *Root* is part of every configuration by definition.

In the two case studies, where *Root* is not the top ranking term, this exact term has a high range of different relevance factors. This does not necessarily mean that the "basic" influence (e.g., initializing, loading data, etc.) of the corresponding configurable software system fluctuates in the same manner. We assume that the fluctuation of the relevance factor is caused by our approach for preventing shifting in the computed performance-influence models (see Section 4.4). When a few specific terms are not considered during the computation of the performance-influence model, the corresponding influence is shifted to the term *Root*. Thus, the term *Root*

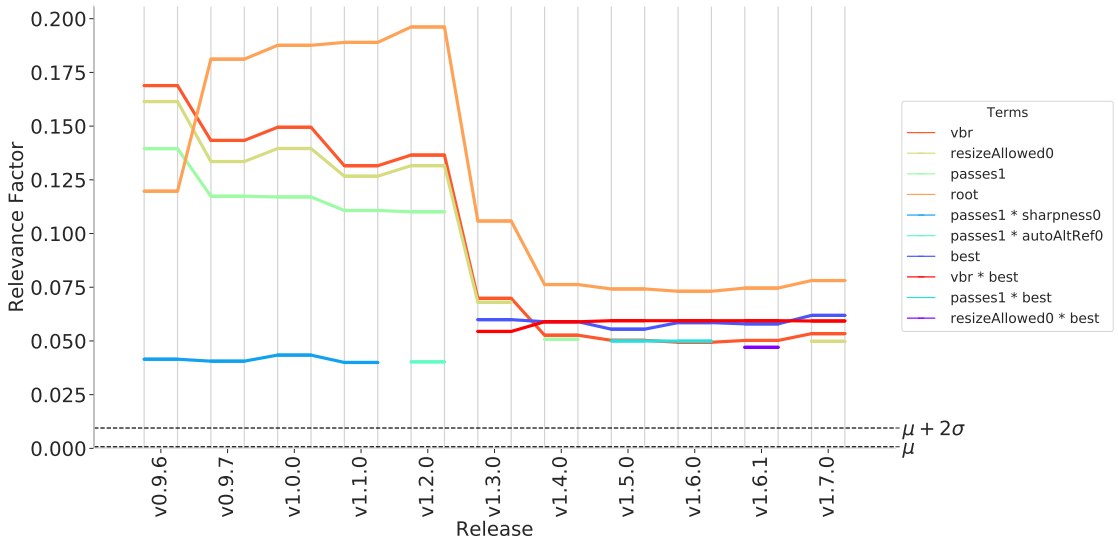

Figure 4.16: Trend over time of the top five terms according to their relevance factor for case study *libvpx VP8*. The y-axis describes the height of the relevance factor, the x-axis contains all releases. For every release, there is a small section in the graph with five thick lines for the five terms with the highest relevance factors. Terms appearing in subsequent releases are connected with a dashed line. There are two dashed black lines, which illustrate (1) the mean value of amounts μ and (2) the threshold for *outstanding* releases $\mu + 2 \cdot \sigma$.

in our case does cause not only the ”basic” influence but also all influences of all removed terms. As a result, this fluctuations for the term *Root* in the case studies can be caused by a high fluctuation in the ”basic” influence of the program or by one or more of the other terms which did not find there way into the performance-influence model.

For the case study *libvpx VP8*, we found 101 *dominating* configuration options or option interactions, which is a quite high number. This may be cause by the fact, that the highest degree of interaction was set to four-way-interactions for *livpx*. This is the reason why we have a vast number of configuration options and option interactions, whereas most of them have a very small relevance factor. Thus, the chosen threshold $\mu + 2 \cdot \sigma$ is below a high number configuration options or option interactions.

We saw that most of the terms appearing in the ranking plots consist of only one configuration option. It seems that individual options have a higher influence on the overall performance of configurable software, then option interactions, which was also shown in [KSK+19].

Summary: Each case study showed only a few configuration options or option interactions which had a high relevance according to the relevance factor. The most relevant terms consist of individual configuration options. We found two *dominating* configuration options or option interactions for *lrzip*, one for *GNU XZ* and 101 for *libvpx VP8*.

4.5 RQ4: Documentation Analysis

The last research question of this work is the investigation of the documentation of the regarded configurable software systems. As already presented in Section 3.2.2.4, we consider both the changelogs which come with every release and the commit messages which are used by the maintainers of the project during development. In both documentation sources, we searched all terms from the performance-influence models of the corresponding case study. This means, we searched single option names as well as multiple option names when dealing with a term which consists of multiple configuration options. The results regarding the investigation of commit messages are shown in Figure 4.17, Figure 4.18, and Figure 4.19. The blue bars depict the number of terms of the corresponding performance-influence model we found in the commit messages between the release of the bar's location and the previous release. The orange bars illustrate how many of those occurences we could retrieve in the performance-influence models, meaning that the coefficient changed *substantially* between the release of the bar's location and the preceding one. If both bars had the same height for one specific release, this means that all terms which appeared in the documentation also had a *substantial* change of their coefficient in the corresponding performance-influence models.

For the case studies *lrzip* and *libvpx VP8*, there are releases where we found matches for more than a half of the terms which appeared in the commit messages. *lrzip* even shows four releases, where we found matches for all terms. The case study *GNU XZ* resulted in no found matches.

Analogous to the investigations of the commit message texts, we considered also the changelogs of the configurable software systems. The way, we count the appearances and matches works the same as before, only the text document is exchanged. Figure 4.20 shows the corresponding results. A similar picture is drawn like before, where we found several matches for *lrzip* and *libvpx*. *GNU XZ* does not provide any applicable changelog. There is only the information, that all changes are documented in the commit messages.

We calculated the match ratios according to our definition in Section 3.2.2.4. For the commit messages, we got 25% for *lrzip*, 21% for *libvpx VP8*, and 12% for *GNU XZ*. The investigations of the changelogs resulted in match ratios of 29% for *lrzip*, and 20% for *libvpx VP8*.

Discussion

In this section, we showed that performance-influence models can be used to detect performance changes for individual configuration options or options interactions. Many terms, which appeared in the documentation provided by the owners of the

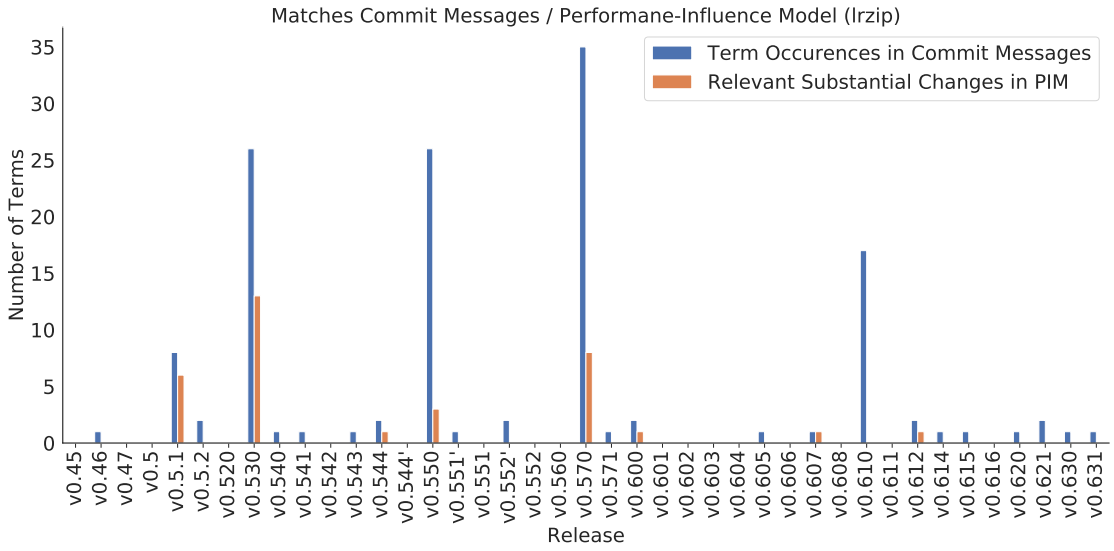

Figure 4.17: Matches of commit messages with performance-influence models of *lrzip*. The blue bars indicate the number of terms found in the commit message texts. The orange bars depict how many of the found terms underlie a *substantial* change in the corresponding performance-influence model.

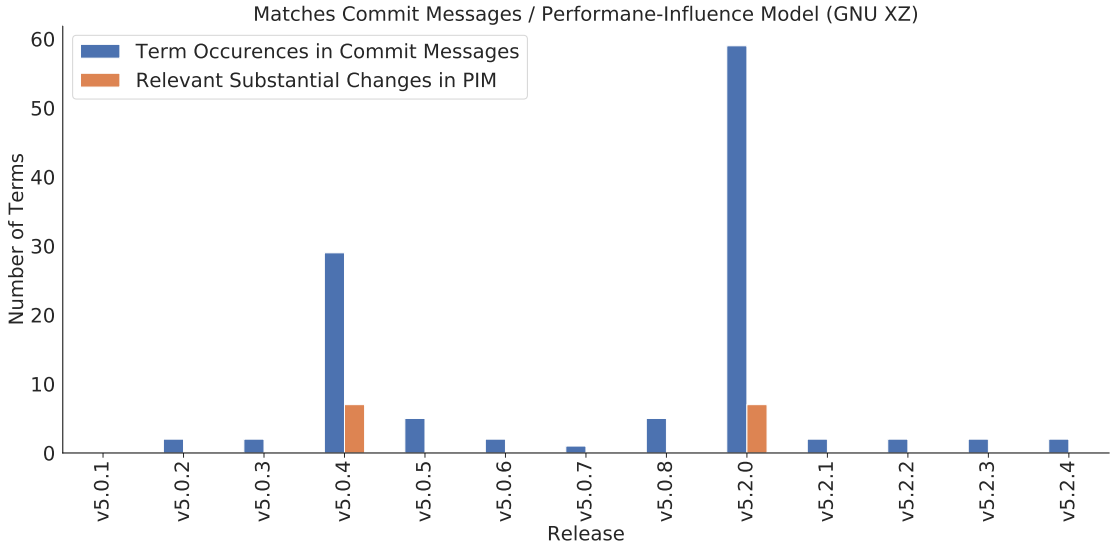

Figure 4.18: Matches of commit messages with performance-influence models of *GNU XZ*. The blue bars indicate the number of terms found in the commit message texts. The orange bars depict how many of the found terms underlie a *substantial* change in the corresponding performance-influence model.

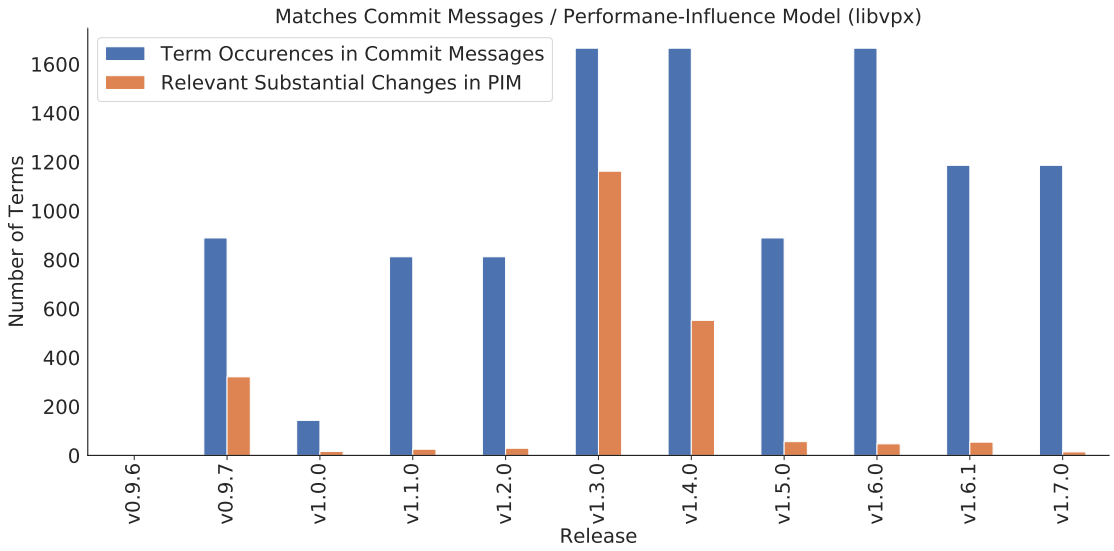

Figure 4.19: Matches of commit messages with performance-influence models of *libvpx VP8*. The blue bars indicate the number of terms found in the commit message texts. The orange bars depict how many of the found terms underlie a *substantial* change in the corresponding performance-influence model.

configurable software systems also had a *substantial* change in the corresponding performance-influence model. The computed match ratios range between 10% and 30%, which seems to be a quite small number, but not all appearances of a configuration option's name in the documentation imply a performance change. Nevertheless, some change in the program code was related to the configuration option and, thus, is a potential indicator for performance changes.

On the other hand, not all performance changes are documented because, often, the developers are not aware of those changes. If the performance for a specific part of the program was improved, it is very likely that this change is documented. Performance bugs are most likely not documented, because nobody is aware of them in the first place. Therefore, performance-influence models can be used to find performance bugs which are related to specific configuration options or option interactions by comparing the performance-influence models of multiple releases. The performance-influence models we computed showed more terms with *substantial* change than we found in the documentation. Of course, this does not mean that these results are wrong, because the developers might not be aware of the performance change and, thus, did not document it, like already mentioned before.

Summary: For the case studies *lrzip* and *libvpx VP8*, we found occurences of terms in the changelogs and in the commit messages between consecutive releases. We found matches between those occurences and the computed performance-influence models, where the exact same terms showed a substantial change in the influence coefficient. The match ratios ranged between 10% (*GNU XZ*) and 30% (*lrzip*).

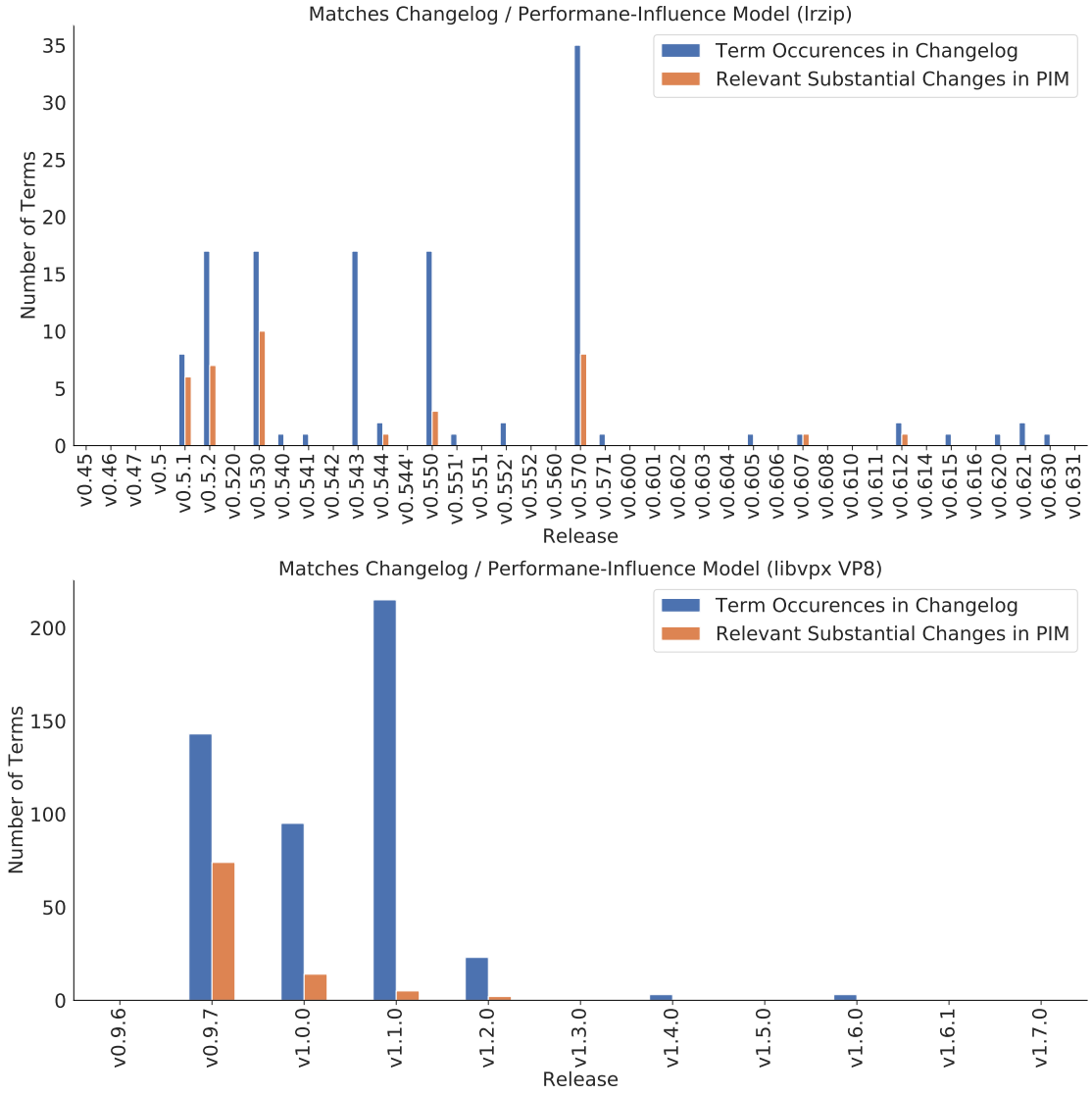

Figure 4.20: Matches of changelogs with performance-influence models of corresponding case study. The blue bars indicate the number of terms found in the changelog text. The orange bars depict how many of the found terms underlie a *substantial* change in the corresponding performance-influence model.

4.6 Summary

In this section, we will briefly show the major findings for our research questions. In Table 4.4, we show all key figures, which we already mentioned in the corresponding chapters.

Table 4.4: Summary of major findings for all research questions.

RQ	**Metric**	**lrzip**	**libvpx VP8**	**GNU XZ**
RQ1	*Outstanding* Releases	3	1	0
RQ2	*Outstanding* Releases	3	1	0
RQ3.1	*Outstanding* Releases	3	1	0
RQ3.2	*Dominant* Terms	2	101	1
RQ4	Match Ratio Commits	29%	20%	12%
RQ4	Match Ratio Changelogs	25%	21%	-

5. Threats to Validity

Internal Validity

Whenever executing performance measurements of all kinds, it has to be ensured that the results do not get distorted by random fluctuation between different iterations. Therefore, we measured every configuration of every considered release three times and computed the standard deviation. Standard deviations higher than 10% lead to the repetition of the corresponding experiment. The execution time measurements where executed in parallel on different workstation PCs. All those PCs are equipped with the exact same hardware of the same age, which, nevertheless, allows us to compare all results.

The choice of the configurations, which we measured the execution times for is crucial and has to be representative regarding the entire configuration space. We focused on individual configuration options and on option interactions consisting of two individual configuration options. This is the reason why we used the option-wise and the pairwise sampling strategy. To get a more representative set of configurations of the entire configuration space, we additionally used random sampling, which evenly distributes the samples on the space of valid configurations.

We computed the performance-influence models for all regarded releases of all case studies. Since we fitted the performance-influence model to a set of given terms, which consisted of all possible term combinations, there is the issue of shifting considering alternative groups (see Section 3.2.2.3). The individual performance-influence models of different releases are not comparable if such shifting appears. Therefore, we prevented the described kind of shifting by following the approach of withdrawing specific terms (see Section 3.2.2.3), which leads to good comparability between the individual performance-influence models.

In this work, we define several thresholds for classifying the observed subjects. We called change rates *substantial* if the change rate is higher then 20%. Using this threshold is a conservative approach, as described in Section 3.2.2.1. Even more performance changes could be found if the threshold was adapted to the real measured standard deviations. The threshold for classifying releases as *outstanding* is an

established method for classifying outliers in normally distributed data. Choosing another threshold could lead to more sophisticated results.

External Validity

For our case studies, we conducted three different configurable software systems of two different fields of application. To get generally applicable results without considering more case studies, we chose software systems with different characteristics. Our case studies consist of a different amount of releases and different sizes of the configuration space. Furthermore, the software systems were developed in different years and differ in the time span between releases. Those different properties allow us to generalize our findings.

6. Related Work

To the best of our knowledge, the evolution of performance in configurable software systems was not investigated by the use of performance-influence models yet in other publications. The evolution of performance across different software releases was also analyzed by *Sandoval et al.*, where a new visualization method for performance evolution is proposed [SBDD13]. Profiling tools can be used to analyze execution times for fine granular aspects of the programs like single methods, which then will be visualized regarding the performance changes from two consecutive releases. Although the evolution of very specific parts of the program can be analyzed, the focus does not lie on configurable software and the influences of configuration options or option interactions.

The work by *Li et al.* [LLC16] is not about the evolution of specific configurations or configuration options of a software system, too. They focus on software architecture, whose evolution is a point of interest in research. Performance simulation is used for evaluating the changes of software architecture, for helping software architects to make profound decisions on software architecture changes concerning execution time and memory.

One key strategy of this work is to use performance-influence models as a base for predicting performance-influences of a configurable software system. There exist other approaches for computing individual influences like *Fourier Learning* [ZGBC15]. Instead of computing performance-influence models, Fourier decomposition is used to learn configurable software performance functions. However, the approach was not used for analyzing the evolution of performance yet.

Transfer Learning is a promising concept whenever knowledge from one domain shall be used in another domain. *Jamshidi et al.* [JSV+17] apply transfer learning to the concept of performance-influence models and show that they can handle small environmental changes (e.g., hardware, workload) without relearning a performance-influence model. One environmental change, which they consider is the change of a software version (e.g., new software release). However, the focus in this work does not lie on the evolution of performance in configurable software systems, like it is in this thesis.

In this work, we used changelogs and commit messages, to strengthen the measured and computed results. *Palomba et al.* [PBP+15] use commits in the history of the version control system to detect code smells (e.g., large complex classes). Although performance is not in their focus, the evolution throughout the history of the software system leads to good results.

7. Conclusion and Future Work

Next, we will summarize the results and findings of this work in Section 7.1 and provide ideas and enhancements for further work on the presented topic in Section 7.2.

7.1 Conclusion

This exploratory thesis aimed at investigating the evolution of performance in configurable software systems. Therefore, we considered three real world software systems and measured the execution times for a variety of configurations for different releases of the programs. Apart from analyzing the pure execution times of configurations, we computed performance-influence models for the different releases of the case studies. These calculations were followed by an in-depth analysis and comparison of the resulting performance-influence models and the individual influences of configuration options. At last, we tried to find congruencies between our results and the documentation included in the software systems.

Firstly, we showed that we are able to identify releases in the history of a configurable software system which underlie a performance change. By comparing the average execution times across all measured configurations, we also are able to make statements on the strength and relevance of the found performance changes. For the case study *lrzip*, we found three releases with *substantial* performance changes, and for *libvpx VP8*, there was one release with a *substantial* performance change.

Furthermore, our investigations lead to the result that, often, performance changes between consecutive releases can only be found in specific configurations, whereas the performance in other configurations did not change. Again, we were able to rate the strength of the performance changes of individual configurations by analyzing the execution time change rates of specific configurations for consecutive releases. There were three *outstanding* releases for *lrzip* and one *outstanding* release for *libvpx VP8*, which showed a high number of configurations with *substantial* performance changes.

After computing performance-influence models for all releases of a configurable software system, we were able to put the performance changes down to individual config-

uration options or option interactions. Comparing the change rates of the influence coefficients for specific configuration options between consecutive releases leads to profound statements on which options caused performance changes exactly and how strong those changes were. In this investigation, we found three *outstanding* releases for *lrzip* and one *outstanding* release for *libvpx VP8*. The *outstanding* releases had a relatively high number of configuration options or option interactions with *substantial* performance changes.

Moreover, we introduced a metric which can be successfully used to rank configuration options and option interactions according to their relevance in the overall performance of a particular release. There were two *dominating* terms for *lrzip*, one *dominating* term for *GNU XZ*, and 101 *dominating* terms for *libvpx VP8*, which had a high relevance ranking throughout the regarded time span.

Finally, we validated our computed results by analyzing the documentation which is provided by the developers for configurable software systems. We found congruencies between the performance-influence model changes and the examined changelogs and commit messages, which strengthens the presented results. We found between 12% and 29% of the occurences of option names in the documentation in the corresponding computed performance-influence models.

7.2 Future Work

The presented work can be used as a base for further research on the performance evolution in configurable software systems. Therefore, we want to suggest a few ideas about improvements and next steps in the topic of this thesis.

First of all, this work demonstrated exciting and promising results regarding the performance evolution of configurable software system based on three case studies of two different domains. To get more resilient results and insights, more case studies from different fields should be conducted.

Moreover, our comparison of performance changes between specific configurations, configuration options or option interactions were based on the percental changes of the execution times. It might be possible to find a profound comparison metric by using machine-learning approaches for novelty detection.

In this thesis, we only investigated binary configuration options. Numeric configuraion options were converted to alternative groups. Further research should also model numeric configuration options as such and provide an appropriate comparison metric for them.

In the end, future work could use the gained insights to develop best-practices and possibly a toolset for software developers, which could help them to keep track of the performance of their maintained configurable software system. This way, performance bugs could be identified and solved early on.

A. Appendix

A.1 Dictionary for Document Search

The following dictionaries are used for the term search in the changelogs and the commit messages. For each configuration option, which we want to replace in the search, we list the substitutes.

lrzip

- *level1 -> level*
- *level2 -> level*
- *level3 -> level*
- *level4 -> level*
- *level5 -> level*
- *level6 -> level*
- *level7 -> level*
- *level8 -> level*
- *niceness0 -> niceness*
- *niceness5 -> niceness*
- *niceness10 -> niceness*
- *niceness15 -> niceness*

GNU XZ

- *level0 -> level*
- *level1 -> level*
- *level2 -> level*
- *level3 -> level*
- *level4 -> level*
- *level5 -> level*
- *level6 -> level*
- *level7 -> level*
- *level8 -> level*

libvpx VP8

- *lagInFrames0 -> lag, frames*
- *lagInFrames5 -> lag, frames*
- *lagInFrames10 -> lag, frames*
- *lagInFrames15 -> lag, frames*
- *lagInFrames20 -> lag, frames*
- *resizeAllowed0 -> resize, allow, allowed*
- *autoAltRef -> auto, alt, ref*
- *noiseSensitivity0 -> noise, sensitivity*
- *noiseSensitivity3 -> noise, sensitivity*
- *sharpness0 -> sharp, sharpness*
- *arnrMaxFrames0 -> arnr, max, frame*
- *arnrMaxFrames5 -> arnr, max, frame*
- *arnrMaxFrames10 -> arnr, max, frame*
- *arnrStrength0 -> arnr, strength*
- *arnrStrength3 -> arnr, strength*
- *rt -> rt, realtime*

Bibliography

[ABKS13] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. Feature-Oriented Software Product Lines. In *Springer Berlin Heidelberg*, 2013. (cited on Page 5 and 6)

[Fou] Blender Foundation. Big Buck Bunny. Website. Available online at https://peach.blender.org/; visited on March 5th, 2019. (cited on Page 19)

[GDT] The Git Development Team. git –fast-version-control. Website. Available online at http://git-scm.com/; visited on February 13th, 2019. (cited on Page 10)

[JSV+17] P. Jamshidi, N. Siegmund, M. Velez, C. Kästner, A. Patel, and Y. Agarwal. Transfer Learning for Performance Modeling of Configurable Systems: An Exploratory Analysis. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 497–508, Oct 2017. (cited on Page 57)

[Kol] Con Kolivas. lrzip - Git Repository. Website. Available online at https://github.com/ckolivas/lrzip; visited on February 26th, 2019. (cited on Page 14)

[KSK+19] Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, Alexander Grebhahn, and Sven Apel. Tradeoffs in Modeling Performance of Highly Configurable Software Systems. *Software & Systems Modeling (SoSym)*, pages 2265–2283, 2019. (cited on Page 8, 9, and 48)

[LLC16] B. Li, L. Liao, and Y. Cheng. Evaluating Software Architecture Evolution Using Performance Simulation. In *International Conference on Applied Computing and Information Technology/International Conference on Computational Science/Intelligence and Applied Informatics/International Conference on Big Data, Cloud Computing, Data Science Engineering (ACIT-CSII-BCD)*, pages 7–13, 2016. (cited on Page 57)

[LvRK+13] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable Analysis of Variable Software. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 81–91, 2013. (cited on Page 7 and 9)

[Pas] Chair Of Software Engineering At University Of Passau. SPLConqueror - Git Repository. Website. Available online at https://github.com/se-passau/SPLConqueror; visited on March 5th, 2019. (cited on Page 18 and 21)

[PBP+15] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia. Mining Version Histories for Detecting Code Smells. *IEEE Transactions on Software Engineering*, pages 462–489, 2015. (cited on Page 58)

[Proa] Tukaani Project. Tukaani - GNU XZ. Website. Available online at https://tukaani.org/xz/; visited on February 26th, 2019. (cited on Page 15)

[Prob] WebM Project. libvpx - Git Repository. Website. Available online at https://github.com/webmproject/libvpx; visited on February 26th, 2019. (cited on Page 16)

[Proc] WebM Project. VP8 Encode Parameter Guide. Website. Available online at https://www.webmproject.org/docs/encoder-parameters/; visited on February 28th, 2019. (cited on Page 16 and 18)

[Prod] WebM Project. WebM: An Open Web Media Project. Website. Available online at https://www.webmproject.org/; visited on February 26th, 2019. (cited on Page 16)

[SBDD13] J. P. Sandoval Alcocer, A. Bergel, S. Ducasse, and M. Denker. Performance Evolution Blueprint: Understanding the Impact of Software Evolution on Performance. In *First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–9, 2013. (cited on Page 57)

[SGAK15] 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)*, pages 284–294, 2015. (cited on Page 1, 6, 7, 8, 9, 18, and 21)

[Sie] Norbert Siegmund. SPL Conqueror: Measuring and Predicting Non-Functional Properties of Highly Configurable Software Systems. Website. Available online at https://www.infosun.fim.uni-passau.de/se/projects/splconqueror/; visited on March 5th, 2019. (cited on Page 18)

[UoC] The University of Canterbury. The Canterbury Corpus. Website. Available online at http://corpus.canterbury.ac.nz/; visited on March 5th, 2019. (cited on Page 19)

[ZGBC15] Y. Zhang, J. Guo, E. Blais, and K. Czarnecki. Performance Prediction of Configurable Software Systems by Fourier Learning. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 365–373, 2015. (cited on Page 57)

Eidesstattliche Erklärung:

Hiermit versichere ich an Eides statt, dass ich diese Masterarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Johannes Hasreiter

Passau, den 17. Juni 2019