Bauhaus-Universität Weimar
Faculty of Media
Degree Programme Computer Science and Media

Supporting the Discovery of Performance Hot-Spots of Configurable Software Systems

Master's Thesis

Max Weber
Born May 04, 1991 in Jena

Matriculation Number 110068

1. Referee: Prof. Dr. Ing. Norbert Siegmund
2. Referee: Junior Prof. Dr. Florian Echtler

Submission date: May 22, 2018

Declaration

Unless otherwise indicated in the text or references, this thesis is entirely the product of my own scholarly work.

Weimar, May 22, 2018

. . . .
Max Weber

Abstract

Today, almost every software system is configurable. Such software systems offer a huge amount of configuration possibilities. It is common practice that for performance analysis some default configurations are tested. Thereby, a huge number of configurations remain untouched. Existing analysis tools does not provide information which parts of software get affected by configuration options. This thesis provides an approach to identify performance hot-spots in configurable software systems. The case study aims to explore the influence of configurations to the individual methods of software systems.

As a result of this thesis, we are able to explain the influence of configurations on performance of about 95% of the methods, which together make up 50% of the overall performance. We investigate influences that bias our measurements and determine the sensitivity of methods to changes in the configuration. Furthermore, we showed how to save a substantial fraction of time while monitoring performance at method level.

Contents

Chapter 1

Introduction

Since the beginning of software engineering, developers try to produce software that performs good. Over the years, software has to run more difficult, complex, and computational-intensive tasks. The used hardware, software, and technology becomes more and more complex. Specifically, developers have to address non-functional requirements like reliability, security, maintainability, and performance. From these properties, performance has always been among the most important aspects, since performance affects user perception, system efficiency, and energy consumption.

We can view performance from an end user's perspective by describing the ability that users can perform a certain task without perceivable delay or without irritation. Moreover, we can view performance also from the developer's perspective as a non-functional property to be optimized (see Molyneaux [2009]). Taking the developers view, we need to ask the question: How can we optimize performance of a configurable software system? There are multiple ways to answer this question. First, we can try to view the system as a black box that produces an output in a certain time (the response time) for a given workload. Finding an optimal configuration to minimize the response time is the main goal from this perspective. Researchers have shown that this perspective is viable and provide different approaches to obtain a black box performance model (see Molyneaux [2009], Weber and Thomas [2005], Siegmund et al. [2015], Smith [1993]). However, the black-box perspective has one main drawback: We can not pin point performance hot-spots or performance bugs related from the configuration to the actual code position. Such kind of analysis requires a white-box perspective, which we will target in this thesis.

Performance hot-spots in software systems are regions of a program where a high portion of executed instructions occur, which refers to a region where most of the execution time is spent. Optimizing these hot-spots leads to a perceptible improvement of the overall performance.

A key property of our work is that we focus on configurable software systems. Nowadays, almost every software system can be configured by numerous options. This configurability provides the possibility to customize a program's behaviour to user's needs. For example, databases typically let users configure the options encryption, compression, cash size, and so forth. While this flexibility improves customizability, it also causes an increased effort during testing, maintenance, and analysis of performance. Not only the size of the configuration space is hard to grasp by users, but also all possible constraints among options make configuring a software systems problematic. For instance, a program with more than 217 binary configuration options provides more configurations than atoms in the universe (see Krueger [2006]), which make testing and analyzing them impracticable. When we consider that there are software projects with more than thousands of options, we can easily imagine that finding a good performing configuration is hard. Users often pick default configurations or leave optimization potential untouched (see Xu et al. [2015]). So, a challenge in our work is not only to develop a white-box performance-analysis method for configurable software systems, but also to provide a generalizable method that requires to analyze only few configurations, but is able to generalize for many unseen configurations.

To sum up, our goal is to analyze performance and model the influence of configuration options of software systems on performance. To understand the influence of configuration options, we profile each subject system with an appropriate profiler to extract the runtime of single methods of the running program. We decided to take Java projects for our subject systems, such that we can also analyze the influence of the environment in which the tests are running. To support the analysis of performance hot spots, we provide a visualization of our results directly in the source code within an Integrated Development Environment (IDE) as well as an overview of the performance distribution over all methods in a Java project.

To analyze *performance* of a configurable software system, we follow a three step approach:

1. Profiling the execution of the subject system with sampled configurations and different workloads.

2. Processing the extracted data to assess performance on method level and learning models that can predict the performance of each method depending on the given configuration.

3. Visualizing performance hot-spots directly in the source code and create suitable visualizations of performance dependencies among the whole code base.

First, we give some background knowledge and provide an overview of related work in Chapter 2. In Chapter 3, we present our approach on discovering performance hot-spots, including an analysis of available Java monitoring tools. Next, we present our subject systems and define the measurement setup in Chapter 4. We analyze the learnability of performance at method level influenced by configurations in Section 4.3. We discuss the measurement bias that influences the performance analysis in Section 4.4. Furthermore, we developed an Eclipse plug-in to visualize performance hot-spots directly in the IDE in Section 4.5.1. Finally, we provide an interactive overview visualization of the performance of methods in Section 4.5.2.

Chapter 2

Background and Related Work

This chapter provides fundamental concepts required to follow this thesis. In section 2.1 we give an introduction about variability models, performance assessment and performance prediction. We summarize related work and approaches that analyse performance similar to our approach in section 2.2.

2.1 Background

We give an overview about configurable software systems (section 2.1.1) and variability models (section 2.1.2). We define the term performance for this thesis in section 2.1.3, describe how performance of software can be measured, and how to learn models in order to be able to predict performance.

2.1.1 Configurable Software Systems

Today, almost every software system provides different options for configuration. Each combination of configuration options results in a *variant* of the program, similar to Software Product Lines (SPLs)[1] (Siegmund et al. [2012b]). Each variant comprises a set of activated configuration options, which are referred to as *features*. The combination of features describes the user desired characteristics of the program (Czarnecki et al. [2000]). For example, a database management system provides features such as compression, encryption, or the use of a specific indexing strategy. Requirements that describe what a system can do are called functional properties Chung et al. [1995]. In addition to the functionality properties of software, there are also non-functional properties to be met, such as performance and energy limitations.

[1]"A SPL is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission which are developed from a common set of core assets in a prescribed way." (Northrop [2010])

In literature, there exist different definitions of non-functional properties as surveyed by Glinz [2007]. Davis [1993] defined non-functions properties as all required attributes of the system like portability, reliability, efficiency, human engineering, testability, understandability, and modifiability. Whereas Kotonya and Sommerville [1998] defined non-functional properties as all requirements which are not directly related to the functionality of the system. Such requirements set restrictions to the development process and specify constraints that the software must meet.

In this thesis, we use the definition of Robertson and Robertson [1999] because these definition is general and has been widely used by others: Nuseibeh and Easterbrook [2000], Jackson [2001], Cohn [2004], Siegmund et al. [2012b]. “A property, or quality, that the product must have, such as an appearance, or a speed or an accuracy property”.

2.1.2 Variability Models

An important requirement when developing and maintaining configurable-software systems is to model the variability in a comprehensible way. *Feature models* define a common way to describe all valid configurations of software systems (Kang et al. [1990], Czarnecki et al. [2000], Apel et al. [2016]). They have been introduced by Kang et al. [1990] and are usually realized with two different representations: textual notations and graphical representations. Feature models in textual from (e.g., SXFM introduced by Mendonca et al. [2009] and Velvet introduced by Rosenmüller et al. [2011]) have been developed to construct and maintain models of systems with a huge amount of features which might be presented poorly with a graphical tool. Whereas feature models with a graphical representation (also called feature diagrams Kang et al. [1990]) provide the possibility to create and manipulate feature models through a graphical user interface. Furthermore, feature diagrams are widely used in scientific literature to present analyzed systems.

Feature diagrams are made of a tree-structured graph, as can be seen in the example diagram shown in Figure 2.1. With this example diagram, we will describe the elements of feature diagrams. The root node of the tree Database_Engine represents the whole database model. All nodes in this tree represent either *abstract* or *concrete* features. Abstract features such as Compression and OS are used to structure the feature model, while concrete features represent functionality that maps to implementation units. Two nodes of the database engine are modelled as *mandatory*. These nodes have to be selected in each valid configuration because they might implement some basic functionality which has to be always available. Optional features such as Indexing or Compression can be selected to create a valid variant with additional function-

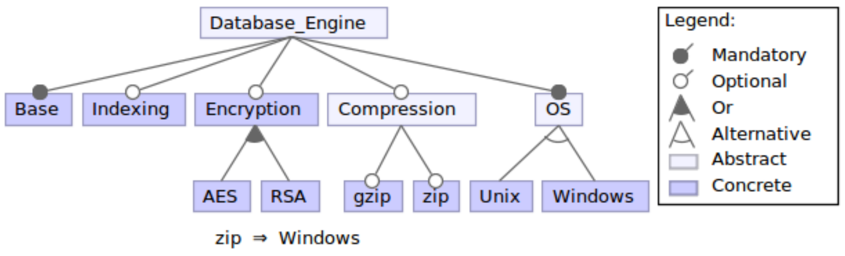


Figure 2.1: Example database model (extended form section 2.1.1).

ality. For example, all databases need to have an operating system specified and can optionally use compression algorithms. There are two more elements available in feature diagrams: *or*- and *alternative* groups. The *alternative* group represents the logical XOR, hence only one child must be selected at a time, whereas the *or* group enables the selection of one or more children. If Encryption is selected in this example, either AES or RSA or both must also be selected. In case of the mandatory abstract feature OS which itself only groups available operation systems, one has to select either Unix or Windows. The structure of the tree, including the parent-child relationship and the different elements, defines a set of constraints for selecting valid configurations. Additionally, there are also *cross-tree constraints* which represent logical expressions over the set of features. They enable the modelling of arbitrary formulas amongst the features in feature models. For example, if users want to have zip-compression enabled, they also have to stick to Windows. Such constraints restrict the amount of valid configurations. In total, our example has 64 valid configurations. If Windows is initially selected there are 24 valid configurations and if Unix is selected we can derive 40 valid configurations. The textual representation of the feature model is available in the appendix A.1.

2.1.3 Performance Assessment

Next, we define the term performance for this thesis and show how to assess performance to learn models from measured configurations. Afterwards, we present different approaches of how these models can predict performance.

Performance in Software Engineering

Performance is an key aspect of software systems. As described in the introduction, definitions of performance differ depending on the perspective. The most used way is to explain performance of software systems as a non-functional property (Molyneaux [2009], Liggesmeyer [2002]). Here, performance is a mean for describing how a system provides its functionality. To decide if a system performs well, non-functional properties are compared to so called non-functional requirementss (NFRs). They constitute a set of requirements which the software should fulfil. NFRs should be defined before the software is tested. Each type of a software system has different metrics which can be analysed. For example, a Web server can be analyzed through the following metrics:

Hit Rate	Total number of requests arriving at a server in a defined time frame.
Latency	Time that has passed from the start of a request message until the response message starts being received.
Response Time	Time that has passed from the start of a request message until the whole response message is received.
Availability	Time that the software is available to the user.

With client-server applications, communication takes place through networks, whereas with offline applications, the focus during the analysis of performance metrics lies on the intrinsic properties. Offline applications usually run on a single device and do not need to communicate directly with other systems. The following are some performance metrics of offline applications:

Resource Utilization	Fraction to which degree a software system uses its resources[2].
Response Time	Time that is needed to complete an operation.
Throughput	Amount of work done in a defined time.
Availability	Time that the software is available to the user.
Scalability	Ability to grow and perform an increasing amount of work.

For this thesis, we will focus on offline systems because we do not want to cope with the measurement inaccuracy which might accompany with the hardware and software of networks resources. Furthermore, we want to define

[2]A resource is a system element that offers some service to other system elements that require them. Resources can be categorized into hardware components (CPU, bus, storage), logical elements (buffers, locks, semaphores) and processing resources (processes, threads) Woodside et al. [2007].

performance as the amount of work done per unit of time like Tsirogiannis et al. [2010]. So, we define performance as:

$$\text{Performance} = \frac{\text{Work Done}}{\text{Execution Time}} \tag{2.1}$$

Because we want to analyze performance of configurable software systems, we execute different variants of each system using the same inputs (benchmarks[3]) for all configurations. Thereby, we are able to compare the different variants and since the Work Done is constant, we are able to simplify our formula like Siegmund et al. [2015]:

$$\text{Performance} = \frac{1}{\text{Execution Time}} \tag{2.2}$$

To identify performance hot-spots, we also measure the run time of each method that is executed. We are interested in the time spent in each method (net-time). We explain the concrete approach we follow in Section 3.2.1.

Software Performance Engineering (SPE)

After defining performance in general, we outline the process of analyzing performance of software systems. The process of analyzing performance of software systems is defined as follows:

> "SPE represents the entire collection of software engineering activities and related analyses used throughout the software development cycle, which are directed to meeting performance requirements." (Woodside et al. [2007]).

There are two general approaches in the literature (Woodside et al. [2007]): *measurement-based* and *model-based* performance assessment. The model-based approach is applied early in the development cycle. It tries to create performance models to adjust design and architecture as early as possible. The measurement-based approach is applied late in the development cycle, because it will run tests and diagnosis to tune the software. The whole process of SPE activities can be structured into six parts (Woodside et al. [2007]). The first identifies the NFR which should be analysed. Defining and analyzing the requirements is the second step. This includes specifying the environment in which the software runs. For the sake of reproducibility, the environment should be defined carefully. To select specific tasks, there are many different benchmarks available (BAPCo, EEMBC, SPEC). These and other benchmarks

[3]Benchmarking is the process of running a software system with standardized workloads to be able to compare the results and assess their relative performance.

provide the possibility to compare one software variant to another with a set of standardized tests and workloads. Another possibility is to use benchmarks which are provided by the software vendor itself (a concrete example is provided in Section 4.2). This may enable analysis of the whole functionality of a the software system. The third activity is to design models that can predict performance from detailed design, architecture, and usage scenarios. These models describe how systems use resources and how resource contention affects operation. These models are able to predict system properties before they are implemented to gather knowledge as early as possible in the software development life cycle. The next SPE activity is performance testing. Tests can cover only parts or the entire software, depending on the selected benchmarks. Here, we obtain actual values for the performance metrics we selected. From the gathered data, we can learn models to predict the effect of changes to the system, which is the fifth step of SPE. A lot of effort has been spent in the last years to understand and improve performance. Section 2.2.1 summarizes more recent measurement-based performance prediction approaches which emphasize variability. The last step of SPE is to analyze the whole system. Here, the final deployed system is analyzed regarding to the defined performance goals.

This thesis identifies concerns, conducts performance tests and learns models for performance prediction of configurable software systems. Because we follow the *measurement-based* approach, we also evaluate the impact of the measurement environment.

2.2 Related Work

This section cope the topics performance prediction and performance analysis of Java programs. The first part deals with different approaches on learning the influence of feature selection on performance and presents different sampling strategies. The second part reviews existing performance analysis tools.

2.2.1 Performance Prediction

The task of learning and predicting performance is subject of research for several years. Since then, several measurement-based performance prediction models were developed. In general, they all work the same way. They sample a subset of configurations because measuring performance of each configuration is not practicable if the configuration space is too big. They split the available measurements into two sets (learning set and test set). For learning a model, they use the learning set and to be able to assess the accuracy of the model they compare the test set with the prediction of the model. In the following, we describe the differences of some models.

One task of SPLs is to express the large number of possible products. Thüm et al. [2012] presented the approach, called family-based verification, in which the whole SPL is encoded in a single *meta-product* which simulates the behaviour of all individual products of the product line. The idea is to verify only the generated meta-product. It contains the implementation and specification of every feature. So, it is able to simulate each product. They verify the meta-product (family-based) instead of verifying all products separately (product-based). Because of the generation of the meta-product, they do not have different products from a product line, but a single system, so they are able to use the single-system theorem prover KeY for verification. They adapted the product simulator introduced by Apel et al. [2011] to translate the Java-based product line into the meta-product. In comparison to the product-based approach, the authors were able to save more than 85% of the verification effort.

Guo et al. [2013] presented a variability-aware approach to performance prediction. The authors want to use few samples (linear in the number of features) to show that this is enough to learn a model that can accurately predict performance of all configurations. They make use of Classification-And-Regression-Trees (CARTs) to recursively partition the configuration space into smaller segments. The mean performance value of the samples of each segment is used as local prediction model. The prediction error at each segment is calculated through the sum of squared error loss. Each split is chosen according to the feature that reduces the entropy of the resulting subsets most. The configuration space will be further subdivided until each leaf of the tree locally describes the configuration best. They introduce two parameter which control the recursive partitioning process to prevent underfitting and overfitting: minibucket is the minimum sample size for any leaf of the tree; and minisplit is the minimum sample size of each segment before it is considered for further partitioning. After that, the algorithm adds iteratively new measured samples and rebuilds the performance model.

Siegmund et al. [2012b] presented an holistic approach, *SPLConqueror*, to optimize learning the influences of features on non-functional properties (NFP). They automate the process of learning the influence of features and their interactions on performance, including sampling, measurement, and creation of performance influence models. They differentiate between feature-wise (i.e., measure an implementation module of a feature in isolation and independent of the remaining software system) and variant-wise assessment (i.e., assessing a property that emerges only when running a valid variant). They also introduce two sampling strategies to analyze the influence of individual features as well as the influence of feature interactions. With *feature-wise* sampling the influence of individual features f to NFPs is analyzed by sampling two variants of each feature. The first variant selects feature f and the other variant deselects feature f while in both variants the number of additionally selected features is minimized. So the difference between these two variants present the impact of feature f on the NFP. The other sampling strategy (*pair-wise* sampling) works similar but uses each combination of two features.

Siegmund et al. [2012a] have further developed the idea of automated detection of feature-interactions. Feature interactions occur when a particular feature combinations have an unexpected influence on performance. They wanted to detect performance-relevant feature interactions to reduce the amount of samples needed to create a performance-influence model. At the start, they identified the set of features that interact with other features by applying a heuristic: If the distance between Δa_{max} and Δa_{min} is within a threshold feature a does not interact with other features. The deltas of a are the two that are most likely to differ because Δa_{min} is the delta of the minimum number of features selected, whereas with Δa_{max} most features are selected. The set of features that are likely to interact with each other is created through applying this heuristic to each feature. Then three additional heuristics are applied to identify feature combinations that cause interactions. First, they investigate candidates for pair-wise interaction. In the second step they used these candidates to identify higher-order interactions (i.e., interactions among three features). Finally, hot-spot features are derived from the set of pair-wise and higher-order interactions. They interact with many other features, and are therefore interesting for optimization.

A further development of the idea of how to build performance-influence models was introduced by Siegmund et al. [2015]. They extend the tool *SPLConqueror* to create human understandable models that describe the performance behavior of features, interactions, and variants. The distinguishing aspect of this work is that they also support numerical features. Therefore, they applied different binary (e.g., pair-wise, option-wise, negative option-wise) and numerical (e.g., Placket-Burman Design) sampling strategies[4] and combined them. The learning procedures consists of a combination of multi-variable regression and feature subset selection as dimensionality-reduction technique. Linear regression tries to fit a line through measurement points by adjusting regression coefficients so that the global error is minimal. The proposed approach incrementally adds new regression terms to the formula in each round until a certain accuracy is reached or improvement in accuracy becomes marginal.

[4]We will give an overview of sampling strategies in section 3.2.1.

Sarkar et al. [2015] proposed another approach to further reduce the number of samples needed to learn an accurate performance model. The authors want to dynamically determine an ideal sample size dynamically, that yields a good prediction accuracy with low measurement effort. They investigate two sampling strategies (projective sampling and progressive sampling) which both are used in data mining to reduce the size of data needed. After sampling, they use CART to build the prediction model. A typically evaluation criteria is prediction accuracy, but Sarkar et al. [2015] use an accumulated cost function consisting of (weighted) prediction error and costs for creation of test and training sets. With their approach, they are able to reduce the number of elements needed for an initial sample.

All aforementioned approaches rely on improving certain aspects of configurable software systems (or SPL) modelling. They provide new strategies for sampling or present tools for learning performance models. However, they all utilize black-box approaches that analyzes the examined software systems as a whole. We, however, want to create and analyze performance model on method-level, which was not addressed so far.

2.2.2 Performance Analysis of Java Programs

To the best of our knowledge there exist no approach that models and predicts performance on method level of a configurable software system (written in Java). Nevertheless, there is related work in monitoring and assessing performance of monolithic Java applications.

van Hoorn et al. [2012] present a framework, *Kieker*, for monitoring and analyzing the runtime behaviour of software systems. They target monitoring the runtime behaviour of concurrent and distributed applications. Their tool is able to collect time and tracing information as well as sampling system-level measurements (CPE utilization and memory usage). *Kieker* also provides logging and streaming mechanisms to save the data for later analysis or to visualize incoming records directly. Furthermore, it achieves an average monitoring overhead of about 10% and is recommended by the SPEC Research Group.

Another tool which adapts Java bytecode is *SPASS-meter* introduced by Eichelberger and Schmid [2012]. Similar to *Kieker*, it instruments the bytecode to profile NFPs such as throughput or response time. It is also capable of logging CPU-usage of the Java Virtual Machine (JVM) or memory usage. The monitoring overhead which accompanies with instrumentation is comparable to that of *Kieker*. Also memory overhead caused by logging the measurements does not differ substantially.

The list of performance monitoring and analysis frameworks is not exhaustive. Here, we focus on open-source tools that approaches recently to motivate our decision on providing additional support in the field of variability-aware performance analysis of software systems. *Kieker* as well as *SPASS-meter*, do not support handling configurable software systems in combination with variability modelling. Nevertheless, they can represent a powerful tool to assess performance of single variants at method level from which we can derive performance models.

Chapter 3

On Finding Performance Hot-Spots

In the following, we provide an overview of our approach for finding performance hot spots. First, we frame our work on performance analysis in the context of configurable software systems. Then, we present existing performance-monitoring techniques we used in this thesis. We compare different monitoring tools regarding their accuracy, monitoring overhead, and other properties. Based on this analysis, we select the Java Interactive Profiler (JIP), which we use for implementing our approach. Afterwards, we describe how to sample configurable software systems to obtain a set of measurements that we use for learning a performance-influence model.

3.1 Problem Statement

Today, almost every software system is configurable. Such software systems offer a huge amount of configuration possibilities. To facilitate the usage of such systems, they were provided with default configurations making it easier for users to utilize them. It is common practice that default configurations are designed by experts or developers themselves. Due to the evolution of source code which comes with adoption of requirements and the development of new features, the configuration space might grow and provided default configurations need to be adapted. Even domain experts are overwhelmed to get through the huge amount of possibilities and constraints which arise thereby.

Existing performance analysis tools (like the ones we presented in section 2.2.2) lack the possibility to incorporate configurability into their analysis. They are often applied only to default configurations. Because it is not feasible to analyze each variant of software with such tools, many configuration possibilities remain untested. This leads users to stick to default configurations, because of the sheer size of possibilities to configure them. Furthermore, if they configure their systems, they might face performance problems which are still undiscovered, due to the lack of performance tests which aim at analyzing performance bugs of configurable software systems. The state of practice of performance testing in Java-based Open-source Software (OSS) projects was conducted by Leitner and Bezemer [2017]. They analyzed 111 projects from GitHub, trying to identify to which extend these software projects take care of their performance. They found that in 48% of the projects only a single developer is responsible for creating and maintaining performance tests. They also discovered that 14% of the analyzed projects have only a single commit for performance tests. The median number of commits for performance tests is 7, which indicates that many OSS projects do not actively maintain performance tests. If this study can be generalized to other OSS projects, the lack of analyzing and dealing with performance problems is an important problem, which has to be solved.

However, we are able to learn the influence of configuration options on performance by using variability modelling (existing approaches are presented in Section 2.2.1). But until now, they are just able to learn performance of variants of software systems as a whole (black-box approach). Existing variability modelling techniques (see section 2.1.2) lack the possibility to pin point performance hot spots or performance bugs to parts of software or to a specific code position.

Currently, there are neither white-box tools for performance-bug detection nor white-box tools for performance analysis of highly configurable software systems. However, there is an increasing need of such support. Han and Yu [2016] conducted an empirical study on performance bugs of configurable software systems. They wanted to investigate to what extend performance bugs remain undetected if only default instances are tested. If there are only a few performance bugs caused by configurations, developers might stick to classical performance analysis of default instances. But if there is a relevant portion of performance bugs sensitive to configuration options, it might be worth exploring these configurations further. Therefore, they investigate 113 real-world performance bugs randomly sampled from OSS projects (e.g. Apache[1], MySQL[2], Firefox[3]). They found that 59% of performance bugs are related to configuration settings, whereas 41% of the bugs are general performance bugs. Another observation was that all studied performance bugs result from legal configuration options (options that are in the valid part of the configuration space). Moreover, they analyzed how these performance bugs were fixed.

[1]Apache is one of the most used HTTP server (The Apache Software Foundation [2018]).
[2]MySQL is the world's most popular OSS database (Oracle Corporation [2018]).
[3]Firefox is a leading web browser (mozilla.org contributors [2018]).

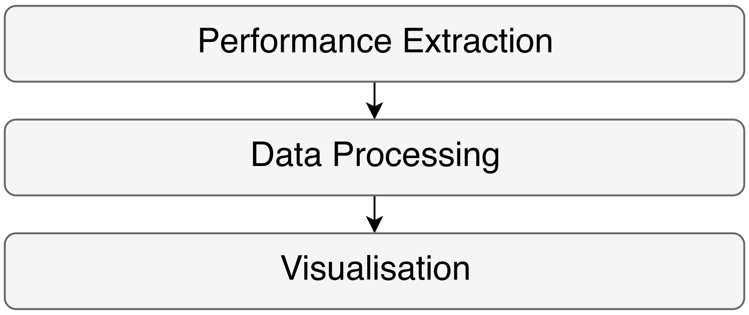


Figure 3.1: Performance Analysis Workflow.

There are two possibilities how fixing can be done: first, through changing configuration values, and second, through patching the source code. A majority of the 113 bugs of their study were patched through adapting the source code of the projects. On top of that, fixing general performance bugs needs to adapt 8 lines of code, on average, whereas patching performance bugs related to configuration options needs adaptation of 30 lines of code, on average. That means fixing configuration-related performance bugs is associated with more effort. Providing tools which are able to support performance bug analysis is all the more important, the more complex software systems become.

3.2 Overall Process

We divide our approach into three parts as shown in Figure 3.1. First, we extract the performance data from our subject systems (see Section 3.2.1) by executing each system in different configurations using different workloads in our test environment, which we describe in Section 4.1. To obtain a list of configurations, we use different sampling techniques as described in Section 3.2.1. During execution, we profile the execution of the Java program of the corresponding configuration and obtain a fine-granular output describing the execution times of each method. For profiling, we compared different monitoring tools, as described in Section 3.2.1, and decided for the tool JIP.

In the second step we process the collected data. An important part is the training of a regression model per method that allows us to predict a method's execution time for unseen configurations.

The last step of our approach focuses on analyzing the performance hot spots via visualizations of the collected performance data (described in Section 3.2.3). To this end, we designed an Eclipse plugin to visualize performance hot spots through the IDE directly in the source code. Additionally, we developed a graph-based visualisation to give an overview of the influence of configuration options and their interactions on the performance of the subject systems.

3.2.1 Performance extraction

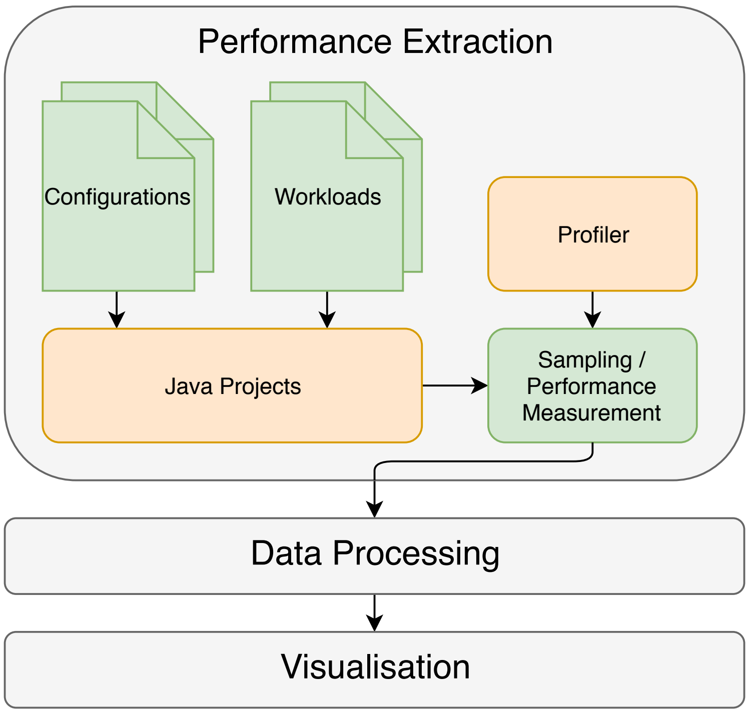


Figure 3.2: Performance Extraction Workflow.

Next, we explain how we extract the performance data from configurable software systems; Figure 3.2 provides an overview. The orange parts of this diagram represent existing tools we used for this thesis and the green parts are results of our implementation. For the evaluation, we choose three Java projects from GitHub as case studies, which we describe in Section 4.2 in more detail. The main tasks that we have addressed here are sampling which configurations to measure, selecting a suitable benchmark as a representative workload for executing a variant (i.e., the program that corresponds to a specific configuration), and preparing the environment for reliable measurement. In order to measure performance during execution, we used the JIP. This profiler is able to measure the runtime of each method during execution of the software. However, we have to deal with the influence of non-determinism that affect overall performance. Our measurements run on real hardware in a Java Runtime Environment (JRE) with other processes being executed in parallel.

Sampling

Sampling is the selection of a subset of individuals to estimate characteristics of the whole population. In the context of configurable software systems, the goal is to select individual configurations from the whole configuration space in order to reduce the amount of samples we have to analyze while not degrading the accuracy of the analysis. There are various sampling strategies, which all have different application areas and purposes. One criterion, which influences the choice of a sampling strategy, is the number of dimension that define the configuration space.

With an increasing number of dimensions, also the volume of the space increases and the sampled data becomes sparse. This sparsity is problematic for any method that requires statistical significance. In order to obtain a statistically sound and reliable result, the additional amount of data needed to support the result often grows exponentially with the dimensionality. This phenomena is called *Course of Dimensionality* (Donoho et al. [2000]).

Furthermore, each individual dimension could represent a different type of data (Stevens et al. [1946]). There are four measurement scales: nominal, ordinal, interval, and ratio. Nominal data are used for labeling data. Usually nominal data is mutually exclusive and none of them have any numerical significance. A special type of nominal scale is binary data (e.g., on/off, true/-false). With ordinal data, we can order the values, but the difference between them is not known (e.g., good, okay, bad). Whereas with interval scales, the ordering as well as the exact difference between the values is known (e.g., temperature scale). Moreover, ratio scales define the ordering, the distance, and also the absolute zero of the values (e.g., height or weight). In the context of configurable software systems, we have to be able to sample each of these dimensions to get a configuration. This is one reason why different sampling strategies were developed. Usually there are two types of data which are used for configuration: binary and numerical values. There are strategies for each of this two types of data, as presented in section 2.2.1.

We choose *Random Sampling* as our sampling strategy, because with it we do not need to know the type of data which we want to sample. Random sampling is only possible because our subject systems do not have any constraints among configuration options. With random sampling we pick one value equally distributed from the values we could choose from, so for each dimension we have to know the start and end value and the step size in between. One drawback of using this approach is that the range of each dimension has the same size.

Java Monitoring Tools

Measuring performance of software is an important aspect of SPE. Many monitoring tools were developed which extract performance values such as hit rate, memory consumption, and runtime. For this thesis, we examined a variety of monitoring tools for Java programs. In order to justify the decision to use JIP for measurements, we compare existing tools.

Table 3.1 provides an overview of selected Java monitoring tools. We examine only OSS tools, because these tools are free to use for all. We also search explicitly for tools which have a command-line interface, because we want to automate the process of performance extraction. The following describes the six properties we used for comparison:

Tool	The name of the monitoring tool.
Performance Type	Summarizes the performance data types that could be extracted with the tool.
Resolution	Identifies the maximum sampling rate of the tool.
Realization	Technical Implementation with witch the data extraction is realized.
Filter	Indicates whether with this tool there is the possibility for defining packages or classes from which the data should or should not be collected.
Output	Shows what kind of data can we get from the tool.
Overhead Factor	Presents the mean time overhead while using the monitoring tool.

We search for performance monitoring tools, that are able to extract the runtime of a method. In total, we have found twelve tools as shown in Table 3.1. Some of those tools are able to extract also other types of performance data, but for this thesis we are interested only in runtime.

Resolution (sampling rate) is the first property on which the tools differ. Most of the tools have a resolution of 1 ms. Exceptions are VisualVM which samples only snapshots initiated by the user, DJProf, which has a resolution of 5 ms, and NetBeans Profiler, which is ten times slower than the others on Windows and Linux. All three monitoring tools are too inaccurate for our purposes. Kieker and SPASS-meter however, provide the highest sampling rate.

The realization of the measurement is important in our context, because we want to automate the process of performance extraction. Six tools use

Tool	Performance Type	Resolution	Realization	Filter	Output	Overhead (Mean)
NetBeans Prof.	runtime, heap, SQL queries, threads	Windows 10 ms Linux 10 ms Solaris 1 ms	aspect	yes	NetBeans GUI	—
VisualVM	runtime, heap, threads, GC	snapshot	aspect	yes	VisualVM GUI	—
HPROF	runtime, heap	1 ms	JVMTI	yes	txt	+5268 %
Patty	runtime, heap, threads, GC	1 ms	JVMTI	yes	GUI, txt	—
JIP	runtime	1 ms	aspect	yes	txt	+420 %
Profiler4J	runtime, heap, threads	—	JVMTI	yes	GUI	—
JRat	runtime	1 ms	JVMTI	yes	GUI	—
EJP	runtime	—	JVMPI	no	GUI	—
JMeasurement	runtime	1 ms	source code	yes	txt, csv	—
DJProf	runtime, heap	5 ms	aspect	no	txt	—
Kieker	runtime, heap	1 μs	aspect	no	txt	+250 %
SPASSmeter	runtime, heap	1 μs	aspect	no	txt	+235 %

Table 3.1: Survey of Java Monitoring Tools.

aspect-oriented programming (AOP) to insert functionality for profiling. Five tools use an interface provided by to JVM. The JVMPI interface exists since J2SE version 1.5 the predecessor of the JVMTI interface. Both are used to inspect the state and to control the execution of applications running in the JVM. JVMPI was always labeled as a native experimental profiling interface and is deprecated since JVMTI was released, hence, EJP does not work any more with todays Java versions.

A possible effect on the monitory accuracy represents our extensions that we made to the subject systems in order to input different configurations. Ideally, we want to exclude this extension from profiling to measure only the performance of the software itself. So, the tool we use has to have the possibility to filter specific classes, hence EJP and DJProf are excluded.

Four out of ten tool present their data only in a Graphical User Interface (GUI). Graphical representations might be helpful in some cases. However, we want to analyze a large number of configurations, so we have to automate the analysis process. Therefore, we want to have the results in a textual representation (e.g., txt, csv, JSON), which excludes NetBeans Profiler, VisualVM, JRat, Profiler4J, and EJP. However, JIP provides a representation of the data-flow which is a great benefit.

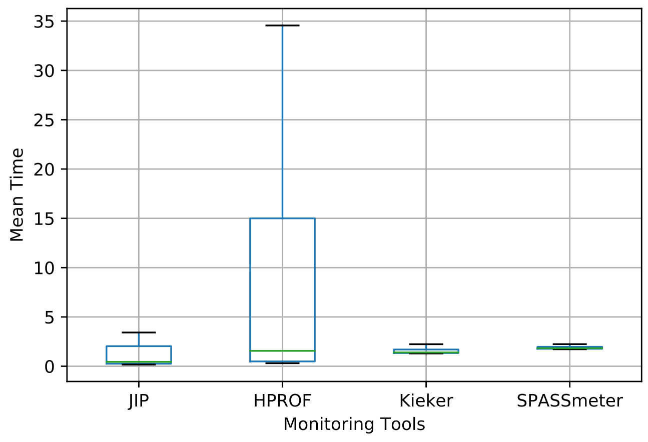


Figure 3.3: Overhead of Performance Monitoring Tools.

Finally, we analyzed the Java monitoring tools. That is, we assess the measurement overhead of JIP, HPROF, Kieker, and SPASS-meter. To this end, an experiment, in which we randomly selected 100 configurations from each subject system and repeated measurements five times with and without monitoring. Based on this comparison we calculated the introduced relative overhead (cf. Figure 3.3) and list the mean in the last column of Table 3.1.

We can see that the tool HPROF introduce a substantial overhead such that this tool cannot give us reliable performance values. Only tools JIP, Kieker and SPASSmeter have low overhead so that we decided for tool JIP (even if the mean execution overhead is higher), because it provides all features that we need for our scenario (including a data-flow graph).

3.2.2 Data Processing

After monitoring each of the subject systems, we obtain several performance log files containing execution times for each method. We obtain one log file for each subject system, each configuration, and each applied workload (a simplified log file can be found in Appendix A.7). Each log file is divided into three sections with different kinds of information. The head of the file shows the time stamp as well as the path to the file itself. We encoded the used configuration and workload in the path. The second area shows an inter-procedural control-flow graph[4] which represents calling relationships between the individual methods of the subject systems. Each line is composed of the number of calls, the time spent in the method, and the method name with the fully qualified class name. The last section contains a list of all methods which are sorted by their net time[5]. Based on these log files, the data processing is divided into three steps: analyzing the measurement influence, calculating relative performance of methods, and creating performance-influence models.

External Measurement Influences

Extracting actual performance values from configurable software systems requires executing the system under a specific configuration, workload, and environment. The execution environment is defined by the used hardware setup, the Operating System (OS), installed software, and several system settings. To enable reproducibility, we will delimit the measurement setup in Section 4.1. Every part of the environment might have an influence on our measurements. This might be a source of non-determinism, which influences the whole process of performance analysis. Therefore, we try to minimize this influence in an two-step approach: Reduce system influence and analyze Java-related influences.

[4]Inter-procedural control-flow graphs are tree-structured graphs representing the control flow of a program.

[5]Net time is the amount of time that was actually spent executing this method. Here, the time taken by calling other methods is factored out.

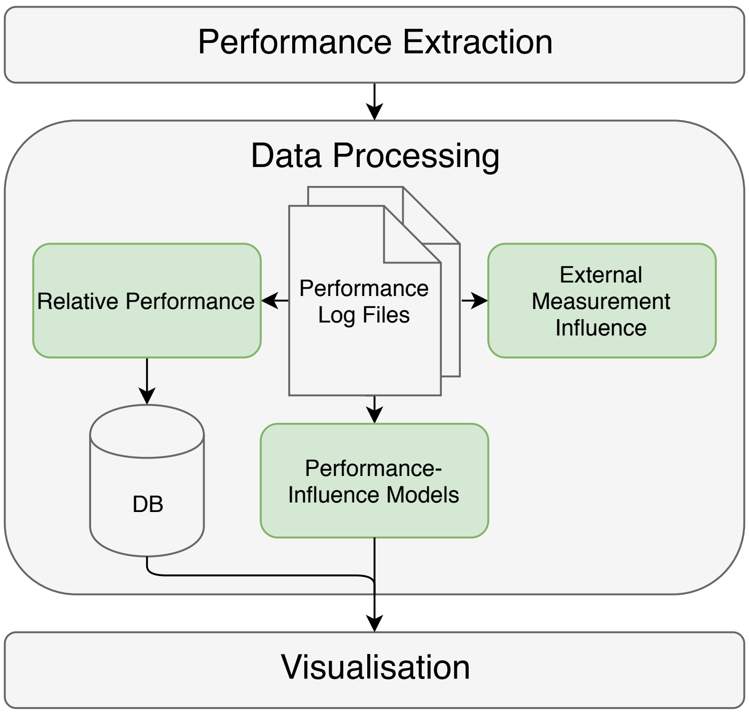


Figure 3.4: Data-processing Worklow.

System Influences. First, we try to reduce the influence of the system on our measurement. Therefore, we use separate machines dedicated to measure all test cases of an individual subject systems. Further, we ensured that there is only software installed which we need for the measurements. Hyper-threading of modern CPUs tries to balance the work between all available CPU cores. With the drawback that assigning a process to a specific core consumes additional, non-deterministic time. Therefore, we restrict the execution of the measurements to one dedicated CPU core. Another property of current CPUs is their ability to overclock. This allows the CPU to temporarily increase their clock frequency for different parts of the software execution. Again, this overclocking features results in non-deterministic performance behaviour and hence, we turned it off in our experiments.

Java Measurement Influences. Profiling of software that is executed in a runtime environment, such as the JVM, is affected by additional factors that influence performance. With Java, the source code first needs to be compiled into platform-independent Java bytecode. The Java bytecode gets interpreted inside the JVM to execute instructions. If parts of the bytecode are frequently used, then these parts were optimized. This process is called Just-In-Time (JIT) compilation. As a consequence, parts of the program may speed up during execution. Another source of measurement bias is the Garbage Collection (GC), which attempts to reclaim memory occupied by objects that are no longer in use by the program. Depending on the Java version, there are different strategies how and when the GC releases memory. However, we do not turn off JIT or the GC, because in real-world scenarios both are also active and we want to investigate performance as close as possible to software running in practice. Furthermore, Georges et al. [2007] surveyed that several researchers also not exclude JIT and GC overhead in their measurements.

However, we analyze the impact of the measurement influences. Therefore, we run the execution of each configuration and workload several times (each time with a five second delay). Afterwards, we are able to calculate the difference of the execution times of these measurements per configuration and workload.

Relative Performance

After monitoring the execution of the subject systems, we calculated the relative performance of each configuration and workload individually. Therefore, we identified the method with the highest execution time and set the relative performance to 1. The execution time of all other methods are scaled thereafter relative to the method with the highest execution time. Hence, we get relative performance values between 0 and 1 for all executed methods. With this approach, we are able to know which methods we first have to investigate to find performance hot spots and to improve overall performance of our subject systems. Furthermore, we can compare the relative performance of each method of different configurations (with the same workload) to calculate their configuration sensitivity and assessing their workload sensitivity by comparing the workloads (for one configuration) of each method.

Performance-Influence Models

To model the influence of configuration options to performance, we have to create performance-influence models (as described in Section 2.2.1). The general, idea behind these models is to learn the influence of individual configuration options, or combinations of options on performance. Therefore, we use a regression approach, which tries to relate dependent variables y (in our case performance) to independent variables X (in our case the configuration). A regression equation is linear when it is linear in the number of its parameter n, so that we can describe y by a function $y(X) = \omega_0 + \sum_{j=1}^{n} \omega_j \cdot x_j$. $X = (x_0, x_1, ..., x_n)$ is a vector containing all configuration options and $\omega = (\omega_0, \omega_1, ..., \omega_n)$ are the unknown parameters. Fitting the model is done by estimating ω. The regression techniques we use differ in the way how they estimate the values for ω.

In the following example, we assume that we have n measurements of one independent variable x yielding one value for y. Then we can calculate the line $\hat{y}$, which is an estimation of the original y, with the equations provided by Hahn and Shapiro [1967] (presented in 3.1).

$$\begin{aligned} \hat{y} &= \hat{\beta}_0 + \hat{\beta}_1 x \\ \hat{\beta}_0 &= \hat{y} - \hat{\beta}_1 \bar{x} \\ \hat{\beta}_1 &= \frac{\sum_{i=1}^{n} y_i x_i - \frac{(\sum_{i=1}^{n} y_i)(\sum_{i=1}^{n} x_i)}{n}}{\sum_{i=1}^{n}(x_i - \bar{x})^2} \end{aligned} \tag{3.1}$$

The fitted value $\hat{y}_i$ for a input value x_i may have a difference to the corresponding measured value y_i. The differences $e_i = y_i - \hat{y}_i$ are the errors (or residuals). The sum of all differences is the error of the model: $E(Y) = \sum_{i=1}^{n} e_i$. This example shows how to calculate models with one independent variable.

Linear Regression. Linear regression fits a linear model with the coefficients $\omega = (\omega_0, \omega_1, ..., \omega_n)$ and tries to minimize the residual sum of squares (RSS) for estimating the goodness of fit to the observed responses in the dataset, and the responses predicted by the linear approximation. One known drawback of this method is that linear regression becomes highly sensitive to outliers in the observed response, which might produce a large variance.

$$RSS(\omega) = \sum_{i=1}^{n}(y_i - \omega^T x_i)^2 \tag{3.2}$$

Ridge Ridge regression also uses RSS, but additionally includes a penalty on the size of coefficients ($l2$-norm). Here, $\alpha \geq 0$ is the complexity parameter that controls the amount of shrinkage. This means, the larger the value of α is, the greater the amount of shrinkage becomes and thus the values of ω become more robust to outliers.

$$RSS(\omega) = \sum_{i=1}^{n}(y_i - \omega^T x_i)^2 \;+\; \alpha \sum_{j=1}^{m}(\omega_j^2) \tag{3.3}$$

Lasso Lasso is a similar linear model like ridge, which estimates sparse coefficients (l_1-norm). The difference is its tendency to prefer ω with more parameters set to 0. Hence, it effectively reduces the number of variables upon which the solution is dependent. A possible drawback might occur if there is a group of highly correlated variables; then the lasso tends to select one variable out of this group, and ignores the others.

$$RSS(\omega) = \sum_{i=1}^{n}(y_i - \omega^T x_i)^2 \; + \; \alpha \sum_{j=1}^{m}(|\omega_j|) \tag{3.4}$$

Elastic Net The elastic net regression model includes the lasso and ridge regression methodology. It combines both penalty approaches into one formula. Elastic net is useful when there are multiple features which are correlated with one another. The strength of penalizing the size of coefficients is controlled with α and the ratio of l_1-norm and l_2-norm is controlled with the parameter p.

$$RSS(\omega) = \sum_{i=1}^{n}(y_i - \omega^T x_i)^2 \; + \; \alpha p \sum_{j=1}^{m}(\omega_j^2) \; + \; \alpha(1-p)\sum_{j=1}^{m}(|\omega_j|) \tag{3.5}$$

Huber Huber regression is another promising technique for learning a performance-influence model. It is used when the dataset contains some strong outliers. The influence of outliers which are far away is greatly reduced. The parameter ϵ regulates the outlier sensibility and α again regulates strength of penalizing the size of coefficients.

$$\begin{aligned} RSS(\omega, \sigma) &= \sum_{i=1}^{n}(\sigma + H_m(\frac{X_i\omega - y_i}{\sigma})\sigma) \; + \; \alpha \sum_{j=1}^{m}(\omega_j^2) \\ H_m &= \begin{cases} z^2 & \text{if } |z| < \epsilon \\ -(n+1)/2 & \text{otherwise} \end{cases} \end{aligned} \tag{3.6}$$

Decision Tree Decision trees are another possibility to model performance of configurable software systems. They split the input features (configuration options) into several regions and assign a prediction value to each region. Every split is chosen according to the highest reduction of impurity. Choosing the best fit means that they try to minimize the distance from the observations to the prediction. Decision trees can handle complex relationships (not only linear) by applying a cascade of rules. The depth of the tree usually describes how fine grained we can learn. If the depth is too high, we are likely to overfit (learn also noise of the input data), but if the depth is too small, we might underfit (we are not able represent our data at all). With decision tree regression, we are able to predict performance values from whole configurations.

Performance-Influence Models Accuracy

As part of the evaluation of our previous work, the trained models need to be determine their accuracy by predicting performance values of configurations that we have not used for learning. After that, we can compare the predicted performance values with the measured values. The distance between these values describes the accuracy with which the model can predict and so the accuracy of the model. In order to have test data available, we randomly divide the set of available measurements into a learning set (90%) and a test set (10%). The following models present the metrics we used for assessing the accuracy of the models.

Mean Absolute Error describes the average distance between observed (measured) values y and predicted value f. It can be divided by the mean value of all measurements to normalize the value, so we can compare MAEs of different value ranges.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - f_i| \tag{3.7}$$

Mean Squared Error (MSE) describes the squared average between observed values y and predicted value f. The squaring causes that errors > 1 have a bigger influence on the overall error. Because of the squaring, we are able to sum the values because they are always positive.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - f_i)^2 \tag{3.8}$$

R squared represents the fraction of predicted values that is explained by the model. The higher the R-squared value, the better the model fits the data. Total sum of squares (TSS) is defined as the sum over the squared distances from the measured values to the mean of all measurements.

$$\begin{aligned} TSS &= \sum_{i=1}^{n}(y_i - y)^2 \\ R^2 &= 1 - \frac{MSE}{TSS} \end{aligned} \tag{3.9}$$

3.2.3 Data Visualisation

The last step of our overall approach consists of novel visualisations presenting performance of configurable software systems. The goal of this thesis includes to provide support for developers with tools that help understanding performance hot-spots of configurable software systems. Therefore we introduce two independent visualisation tools (overview of the tools in Figure 3.5), which we will present in the following.

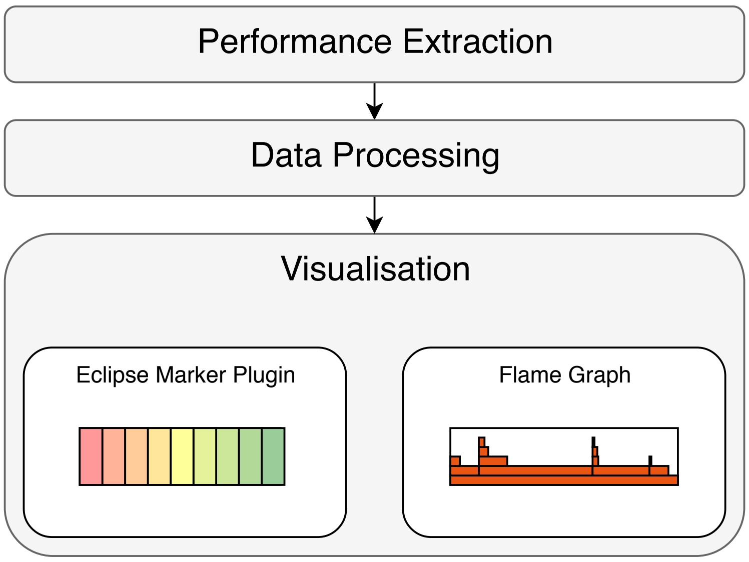


Figure 3.5: Visualisation Workflow.

Eclipse Marker Plugin

The Eclipse IDE is a platform for developing in different programming languages. It is an OSS framework which provides mechanisms to extend it with plugins. We developed an Eclipse marker plugin which is able to highlight lines of source code inside of the Eclipse workbench. The main purpose of our plugin is to highlight regions of source code to identify performance hot-spots. There are different modes for highlighting implemented. First, it is able to color methods of selected configurations according to their relative performance. This might be useful if developers want to analyze default configurations of their software. The next modes color the source code according to mean and median relative performance over all configurations. These modes identify performance distribution over all sampled configurations. We are also able to color the method's configuration sensitivity. Therefore, the differences of the methods relative performance values over all configurations is calculated by their standard deviation. These different modes are selectable via a drop down menu, which is integrated in the menu bar. To increase comprehensibility while working with this plugin we created an overview area for all modes, containing a list of all marked methods sorted by descending performance.

Flame Graphs

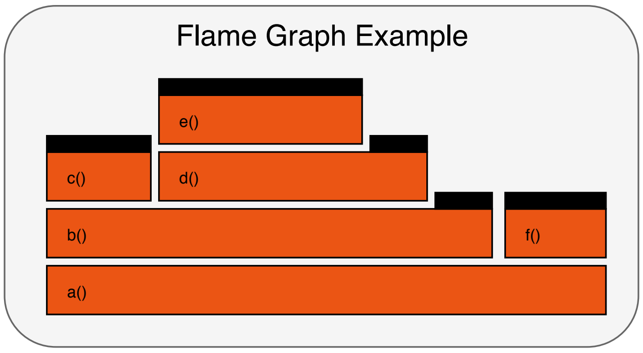


Figure 3.6: Example Flame Graph.

Flame graphs are designed to visualize different non-functional properties (e.g., runtime, heap) of software systems Gregg [2016]. They are usually presented with warm colors, because these visualisation should explain why CPUs were *hot* (busy). The flame-like shape also supports this naming. Initially, they were designed to visualize the sampled stack traces[6] (which can be extracted through Linux perf_events and DTrace). We adapted the idea of stack-trace-based flame graphs to encode the runtime of methods in flame graphs. Figure 3.6 shows an example flame graph. The y-axis encodes the call stack depth and the x-axis shows the the time of methods. Each rectangle represents a stack frame (a method), where the width shows the runtime of the method in the profile. The right-to-left ordering is alphabetical. The bottom to top ordering decodes the call flow of the program. For example, function *a()* from Figure 3.6 has called the functions *b()* and *f()*. The top edge (black bar) of the top rectangles shows how much CPU time each has spent. That means, even if the width of *b()* is bigger than the width of *f()*, the latter has spent more time on CPU. With the help of these visualisations we are able to understand a whole profile in one diagram.

[6]A stack trace is a list of method calls represented in a graph structure.

Chapter 4

Evaluation

The main question of this chapter and of this thesis is the following: *Can we identify performance hot-spots in configurable software systems?* In order to answer this question, we implement the approach described in the previous chapter and evaluate the following research questions:

RQ $_1$ How accurate are performance-influence models at method level learned by different techniques?

Here, we try to learn influence from configuration options on performance of each method. Because, we assume that performance hot-spots emerge on method level. Therefore, we utilize the presented regression models to indicate whether we can learn performance-influence models. A dependent question is:

RQ $_{1.1}$ What influences the learning of performance-influence models?

This question will help explaining the accuracy with which we can learn performance-influence models.

RQ $_2$ To which degree is the performance of individual methods sensitive to the used configuration?

The sensitivity of methods to the different configurations might be an indicator for performance hot-spots.

RQ $_3$ How severe are external influences on performance measurements for identifying hot-spots in the source code?

To answer this question, the non-determinism of the measurements is discovered by analyzing the differences in the repetitions of the performance extraction process with each configuration.

RQ $_{3.1}$ What external influences exist that bias our measurements?
We sum up the external influences to be able to explain which parameter bias the extracted performance data.

RQ $_{3.2}$ Are the external influences workload sensitive?
Here, we reason about the choice of the different workloads that are used to run our experiments.

This chapter is organized as follows: Section 4.1 defines the test environment we used to run our experiments. In Section 4.2 we present our subject systems. This includes a general description of each system motivating the selection of these three systems. After that, we present the results in Section 4.3 and answer the research questions in Section 4.4. Finally, we analyze the sources of measurement bias and discuss how to support a developer in finding performance hot-spots using tailor-made visualizations.

4.1 Measurement Setup

All experiments were conducted on three dedicated computers, each with an Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz, 32GB DDR4 RAM, and 500 GB SSD. We run Ubuntu 16.04.3 LTS 64 bit with Python 3.5.2 and Java OpenJDK Runtime Environment version 1.8.0_171. Except from Python and Java, there is no additional software installed which could influence the measurements.

As already mentioned in Section 3.2.2, we configured each experiment execution to use one physical CPU core without overclocking. Before the measurements, we execute the `turbo_boost.sh` script (the script is presented in the Appendix A.6). This script searches for all available CPU cores and disables or enables the turbo boost functionality of Intel CPUs with help of the `wrmsr` tool. Each measurement refers to the execution of a single subject system with one specific configuration and one workload. We automated the measurements and data collection using python. An example execution is: `taskset 0x1 python3 profiling_script_random.py` which forces the Linux scheduler to keep the called process on the defined CPU. After this, other processes will be assigned to the remaining cores.

4.2 Subject Systems

In the following, we present the subject systems we use in our evaluation. We selected three representative Java projects in order to analyze their performance behaviour and to subsequently find performance hot spots. These subsystems are the password-hashing framework Catena, the SQL database H2 and Sunflow, a rendering system for photo-realistic image synthesis. All three systems are OSS written in Java. Each of them is a configurable software system with the possibility to customize its workload.

To seek out the configuration space of the subject systems, we followed the guideline provided by Han and Yu [2016]. Because each software system can be implemented and documented in different ways, we had to study all artifacts that are publicly available to users including documentations (e.g., readme files, user manuals, and online help pages), configuration files (e.g., default configuration examples, user manuals), and the source code itself. The ability to search the source code and test files, is another advantage of using OSS. After identifying all configuration options, we created the corresponding feature models. We use these feature models to calculate the whole configuration space and to sample a set of configurations for the experiments. Besides the identification of the configuration options, the possible workload of each system must be identified for building a proper benchmark. In the following, we explain each subject system in detail.

4.2.1 Catena

Catena is a flexible and provable secure password-hashing framework. As a finalist of the *Password Hashing Competition*[1], Catena has convinced because of its flexibility of the internal graph-based structure (brings a huge time-memory tradeoff). With different configuration options, users can tune Catena to consume more time and/or memory to hash passwords. This property prevents hashes from being computed massively in parallel on clusters of graphics cards, which is a common attack scenario.

The Catena project provides different versions of the documentation (provided by Forler et al. [2014]) as well as with the implementation itself (available on `https://github.com/medsec/catena-java`) which contains the implementation of Catena and some default variants. We derive the set of configuration options by searching the documentation files and by analyzing the source code. The result of our analysis represents the feature model in Figure 4.1.

The Catena password-hashing framework has eight features, three binary and 5 numerical (numerical features are represented by their name, min value and max value; separated with underscores) ones. This results in a configuration space of **3,145,728,000** different possibilities (variants) to configure Catenas password hashing mode. We extracted the following features:

[1]Password Hashing Competition ran from 2013 to 2015 as an open competition to establish a standard in hashing passwords to be secure against attackers (Aumasson [2015]).

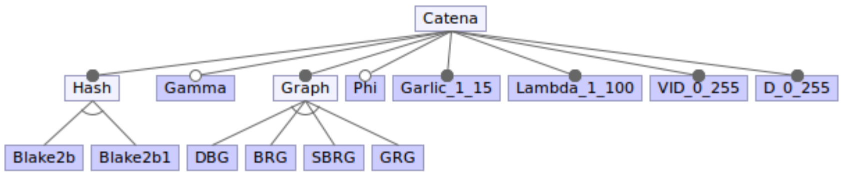


Figure 4.1: Catena Feature Model.

Hash	Boolean parameter defining the used hash function.
Gamma	Boolean parameter enabling the selection of a salt-dependent memory accesses layer to be resistant against attacks on dedicated hardware (e.g., ASICs[2]).
Phi	Boolean parameter enabling additional security properties[3].
Graph	Defines one out of four different graph structures, which, in turn, defines the amount of necessary memory and time to compute the final hash.
Garlic	defines time and memory requirements in a range from 0 to 255. We restricted the range from 1 to 15 to have a reasonable execution time.
Lambda	Defines the depth of the graph in a range from 1 to one 100.
VID	Encodes the unique version of Catena in a range from 0 to 255.
D	Numerical parameter that is hashed together with other parameters to make up the tweak (increasing the additional computational effort for an adversary) in a range from 0 to 255.

To execute Catena, we have to pick a configuration and we have to define a workload. A workload for Catena consists of three parameters: the password to be hashed, an user-defined salt, and the output length of the hash. Table 4.1 presents the three workloads we used. The first workload consists of an empty password string, an empty salt string, and an output length of one. With this workload we tried to minimize the amount of computations Catena has to perform. The second workload is exactly the half in each parameter and

[2]ASICs are hardware implementations of functions which are specialized to perform exactly a specific task.

[3]If Phi is enabled, Catena is resistant against parallel computations of hashes. But Phi has the drawback that Catena becomes vulnerable against other attacks (Lucks and Wenzel [2016]).

the last workload is an example workload we have taken from the Catena test vectors repository[4].

#	Password	Salt	Output Length
1	”	”	1
2	’012345012345012345012345’	’6789ab6789ab’	32
3	’012345012345012345012345’+ ’012345012345012345012345’	’6789ab6789ab’+ ’6789ab6789ab’	64

Table 4.1: Catena Workloads.

4.2.2 H2

H2 is a database engine written in Java with the aim to be one of the fastest OSS databases. In comparison to other databases (e.g., HSQLDB, Derby, PostgreSQL, and MySQL) H2 is faster but consumes more memory[5].

The H2 project provides a detailed online documentation[6] for configuration options, usage scenarios, benchmark results, and other useful information. We extracted the set of configuration options from the documentation and created the feature model (shown in Figure 4.2). In total, we extracted fourteen binary options and two numerical options which result in a configuration space of 3,920,000,000 different variants.

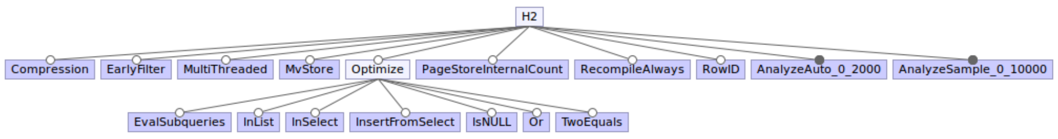


Figure 4.2: H2 Feature Model.

We excluded some options, because they do not influence the performance behaviour of the database (e.g., *databaseToUpper* writes database names always in uppercase or *defaultEscape* defines the default escape character) or produce exceptions during execution of the performance tests (e.g., changing

[4]The Catena test vectors are available on `https://github.com/medsec/catena-test-vectors`.

[5]Performance benchmark results and descriptions of the used test cases are available on `http://h2database.com/html/performance.html`.

[6]Documentation of parameters is available on `http://h2database.com/javadoc/org/h2/engine/DbSettings.html`.

the default value of *maxQueryTimeout* causes some queries to not be executed). We extracted the following configuration options:

Compression	Compresses data when stored.
EarlyFilter	Allows table implementations to apply filter conditions early on.
MultiThreaded	Enables multi threading.
MvStore	Enables the use of the MVStore storage engine.
OptimizeEvalSub-queries	Optimizes subqueries that are independent of the outer query.
OptimizeInList	Optimizes the *IN(...)* and *IN(SELECT ...)* operations.
OptimizeInSelect	Optimizes the *IN(SELECT ...)* operation.
OptimizeInsert-FromSelect	Enables insertion into tables from queries directly by passing temporary disc storage.
OptimizeIsNull	Enables the use of indexing with *IS NULL* checks.
OptimizeOr	Optimizes the *OR* operation.
OptimizeTwoEquals	Enables the optimization of the equality check.
PageStoreInternal-Count	Enables updating the internal row counts.
RecompileAlways	Enables that prepared statements always gets recompiled.
RowID	Adds the pseudo-column *_ROWID_* to each table.
AnalyzeAuto	Number which defines the necessary number of changes of rows in a table before the table gets analyzed.
AnalyzeSample	Number which defines how many samples of a table gets analyzed.

To create a meaningful workload, we vary the size of the data stored in the databased for each workload. To this end, we initialized the tables of the database with 10,000, 55,000, and 100,000 rows for the respective workloads. To generate load on the database, we use the benchmark that was provided by the vendor. This benchmark has been used previously to compare different database engines.

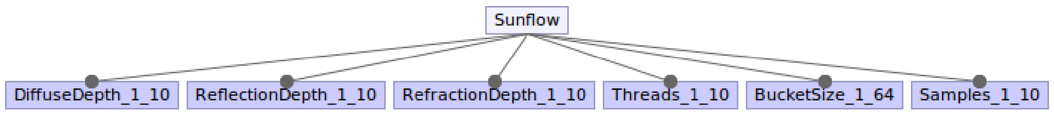


Figure 4.3: Sunflow Feature Model.

4.2.3 Sunflow

The third subject system is the rendering engine Sunflow. It is built around a ray tracing core to create photo-realistic images with the help of many different features such as camera motion blurring, lightmap generation, texture mapping, normal mapping, and other features.

We analyze the performance of the process of rendering an image. Therefore, Sunflow already provides a benchmark which analyzes the resulting image (by time and by image quality) while modifying different parameters. We extracted all parameter that configure (*DiffuseDepth*, *ReflectionDepth*, and *RefractionDepth*) to create the feature model of Sunflow. Moreover, we identified three more parameters (*Threads*, *BucketSize*, and *Samples*) that were hard-coded in the source code of the benchmark. Although Sunflow provides additional configuration options for different use cases, we wanted to stick to the process of generating one image by using the standard scene (the Cornell box with two tea pods and two glass balls).

In total, we have a configuration space of 6,400,000 variants. In the following, we present a short description of all configuration options we used:

DiffuseDepth	Controls the maximum depth of diffuse light paths in the scene.
ReflectionDepth	Controls the maximum number of reflections a ray can do in the scene.
RefractionDepth	Controls the maximum number of refractions a ray can do in the scene.
Threads	Number of threads used for calculating the image in parallel.
BucketSize	Defines the size of a moving window.
Samples	Specifies the number of times the scene gets rendered.

As the different levels of workload, we vary the resolution of the generated image (64×64 pixel, 128×128 pixel, and 256×256 pixel).

In total we performed 21.000 measurements with each subject system. That is made up of 1000 random sampled configurations, 3 different workloads, and 7 repetitions of the measurement. We parallelize the measurements of the three subject systems, but at the end, we measured a total of 21 days for each subject system in parallel. During this time, we collected 100 GB of profiling data.

4.3 Results

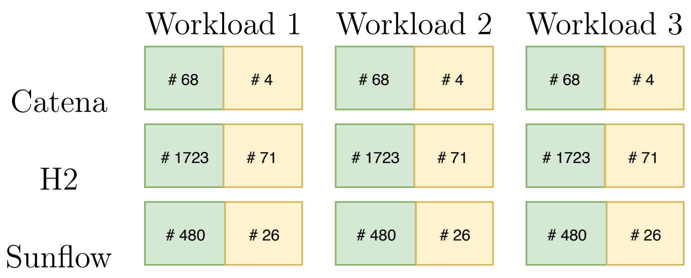
4.3.1 Performance-influence Models at Method Level

To learn the influence of the individual configuration options on performance, we applied the regression models presented in Section 3.2.2. Before we start training the model, we check if we have enough data to train and test the model. We learn our models only for methods if we have at least 11 measurements.

$< f_1, f_2, ..., f_8, interaction_{f1,f2}, ... >$	performance (ms)
$<$ 0, 0, 1, 0, 10, 36, 144, 48, 0, ... $>$	345.75
$<$ 0, 0, 1, 0, 11, 1, 144, 35, 0, ... $>$	30.1
$<$ 0, 0, 1, 0, 12, 25, 213, 200, 0, ... $>$	1129.5
...	...

Table 4.2: Catena – Example Input/Output Vector.

An example set of input-output vectors is shown in Figure 4.2. To build the input vector for our regression models, we encoded the configuration as input (f_1 corresponds to configuration option 1), extended this input vector by all pair-wise interactions of the configuration options, and learned the corresponding performance value of the method.

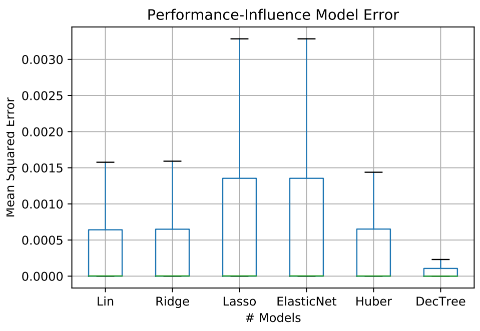


	Workload 1		Workload 2		Workload 3	
Catena	# 68	# 4	# 68	# 4	# 68	# 4
H2	# 1723	# 71	# 1723	# 71	# 1723	# 71
Sunflow	# 480	# 26	# 480	# 26	# 480	# 26

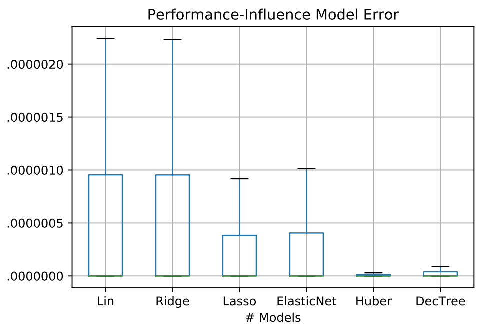
Table 4.3: Proportion of Learned Performance (# is the Number of Methods).

We train each method with all regression techniques and assessed their individual prediction accuracy by predicting all configurations in our test set.

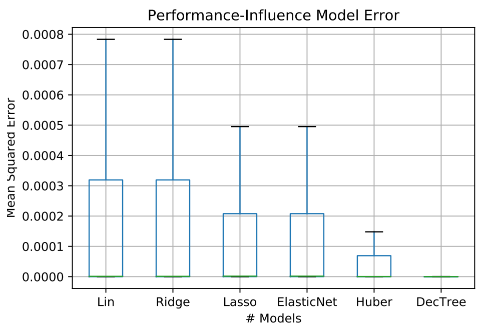
The result of the training process is shown in Table 4.3. This table visualizes the proportion of performance that we have learned (green) with our model, in contrast to the performance that we could not learn (yellow). Each field consists of the number of methods (#) that together make up the portion of performance. It is noticeable, that we are able to predict half of the performance of each system with each workload even if we can assess the performance of about 95% of the methods.



(a) Catena – Prediction Error.



(b) H2 – Prediction Error.

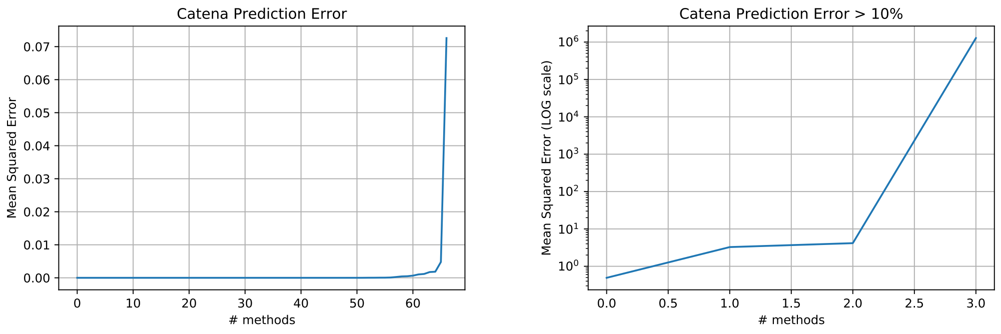


(c) Sunflow – Prediction Error.

Figure 4.4: Comparison of Prediction Error over all Subject Systems and Regression Model without Outliers.

An overview of the regression models is presented in Figure 4.4. Each boxplot comprises the error of all methods that we tried to learn with the individual regression models. We can see, that the median error of all models over all subject systems is close to zero. We also see that Linear Regression and Ridge Regression as well as Lasso and Elasic Net perform almost equally. However, Huber Regression is always better than former regression models. With decision trees, we achieve the best results of prediction performance by configuration. Nevertheless, it is not possible to get the influence of the individual features or interactions with decision trees.

Next, we analyze the errors of the regression techniques that predict the performance of a method. Therefore, we assessed the MSEs between the predicted and the actual performance values of each method using the test set. We utilized the technique with the smallest error, to have always the best for further evaluation. We split the figures by 10% prediction error to provide a better illustration of the error distribution.



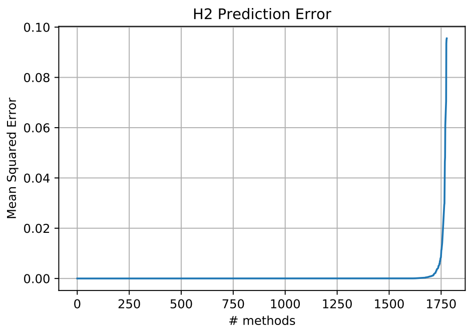
(a) Learned Methods. **(b)** Unlearned Methods.

Figure 4.5: Catena – Comparison of Learnability over all Methods (# 72).

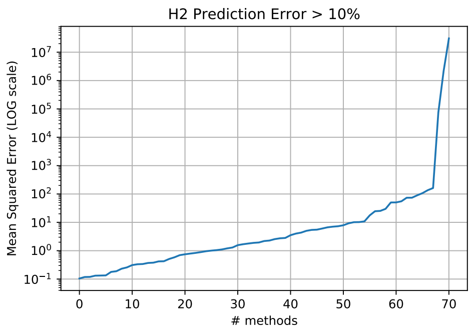
For Catena we had in total 82 methods available for learning. We had to exclude 10 of them from our learning approach, because they were less than 11 times executed under different configurations. The results of learning the influence of the remaining 72 methods are shown in Figure 4.8. In Figure 4.5a, we show the 68 methods that we are able to learn with an MSE of less than 10%. Also the 23 methods which are not influenced by any configuration option (presented in Section 4.3.2 and in Appendix A.2.1) belong to those methods. Nevertheless, we were not able to predict the performance of 4 methods, that makes 5.5% out of all methods available for learning. These methods are presented in Figure 4.5b aligned with an logarithmic MSE. Later (in Section 4.4), we discuss possible reasons why we could not learn these methods.

Figure 4.6 shows the results of the learning the influence of configuration options of H2. In total, we had 1865 methods available from which we excluded 15 methods because of the lack of data. We were able to predict the performance of 1794 methods, but just like with Catena, we are not able to predict 3.8% (71) of the methods with an MSE of less than 10%. Figure 4.6b also show that the error of 5 methods increases very strongly.

The results of learning the influence of Sunflow are shown in Figure 4.7. We tried to learn 496 methods (we had to exclude 10 methods) from which we are able to predict 480. Again, we were not able to predict the performance

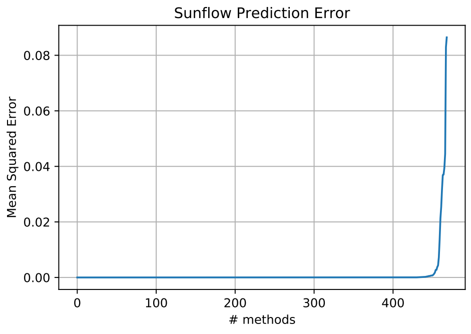


(a) Learned Methods.

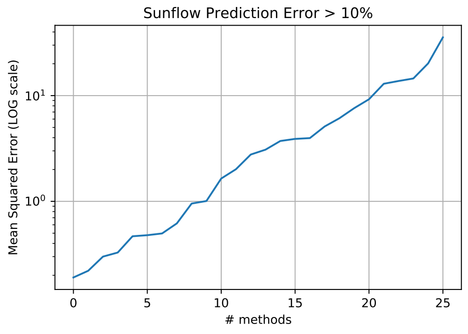


(b) Unlearned Methods.

Figure 4.6: H2 – Comparison of Learnability over all methods (# 1850).



(a) Learned Methods.

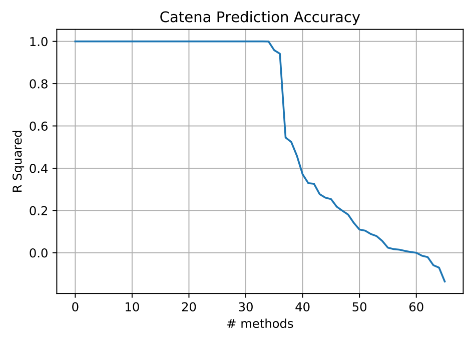


(b) Unlearned Methods.

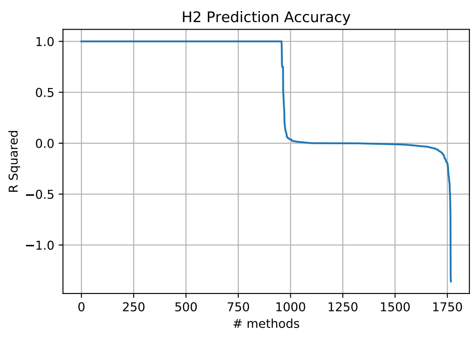
Figure 4.7: Sunflow – Comparison of Learnability over all methods (# 496).

of 26 methods (which are 5.1%). But in contrast to the unlearned methods of Catena and H2, the MSE increases nearly linearly.

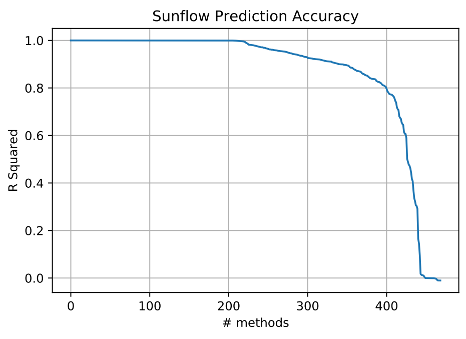
To evaluate how much of the variance in the data we can explain with our models, we assessed the R squared coefficient of each model for the methods of the subject systems. Figure 4.8 shows proportion of the variance in the performance values that is predictable from the configurations. We can see that the R squared score of the methods of each subject system goes beyond zero. This indicates that, especially with the methods whose score is beyond zero, all regression techniques of our model are not able to explain the performance based on the configuration. Hence, we are not able to predict the performance values of the affected methods. That coincides with our observations of the sum of squared errors in Figures 4.7, 4.6, and 4.8.



(a) Catena – R squared.



(b) H2 – R squared.



(c) Sunflow – R squared.

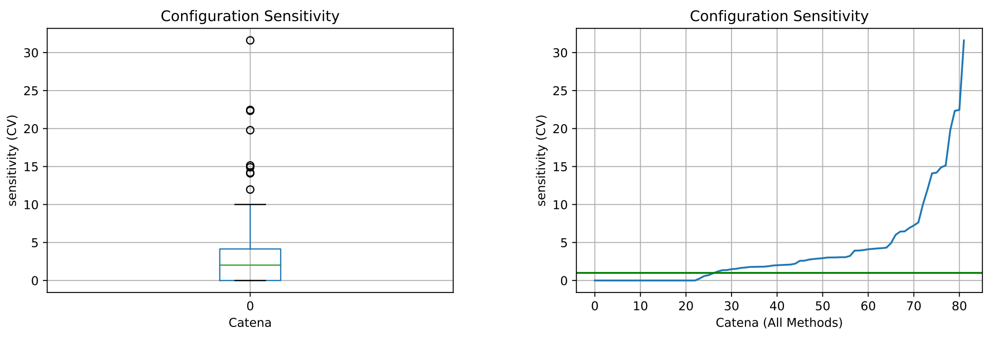
Figure 4.8: Performance-Influence Model Validation.

4.3.2 Configuration Sensitivity

To assess the sensitivity of each individual method regarding the chosen configuration of the respective configurable software system, we calculated the difference in the performance values of the methods over each configuration. To this end, we calculate the standard deviation of the performance values to get the amount of variation of the performance of each method depending on the configurations. Then, we divide the standard deviation by the mean to calculate the coefficient of variation (CV)[7].

We summarize the configuration sensitivity of the methods of Catena in Figure 4.9. The Subfigure 4.9a shows the distribution of the CV-values of all methods as a box plot with outliers. Subfigure 4.9b marks the CV-value of 1 (the green line). This value divides all methods into a configuration insensitive (below) area and configuration sensitive (above) area. We can see that there

[7]CV is also known as the relative standard deviation, which describes the ration of standard deviation to the mean. Values above 1 have a high variance, which describes in our case a high configuration sensitivity

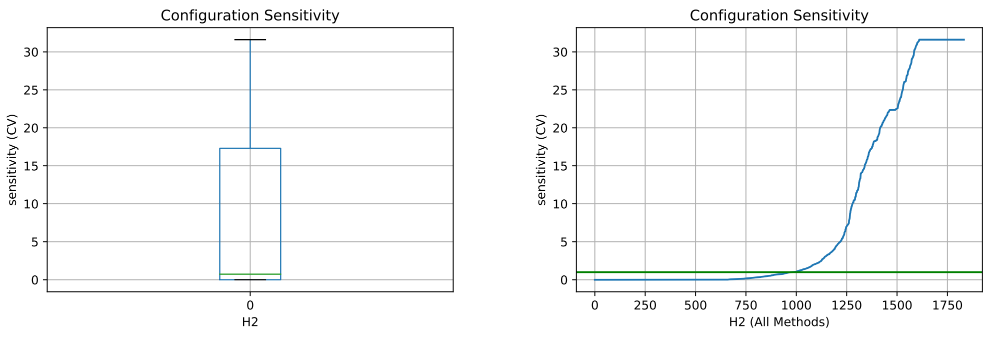


(a) Boxplot with Outliers.

(b) Whole Distribution

Figure 4.9: Catena - Coefficient of Variation in Configuration Sensitivity of Methods (82).

is a great difference in the configuration sensitivity of the different methods. Most methods are weakly sensitive, but there are some methods which are heavily sensitive to different configuration options. That is, we observe that more than half of the methods' performance changes depending on the chosen configuration.



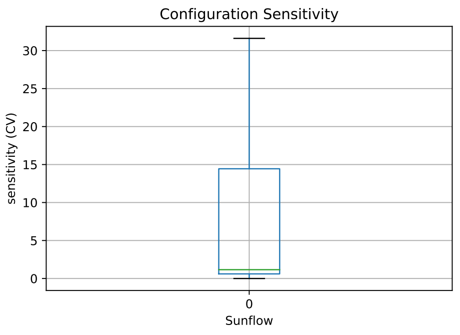
(a) Boxplot with Outliers.

(b) Whole Distribution

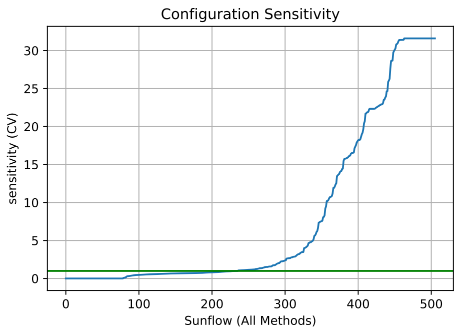
Figure 4.10: H2 – Coefficient of Variation in Configuration Sensitivity of Methods (1865).

Figure 4.10 show the configuration sensitivity of the H2 database engine. We investigated the 1850 methods that are executed during the test cases. We observe that about half of the methods are weakly sensitive to different configurations, which is shown by the height of the mean in the box plot and the CV-marker line. For the remaining methods, the sensitivity increases rapidly until they plateau, where around 250 methods have the same sensitivity.

The configuration sensitivity of the methods of the rendering engine Sun-



(a) Boxplot with Outliers.



(b) Whole Distribution

Figure 4.11: Sunflow – Coefficient of Variation in Configuration Sensitivity of Methods (506).

flow is shown in Figure 4.11. Again, half of the methods are insensitive to the chosen configuration. Furthermore, the sensitivity increases rapidly until a plateau with the highest sensitivity. The difference to H2 is that there are less methods with no sensitivity, but more methods that are weak sensitive.

4.3.3 External Measurement Influence

External influences such as the garbage collector or operating-system processes can substantially influence the reliability of our fine-grained performance measurements. Remember that we measure at method level with a resolution within milliseconds. For example, a context switch in the operating system can lead to severe measurement bias. Hence, we restricted possible sources of non-determinism to reduce their influence (fixed the hardware setup and minimal amount of installed software). To further quantify the remaining influence of measurement bias, we repeated each test case 7 times.

Figure 4.12 shows the standard deviation of Catena for the three defined workloads (defined in Section 4.1) for all sampled configurations using box plots. In about half of the configurations the mean relative standard deviation is only about 2.4%. Furthermore, changing the workload does not influence the measurement bias.

For H2, the picture changes. Figure 4.13 shows that there is a difference in the variance of the standard deviations depending on the used workload. The variance of the standard deviations is the biggest with the first workload, the lowest with the second workload. We believe that with increasing runtime the relative influence of measurement bias decreases. For example, JIT compilation should have the highest influence at the start of the program and the longer the program runs, the smaller is the relative influence on the overall runtime.

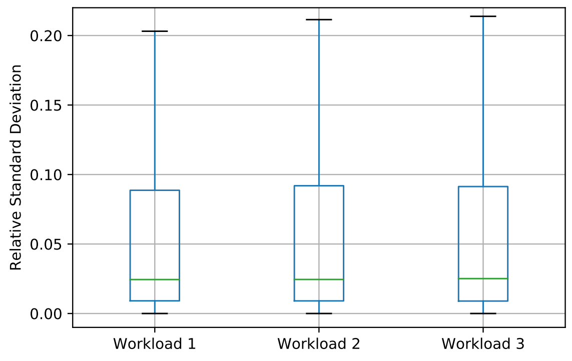


Figure 4.12: Catena - Distribution of the relative standard deviations for each configuration of a specific workload.

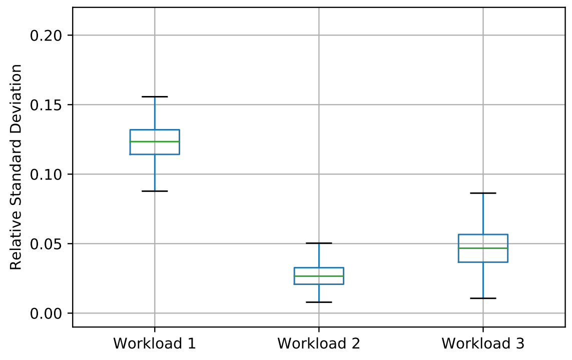


Figure 4.13: H2 - Distribution of the relative standard deviations for each configuration of a specific workload.

For Sunflow, we see a similar picture as for H2 (cf. 4.14). The influence of external factors, as well as the variance in the influences decrease from the first workload to the third.

Since Catenas measurement variance remains the same for all workloads, we changed the parameter *Garlic* from a maximum of 15 to 17 (only for this experiment). Hereafter, the tests run for four times the time (increasing *Garlic* by one doubles the runtime). We observe in Figure 4.15 now a similar trend as for the other subject systems: a decrease in the measurement bias.

Analyzing the measurement bias shows that there is a proportion of non-determinism in our measurements over all subject systems and workloads. Overall, we can see that for all subject systems the measurement bias is low (usually between 2% and 6%) such that we conclude that we obtain robust

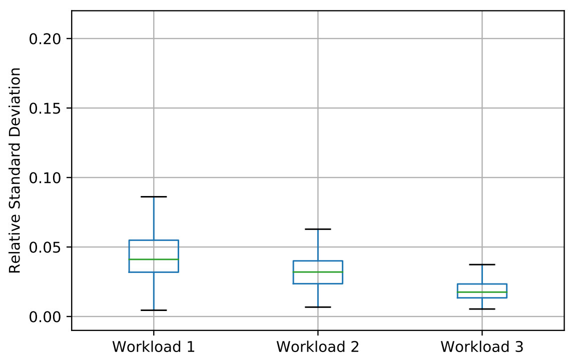


Figure 4.14: Sunflow - Distribution of the relative standard deviations for each configuration of a specific workload.

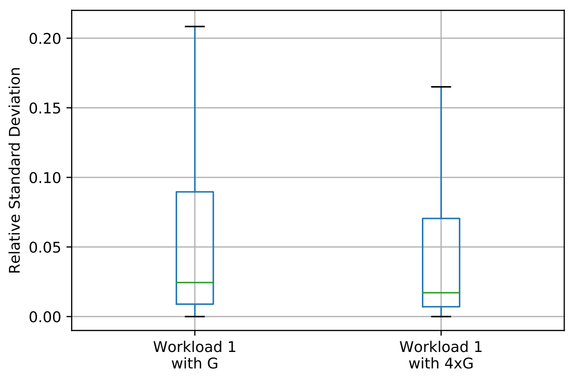


Figure 4.15: Catena - Measurement Influence with all Configurations (#820).

and reproducible performance measurements.

4.4 Discussion

RQ $_1$ *How accurate are performance-influence models at method level learned by different techniques?*

In order to answer RQ $_1$ we learned the influence of configuration options of three software systems for each of their methods. We created a dataset, which combines each configuration option and each pair-wise combination of these options as independent variables and the performance of each method as dependent variable. We utilize different regression techniques to model the performance of each method. By picking the best regression technique per method, we are able to learn the performance of about 95% of the methods. For about 5% of the methods (Catena: 5.5%, H2: 3.8%, and Sunflow: 5.1%), our model was not able to predict within an acceptable precision. But it is noticeable that, while successfully learning the performance-influence models of 95% of the methods, we are only able to explain half of the performance of each system. To this end, we identified the performance hot-spots of the three subject systems: These 5% of methods that we could not learn with our model are also those, which together unite half of the performance.

RQ $_{1.1}$ *What influences the learning of performance-influence models?*

For each subject system, we were not able to learn the performance of about 5% of the methods. RQ $_{1.1}$ will explain possible sources that influence the learning process. Therefore, we investigate the methods of Catena which we are unable to learn.

Table 4.4 presents the five methods which are above our error threshold (also presented in Figure 4.5b).

Best Model	Prediction Error (MSE)	Method
Decision Tree	955,754.56	...DoubleButterflyGraph:graph
Linear Regression	3.41	...Catena:flap
Elastic Net	1.43	...SaltMix:gamma
Linear Regression	0.14	...Catena:phi

Table 4.4: Catena - Unlearnable Methods.

We suspect three different reasons:

1. The GC activity might be the first source of non-deterministic behaviour. During our measurements, we did not log the GC activity during the measurements, but while examining the problematic methods of Catena (presented in the Appendix A.2 – A.5), we found out that in three out of the four methods the command `System.arraycopy(...);` is involved. It frees in each iteration of the surrounding loop a bunch of memory. In case of the DoubleButterflyGraph this command is nested in two loops which might be a reason for the high prediction error.

2. The next reason might be that our model is too weak for those methods. We utilized only individual features and their pair-wise interactions. Those methods might be affected by higher-order interactions. Another possibility is that the models have not learned that some features deactivate the execution of some methods. But at least the decision tree regression should be able to identify this behaviour. However, this leads us to the third possible reason.

3. The third reason might be that we have too few data. The Graph feature for example decides between 4 different graphs, that are implemented by default in Catena. The DoubleButterflyGraph is one of them and the GenericGraph represents the other three graphs with its three different indexing mechanisms. Therefore, we have only about 250 configurations (1000 randomly sampled configurations divided by 4 different graphs) in which the DoubleButterflyGraph is selected. This might be not enough.

RQ $_2$ ***To which degree is the performance of individual methods sensitive to the used configuration?***

To answer this question, we considered each subject system individually. Catena has 82 methods in total, 23 (28%) of them (listed in Appendix A.2.1) are not influenced by any configuration option. This may come from the accuracy (1 ms) during measuring the performance of the systems. But even if we had used another monitoring tool with a higher precision, the influence would remain very low. Furthermore, we identified 5 (6%) methods (listed in Appendix A.2.1) with the highest relative standard deviation. These methods are strongly influenced by the choice of configuration options. Also, the methods of the subject systems H2 and Sunflow differ in their sensitivity to configurations. With H2, we analyzed 1865 methods in total, 673 (37%) of them are not influenced, about 296 (15%) are weakly influenced but the remaining 896 (48%) methods are highly influenced by configuration options. A similar distribution also applies to Sunflow. Here, 82 (16%) out of 506 methods are not influenced by any configuration options and 165 (32%) methods are weakly influenced, which corresponds to 48% of all methods. The other methods (52%) are strongly influenced. Finally, we can conclude that about half of the methods are not related to configuration options with respect to a relevant performance behaviour and that those methods, which are most sensitive to configurations are also the same which we could not learn with our performance influence model – the performance hot-spots.

The analysis of the configuration sensitivity of methods helps identifying relevant methods that are influenced by configuration options. To this end, in future analysis, we can concentrate our approach to this methods that are strong influenced. Methods that are not influenced by any option can be measured once, to know their performance, so we can reduce measurement effort explicitly. With the coefficient of variation as estimator for configuration sensitivity we also know how sensitive the methods to configurations are. When the mean value is close to zero, the coefficient of variation will approach infinity and is therefore sensitive to small changes in the mean.

RQ $_3$ *How severe are external influences on performance measurements for identifying hot-spots in the source code?*
The mean external influence on our measurements is below five percent for all subject systems and workloads except for the H2 database engine. This high variance in the measurements of H2 with workload 1 might lead to difficulties when learning an performance-influence model. Sunflow as well as Catena reached a mean variance of 2.4% (Catena with increased *Garlic* even 1.7%) for their measurements with the third workload.

The variance of the input sensitivity of Catena does not change, because we defined the workload as the length of the input that is to be hashed. Since Cantena's hash function has a standard output length, the workload inside Catena is always the same. With Sunflow, the variation of the measured performance decreases with increasing workload, which means that the measurement of performance stabilizes with increasing runtime. So Sunflow and H2 are both sensitive to different workloads.

RQ $_{3.1}$ *What external influences exist that bias our measurements?*

The list of external influences includes all possible sources of non-determinism that bias our measurements. During our measurements we fixed the test setup to exclude the hardware that we used. We also fixed the hyper-treading and the over clocking of the CPU. A part of the measurement influence might come from the OS and other installed software because it runs simultaneously with our measurements. The load balancing module of the Intel processor forces concurrent processes to run on the other CPUs, but all processes still use the same memory. We think that a big portion of the influence is Java-specific. The JIT does have an influence on short running experiments. The GC however, is called by the JVM and stops the program execution to free memory.

RQ $_{3.2}$ *Are the external influences workload-sensitive?*

As we showed in Section 4.3.3, the portion of the measurement bias tends to decrease with increasing runtime. The workload usually controls overall runtime. If the runtime increases, the measurement bias seems to reduce. However, with the database engine H2, the measurement bias increases again with the third workload, which indicates that the measurement bias does behave differently for different software systems. But to sum up we can confirm that the external influences are sensitive to changing workloads.

4.5 Detailed Analysis

To be able to understand possible performance hot spots of software systems, we designed two visualisation tools. Both are able to frame the hot-spot methods in contrast to the other methods of the corresponding software. We used these tools to discuss possible performance issues of the Catena password hashing framework with one of Catenas developers. He knew that the graphs consume a lot of the performance, but with the visualisation tool, he understands how much.

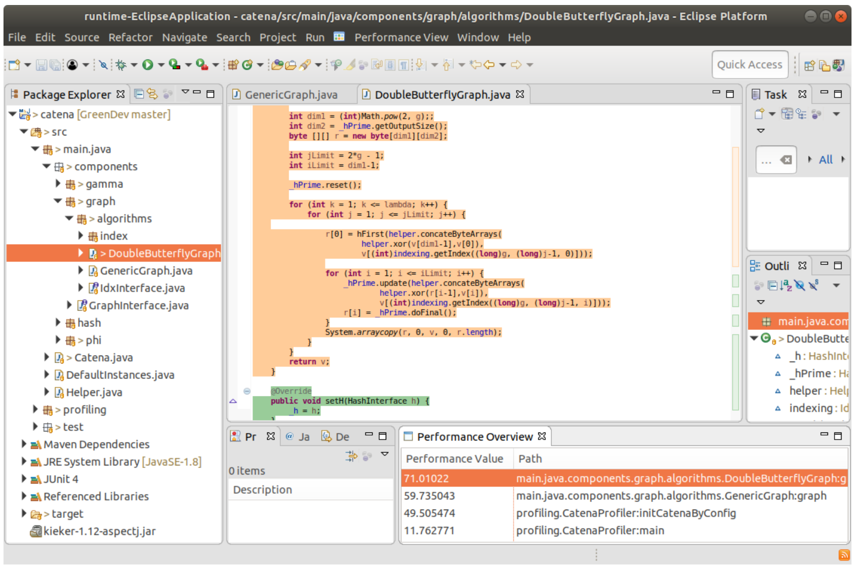
4.5.1 Eclipse Marker Plugin

The first tool we want to present is an Eclipse plugin for showing results of the performance analysis approach. The plugin provides the possibility to color each method of the open project. Therefore we assign each method a value between 0 and 1, which maps then to a color range (from red to green). Our plugin provides different modes which performance metric to show from the project. We provide:

- The relative performance of each method of one defined configuration. All methods are mapped relative to the method with the highest runtime, which gets the value of 1 (red).
- The mean value of each method over all configurations (see Figure 4.16). Provides an overview of the mean performance.

- Maximum value of all configurations (see Figure 4.17). Here, each method gets its color according to its worst performance in all measured configurations.

- Configuration sensitivity of all methods is provided by the standard deviation of the performance of each individual method.



```java
        int dim1 = (int)Math.pow(2, g);;
        int dim2 = _hPrime.getOutputSize();
        byte [][] r = new byte[dim1][dim2];

        int jLimit = 2*g - 1;
        int iLimit = dim1-1;

        _hPrime.reset();

        for (int k = 1; k <= lambda; k++) {
            for (int j = 1; j <= jLimit; j++) {

                r[0] = hFirst(helper.concateByteArrays(
                        helper.xor(v[dim1-1],v[0]),
                        v[(int)indexing.getIndex((long)g, (long)j-1, 0)]));

                for (int i = 1; i <= iLimit; i++) {
                    _hPrime.update(helper.concateByteArrays(
                            helper.xor(r[i-1],v[i]),
                            v[(int)indexing.getIndex((long)g, (long)j-1, i)]));
                    r[i] = _hPrime.doFinal();
                }
                System.arraycopy(r, 0, v, 0, r.length);
            }
        }
        return v;
    }

    @Override
    public void setH(HashInterface h) {
        _h = h;
```

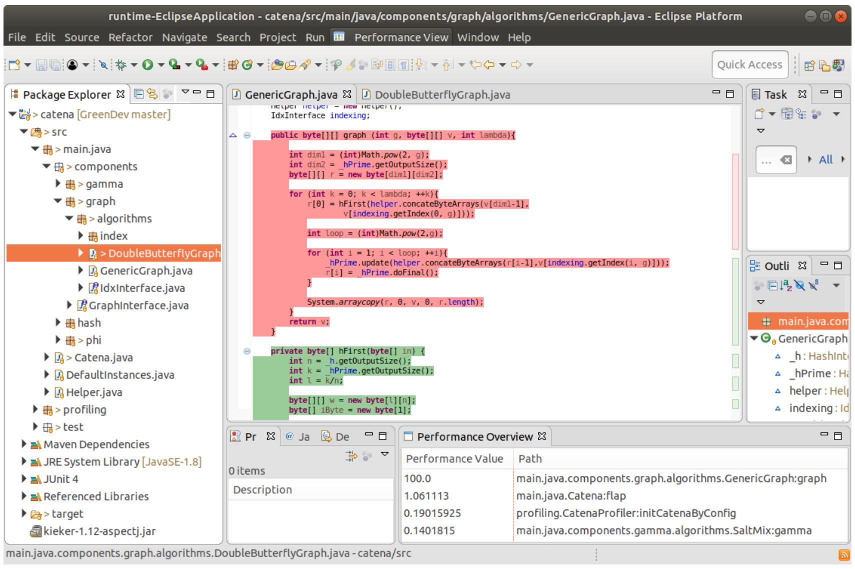
Performance Value	Path
71.01022	main.java.components.graph.algorithms.DoubleButterflyGraph:g
59.735043	main.java.components.graph.algorithms.GenericGraph:graph
49.505474	profiling.CatenaProfiler:initCatenaByConfig
11.762771	profiling.CatenaProfiler:main

Figure 4.16: Eclipse Marker Plugin – Mean Over all Configurations.

The plugin integrates itself into the IDE if the Java perspective is active. It provides a menu for seleting the different modes and an overview area of all colored methodes in the bottom right area of the IDE. In the overview, the methods are sorted in descending order by the performance value. Each line consists of the performance value and the fully qualified name of the method. By double-clicking on a line the corresponding class opens in the main window and scrolls to the clicked method.

Together with a developer of Catena we investigated its performance hot-spots. He confirmed that the graphs as well as the flap-method are known as the parts of Catena which are performance critical.

There are some things still to develop to use this plugin efficiently. Currently it only reads preprocessed data from a database. In the future it would be helpful to automate the process of identifying performance hot-spots with our approach and saving the results such that the plugin can use the information directly.



```
    helper helper = new helper();
    IdxInterface indexing;

    public byte[][] graph (int g, byte[][] v, int lambda){

        int dim1 = (int)Math.pow(2, g);
        int dim2 = _hPrime.getOutputSize();
        byte[][] r = new byte[dim1][dim2];

        for (int k = 0; k < lambda; ++k){
            r[0] = hFirst(helper.concateByteArrays(v[dim1-1],
                    v[indexing.getIndex(0, g)]));

            int loop = (int)Math.pow(2,g);

            for (int i = 1; i < loop; ++i){
                _hPrime.update(helper.concateByteArrays(r[i-1],v[indexing.getIndex(i, g)]));
                r[i] = _hPrime.doFinal();
            }

            System.arraycopy(r, 0, v, 0, r.length);
        }
        return v;
    }

    private byte[] hFirst(byte[] in) {
        int n = _h.getOutputSize();
        int k = _hPrime.getOutputSize();
        int l = k/n;

        byte[][] w = new byte[l][n];
        byte[] iByte = new byte[1];
```

Performance Overview

Performance Value	Path
100.0	main.java.components.graph.algorithms.GenericGraph:graph
1.061113	main.java.Catena:flap
0.19015925	profiling.CatenaProfiler:initCatenaByConfig
0.1401815	main.java.components.gamma.algorithms.SaltMix:gamma

Figure 4.17: Eclipse Marker Plugin – Maximum value Over all Configurations.

4.5.2 Flame Graphs

Flame Graphs are the second analysis tool we used to present performance of configurable software systems. We used the D3.js plugin[8] developed by a Netflix performance engineer.

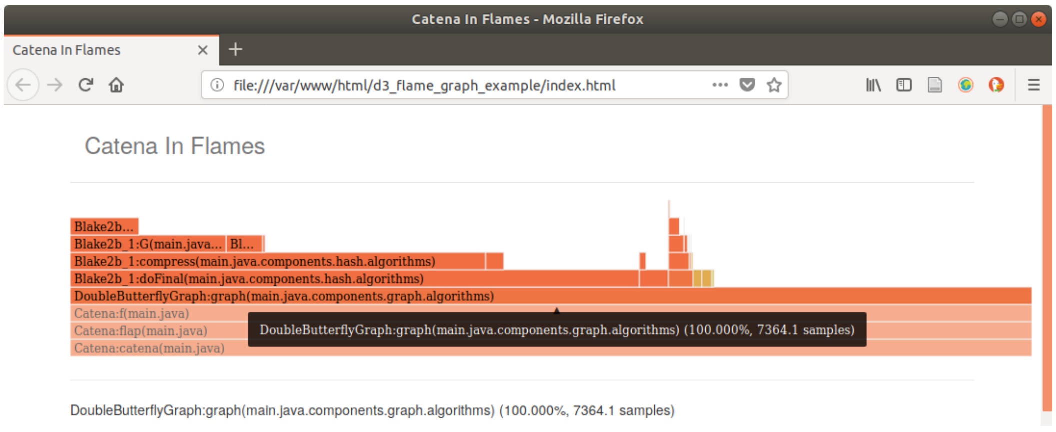

Catena In Flames - Mozilla Firefox
Catena In Flames
file:///var/www/html/d3_flame_graph_example/index.html
Catena In Flames
Blake2b...
Blake2b_1:G(main.java...
Bl...
Blake2b_1:compress(main.java.components.hash.algorithms)
Blake2b_1:doFinal(main.java.components.hash.algorithms)
DoubleButterflyGraph:graph(main.java.components.graph.algorithms)
Catena:f(main.java)
Catena:flap(main.java)
Catena:catena(main.java)
DoubleButterflyGraph:graph(main.java.components.graph.algorithms) (100.000%, 7364.1 samples)
DoubleButterflyGraph:graph(main.java.components.graph.algorithms) (100.000%, 7364.1 samples)

Figure 4.18: Flame Graph – Catena Performance Visualisation.

In Figure 4.18 we show the performance distribution of the methods of one configuration. This example shows the performance of each method of its mean execution time. In this example, we clicked on the DoubleButterflyGraph to

[8]Available at GitHub: `https://github.com/spiermar/d3-flame-graph`.

zoom in and present the methods performance with the whole width of the visualisation. Thus, we can see that with this specific configuration the most execution time of Catena is consumed by the DoubleButterflyGraph, but also a lot of performance by the `compression()` and the `doFinal()` method of Blake2b_1. With this visualisation, we have the possibility to investigate all methods in relation the their performance of other methods.

4.6 Threads to Validity

4.6.1 Internal Validity

For identifying variants of our subject systems, we used only random sampling in each dimension. There are other sampling strategies that are specialized to sample binary or numerical features. It might be that with such strategies, we were able to detect other pattern in the performance data. There are methods in our set of measurements that are biased by more than 20% by external influences. These methods might introduce a learning bias into our model. In the future, we should analyze the reason for that measurement bias and probably exclude them from our further analysis. The JIP measures the execution time of methods with millisecond accuracy. There are profilers like Kieker and SPASSmeter that measure within microseconds. This would refine our measurement accuracy by factor 1000. Another issue might be that we only consider pairwise interactions. Siegmund et al. [2012a] proves that there might be a number of interactions among three features that could improve performance. It might be that this finding also applies on method level. To model different use cases of software, we varied the workloads of our subject systems. But analyzing a databases workload in a real scenario, requires more different workloads and other test scenarios like stress tests or in a client server scenario, other non-functional properties.

4.6.2 External Validity

To achieve a reasonable external validity we used configurable software systems of different application areas (hashing, raytracing, and database) which uses different configuration mechanisms. But to generalize our findings, software systems from other application areas (like flow simulation or even server software) should be analyzed as well. Furthermore, we only used Java-based software systems. It may be that or results are not generalizable to projects that are written in other languages.

Chapter 5

Conclusion

5.1 Concluding Remarks

With this thesis, we present an approach for automating the currently static performance analysis process of configurable software systems. This work provides reasonable guidelines and issues to each stage of the performance analysis process. We surveyed different monitoring tools and sampled the configuration space of our subject systems. We utilized existing techniques to build performance influence models for methods and reason about the used regression techniques and analyzed sources of bias in of our approach.

We found out that we can reduce performance monitoring time by excluding at least 30% of the methods. If we concentrate only on those methods with the highest configuration sensitivity (that we could not learn with our models), we can even reduce the time to monitor performance by 94%. Profiling needs at least triples the time so we can achieve at least triple measuring speed, if we do the data flow analysis apart from the profiling process. Therefore, we could use Kieker or SPASS–meter to speed up. To integrate the results of our approach to the SPE process we make use of novel visualisations that present detected hot-spots integrated into existing development tooling. Finally, we can say that we understand the behaviour of performance hot-spots of configurable software systems better than before.

5.2 Outlook and Future Work

Most effort should be spent in the analysis of methods that are strongly sensitive to configurations, because these methods characterize a huge fraction of the overall performance of software systems. Therefore, researchers could adapt the sampling strategy. To determine which features influence which method, we could use feature wise sampling of configuration options. Another issue of in our approach might be that we only analyzed individual features and pair-wise interactions. Through the discussion with the Catena developer, we know that Garlic, Lambda, and the chosen Hash function are the main reason for the performance of the graphs. So, in an reproduction of this study, at least interactions among three features should be considered. We analyzed the overall measurement bias to the performance measurements. But until now, it remain unclear which proportion the individual sources of non-determinism have. We suppose the GC to influence the performance substantial. For clarification, the activity of the GC could be logged during the measurements. We choose to measure the performance of 1000 configurations for each of the three subject systems. We do not differentiate between the size of the projects and the number of methods to be analyzed. With Catena, we decide between four different graphs, which results in 250 measurements per graph. In further experiments, the focus should be on a proper sample size, which provides enough data to be able to learn the influence models. Finally, to be able to generalize the results, other configurable software systems, developed in Java or other languages, should be analyzed as well.

Bibliography

Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander Von Rhein, and Dirk Beyer. Detection of feature interactions using feature-aware verification. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 372–375. IEEE Computer Society, 2011.

Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-oriented software product lines.* Springer, 2016.

Jean-Philippe Aumasson. Password Hashing Competition. `https://password-hashing.net/`, 2015. accessed on 2018-05-12.

Lawrence Chung, Brian A Nixon, and Eric Yu. Using non-functional requirements to systematically support change. In *Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on*, pages 132–139. IEEE, 1995.

Mike Cohn. *User stories applied: For agile software development.* Addison-Wesley Professional, 2004.

Krzysztof Czarnecki, Ulrich W Eisenecker, and Krysztof Czarnecki. *Generative programming: methods, tools, and applications*, volume 16. Addison Wesley Reading, 2000.

Alan M Davis. *Software requirements: objects, functions, and states.* Prentice-Hall, Inc., 1993.

David L Donoho et al. High-dimensional data analysis: The curses and blessings of dimensionality. 2000.

Holger Eichelberger and Klaus Schmid. Erhebung von produkt-laufzeit-metriken: Ein vergleich mit dem spass-meter-werkzeug. In *Proceedings of the DASMA Metrik Kongress (MetriKonâĂŽ12)*, pages 171–180. Shaker Verlag, 2012. In German.

Christian Forler, Stefan Lucks, and Jakob Wenzel. Catena. https://www.uni-weimar.de/de/medien/professuren/medieninformatik/mediensicherheit/research/catena/, 2014. accessed on 2018-05-12.

Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, 2007.

Martin Glinz. On non-functional requirements. In *Requirements Engineering Conference, 2007. RE'07. 15th IEEE International*, pages 21–26. IEEE, 2007.

Brendan Gregg. The flame graph. *Queue*, 14(2):10:91–10:110, 2016. ISSN 1542-7730. doi: 10.1145/2927299.2927301. URL http://doi.acm.org/10.1145/2927299.2927301.

Jianmei Guo, Krzysztof Czarnecki, Sven Apely, Norbert Siegmundy, and Andrzej Wasowski. Variability-aware performance prediction: A statistical learning approach. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 301–311. IEEE Press, 2013.

G.J. Hahn and S.S. Shapiro. *Statistical Models in Engineering.* Wiley Series on Systems Engineering and Analysis Series. John Wiley & Sons, 1967. ISBN 9780471339151. URL https://books.google.de/books?id=4WzbAAAAMAAJ.

Xue Han and Tingting Yu. An empirical study on performance bugs for highly configurable software systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '16, pages 23:1–23:10, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4427-2. doi: 10.1145/2961111.2962602. URL http://doi.acm.org/10.1145/2961111.2962602.

MA Jackson. Problem frames: Analysing and structuring software development problems pdf. 2001.

Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

Gerald Kotonya and Ian Sommerville. *Requirements engineering: processes and techniques.* Wiley Publishing, 1998.

Charles W Krueger. New methods in software product line development. In *Software Product Line Conference, 2006 10th International*, pages 95–99. IEEE, 2006.

Philipp Leitner and Cor-Paul Bezemer. An exploratory study of the state of practice of performance testing in java-based open source projects. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 373–384. ACM, 2017.

P. Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum, Akad. Verlag, 2002. ISBN 9783827411181. URL `https://books.google.de/books?id=ujNVAAAACAAJ`.

Stefan Lucks and Jakob Wenzel. Catena variants. In Frank Stajano, Stig F. Mjølsnes, Graeme Jenkinson, and Per Thorsheim, editors, *Technology and Practice of Passwords*, pages 95–119. Springer International Publishing, 2016. ISBN 978-3-319-29938-9.

Maythux. Disabling Intel Turbo Boost in ubuntu. `https://askubuntu.com/questions/619875/disabling-intel-turbo-boost-in-ubuntu`, 2015. accessed on 2018-05-12.

Marcilio Mendonca, Moises Branco, and Donald Cowan. S.p.l.o.t.: Software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 761–762, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-768-4. doi: 10.1145/1639950.1640002. URL `http://doi.acm.org/10.1145/1639950.1640002`.

Ian Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance.* O'Reilly Media, Inc., 1st edition, 2009. ISBN 0596520662, 9780596520663.

mozilla.org contributors. The new, fast browser for Mac, PC and Linux | Firefox. `https://www.mozilla.org/en-US/firefox/`, 2018. accessed on 2018-04-28.

Linda M. Northrop. Introduction to software product lines. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond*, pages 521–522, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-15579-6.

Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 35–46. ACM, 2000.

Oracle Corporation. MySQL. `https://www.mysql.com/de/`, 2018. accessed on 2018-04-28.

S Robertson and J Robertson. Mastering the requirements process. 1º edição, 1999.

Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, and Gunter Saake. Multi-dimensional variability modeling. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, pages 11–20. ACM, 2011.

Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. Cost-efficient sampling for performance prediction of configurable systems (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 342–352. IEEE, 2015.

Norbert Siegmund, Sergiy S Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 167–177. IEEE, 2012a.

Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. Spl conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal*, 20(3-4):487–517, 2012b.

Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 284–294. ACM, 2015.

Connie U Smith. Software performance engineering. In *Performance Evaluation of Computer and Communication Systems*, pages 509–536. Springer, 1993.

Stanley Smith Stevens et al. On the theory of scales of measurement. 1946.

The Apache Software Foundation. The Apache HTTP Server Project. `https://httpd.apache.org/`, 2018. accessed on 2018-04-28.

Thomas Thüm, Ina Schaefer, Sven Apel, and Martin Hentschel. Family-based deductive verification of software product lines. In *ACM SIGPLAN Notices*, volume 48, pages 11–20. ACM, 2012.

Dimitris Tsirogiannis, Stavros Harizopoulos, and Mehul A Shah. Analyzing the energy efficiency of a database server. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 231–242. ACM, 2010.

AndrÃľ van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*, pages 247–248. ACM, 2012.

Al Weber and R Thomas. Key performance indicators. *Measuring and Managing the Maintenance Function, Ivara Corporation, Burlington*, 2005.

Murray Woodside, Greg Franks, and Dorina C Petriu. The future of software performance engineering. In *Future of Software Engineering, 2007. FOSE'07*, pages 171–187. IEEE, 2007.

Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 307–319. ACM, 2015.

Acronyms

JVM	Java Virtual Machine
KPI	Key Performance Indicators
IDE	Integrated Development Environment
SPL	Software Product Line
NFR	non-functional requirements
NFP	non-functional properties
SPE	Software Performance Engineering
CART	Classification-And-Regression-Tree
OSS	Open-source Software
OS	Operating System
JIT	Just-In-Time
GC	Garbage Collection
JIP	Java Interactive Profiler
JRE	Java Runtime Environment
GUI	Graphical User Interface
AOP	aspect-oriented programming
RSS	residual sum of squares
RAM	random access memory
CV	coefficient of variation
MSE	Mean Squared Error

Appendix A

Appendix

A.1 Feature Diagrams

A.1.1 Running Example

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
    <featureModel chosenLayoutAlgorithm="1">
        <struct>
            <and abstract="true" mandatory="true" name="Database_Engine">
                <feature mandatory="true" name="Base"/>
                <feature name="Indexing"/>
                <or name="Encryption">
                    <feature mandatory="true" name="AES"/>
                    <feature mandatory="true" name="RSA"/>
                </or>
                <and abstract="true" name="Compression">
                    <feature name="gzip"/>
                    <feature name="zip"/>
                </and>
                <alt abstract="true" mandatory="true" name="OS">
                    <feature mandatory="true" name="Unix"/>
                    <feature mandatory="true" name="Windows"/>
                </alt>
            </and>
        </struct>
        <constraints>
            <rule>
                <imp>
                    <var>zip</var>
                    <var>Windows</var>
                </imp>
            </rule>
        </constraints>
        <calculations Auto="true" Constraints="true" Features="true"
Redundant="true" Tautology="true"/>
        <comments/>
        <featureOrder userDefined="false"/>
    </featureModel>
```

Listing A.1: Textual representation of example database model.

A.2 Results

A.2.1 Configuration Sensitivity

Catenas 23 methods which are not sensitive to configurations:

- main.java.Catena:<init>
- main.java.Catena:erasePwd
- main.java.Catena:gamma
- main.java.Catena:init
- main.java.Catena:setD
- main.java.components.graph.algorithms.GenericGraph:<init>
- main.java.components.phi.algorithms.CatenaPhi:<init>
- main.java.components.phi.algorithms.CatenaPhi:setH
- main.java.components.phi.algorithms.CatenaPhi:setHPrime
- main.java.components.phi.algorithms.IdentityPhi:<init>
- main.java.components.gamma.algorithms.IdentityGamma:<init>
- main.java.components.graph.algorithms.index.IndexDBG:<init>
- main.java.components.hash.algorithms.Blake2b_1:<init>
- main.java.components.gamma.algorithms.IdentityGamma:setH
- main.java.components.gamma.algorithms.SaltMix:<init>
- main.java.components.gamma.algorithms.SaltMix:bytes2long
- main.java.components.gamma.algorithms.SaltMix:transformBytesToLong
- main.java.components.graph.algorithms.DoubleButterflyGraph:<init>
- main.java.components.graph.algorithms.index.IndexGRG:<init>
- main.java.components.hash.algorithms.Blake2b:<init>
- main.java.components.phi.algorithms.index.LSBIndex:<init>
- main.java.Helper:intToByteArrayLittleEndian
- main.java.Helper:string2Bytes

A.2.2 Worst Methods Catena

```
public byte[][] graph(int g, byte[][] v, int lambda) {

    int dim1 = (int)Math.pow(2, g);;
    int dim2 = _hPrime.getOutputSize();
    byte [][] r = new byte[dim1][dim2];

    int jLimit = 2*g - 1;
    int iLimit = dim1-1;

    _hPrime.reset();

    for (int k = 1; k <= lambda; k++) {
        for (int j = 1; j <= jLimit; j++) {

            r[0] = hFirst(helper.concateByteArrays(
                helper.xor(v[dim1-1],v[0]),
                v[(int)indexing.getIndex((long)g, (long)j-1, 0)]));

            for (int i = 1; i <= iLimit; i++) {
                _hPrime.update(helper.concateByteArrays(
                    helper.xor(r[i-1],v[i]),
                    v[(int)indexing.getIndex((long)g, (long)j-1, i)]));
                r[i] = _hPrime.doFinal();
            }
            System.arraycopy(r, 0, v, 0, r.length);
        }
    }
    return v;
}
```

Listing A.2: Catena – Double Butterfly Graph.

```
public byte[][] phi(int garlic, byte[][] b, byte[] m) {
    int j = _idx.getLsbIndex(m, garlic);
    byte[][] tmp = new byte[b.length][_h.getOutputSize()];

    System.arraycopy(b, 0, tmp, 0, b.length);

    _hPrime.update(helper.concateByteArrays(b[b.length-1], b[j]));
    tmp[0] = _hPrime.doFinal();
    _hPrime.reset();

    for (int i = 1; i < b.length; ++i){
        j = _idx.getLsbIndex(tmp[i-1], garlic);
        _hPrime.update(helper.concateByteArrays(tmp[i-1], tmp[j]));
        tmp[i] = _hPrime.doFinal();
        _hPrime.reset();
    }
    return tmp;
}
```

Listing A.3: Catena – Phi.

```
    private byte[] flap(int g, byte[] xIn, byte[] gamma){
        _hPrime.reset();

        byte[] xHinit;
        int iterations = (int)Math.pow(2, g);
        byte[][] v = new byte[iterations+2][_k];
        xHinit = hInit(xIn);

        System.arraycopy(xHinit, 0, v[0], 0, _k);
        System.arraycopy(xHinit, _k, v[1], 0, _k);

        for (int i=2; i<iterations+2; ++i){
            _hPrime.update(helper.concateByteArrays(v[i-1], v[i-2]));
            v[i] = _hPrime.doFinal();
        }

        byte[][] v2 = new byte[iterations][_k];
        System.arraycopy( v, 2, v2, 0, v2.length );

        _hPrime.reset();
        v2 = gamma(g, v2, gamma);
        _hPrime.reset();
        v2 = f(g, v2, _lambda);
        _hPrime.reset();
        v2 = phi(g, v2, v2[v2.length-1]);
        return v2[v2.length-1];
    }
```

Listing A.4: Catena – Flap.

```
public byte[][] gamma (int g, byte[][] x, byte[] gamma){

    byte[] gammaByte = gamma;
    byte[] tmp1;
    byte[] tmp2;

    _h.update(gammaByte);
    tmp1 = _h.doFinal();
    _h.reset();
    _h.update(tmp1);
    tmp2 = _h.doFinal();
    _h.reset();

    transformBytesToLong(tmp1, tmp2);

    p=0;
    long j1 = 0;
    long j2 = 0;
    int loopLimit = (int)Math.pow(2, Math.ceil(3.0*g/4.0));

    _hPrime.reset();

    for (int i = 0; i < loopLimit; ++i){
        j1 = xorshift1024star() >>> (64 - g);
        j2 = xorshift1024star() >>> (64 - g);

        _hPrime.update(helper.concateByteArrays(x[(int)j1], x[(int)j2]));
        x[(int)j1] = _hPrime.doFinal();

    }
    return x;
}
```

Listing A.5: Catena – Salt Mix.

A.3 Scripts

```
#!/bin/bash

# ./turbo-boost.sh disable
# ./turbo-boost.sh enable
# if error: "rdmsr:open: No such file or directory"
# -> sudo modprobe msr


if [[ -z $(which rdmsr) ]]; then
    echo "msr-tools not installed. Run 'sudo apt install msr-tools'." >&2
    exit 1
fi

if [[ ! -z $1 && $1 != "enable" && $1 != "disable" ]]; then
    echo "Invalid argument: $1" >&2
    echo ""
    echo "Usage: $(basename $0) [disable|enable]"
    exit 1
fi

cores=$(cat /proc/cpuinfo | grep processor | awk '{print $3}')
for core in $cores; do
    if [[ $1 == "disable" ]]; then
        sudo wrmsr -p${core} 0x1a0 0x4000850089
    fi
    if [[ $1 == "enable" ]]; then
        sudo wrmsr -p${core} 0x1a0 0x850089
    fi
    state=$(sudo rdmsr -p${core} 0x1a0 -f 38:38)
    if [[ $state -eq 1 ]]; then
        echo "core ${core}: disabled"
    else
        echo "core ${core}: enabled"
    fi
done
```

Listing A.6: Script for enabling/disabling Turbo Boost in Ubuntu (Maythux [2015]).

A.4 Performance Log File

```
+----------------------------------------------------------------------
|  File:
    ./0_0_1_0_1_23_250_178_Butterfly-Full-adapted_6789ab/20180315-003731.txt
|  Date: 2018.03.15 00:37:31 AM
+----------------------------------------------------------------------

            Time        Percent
         ----------   ----------
 Count    Total  Net   Total  Net    Location
 =====    =====  ===   =====  ===    =========
     1     62.1  0.1   100.0         +--Catena:catena (main.java)
     1      9.6  0.0    15.4         | +--Catena:compTweak     (main.java)
     2      0.0  0.0     0.0         | | +--Helper:intToByteArray (main.java)
...

+---------------------------------------
| Most expensive methods (by net time)
| Frame Count Limit: Unlimited
+---------------------------------------

               Net
          ------------
 Count      Time    Pct  Location
 =====      ====    ===  ========
  4416      35.7   55.8  main.java.components.hash.algorithms.Blake2b:G
     2       3.9    6.1  main.java.components.hash.algorithms.Blake2b:init
   384       2.7    4.2  main.java.components.hash.algorithms.Blake2b:rotr64
...
```

Listing A.7: Example Log File of Catena Password Hashing Framework.