Bachelor

June 27, 2017

Distributed Execution of Performance Tests on Cloud Instances

Selin Fabel

of Buchrain, Switzerland (13-121-140)

supervised by

Prof. Dr. Harald C. Gall

Christoph Laaber

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Bachelor

Distributed Execution of Performance Tests on Cloud Instances

Selin Fabel

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Bachelor

Author: Selin Fabel, selin.fabel@uzh.ch

Project period: 20.02.2017 - 20.08.2017

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

Before diving into the fields of performance testing and cloud computing, I would like to shortly recap the process of establishing this bachelor thesis. It was an intensive period – very instructive, although stressful at times. This thesis is the result of over 650 working hours, but it marks the end of my undergraduate studies and I consider it also as a summary of them. Concepts once heard in a lecture re-appeared, and I transformed this theoretical knowledge into practical experience.
However, there would be no thesis without a lots of helping hands. First of all, I would like to thank my supervisor Christoph Laaber. During the establishment of this thesis, he gave me valuable feedback, provided frequent support and advice. Together with Dr. Philipp Leitner, he also developed the tool this thesis is based on. A thank-you to Dr. Leitner at this point for initial organization and brief comments. Furthermore, I would like to thank Professor Dr. Harald Gall for offering the opportunity to write this bachelor thesis at his institution. And lastly, an expression of gratitude to my family and especially to my boyfriend for their patience, support and comfort if I was literally having my head in the cloud all day.

Abstract

Software performance testing is a very important task in the development cycle of applications and services. Regression between versions such as increased response time and lowered throughput can lead to an inappropriate usage of resources, unsatisfied users, and eventually also a loss of money. To make matters worse, performance testing is a tedious process; the test suites take long to execute, but must be repeated several times to obtain expressive results. Additionally, software is changing at a pace which makes it almost impossible to thoroughly test the performance of the whole application before every release.

This thesis investigates the impacts of parallel execution of performance tests in cloud environments. Initially, it examines how performance tests suites can be split, distributed and executed on several remote instances. For this purpose, the thesis introduces a tool called *clopper* which stands for *cloud-extended hopper* and is based on a framework for performance history mining of software projects. Clopper implements four different distribution algorithms which either split the test suite on version- or test-level. In a further step, clopper is used to extract performance metrics from three different projects. By means of these measurements, the distribution methods are compared in terms of time, cost and quality.

The results reveal that parallel is always faster than non-parallel execution but at the same time, that this does not imply savings of money. Depending on the use-case, one method is more suitable than another. If the aim is to quickly obtain measurements which possibly contain inaccuracies, one should distribute groups of consecutive versions. On the other hand, if the results should be as stable as possible and time is not an urgent matter, the method which randomly distributes version-test-tuples should be chosen. Distribution by versions obtains in parallel execution with six cloud instances a gain in time of factor 4.78. The completely randomized approach is 5.30 times faster when using six instances instead of one.

Zusammenfassung

Das Testen von Performanz ist eine wichtige Aufgabe in der Software-Entwicklung von Applikationen und Diensten. Leistungseinbussen zwischen Versionen, wie zum Beispiel erhöhte Antwortzeit oder verminderter Datendurchsatz können die Nutzerzufriedenheit beeinträchtigen und zu unnötigem Ressourcenverbrauch sowie Kapitalverlust führen. Zu allem Überfluss sind solche Tests mühselig durchzuführen und mit viel zeitlichem Aufwand verbunden. Nebst der langen Ausführungszeit, sind viele Wiederholungen der Tests notwendig um aussagekräftige Resultate zu erhalten. Ausserdem verändert sich Software so schnell, dass es nahezu unmöglich ist vor jedem Release die Performanz einer ganzen Applikation zu testen.
Aus diesen Gründen untersucht die vorliegende Bachelorarbeit, wie sich Software-Leistungstests parallelisieren und in Cloud-Umgebungen ausführen lassen und welche Auswirkungen dies hat. In einem ersten Schritt wird erforscht, wie Test-Sammlungen aufgeteilt und zur Ausführung einer Gruppe von virtuellen Cloud Instanzen zugeschrieben werden können. Für diese Aufgabe, präsentiert diese Arbeit ein Skript namens *clopper*, das auf einem Programm zur Erstellung eines Performanz-Verlaufs von Software Projekten basiert. Clopper steht für *cloud-extended hopper*, ist also eine Erweiterung des *hopper*-Programms für Cloud Computing. Clopper stellt vier verschiedene Distributions-Algorithmen zur Verfügung. Die Testsammlung wird dabei entweder nach Versionen oder Tests aufgeteilt. In einem weiteren Schritt wird clopper dann zur Performanzmessung dreier Software-Projekte eingesetzt. Mithilfe der gesammelten Messdaten werden die Distributions-Algorithmen auf Zeit-, Kosten- und Qualitätsunterschiede untersucht.
Schlussendlich zeigt sich, dass parallele Ausführungen immer schneller als sequentielle Ausführungen sind. Aus den Ergebnissen lässt sich aber auch schliessen, dass eine schnellere Ausführung nicht unbedingt mit geringeren Kosten verbunden ist. Je nach Anwendungsfall ist deshalb eine andere Methode zu empfehlen. Falls möglichst schnell Messwerte vorliegen sollten, die aber auch Inkonsistenzen enthalten dürfen, sollte jene Methode gewählt werden, die Grüppchen aufeinanderfolgender Versionen verteilt. Wenn die Werte jedoch so stabil wie möglich sein sollten und dabei die Ausführungsdauer nur eine geringe Rolle spielt, sollte man zu jener Methode tendieren, die Tupel von Versionen und Tests generiert und diese zufällig an die Instanzen verteilt. Die erste Methode erzielt in einer parallelen Durchführung auf sechs virtuellen Instanzen einen Zeitgewinn von Faktor 4.78. Mit dem komplett zufälligen Ansatz sind sechs Instanzen rund 5.30-mal schneller als eine einzelne.

Contents

1 Introduction **1**

2 Background **3**
2.1 Definition . . . 3
2.2 Related Work . . . 3
2.2.1 Concepts and Tools . . . 4
2.2.2 Optimization Approaches . . . 5
2.2.3 Distributed and parallelized testing . . . 7
2.3 Hopper . . . 8

3 Description **11**
3.1 Architecture . . . 11
3.2 Workflow and State Concept . . . 12
3.3 Implementation . . . 15
3.3.1 Communication . . . 15
3.3.2 Preparation and Execution . . . 15
3.3.3 Test Suite Distribution . . . 17
3.3.4 Output and Storage . . . 20

4 Evaluation **23**
4.1 Experimental setup . . . 23
4.1.1 Cloud Provider . . . 23
4.1.2 Approach . . . 23
4.2 Overall execution time . . . 26
4.2.1 Cost Differences . . . 28
4.3 Quality . . . 29
4.3.1 Range of Benchmarks . . . 30
4.3.2 Similarity of Distribution Methods . . . 32
4.3.3 Quality of Methods . . . 34
4.4 Findings . . . 37
4.4.1 Recommendation . . . 38
4.4.2 Threats to Validity . . . 38
4.4.3 Future Work . . . 39

5 Conclusion **41**

List of Figures

2.1 Sample Workflow of *hopper* [LL17] 8

3.1 Architecture of the *clopper*-tool 12
3.2 State concept implemented by the remote instances which are controlled by *clopper* 13
3.3 Workflow of *clopper* 14
3.4 *Random Version Distribution* where A to H denote single versions of a project 18
3.5 *Version Range Distribution* where A to H denote single versions of a project 18
3.6 *Test Distribution* where I to VIII denote benchmarks of a project 19
3.7 *Randomized Multiple Interleaved Trials* where letters denote versions, and Roman numbers benchmarks of a project 19

4.1 Version Range Distribution in project RDF4J with P99 threshold over all benchmark measurements. The y-axis for *maxSpread*-display is cut at 0.7. 31
4.2 Comparison of projects in method Test Distribution. y-axis is cut at 1.65. 31
4.3 RDF4J RMIT Distribution 33
4.4 RDF4J Version Distribution 33
4.5 Comparison of Distribution-methods over all projects. The *maxSpread*-display is limited to 1.6. 33
4.6 JCTOOLS Random Version Distribution with p-value of .036. Values are displayed until *maxSpread* of 1.2. 35
4.7 RDF4J Random Version Distribution with p-value <.001 which indicates instability over versions. *maxSpread*-values are displayed between 0.0 and 0.7. 36
4.8 JCTOOLS RMIT Distribution with p-value .936. With a p-value above .05, this method produces output which is robust over versions. The y-axis is cut at 1.2. . . 36

List of Tables

4.1 Performance test suites used for the experiments 25
4.2 Comparison of the overall execution time between parallel execution (6 instances) and singular (1 instance). Displayed are the mean and standard deviation of execution time. All values are shown in minutes except for the column *Factor* which describes how many times parallel - is faster than non-parallel execution. 26
4.3 Difference between fastest and slowest instance in parallel execution mode. Displayed are the mean and standard deviation of execution time. All values are shown in minutes. 27
4.4 Differences in total costs between 1 instance, and 6 instances executing the same workload in a parallel manner. The price difference is given in percentage. Positive percentages imply higher costs for the parallel approach. 29
4.5 Display of different percentiles of *maxSpread*. A lower maxSpread indicates less variability of benchmark measurements. 30
4.6 *p-value* obtained by the Wilcoxon signed-rank test in which the *maxSpreads* of a benchmark in one method was compared with the maxSpread in another method. RV stands for Random Version Distribution, Version refers to Version Ranges. A p-value <.005 implies a rejection of the null-hypothesis and indicates that the two examined methods output different measurements. 32
4.7 The p-values obtained by the ANOVA-tests. Values below .05 imply a rejection of the null-hypothesis and indicate instability over versions. 35

List of Listings

2.1 JMH-benchmark sample from project *RDF4J* 5
2.2 Sample JMH output 6
3.1 Sample JSON-configuration file for execution of the *clopper*-script 16
3.2 Sample log file produced by the *clopper*-script 21
4.1 ANOVA sample for version *2cfb106* in project RDF4J 34

Chapter 1

Introduction

Performance is an important quality attribute of software applications and services. We experience performance for example in terms of response time when launching a web page: 0.1 seconds are stated as *immediate responding*, 1 second is considered as *seamless*, but everything above is sensed as a *delay* and the likelihood of aborting the task is increased [Nie10]. If a task has been executed in the past and its response-time is thus known, we expect the process to behave accordingly in all future repetitions. In case, a future execution shows worse behaviour this is considered as *Performance Regression*.
As demonstrated by Google, performance regression can lead to significant losses. Researchers simulated a regression by increasing latency in display of search result from 0.4 to 0.9 seconds. As a consequence, they encountered a traffic loss of over 20% [May09]. Performance regression therefore results in dissatisfaction among users and can lead to financial deficiencies [CS17].
For these reasons, it is important for software engineers to focus on measuring performance when developing applications. Unfortunately, performance testing is not yet a well-established practice since functional correctness is considered more important by developers [LB17]. Chen and Shang even state that regression is often a consequence of functional bug fixing [CS17]. As a result, many regression provoking root causes might not be detected [FJA+10].
To circumvent such problems, developers could use different approaches for automating performance testing and regression detection. For example, there exist methods which continuously monitor applications at runtime and obtain measurements for analysis [KWZK16]. Other approaches isolate different versions of applications and compare their outcomes [ABV16], [CS17], [HMSZ14]. However, in-depth performance-testing of an application is time-consuming. Execution of performance tests can take several hours, for large test suites even days [FJA+10]. Moreover, performance tests need to be repeated until the results become stable, and sources of non-determinism such as *garbage collectors* or *just-in-time-compilers* can additionally affect execution [HMSZ14], [ABV16]. To make matters worse, software is changing at a pace which makes it impossible to thoroughly test all changes and the system as a whole until the next release [HMSZ14]. As a consequence, despite the plethora of tools and concepts, there exists no state-of-the-art in performance testing [LB17].

It is a vicious circle: performance testing seems to be expensive and time-consuming, but if on the other hand, regression occurs, the costs for finding and fixing its cause quickly explode [HHF13]. This bachelor thesis aims to satisfy the need for efficient gathering of performance metrics and evaluates how a distributed approach with parallel measurement gathering suits the task.
In detail, the following research questions are investigated:

RQ 1: *In what ways is it possible to distribute a performance test suite for parallel execution?*

RQ 2: *How much time and costs can be saved by executing performance test suites in parallel?*

RQ 3: *Are the results obtained by different distribution techniques similar in terms of quality?*

To answer the stated questions, this thesis first establishes a literature review. It differentiates between existing optimization techniques for performance measuring and parallelized testing approaches. The findings are used to design a tool which is based upon a distributed architecture. The tool is called *clopper*. Clopper stands for *cloud-extended hopper* and encapsulates a framework for performance mining of software projects. It is able to fully automate the process of parallel performance measurement gathering. By means of the obtained knowledge from the literature review, clopper implements different algorithms to distribute a performance test suite among several workers. In this context, the first research question is examined.
Besides test suite distribution, clopper provides functionality to monitor execution and assembly of the final output. The tool is used in a next step to conduct an experimental evaluation of the implemented methods. In order to provide answers to research question 2, this thesis sets the focus on quantifiable measurements and compares the implemented methods in parallel and non-parallel mode. For research question 3, the results are additionally examined in terms of quality. It is measured how stable results are within a method, and whether there exist significant differences between the alternatives.

The remainder of this thesis is structured as follows:
Chapter 2 sets the thesis into context and starts with a literature review in which it briefly defines the concept of performance testing. It further refers to related work, existing tools and compares them with the proposed solution. The design and implementation of the solution is discussed in chapter 3 alongside supporting diagrams and graphics. There follows the setup and results of the experimental evaluation in chapter 4. The research questions are also revisited in this chapter. Chapter 5 finally summarizes and concludes this thesis.

Chapter 2

Background

This chapter gives an overview of the different notions of performance testing and characterises the concept used in this thesis. Further presented are related concepts and tools in general, as well as optimization approaches and performance testing research in distributed environments. The chapter concludes with a description of the thesis' parent tool *hopper*.

2.1 Definition

Performance testing can be ranked as a subset of *software testing* and *performance engineering* [ZLZY13]. Whereas performance engineering approaches solely aim to improve performance, performance testing also includes analytic activities (Woodside et al. in [MTHG14]). In general, workload is put on a system and its behaviour examined. In this context, the term *load testing* often occurs which in the traditional sense, focuses on the performance of applications as a whole [MFB+07]. Besides conducting performance tests on application level, it is also possible to apply the principle of *microbenchmarking* on a lower level. In microbenchmarking, small but critical pieces of code (e.g. functions) are isolated and performance measures taken [RCCB16], [GLS11]. Typically, measurements describing software quality attributes are obtained. These are for example stability, speed, response time, memory or CPU utilization [KWZK16], [CH14].
The obtained measurements can subsequently be used for determination of performance regression. To do so, the results from different versions must be gathered and compared. If regression occurs, the newer version shows in comparison with the prior one worse behaviour, e.g. uses more resources or has less throughput [CH14], [Luo16b], [MFB+07], [FJA+10], [NAJ+12]. In literature, regression is also observed in the scope of *functional testing* where it defines a change which leads to a defect [BMZP14], [CH14].
As it is a fact, performance tests have a long execution time. According to Foo et al.,a performance test suite can run several days [FJA+10]. As a result, no commonly established standard exists [LB17]. This thesis therefore aims to optimize the gathering of performance metrics and presents in chapter 3 a parallelized approach.

2.2 Related Work

The performance test suite used in this thesis consists of a collection of microbenchmarks. Besides displaying topics related to the approach in this thesis, the current section briefly evaluates on a framework for creation and execution of such benchmarks on code-level. This section further states optimization approaches for determination of regression. It must be noted that functional

and performance regression do not examine the same kind of regression. The used approaches are nonetheless similar and comparable with the approach used by this thesis.

2.2.1 Concepts and Tools

One of the basic concepts of measuring performance is *monitoring*. On the market, there are a large number of tools available which gather measurements such as response time, memory, or CPU utilization during execution of a program [KWZK16]. Popular are for example *New Relic*, *Dynatrace*, and *AppDynamics* [HDS16].
Unfortunately, monitoring often results in the creation of big log files which are not just hard to analyse but can also be a storage issue. Additionally, the complexity and the development-pace of applications increases and makes thorough testing impossible [KWZK16].
A possible solution has been proposed by Kross et al. In their 2016 published paper, they present a tool named *PET* which monitors an application and continuously evaluates the produced measurements. The tool is suitable for big data since it implements a distributed database management system [KWZK16]. This idea of continuously writing measurements to a central repository is similar to the approach used by this thesis. The tool in this thesis however, uses a cloud storage bucket for this purpose .
Another common approach in performance testing is described by the *unit-test assumption*. It assumes that "the performance of relevant use cases of a program correlates with the performance of at least a part of its unit tests" [RK16]. Or in other words, it is possible to focus on system components rather than on the whole system when testing. This is exactly what *profiling* methods are based on. While in monitoring, the whole application is examined, profiling methods focus only on parts of codes. Profiling consists of *instrumentation* and *sampling*. Sampling tools such as the *HPCToolkit* statistically determine samples of code and measure its performance [DNM15], [ABV16]. In instrumentation on the other hand, marks are placed in the source code which trigger a measurement during program execution and produce a trace [Luo16b]. In assumption that some important piece of code might need longer to execute, Maplesden et al. also take into account the cost and benefits of certain functions [MTHG16].
The concept of microbenchmarking goes into the same; components of code are isolated and only these are measured. Since the design of such a test suite needs to take into consideration issues as just-in-time-compiler (JIT-compiler), or multi-threading, it is not a trivial task [GLS11], [RCCB16]. The frameworks *Java Microbenchmark Harness* (JMH)[1] and *Google Caliper*[2] are commonly used therefor [LB17], [RCCB16]. Since version 1.7, JMH is part of the *Java Development Toolkit* (JDK), so it might be an emerging standard [LB17].

Java Microbenchmark Harness

This thesis mines microbenchmarks established by the JMH-framework. The following gives an insight into the structure of such tests.
JMH-benchmarks are simple methods equipped with annotations [LB17]. It is mandatory to set an *@Benchmark*-flag on top to clearly mark all methods which will be executed by JMH. In addition, there are several optional annotations available. With the *@Param*-flag for example, different configurations can be assigned to a benchmark.
For some annotations, it is possible to set them in code or also by means of command-line parameters when starting the program. The difference is though that method-annotations are only valid for a particular method whereas command-line parameters are globally applied and overwrite

[1] `http://openjdk.java.net/projects/code-tools/jmh/`
[2] `https://github.com/google/caliper`

all individual annotations. A sample benchmark obtained from the project *RDF4J*[3] is shown in listing 2.1. The chosen benchmark-mode is *AverageTime* in which methods annotated with the *@Benchmark*-flag are called and the average calling time over all worker threads is counted.[4] The output is given in milliseconds as specified by the flag *@OutputTimeUnit*.

```
@Benchmark
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
public void noReasoning()
throws IOException
{
SailRepository sail = new SailRepository(new MemoryStore());
sail.initialize();

try (SailRepositoryConnection connection = sail.getConnection()) {
connection.begin();

connection.add(resourceAsStream("schema.ttl"), "", RDFFormat.TURTLE);
addAllDataSingleTransaction(connection);

connection.commit();
}
}
```

Listing 2.1: JMH-benchmark sample from project *RDF4J*

Listing 3.3.4 displays the console output obtained by the execution of the benchmark in listing 2.1. Literature recommends to first run the entire test suite in several warmup-iterations to ensure all tests are fully initialized and optimized by the JIT-compiler [GLS11], [RCCB16]. Since the execution of benchmarks is non-deterministic by nature, it is also common practice to collect the output from multiple measurement runs. Such an approach should equilibrate influences of the garbage collection [GLS11]. In order to assess independence during execution, JMH forks one or several new Java processes for each group of iterations.[5] In the example, the benchmark is run in 10 warm-up - and 10 measurement iterations using 3 forks. From the metrics obtained by the measurement iterations, the harness eventually calculates statistics. These are the minimum, average and maximum alongside the standard deviation of the results, and the 99.9% confidence interval.

2.2.2 Optimization Approaches

Approaches to optimize the execution of test suites are often applied in the different fields of testing for regression. Code analysis techniques to find bottlenecks are very common. In the context of load testing, Luo for example use genetic algorithms as well as machine learning techniques

[3] https://github.com/eclipse/rdf4j
[4] http://javadox.com/org.openjdk.jmh/jmh-core/0.9/org/openjdk/jmh/annotations/Mode.html
[5] http://java-performance.info/jmh/

```
# Run progress: 91.67% complete, ETA 00:00:28
# Fork: 3 of 3
# Warmup Iteration 1: 1.120 ms/op
# Warmup Iteration 2: 0.411 ms/op
# Warmup Iteration 3: 0.363 ms/op
(...)
Iteration 8: 0.113 ms/op
Iteration 9: 0.115 ms/op
Iteration 10: 0.107 ms/op

Result: 0.114 ±(99.9%) 0.007 ms/op [Average]
Statistics: (min, avg, max) = (0.106, 0.114, 0.162), stdev = 0.011
Confidence interval (99.9%): [0.107, 0.122]

# Run complete. Total time: 00:05:47

Benchmark (param) Mode Samples Score Score error Units
o.e.r.b.ReasoningBenchmark.noReasoning moreRdfs::12180 avgt 30 7.841 0.149 ms/op
o.e.r.b.ReasoningBenchmark.noReasoning longChain::5803 avgt 30 1.563 0.038 ms/op
o.e.r.b.ReasoningBenchmark.noReasoning medium::544 avgt 30 0.454 0.006 ms/op
o.e.r.b.ReasoningBenchmark.noReasoning simple::152 avgt 30 0.114 0.007 ms/op
```

Listing 2.2: Sample JMH output

to determine and thus eliminate regression patterns [Luo16b], [Luo16a]. Other researchers take a selection upon the testing amount and likewise reduce the execution time [ABV16], [SYGM15], [HMSZ14].
Shi et al. who examine functional regression, apply a principle called *selection of reduction*. They first create a baseline performing an analysis on a single version of software. Based on this analysis, superfluous tests are removed from the test suite. They then perform regression testing, once again remove inconspicuous tests and end up with a small collection of regression provoking tests [SYGM15]. Concerning performance regression, Huang et al. apply *performance risk analysis* to determine potentially regressive commits which are added to the test list whereas stable commits are delayed or not tested at all [HMSZ14]. These approaches do not correspond to the methods used in this thesis since it does not aim to determine regressive code and Shi et al. moreover do this in the context of functional regression testing. Additionally, execution time should not be reduced at the expense of the overall testing amount.
Closer is therefore the approach of Alcocer et al. With the sampling technique *horizontal profiling*, they only consider every k-th version to collect run-time performance metrics and reduce execution time by this particular factor k [ABV16]. Alcocer's et al. method is probably most similar to the idea of distributing versions among different hosts used in this thesis; The total number of versions to test is not reduced, but every k-th version is assigned to an instance and executed there.
Besides Huang et al, all of the above stated researcher refer to *releases* when the term *version* is used. In the context of this thesis, a version though denotes its sub-unit, i.e. a commit. The approach which will be presented at the upcoming *International Conference on Software Maintenance and Evolution 2017* (ICSME '17) goes into the same direction. In order to determine performance regression provoking patterns, Chen and Shang examined commits of several software projects. With this approach on commit-level, they identified six code-patterns causing regression [CS17].
A technique proposed by Nguyen et al. goes a step beyond the scope of pure performance test-

ing and focuses on optimizing the evaluation of the test output produced by load tests. The researchers use statistics to determine regression and visualize the outcomes in so-called *control charts* [NAJ+12]. Likewise, Heger et al. rather try to improve the schedule than overall testing time. They execute performance tests in parallel to development. As a result, measurements are always available and for example regression becomes immediately evident [HHF13].
Testing applied at an early stage of development has also been proposed by Mayer et al. But while Heger et al. test the whole application, the latter split the test suite and separately concentrate on testing algorithms, methods and libraries [MSWM12]. The idea of splitting the test suite reappears in this thesis. Kumar et al. would even claim that originally, test suites were generated for a tester who had to execute them, gather the results and deliver the final output. Nowadays, the test suite is split among multiple testers. The testers work in parallel and in the end, multiple small groups of results must be combined [KKSLM14]. Even though the cited authors have worked in the field of functional regression testing their words exactly summarize the concept applied in this thesis.

2.2.3 Distributed and parallelized testing

A distributed system is often structured as a *client-server architecture* where a central server controls and coordinates a group of (remote) machines [Gho14]. With such an approach, execution time can be drastically lowered. This has been demonstrated by Garg et al. in the context of regression testing of methods. The authors establish a functional dependency graph of an application, re-order its test cases and execute them distributively. Eventually, they are able to reduce the execution time by 66% in comparison to a non-parallel run [GD13].
Kumar et al. achieve in similar experiments a reduction of execution time of even more than 80%. To do so, they first analyse and establish a graphical representation of the source code. From these graphics then potentially regressive sequences are extracted and distributed among a group of instances. Hence, the named authors also apply a sort of test reduction [KKSLM14].
Another tool based on a distributed architecture is *DiPerF*. However, DiPerF is a framework for testing of service' performance, i.e. load testing. It therefore does not split a test suite. The workflow of DiPerF is nevertheless related to the one of the tool presented in this thesis: DiPerF first connects to a group of machines to which it then deploys the testing code. It further coordinates execution of the code, and collects the measurements. At the end of the run, the tool cleans up the remote hosts and aggregates the results. DiPerF additionally creates visualizations of the results [RDRF06].
While all the cited researchers use a group of dedicated servers in a lab, the tool of this thesis can also run in a virtual environment. The computational tasks are executed on remote instances hosted by a public cloud provider. Using a cloud for testing tasks is not a novel approach. The principle is known as *Testing as a service* (TaaS) and it mainly refers to use cloud infrastructure for functional testing [GBT11], [RKTSR16]. Similar to DiPerF, there exist frameworks to carry out load tests of applications with cloud instances [ZCTA11], [ZLZY13]. However, none of these researchers have explicitly used cloud solutions to optimize performance test execution by dividing a test suite.
That the performance of the cloud itself could be an issue has been examined by a handful of researchers. Especially the leading cloud provider *Amazon EC2* experiences relevant variation in execution when resources are accessed from many tenants [FJV+12], [MDH+12], [LC16], [AB17].
A recently published paper tries to circumvent such inconsistencies with the concept of *Randomized Multiple Interleaved Trials* (RMIT) which should ensure to create repeatable results in a distributed environment. A trial in RMIT is defined as a measurement of one benchmark iteration. The idea is to execute several trial alternatives in different orders. Alternatives are for example versions [AB17]. For simplification, this thesis projects the concept to a different level: A trial

is defined as consisting of a benchmark which is iteratively run in a certain version. Chapter 3 describes the method in more detail.

2.3 Hopper

The tool which is used in this thesis for gathering performance metrics is a command-line tool named *hopper*. It has been developed by Laaber and Leitner in 2016 for the purpose of mining software project with extraction of performance metrics. Its source code is available on the development platform GitHub.[6]
The projects which can be examined by hopper must be written in a language which runs on the Java Virtual Machine and is built with *Maven* or *Gradle*, respectively. A collection of either JMH-benchmarks or JUnit tests needs to be included. The hopper-script further requires the project to be built on top of the version control system *git*.
In order to execute the script, the git-repository needs to be cloned to the local computer, the JMH-benchmarks extracted and a configuration file prepared. The configuration file includes specifications of the path to the benchmarks, the version IDs to execute, the number of test iterations, and some of the JMH-specific arguments as specified in section 2.2.1. In addition, the script requires several execution parameters telling it which test types to execute and where to place the results. These parameters can directly be defined by means of flags at program call. For hopper, several run configurations are available. It can for example be chosen whether to mine only commits containing code changes, or to skip every n-th version. As build system, either Maven or Gradle can be used [LL17].

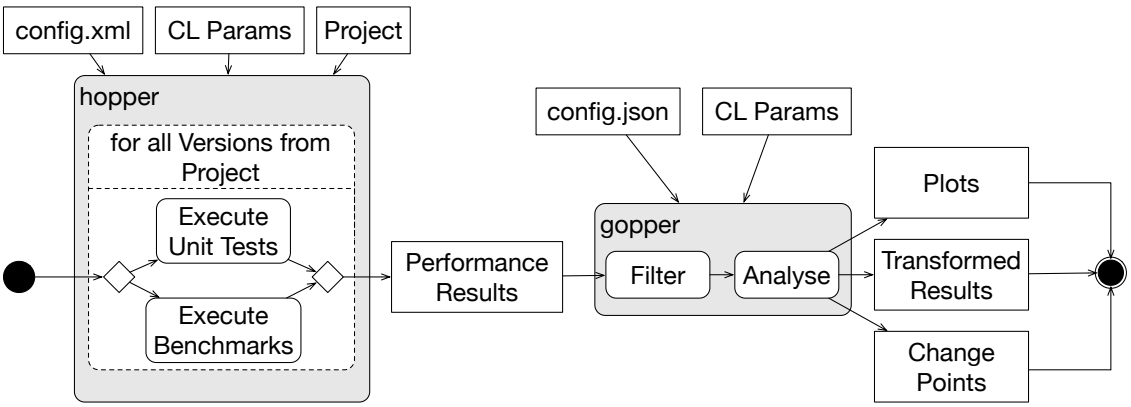

Figure 2.1: Sample Workflow of *hopper* [LL17]

During execution, hopper displays the JMH-log information in the console. Afterwards, the gathered measurements are written to an output file in CSV-format. The file concretely contains the specified command line parameters and lists the results for each benchmark in the columns *Project*, *Version*, *SHA*, *Configuration*, *Test* and *RawVal*. While the field *Project* describes the name of the project, *Version* refers to the mined Maven-version or git-commit. *SHA* stands for *Secure Hash*

[6] `https://github.com/sealuzh/hopper`

Algorithm, thus in case, Maven-versions are mined, this field contains the commit hash of the performance metric. Otherwise, it defaults to the value in field *Version*. The executed benchmark is stated in field *Test*. If it was run with different configuration-parameters, the corresponding value is set in field *Configuration*. *RawVal* finally indicates the obtained measurement for a single iteration. For more than one measurements iterations, the values are listed in successive lines.
The left part of figure 2.1 shows the sample workflow of hopper as defined by Laaber and Leitner. Additionally attached to the pipeline is the tool *gopper* which is designed to examine performance measurements. The tool presented in the next chapter preserves the existing workflow as well as all possible configuration parameters. The difference lies in the deployment- and storage-concept: the new tool deploys hopper on a remote machine and let it write its result to a central repository. The format of the final output file is thereby the same as described above. The evaluation in chapter 4 is hence done by means of these values and compares benchmarks and versions.

Chapter 3

Description

This chapter states the requirements of the presented solution. It provides specifications of the system architecture with all its components and processes, describes the used state concept and the theoretical workflow. A section with technical details about the implementation rounds off this chapter. It includes a note about the used approaches to split the test suite as well as the applied storage concept.

3.1 Architecture

In order to face the stated problems and challenges of the previous chapter, this thesis proposes a parallel approach in form of the script *clopper* with which it aims to reduce the overall execution time compared to a non-parallel version. An overview of the overall system architecture of clopper is depicted in figure 3.1. The script can be deployed from a local host which is situated on the left in the figure and acts as the managing instance. For execution, it requires a project, an XML-configuration file used for the hopper-script, and a general configuration-file in JSON-format. The local host handles automatic connection, and set up of the remote instances.
It is a crucial requirement of the local host to equally divide a test suite. Besides this task, the host composes a work order in form of a simple listing with command-line parameters. The test suite splits are then distributed among the available instances. The depicted data-flow from the clopper-script to the remote instance thereby consists of a test suite split, the prepared parameters, the XML-configuration file and a project.
The local host is further responsible for supervision of execution on the remote instances. It therefore runs a communication proxy stub. This stub opens a port for each instance, and creates channels to the instances. Via these channels, the stub can send a request to trigger execution of the hopper-script, and receives status updates from the working instances.
The remote instances consist of a group of virtual cloud instances which can origin from any cloud provider. It is also possible to use a group of dedicated computing stations. However, the instances are all equally set up and most importantly, they run a copy of the recycled hopper-script. For installation of software required for hopper, the communication- and result-saving-components, there is an installation configuration script provided which can be run at first start up of the instances. As a counter-part to the proxy stub on the local host, a proxy skeleton is implemented on the remote instances. This skeleton – or *gRPC-server* – handles requests from the local computer; it prepares the input for hopper, starts its execution by calling the received work order, and reports the current status of the instance. The instances are connected to a central repository in which the hopper-script stores intermediate results. Upon request, the local host can finally access the repository and download the stored metrics.

In summary, the implementation is based on the following principles:

- Reduction of the overall execution-time in comparison to the non-parallel version
- Automatic set up, deployment and management of remote hosts
- Decomposition and distribution of the test suite to the remote instances
- Extendability to other cloud providers
- Interoperability with and re-usability of the existing hopper-script
- Continuous result-saving to a central repository

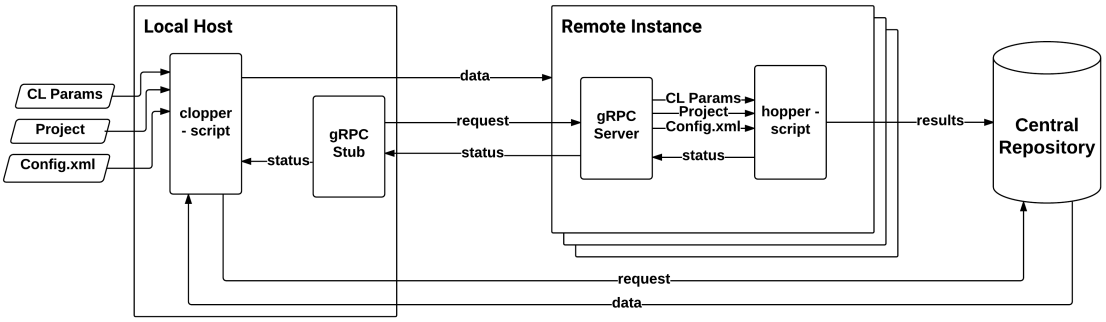

Figure 3.1: Architecture of the *clopper*-tool

3.2 Workflow and State Concept

In order to provide a constructive feedback to the experimenter, the remote instances implement a state concept. Every five minutes or whenever entering a new state, the instance pushes a status update to the server signaling it is still working. The following paragraph describes the workflow of the script alongside with the purpose of the used states. The states are also displayed in figure 3.2, the simplified workflow is shown in figure 3.3.
At the beginning (*Start Execution* in figure 3.2), the instances are running but in state *SLEEPING*. If necessary, the local host executes an installation script on the instances. Subsequently, the depicted workflow in figure 3.3 starts. In order to parallelize the execution, the provided test suite is splitted. The local host connects to the available instances and distributes the generated splits as well as the project to mine using *Secure Shell* (SSH) alongside with *Secure Copy* (SCP). Afterwards, the local host establishes a dedicated communication channel. This triggers the instances to wake up. They start listening to port 8080 for incoming data requests. If a connection attempt fails the local host will report a critical log message and terminate execution with a note to the experimenter to check connection. Otherwise, it sends a first request which provokes the connected instances to send a *Hello*-message (state *HELLO*). The remote instances are

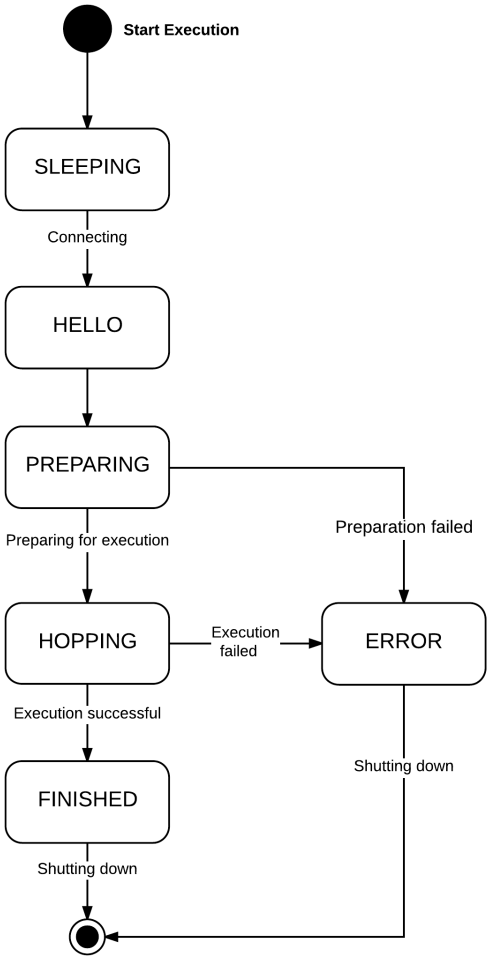

Figure 3.2: State concept implemented by the remote instances which are controlled by *clopper*

now ready for computation. If the local host has received positive feedback from all instances, it broadcasts a preparing-request. Upon this request, the instances look for three compressed directories containing a project with benchmarks to mine, command-line parameters and one or more configuration-files needed for execution of the hopper-script. They signal state *PREPARING* and unzip these directories.
After successful preparation, the local host sends a request to start computation. The remote instances call the command they have found in the unzipped command-line-parameter file and trigger execution of hopper. The original hopper-script was extended in such a way that it writes intermediate results to a remote repository after execution of each version.
In order to ensure the instance has terminated work, the status *FINISHED* was introduced. If an instance has processed all configuration files and all results have been stored, it signals status *FINISHED* and releases the communication-port. Status *FINISHED* is also suitable to catch the moment an instance might be ready to be used for different tasks.
The local host, however, waits for all instances to terminate work. From the remote repository, it

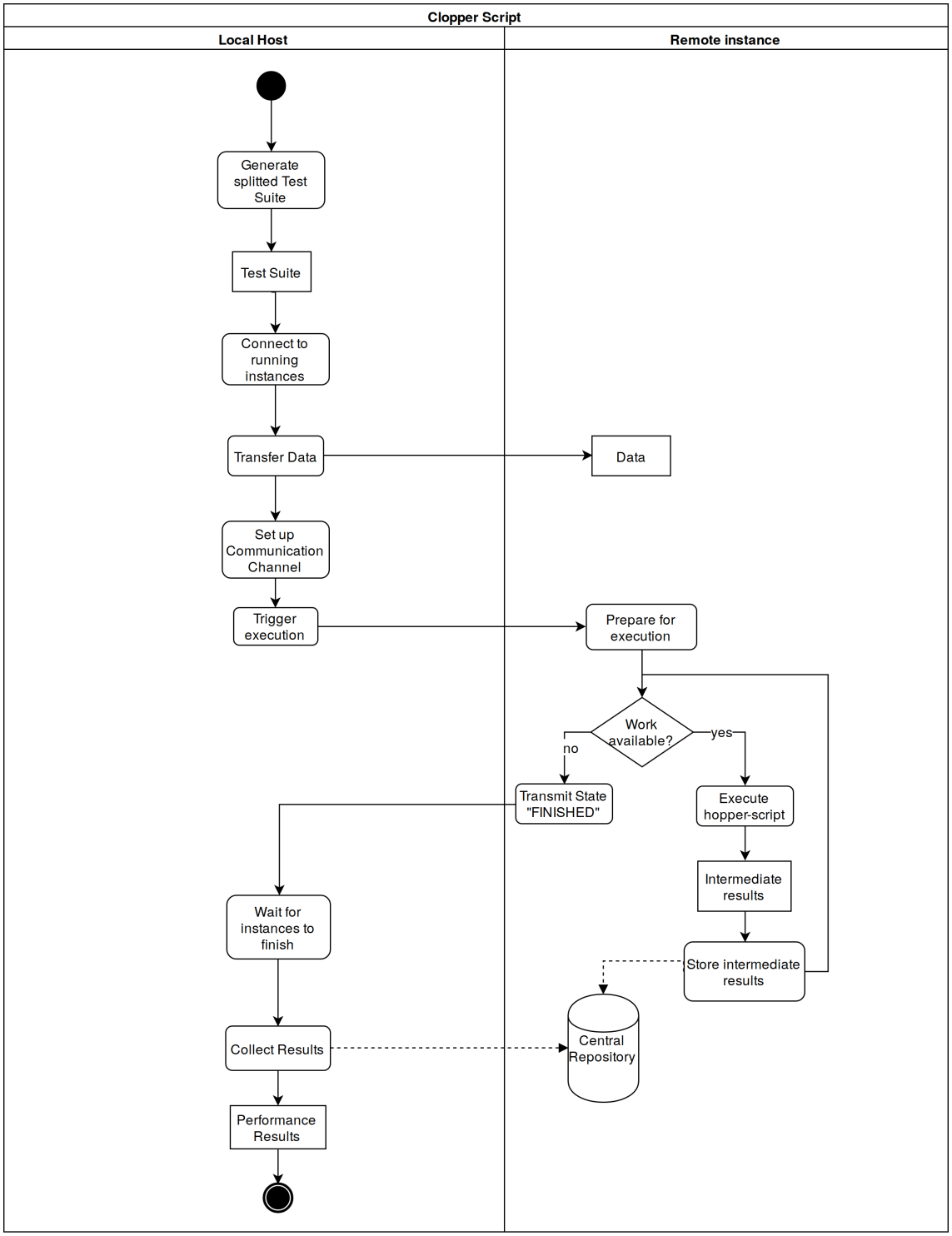

Figure 3.3: Workflow of *clopper*

eventually downloads all the results and compiles a single output file.

In case, a cloud instance encounters an erroneous situation, it transmits status *ERROR*. However, in such a case the clopper-script provides a feedback to the experimenter displaying the instance's workload and status log. Alongside with the intermediate results stored on the central repository, he is able to determine the moment of crash on the instance. With this information available, it is left to the experimenter to decide whether to try a task repetition or not. By all means, the recovery-script in the clopper-directory should be executed to reset the instances to a prior execution state, release occupied ports and terminate any other running scripts.
The described states are transmitted using gRPC. Section 3.3.1 provides more detail for its implementation.

3.3 Implementation

3.3.1 Communication

For enabling the remote communication from the local managing machine to the remote instances, the publicly available framework *gRPC*[1] is used. It is based on a client-server-principle where a client requests data from a remote server and gets a response – either a single message or a stream of messages. In the used architecture, the client code is deployed on the local host requesting states from multiple servers. The servers are represented by the remote instances.
For communication setup, gRPC uses so-called *protocol buffers*.[2] Protocol buffers provide a simple way to define a service interface and structure the content of the messages with different fields. The gRPC-tools then automatically generate the appropriate code for client and server. One part is implemented by the local host who sets up a communication channel and handles all the requests. The remote instances on the other hand, implement a skeleton which offers the counterpart of the methods defined in the service interface (see also figure 3.1).
In order to transfer messages, gRPC applies the revision of the *Hypertext Transport Protocol*, namely HTTP/2. The application is not bound to a dedicated port but the instances listen to one of the standard HTTP ports – namely port 8080 – for incoming requests. The local host however, uses different port numbers for the individual instances. The port numbers are generated from the instances' name. Since a secure shell is used the local host must additionally forward its local port to the remote on the instances at connection setup.
Besides the light and easy implementation, the advantage of gRPC lies in is its extensibility to other environments. Hence, if it is required to deploy clopper from a different platform or use another programming languages, the communication part is recyclable.

3.3.2 Preparation and Execution

The clopper-script requires for execution a number of packages. Since the script was developed and tested on a machine running *Ubuntu Gnome 17.04 (Zesty Zapus)*, the packages and versions refer to this operation system. First of all, Python 2.7 must be installed on the machine. Also, the packages *paramiko* version 2.1.2 for enabling of SSH networking, the *scp*-module version 0.10.2 to securely copy files to the instances, as well as *grpcio* and *grpcio-tools* version 1.2.1 for the communication-part must be installed. Further packages needed for the version-extraction are *git* version 2.11.0, *pygit2 v 0.25.0* with all its requirements[3], and *untangle* version 1.1.0 for XML-

[1] `http://www.grpc.io/`
[2] `https://developers.google.com/protocol-buffers/`
[3] according to the instructions on: `http://www.pygit2.org/install.html`

```
{
"total":3,
"ip-list": {
"instance-1": "12.345.567",
"instance-2": "12.345.678",
"instance-3": "12.345.789"},
"project": "/home/selin/project",
"username":"cloudmanager",
"distribution": "VersionDistributor",
"ssh-key":"/home/selin/.ssh/ssh-key-file",
"setup":"True",
"CL-params": {
"-f": "/home/selin/config.xml",
"-o": "/home/selin/final-output.csv",
"-t": "benchmark",
"-b": "commits",
"--tests": "'BenchA$|BenchB$BenchC$'",
"--cloud": "/home/selin/storage-credentials.json storage-bucket"}
}
```

Listing 3.1: Sample JSON-configuration file for execution of the *clopper*-script

file extraction. Lastly, the package *google-cloud storage*[4] version 1.0.0 is needed for bucket-storage writing.
For this purpose, it is also necessary to establish a project on the *Google Cloud Platform* and to create a storage bucket alongside a storage credentials key. On this, more details are given in section 3.3.4.
Clopper can either be executed on a group of remote computers or on a number of virtual instances on a cloud-platform. In any case, they must run *Ubuntu 16.04 LTS* and allow SSH-access, i.e. a SSH-key must be available.
The clopper-script is invoked with a configuration file in the JSON-format where all required and optional parameters for the clopper- as well as the hopper-script are specified. After parsing the JSON-file, the fields are checked for completeness and validity. A sample file is shown in listing 3.1. The mandatory fields are:

- total: This field specifies the total number of remote instances.
- ip-list: Below the term *ip-list* a dictionary of the form { instance-name : ip } is specified. The instance-name must end with a hyphen and a unique number of maximal five digits length (e.g. instance-22221). The digits specify the port for communication hence it is important not to use the system-port numbers between 0 and 1023 since these require super-user privileges.
- project: A prerequisite to run the hopper-script, is a git-repository with a collection of either JMH-Benchmarks or JUnit Tests available. The field *project* specifies the path of the project-directory containing the sub-folders with the JMH-root directory and the git-repository.
- distribution: The field *distribution* describes the splitting method which is used to generate equally splitted test suite parts. Available options are *RandomVersionDistributor*, *Ver-*

[4] following the instructions on: `https://cloud.google.com/storage/docs/xml-api/gspythonlibrary`

sionDistributor, *TestDistributor* or *RMIT*. Section 3.3.3 provides more details about the splitting methods.

- ssh-key: When connecting to remote hosts using SSH, an identification file is needed. This field specifies the absolute path to the public SSH-key.
- setup: If an instance is started up for the first time and needs configuration, the field *setup* must be set to *True*.
- username: This field is optional as by default, clopper uses the hostname on the system to access the remote instances. If the username on the instances is different though, it must be specified accordingly.
- CL-params: *CL-params* are another dictionary containing the following command line parameters that should eventually be passed to hopper:

  -f: the path to hopper's XML-configuration file

  -o: the path to the final output-file in CSV-format

  -t: the test type to execute, either *benchmark* or *unit*

  -b: the version type, either *commits* for git-commits, or *versions* for Maven versions.

  -- cloud: This field is a tuple containing the name of the central storage repository and the path to its credentials file. The two parameters can occur in any order but must be separated by a whitespace character.

Optional CL-params are:

-- tests: This is a list of selected tests to mine. For benchmarks, it should be of the form 'BenchA$|BenchB$|BenchC$' which refers to a regular expression, and for unit tests 'TestA, TestB, TestC'.

-- step n: If this field is specified, only every n-th version is executed. The step defaults to 1.

-- build-type: This field defines if builds between versions are clean or incremental. Available options are *clean* and *inc*, the default is *clean*.

-- skip-noncode: In case *skip-noncode* is set to *True*, the hopper-script skips versions without code change (e.g. change only in comments).

After preparation of the configuration file, the actual script can be executed by running the following command:

```
$ python ./clopper.py config.json
```

The script must be executed from the folder in which the clopper-script is stored. The command-line argument *config.json* refers to the prepared JSON-file.

3.3.3 Test Suite Distribution

The hopper-script takes ranges of git-commits or a list of Maven-versions as input as well as a collection of JMH-benchmarks or JUnit-tests. The test suite thus consists of a number of benchmarks which are iteratively executed over a sequence of versions. For division of the test suite, four different methods are implemented:

Random Version Distribution: For execution of the first method, the field *distribution* in the JSON-configuration file must be set to *RandomVersionDistributor*. The method generates a list of versions based on the specification in the hopper-configuration file. The version-list is then shuffled, cut, and distributed among the instances (see also figure 3.4). The list of tests is not manipulated, thus, each instance executes all tests. This method aims to distribute the work load in a random manner which should ensure that each instance is always equipped with different subsets of the test suite – especially when conducting multiple experiments. Such an approach increases the statistical expressiveness.

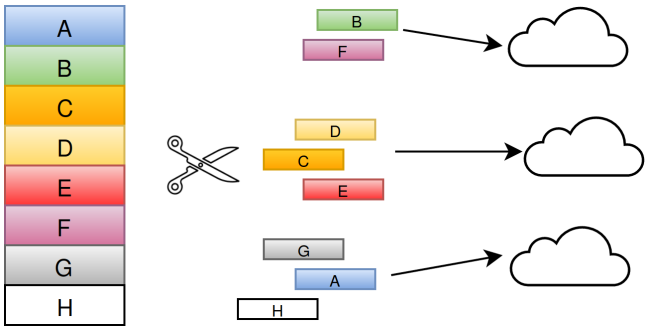

Figure 3.4: *Random Version Distribution* where A to H denote single versions of a project

Version Range Distribution: The second method is triggered by setting the *distribution*-field in the JSON-file to *VersionDistributor*. It is similar to the prior one but assigns packages of consecutive units, i.e. version ranges, instead of randomly selected versions (see figure 3.5). Each instance executes all tests. Intuitively, the workload of the instances as well as the execution time should be equal to distribution by Random Version. Version Range Distribution aims to determine whether the obtained measurements show equal variability between versions as the ones gathered by Random Version Distribution.

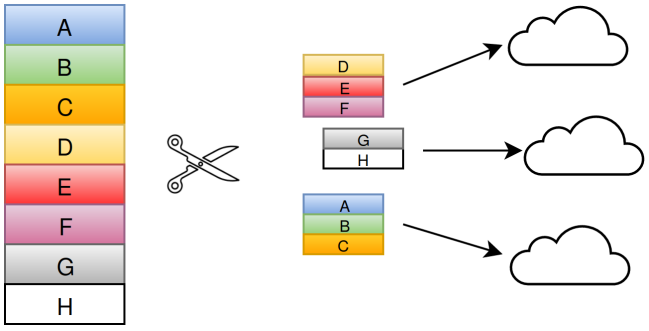

Figure 3.5: *Version Range Distribution* where A to H denote single versions of a project

Test Distribution: When applying Test Distribution, the corresponding field in the JSON-file must be set to *TestDistributor*. In this method, the range of versions is not changed. Instead, a list of available unit-tests or benchmarks is generated, shuffled and evenly distributed among the instances. The principle is shown in figure 3.6. Similar to Random Version Distribution, the shuf-

fling ensures to establish unique test suite subsets for more robust results. Test Distribution aims to preserve the testing environment during execution. Such an approach avoids false-positive performance regression between versions since benchmark results for different versions origin from a single instance.

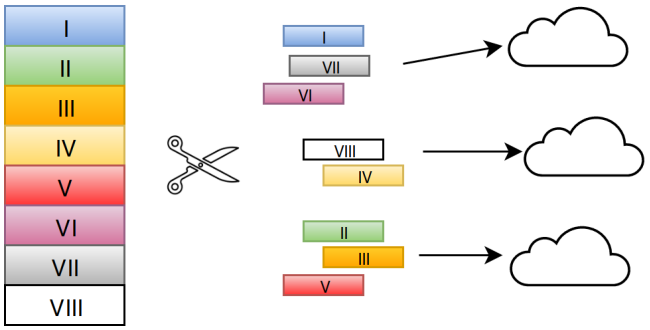

Figure 3.6: *Test Distribution* where I to VIII denote benchmarks of a project

Randomized Multiple Interleaved Trials (RMIT): Reliability and repeatability are key requirements in performance benchmarking ensuring the trustworthiness of the obtained performance results [FP16]. The concept of RMIT seems therefore suitable for this task. In this thesis, a trial is defined as consisting of a benchmark which is iteratively run in a certain version. For this purpose, all possible version-test-combinations of the project are generated. In reference to figure 3.7, where versions are marked with letters A to C, and tests labeled with the Roman numbers I and II, all possible version-test-tuples are: A-I, A-II, B-I, B-II, C-I, and C-II. When applying RMIT distribution, these tuples are shuffled and evenly distributed among the instances. So a possible distribution among three instances would be: instance 1: B-I, A-II, instance 2: C-I, A-I, and instance 3: B-II, C-II. The intuition of RMIT differs from the other distribution methods as it splits the test suite in two dimensions whereas all others preserve at least version or test dimension. In the JSON-configuration, clopper requires the keyword *RMIT* to trigger this distribution-method.

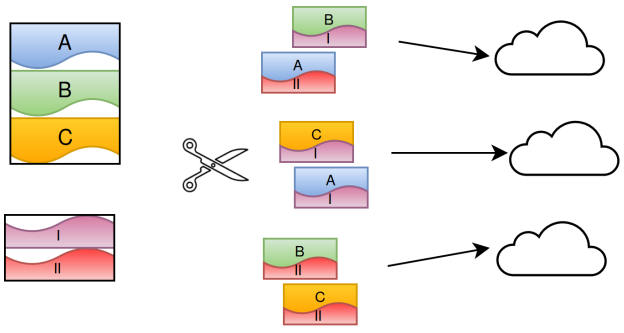

Figure 3.7: *Randomized Multiple Interleaved Trials* where letters denote versions, and Roman numbers benchmarks of a project

The generated test-suite is cut into as many pieces as specified by the field *total* in the JSON-configuration file. In order to use the instances to their full capacity, the pieces are desired to be of equal size. If no perfect distribution is possible, some instances receive a test suite being bigger by one version, test or tuple. The smallest possible test suite split contains one unit. In case the test suite is too small, a number of instances receives no work and is released by the script.
Currently, four distribution methods are implemented but in order to be open for future extension, the *Strategy-pattern* [VHJG95] has been applied. Advantageously, the strategy-pattern provides a unified access point to the distribution-method. The user only needs to define the field *distribution* in the configuration-file and the script will take care of evoking the appropriate algorithm. For future extension, thanks to this pattern, additional distribution methods can simply be plugged into the mechanism by appending to the code without having to touch the existing algorithms.

3.3.4 Output and Storage

In the original version, the hopper-script writes the calculated metrics to a single output file in CSV-format. As continuous result-saving to a central repository is a key requirement of the tool, the original script was extended accordingly. Whenever the hopper-script is called with -- *cloud*, it continuously writes the gathered metrics version by version to a central repository, which is in this case the *Google Cloud Storage*.[5] To read from or write data to the bucket, the instances as well as the local host need to have access to the bucket's name and a Google service account key in JSON-format. The -- *cloud*-flag is followed by a tuple containing the absolute path of the key file and the bucket name.
In addition to the output file, clopper generates a log-file. The log-file gives feedback during the installation progress, displays the workload assigned to the instances and provides the opportunity to monitor execution. A sample is depicted in listing 3.2.
As the log file does not give any information about the current running task, it is also possible to monitor working progress in the storage bucket. After execution of each version, the instance uploads an output-file named with its name, the processed version and a time-stamp.

[5] `https://cloud.google.com/storage/`

```
2017-05-24 16:36:27 [INFO] Starting execution
2017-05-24 16:36:27 [INFO] Json-file is valid.
2017-05-24 16:36:27 [INFO] Start setting up hosts.
2017-05-24 16:36:27 [INFO] Connected (version 2.0, client OpenSSH_7.2p2)
2017-05-24 16:36:28 [INFO] Authentication (publickey) successful!
2017-05-24 16:36:28 [INFO] Start installation on instance-2
2017-05-24 16:38:29 [INFO] Installation in progress on instance-2
2017-05-24 16:39:58 [INFO] Installation on instance-2 completed.
2017-05-24 16:39:58 [INFO] Instances successfully configured.
2017-05-24 16:39:58 [INFO] Test suite generated and splitted.
2017-05-24 16:39:58 [INFO] VersionDistributor
2017-05-24 16:39:58 [INFO] [[['90c1a63', '4babc4b', 'a03ca23', 'ff99a8a',
'df3823f', 'da408f7', 'd0e9d70'], [None]]]
2017-05-24 16:40:16 [INFO] Config-files generated.
2017-05-24 16:40:16 [INFO] Commandline parameters prepared.
2017-05-24 16:40:16 [INFO] Connected (version 2.0, client OpenSSH_7.2p2)
2017-05-24 16:40:16 [INFO] Authentication (publickey) successful!
2017-05-24 16:40:40 [INFO] Splits distributed among instances.
2017-05-24 16:40:40 [INFO] Waiting for instances to start grpc server...
2017-05-24 16:40:45 [INFO] Starting cloud-manager-client...
2017-05-24 16:40:45 [INFO] Channel created.
2017-05-24 16:40:45 [INFO] Hello from instance-2
2017-05-24 16:40:45 [INFO] Trigger hopper execution...
2017-05-24 16:40:50 [INFO] instance-2 --- PREPARING
2017-05-24 16:40:51 [INFO] instance-2 --- HOPPING...
2017-05-24 17:51:04 [INFO] instance-2 --- HOPPING
2017-05-24 17:55:13 [INFO] instance-2 --- FINISHED
2017-05-24 17:55:13 [INFO] Shutting down cloud-manager-client...
2017-05-24 17:55:13 [INFO] Grabbing results...
2017-05-24 17:55:17 [INFO] Execution finished.
```

Listing 3.2: Sample log file produced by the *clopper*-script

Chapter 4

Evaluation

This chapter describes and evaluates distributed performance tests on cloud instances conducted by clopper. The goal is to state the differences between the execution modes as well as the distribution methods and assess the produced output. Eventually, this chapter provides answers to the following research questions:

RQ 1: *In what ways is it possible to distribute a performance test suite for parallel execution?*

RQ 2: *How much time and costs can be saved by executing performance test suites in parallel?*

RQ 3: *Are the results obtained by different distribution techniques similar in terms of quality?*

4.1 Experimental setup

4.1.1 Cloud Provider

As mentioned in chapter 2, some of the public cloud provider suffer from unstable resource provision. Besides others, the *Google Compute Engine* (GCE)[1] is considered to be more robust against multi-tenancy [LC16]. As it is further stated by [LC16], neither the time of the day nor the day of the week seem to have measurable impact on performance of GCE instances. For these reason, GCE was chosen for the experiments also because manual parallelization-experiments of hopper have already been executed on this cloud platform. Furthermore, there is a 12 month's free trial with a credit of 300$ available. In this free trial, per account a total of 8 virtual cores is allowed which enables testing with up to four double- or eight single-cored instances.

4.1.2 Approach

The experiments were conducted on three different Google accounts between the 23rd and the 31st of May 2017 from the writer's personal computer running *Ubuntu Gnome 17.04*. All experiments were repeated three times, i.e. in total, measurements from three runs were obtained. The potential influence of the time of the day was not explicitly addressed. However, the experiments happened to run at different times, on a variety of instances, and in no distinct order.
Non-parallel tasks used a single cloud-instance of type *n1-standard-1* in the zone *europe-west1-b*[2]. n1-standard-1 instances have a virtual CPU of 3.75 GB memory on a 2.6 GHz Intel Xeon E5 (Sandy

[1] `https://cloud.google.com/compute/`
[2] `https://cloud.google.com/compute/docs/machine-types`

Bridge) processor implemented. The boot disk type was a SSD persistent disk. As operating system, *Ubuntu 16.04 LTS* was chosen. For parallel tasks, the same machine type and configurations were used. On each of the three Google accounts, six of such instances were created.

The requirements for the projects to mine were to be open-source, i.e. on GitHub, and having a testsuite of JMH benchmarks available. One project was the one used by [LL17] in the presentation of the hopper-tool. The others were selected by manually filtering GitHub for pom.xml-files in Java projects which contain the tag *group-id* with text *org.openjdk.jmh*. Some were not suitable since they contained only a few commits or benchmarks. Others, on the other hand were not executable with hopper. The final selection consisted of:

- **JCTOOLSproject**: *JCTOOLS* stands for *Java Concurrency Tools* and is a project which adds a range of concurrent data structures to the *Java Developer Kit* (JDK). JCTOOLS originally consists of 61 benchmarks. A random selection of 23 of JCTOOLs implemented benchmarks serves as the largest test suite. This project was chosen since its code is clearly laid out which made it easy to remove a certain number of benchmarks.[3]
- **protostuff**: *protostuff* is a serialization library which supports tasks such as validation, and is forward and backwards compatible. protostuff's 16 benchmarks represent the medium test suite of this thesis.[4]
- **RDF4J**: *RDF4J* is an Eclipse project. The framework was well known as *Sesame* and includes methods to process and query RDF data. RDF4J was selected since it originally consisted of 11 benchmarks which is one of the smallest benchmark test suites on GitHub but still has an adequate number of commits.[5]

The small and large test suite sizes were aligned with the medium one: First, project protostuff was selected to represent the medium test suite, and from this project, all 16 benchmarks were used. The other test suites should not be too small to provide adequate statistical expressiveness but also not too big since execution should not last for days. Furthermore, the test suites should be different enough in size from protostuff to observe variations in execution. Finally, the compromise were 9 benchmarks as small, and 23 tests as large test suite; 16 lies exactly between the two numbers, and the results are visualizable in tables and violin plots.
In JCTOOLS and protostuff, three benchmarks additionally have multiple configurations but since the clopper-script cannot distinguish them, they are counted as one benchmark. For evaluation, the command "git rev-list --all" was used to list all version, i.e. git-commits. From this list, 10 consecutive commits were extracted in randomly choosing a beginning and an end from the list. The test suite for each project is displayed in table 4.1.
After determination of the version range to mine, the XML-input file was prepared. In order to get comparable results, each benchmark was run with 10 warm up- and 20 test iterations. The applied JMH-benchmark mode was average execution time.
For examination of the methods in parallel mode, the test suite was distributed among six instances. The non-parallel measurements were obtained by one instance which was equipped with the same workload.

[3] `https://github.com/Buzzerio/JCTOOLSproject`
[4] `https://github.com/protostuff/protostuff`
[5] `https://github.com/eclipse/rdf4j` and `http://rdf4j.org/about/`

Project	Versions	Benchmarks
JCTOOLS	f03ae9a e8d6bd9 179309f e58610b 8646698 9e3c2c9 3813555 9a0ee76 8d447a5 a158e5b	QueueOfferPoll.offerAndPollLoops BaselinePingPong.pingpong SingleThreadedPoll.poll ConcQOfferPoll.offerAndPoll SingleThreadedOffer.offerLoop SetOps.contains SetOps.remove SetOps.sum SetOps.add CountersBenchmark.rw ConcurrentMaprandomGetPutRemove QueueBurstCost.burstCost RingCqBurstRoundTripWithGroups.ring QueueThroughputBackoffNano.tpt QueueThroughputBackoffYield.tpt IntrusiveQueueThroughputBackoffNone.tpt MpqThroughputBackoffNone.pr MpqThroughputBackoffNone.nor QueueThroughputBackoffConsume.tpt MpqThroughputBackoffNone.cr QueueThroughputBackoffNone.tpt MpqDrainFillThroughputBackoffNone.normal MpqThroughputBackoffNone.bothr
protostuff	da27225 2c252e4 e48ab1d ace2a01 30d6024 c972d94 8e31181 323ae10 a5d36a9 db1ef89	RuntimeSchemaBenchmark.generated_deserialize_1_int_field RuntimeSchemaBenchmark.runtime_serialize_10_int_fields RuntimeSchemaBenchmark.runtime_deserialize_1_int_field RuntimeSchemaBenchmark.baseline RuntimeSchemaBenchmark.runtime_sparse_serialize_1_int_field RuntimeSchemaBenchmark.runtime_sparse_deserialize_10_int_field RuntimeSchemaBenchmark.generated_serialize_10_int_field RuntimeSchemaBenchmark.runtime_sparse_deserialize_1_int_field RuntimeSchemaBenchmark.generated_serialize_1_int_field RuntimeSchemaBenchmark.runtime_deserialize_10_int_field RuntimeSchemaBenchmark.runtime_serialize_1_int_field RuntimeSchemaBenchmark.runtime_sparse_serialize_10_int_fields RuntimeSchemaBenchmark.generated_deserialize_10_int_field StringSerializerBenchmark.bufferedSerializer StringSerializerBenchmark.builtInSerializer StringSerializerBenchmark.bufferedRecycledSerializer
RDF4J	90c1a63 4babc4b a03ca23 ff99a8a 2cfb106 5391dfd df3823f da408f7 d0e9d70 b8fd143	ForwardChainingRDFSInferencerBenchmark.initialize ForwardChainingSchemaCachingRDFSInferencerBenchmark.initialize NoReasoningBenchmark.initialize ReasoningBenchmark.noReasoning ReasoningBenchmark.forwardChainingRDFSInferencer ReasoningBenchmark.forwardChainingSchemaCachingRDFSInferencer ReasoningBenchmark.forwardChainingSchemaCachingRDFSInferencerMultipleTransactions ReasoningBenchmark.forwardChainingSchemaCachingRDFSInferencerSchema ReasoningBenchmark.forwardChainingSchemaCachingRDFSInferencerMultipleTransactionsSchema

Table 4.1: Performance test suites used for the experiments

4.2 Overall execution time

Research question 2 aims to determine the gain between non-parallel and parallel execution in terms of execution time and costs. To evaluate overall execution time, the log-files generated by the clopper-script were used. For each experiment, the duration from the first instance transmitting *HOPPING* until the last instance sending *FINISHED* were calculated. Due to potential network latencies, this duration might not exactly correspond to the actual execution time but is accurate enough for comparison of the results. For each method and execution mode (parallel or non-parallel), the average value as well as the standard deviation from the runs were built and subsequently compared using the *gain factor*. The gain factor is the result of dividing singular -, i.e. non-parallel, by parallel execution time. Values bigger than 1.00 correspond to a gain of time; the higher the value, the bigger the gain. The results are displayed in table 4.2. In addition to the overall execution time, the difference between the fastest and the slowest instance was calculated. Table 4.3 shows the corresponding outcomes. Except for the gain factor describing the proportions, all displayed values are represented in minutes.

When looking at table 4.2, it is not surprising, that all the examined methods perform better in

Project	Method	Benchs	Execution Time (Mean)		Factor	Execution Time (StDev)	
			Parallel (6)	Singular (1)		Parallel (6)	Singular (1)
JCTOOLS	Random Version Distributon	23	37.75	186.92	4.95	0.48	0.90
	Version Range Distributon	23	37.72	186.45	4.94	0.32	0.62
	Test Distributon	23	54.37	186.10	3.42	2.00	0.90
	RMIT	23	90.45	491.73	5.44	5.23	2.03
protostuff	Random Version Distributon	16	24.93	123.48	4.95	0.47	3.82
	Version Range Distributon	16	26.42	120.43	4.56	3.58	1.37
	Test Distributon	16	63.15	121.53	1.92	1.25	1.70
	RMIT	16	93.95	490.85	5.22	2.45	49.90
RDF4J	Random Version Distributon	9	16.55	80.23	4.85	0.35	5.08
	Version Range Distributon	9	16.83	79.88	4.75	0.15	4.82
	Test Distributon	9	36.92	77.42	2.10	3.10	3.68
	RMIT	9	19.92	99.70	5.01	1.95	4.18
Overall	Random Version Distributon	$\emptyset$	26.41	130.21	4.93	8.72	43.82
	Version Range Distributon	$\emptyset$	26.99	128.92	4.78	8.54	43.92
	Test Distributon	$\emptyset$	51.48	128.35	2.49	10.90	44.63
	RMIT	$\emptyset$	68.11	360.76	5.30	34.10	184.60

Table 4.2: Comparison of the overall execution time between parallel execution (6 instances) and singular (1 instance). Displayed are the mean and standard deviation of execution time. All values are shown in minutes except for the column *Factor* which describes how many times parallel - is faster than non-parallel execution.

parallel than in singular execution mode. The four distribution methods are not equal, though. On average, Random Version Distribution takes 26.41 minutes in parallel and 130.21 on a single instance. Version Range Distribution needs with six instances 26.99 minutes for a task, one instance used 128.92 minutes. Distribution by Tests takes on average 51.48 minutes in parallel and 128.35 on a single instance. And finally, the longest execution time on average has RMIT Distribution with 68.11 minutes in parallel and 360.76 minutes in non-parallel mode. However, even if RMIT Distribution needs the most time, it achieves the biggest gain. When computing in parallel, the six instances are 5.30 times faster than a single one. Distribution of Random Version and Version Ranges are close behind with a gain of 4.93 and 4.78, respectively. The weakest in performance is Test Distribution with an average gain of 2.49. These observations are independent of the test suite size.

In general, the two methods which distribute the versions among the instances are very close to each other in terms of execution time. Their results only differ in a few seconds which is due to the fact that most of the execution time is spent for compilation of versions. In both methods, two instances are assigned one version and four hosts process two versions. This is also the reason why these two methods are faster than the other ones. In Test Distribution for example, each instance processes 10 versions and hence, compiles 10 times. In RMIT, even more time is needed since each assigned version-test-tuple must be compiled individually. When for example mining JCTOOLS in non-parallel mode, the instance has to process 23*10 version-test-tuples. This explains why RMIT has such a long computation time.
Distribution by Random Versions eventually seems to behave most stable. RDF4J with the smallest test suite shows a factor of 4.85 with a standard deviation of only 0.35 minutes, medium and large test suite both have factor 4.95 and a standard deviation of 0.47 and 0.48 minutes, respectively. On the other hand, the method with the biggest spread is Test Distribution. JCTOOLS with the biggest test suite states a standard deviation of 2.00 minutes and a gain of factor 3.42 whereas the medium test suite only shows a gain of 1.92.
A question which might arises when looking at the table is why RMIT distribution shows in protostuff and over all projects such a high standard deviation? This must be the result of multi tenancy or network latency which was encountered in one of the three runs and has caused the mean and standard deviation to show such a high deviation.

The highest gain, was achieved by RMIT distribution in project JCTOOLS where six instances completed the workload 5.44 times faster than a single one. This stands in contrast to Test Distribution in project protostuff where six instances only encounter a gain of factor 1.92 in terms of execution time.
An interesting fact is, that overall execution time for the two methods Random Version and

Project	Method	Benchs	Exec. Time (Mean)		Difference Exec. Time	
			Fastest	Slowest	Mean	StDev
JCTOOLS	Random Version Distributon	23	18.60	37.75	19.22	0.43
	Version Range Distributon	23	18.75	37.77	18.95	0.27
	Test Distributon	23	33.88	54.37	20.48	3.30
	RMIT	23	79.67	90.45	13.30	4.48
protostuff	Random Version Distributon	16	12.28	24.93	12.67	0.33
	Version Range Distributon	16	11.93	26.42	14.47	3.53
	Test Distributon	16	35.38	63.15	27.77	2.13
	RMIT	16	74.93	93.95	17.77	2.50
RDF4J	Random Version Distributon	9	8.18	16.55	8.23	0.33
	Version Range Distributon	9	7.92	16.83	8.92	0.15
	Test Distributon	9	7.07	36.92	29.83	3.16
	RMIT	9	12.62	19.92	7.30	1.23

Table 4.3: Difference between fastest and slowest instance in parallel execution mode. Displayed are the mean and standard deviation of execution time. All values are shown in minutes.

Version Range Distribution can be estimated, if the duration of one version is known. This prediction can be made when comparing the information of table 4.3 with the factor of table 4.2: The test suite was executed over 10 versions. Using six instances, the smallest split of the test suite contains 1 version, the biggest one fifth, namely 2 versions. When looking at table 4.3 project protostuff and method Version Range Distribution, the instances show consistent differences in execution time with a standard deviation of 0.47. The fastest instance who processes a small split

needs for execution 12.28, and the slowest instance for a big split 24.93 minutes on average. The logical conclusion is thus, that the size of the biggest test suite split correlates with the overall execution time. The biggest split consists of one fifth, and parallel execution takes roughly one fifth of the singular execution time, i.e. table 4.2 states that it is exactly 4.95 times faster than singular execution.
For distribution of the tests, no such prediction can be made since the setup time is hard to determine. If in addition, multiple configurations are applied to a benchmark overall execution time further increases. The distribution-methods of the clopper-script cannot distinguish between benchmarks with and such without configuration and neither determine the number of different parameters applied.
In the experiments, for RDF4J, between the first and the last instance sending *FINISHED* up to half an hour passed by whereas each instance had to mine 3 or 4 benchmarks. For RMIT distribution, this difference also occurs but only consists of 7.30 minutes on average with a standard deviation of 1.23. In JCTOOLS though, the results are closer to each other. Test Distribution shows an average difference of 20.48 minutes with a standard deviation of 3.30. In RMIT, the fastest instance is roughly 13.30 minutes faster which can vary up to 4.48 minutes.
In conclusion, it can be stated that RMIT Distribution spreads the computing-intensive tests more evenly which is the reason why it achieves the biggest gain-factor and encounters less difference between the slowest and the fastest instance than Test Distribution. Distribution by Random Versions shows the most stability in execution time. It is the fastest method very closely followed by Distribution of Version Ranges.

4.2.1 Cost Differences

The values for comparing the total costs of execution time are calculated by multiplying Google's standard per hour-price[6] by total execution. For a *n1-standard-1* instance in zone *europe-west1-b* the per-hour rate is 0.0523$. The price for the SSD persistent disk is only given per gigabytes (GB) per month. For a 10 GB SSD disk and 30*24 hours per month, the hourly rate thus comprises 0.0024$. The total price per hour eventually adds up to 0.0547$.
Table 4.4 displays the costs for singular execution and the sum of the costs for all six instances. JCTOOLS executed on six instances with method Test Distribution took for example 54.37 minutes on average. Google only charges full minutes, 54.37 is therefore rounded up to 55 minutes or rather 55/60 hours. The total costs are therefore:

6 instances * 55/60 hours * 0.0547$ per hour = 0.3009$.

The last column in the table corresponds to the price difference between parallel and singular execution costs. A positive percentage thus implies that parallel execution is more expensive than singular execution, a negative value indicates smaller costs, i.e. a price gain.
Since in the previous section, the gain-factor of parallelization was always smaller than the number of used instances, there is no cost reduction observable. Random Version Distribution for example costs on average 23.66 % more when using six instead of one cloud instance. In project JCTOOLS, the two methods are equally expensive. When inspecting the raw costs, RMIT is the most expensive method since it also has the longest execution time. With on average 0.3291$ for non-parallel execution, its costs are about three times higher than the ones from the other methods. In project protostuff, the difference is even bigger. Nonetheless, when looking at the difference between non-parallel and parallel execution, RMIT distribution achieves on average a plus in costs of 14.68 % which is the smallest difference. In JCTOOLS, the difference is only +10.98%. Version Range and Random Version Distributions are between 20 % and 33 % costlier when executed in parallel than on a single instance. Test Distribution is with differences of +76.47

[6] `https://cloud.google.com/compute/pricing`

% in JCTOOLS, +214.75 % in protostuff, and +184.62 % in RDF4J much more expensive. Table 4.4 summarizes the findings.

Project	Method	Measured Benchs	Total Execution Costs $		Price Difference %
			Sum of (6)	1 Instance	
JCTOOLS	Random Version Distributon	23	$ 0.2079	$ 0.1705	+21.93%
	Version Range Distributon	23	$ 0.2079	$ 0.1705	+21.93%
	Test Distributon	23	$ 0.3009	$ 0.1705	+76.47%
	RMIT	23	$ 0.4978	$ 0.4485	+10.98%
protostuff	Random Version Distributon	16	$ 0.1368	$ 0.1130	20.97%
	Version Range Distributon	16	$ 0.1477	$ 0.1103	+33.88%
	Test Distributon	16	$ 0.3501	$ 0.1112	+214.75%
	RMIT	16	$ 0.5142	$ 0.4476	+14.87%
RDF4J	Random Version Distributon	9	$ 0.0930	$ 0.0738	+25.93%
	Version Range Distributon	9	$ 0.0930	$ 0.0729	+27.50%
	Test Distributon	9	$ 0.2024	$ 0.0711	+184.62%
	RMIT	9	$ 0.1094	$ 0.0912	+20.00%
Overall	Random Version Distributon	∅	$ 0.1477	$ 0.1194	+23.66%
	Version Range Distributon	∅	$ 0.1477	$ 0.1176	+25.58%
	Test Distributon	∅	$ 0.2844	$ 0.1176	+141.86%
	RMIT	∅	$ 0.3774	$ 0.3291	+14.68%

Table 4.4: Differences in total costs between 1 instance, and 6 instances executing the same workload in a parallel manner. The price difference is given in percentage. Positive percentages imply higher costs for the parallel approach.

4.3 Quality

Research question 3 is concerned about the quality of the measurements the distribution methods produce. In order to assess the quality, the results from the different runs for each benchmark were set off against each other. For comparison of the collected values, the measurement-function *maximum spread* (*maxSpread*) was used. This function is based on yet unpublished work of the supervisor of this thesis (Laaber, together with Leitner). The intention of maxSpread is to capture the variability of a benchmark over different runs, and thus defines the stability of a method. The lower the value, the more stable is the execution of this benchmark and consequently, the better the quality of the method. maxSpread is given in percentage. As an example: Benchmark *runtime_serialize_1_int_field* of project protostuff encountered in version *ace2a01*, method RMIT Distribution a maxSpread of 0.6212. The results of the different runs for this benchmark varied therefore by 62.12 %.
This section further contains the results of three statistical analyses. The applied tests are briefly described in the corresponding subsection, its outcomes discussed and illustrated by means of *violin plots*. Violin plots show the relative distribution of a value – maxSpread in the context of this thesis. If a plot is small but bulgy, the examined measurement is consistent. If the plot is tall and narrow, the measurement is widely distributed. The plots are established by means of the Python *seaborn*-library[7]. Since seaborn uses the *Gaussian normal distribution* for calculation of the shape, values can lie below the x-axis. maxSpread is by nature always positive. Therefore, the y-axis has been cut at 0.0 and a comparison-threshold which is defined accordingly.

[7] `https://seaborn.pydata.org/`

4.3.1 Range of Benchmarks

The range of benchmarks refers to percentiles defining thresholds below which a certain percentage of all measurements lies. The percentiles P50, P90, P95, P99 and P99.5 of the maxSpread-values are listed in table 4.5. The measured values used for calculation are the max-Spreads of the benchmarks for each version, e.g. JCTOOLS has with 23 benchmarks executed over 10 versions 230 measurements available. The percentiles indicate what results can be expected when using a certain method.
When looking at the results in table 4.5, for each project a best and a worst performing method can be distinguished. In RDF4J, the method which obtains the smallest maxSpread is for example Version Distribution. 99 % of the obtained values lie below 0.226. Figure 4.1 visualizes this observation. Benchmark 3 encounters outliers, but besides that the vast amount of measurements lies below the red dotted threshold of 0.226. The maxSpread-display is limited to 0.7.
The worst performing method in RDF4J is though not as distinct as the best one. The values in the percentiles are close to each other and state different methods. In most cases, it is RMIT having for example a P99 value of 0.490.
In protostuff, Random Version Distribution obtains in all but one examined percentiles the low-

Project	Method	Measured Values	Percentiles				
			P50	P90	P95	P99	P99.5
RDF4J	Random Version Distribution	90	0.127	0.294	0.357	0.368	0.369
	Version Range Distribution	90	0.067	0.167	0.180	0.226	0.312
	Test Distribution	90	0.136	0.317	0.351	0.368	0.374
	RMIT	90	0.183	0.312	0.345	0.490	0.507
protostuff	Random Version Distribution	160	0.170	0.537	0.613	0.726	0.782
	Version Range Distribution	160	0.215	0.552	0.648	0.887	1.030
	Test Distribution	160	0.221	0.740	0.836	1.027	1.067
	RMIT	160	0.214	0.530	0.649	0.785	0.850
JCTOOLS	Random Version Distribution	230	0.141	0.584	0.690	0.975	1.238
	Version Range Distribution	230	0.145	0.491	0.662	0.827	0.922
	Test Distribution	230	0.220	0.522	0.944	1.042	1.052
	RMIT	230	0.124	0.490	0.635	0.679	0.680
ALL	Random Version Distribution	480	0.146	0.472	0.553	0.690	0.796
	Version Range Distribution	480	0.142	0.403	0.496	0.647	0.755
	Test Distribution	480	0.192	0.526	0.710	0.812	0.831
	RMIT	480	0.174	0.444	0.543	0.651	0.679

Table 4.5: Display of different percentiles of *maxSpread*. A lower maxSpread indicates less variability of benchmark measurements.

est maxSpread. 99 % have a maxSpread of less than 0.726. Test Distribution is on the other hand, the method with the highest maxSpread. Here, the 99th percentile states a maxSpread of 1.027. Version Range Distribution and RMIT obtain in protostuff more or less the same percentiles.
In JCTOOLS, the worst method for P50, P95 and P99 is also Test Distribution. Similar to protostuff, 99 % of the benchmark spread distributed by Tests lie below 1.042. This method shows around 10 % of outliers, since maxSpread shows a gap of 0.5 between P90 and P99. The leading method in JCTOOLS is RMIT. Furthermore, it seems to encounter only a few outliers. The difference between P90 and P99 only comprises 0.189.
When comparing the projects with each other, there are contradictions: The mostly worst performing method in RDF4J is for example RMIT whereas in JCTOOLS, this is the technique which

obtains the lowest maxSpread. On the other hand, RDF4J's best performing method is Version Range Distribution and this is also the method which obtains the lowest maxSpread in the first four percentiles over all projects. Nonetheless, the last percentile ranks RMIT as the best performing method which leads to the first conclusion that Version Distribution performs good on average but experiences a bigger amount of outliers than RMIT does.

The worst performing results are on average obtained by Test Distribution. This is in fact a

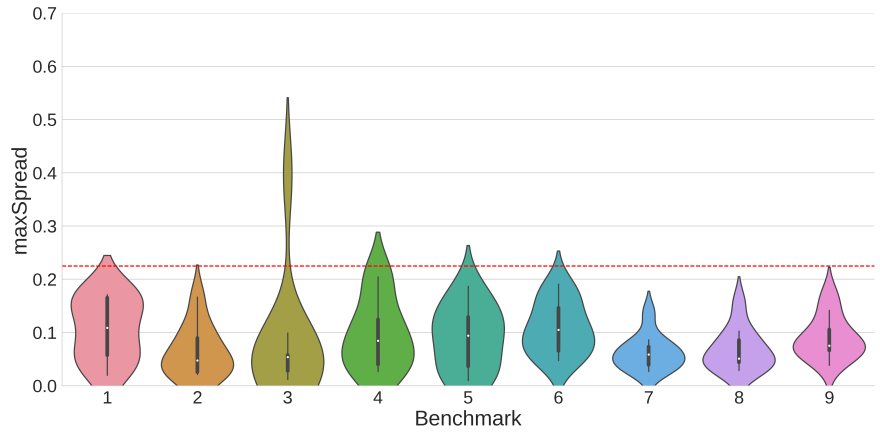

Figure 4.1: Version Range Distribution in project RDF4J with P99 threshold over all benchmark measurements. The y-axis for *maxSpread*-display is cut at 0.7.

surprising result. Test Distribution was expected to behave more stable than Version Range - and Random Version Distribution since results from a benchmark are always obtained from one single instance and not from different ones. This is a matter which needs more investigation (see future work in section 4.4.3). What further stands out, is the fact that the measurements of maxSpread in RDF4J are much smaller than in the other projects (see also figure 4.2). However, this has nothing to do with the test suite size but might origin in the structure of the mined project itself (e.g. many input-output-operations).

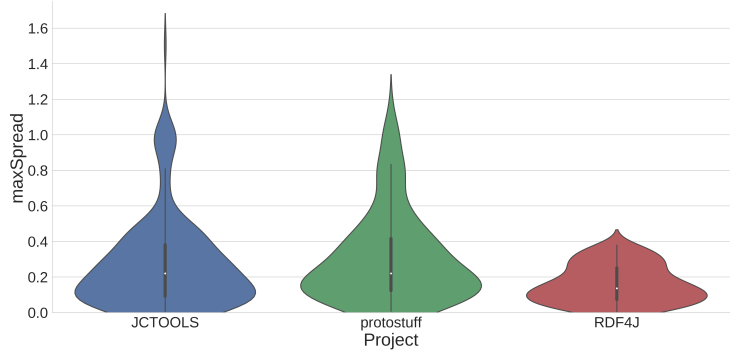

Figure 4.2: Comparison of projects in method Test Distribution. y-axis is cut at 1.65.

4.3.2 Similarity of Distribution Methods

How similar to each other are the results produced by the different methods? In order to answer this question, the maxSpread of a benchmark in one method was compared with the maxSpread in another method. For comparison, the *Wilcoxon signed-rank test* provided by *SciPy*[8] was used. The Wilcoxon signed-rank test is a statistical hypothesis test which pair-wisely compares two independent samples and determines whether the two sample origins from the same population. In this bachelor thesis, the test should give an indication of how similar to each other the methods perform. The null-hypothesis H_0 used in this context assumes that the compared samples belong to the same population, and the examined methods are similar.
The alternate hypothesis H_1 states that the samples are not obtained from the same population, and thus, the methods are not similar to each other. The significance threshold of the obtained probability – the *p-value* was set at .05. Any measurement below .05 signifies a rejection of the null-hypothesis. If the obtained p-value is bigger than .05, the null-hypothesis holds and the methods are considered as similar. To avoid false-positive results, continuity correction was applied to the p-values. The calculated p-values are displayed in table 4.6.
In order to obtain the values displayed in subtable *All projects*, the Wilcoxon-test was executed with a file containing the maxSpreads of all benchmarks from the projects JCTOOLS, RDF4J and protostuff in a particular method. According to the test results, distribution by Random Version vs. RMIT lies for the concatenation of the projects with a p-value of .044 below the significance level. The null-hypothesis is rejected and the two methods can be considered as delivering different results.

JCTOOLS	p-value			
	RV	**Version**	**Test**	**RMIT**
RV	X	.117	.494	.080
Version	X	X	.843	.915
Test	X	X	X	.843
RMIT	X	X	X	X

protostuff	p-value			
	RV	**Version**	**Test**	**RMIT**
RV	X	.552	.979	.245
Version	X	X	.394	.041
Test	X	X	X	.737
RMIT	X	X	X	X

RDF4J	p-value			
	RV	**Version**	**RMIT**	**RMIT**
RV	X	.013	.097	.636
Version	X	X	.343	.033
Test	X	X	X	.058
RMIT	X	X	X	X

All projects	p-value			
	RV	**Version**	**Test**	**RMIT**
RV	X	.051	.303	.044
Version	X	X	.890	.914
Test	X	X	X	.810
RMIT	X	X	X	X

Table 4.6: *p-value* obtained by the Wilcoxon signed-rank test in which the *maxSpreads* of a benchmark in one method was compared with the maxSpread in another method. RV stands for Random Version Distribution, Version refers to Version Ranges. A p-value <.005 implies a rejection of the null-hypothesis and indicates that the two examined methods output different measurements.

However, when comparing the projects, there is no strong trend distinguishable. VersionDistribution vs. RMIT for example scores high in project JCTOOLS (.915) whereas the other two projects obtain a p-value below the significance level, namely .014 in protostuff and .033 in RDF4J. Therefore, the methods show in these two projects statistically significant differences. Random Version vs. RMIT lies for project RDF4J with .013 also below the significance threshold, and H_0 is

[8] https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.stats.wilcoxon.html

rejected. For RDF4J, figure 4.3 and 4.4 where *maxSpread*-display is limited to 0.7, show the violin plots for the benchmarks of RMIT and Version Distribution next to each other. The shapes of the violin plots differ from each other and there are some abnormalities observable: For example in RMIT distribution (figure 4.3), the results show a wider spread with an evident amount of outliers whereas in Version Distribution (figure 4.4), the results are of small variability. Only benchmark number 3 shows outliers.

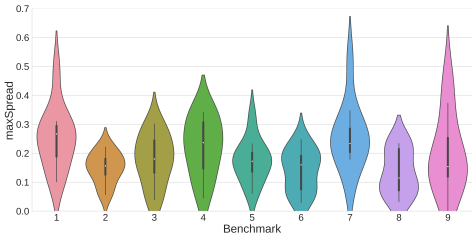

Figure 4.3: RDF4J RMIT Distribution

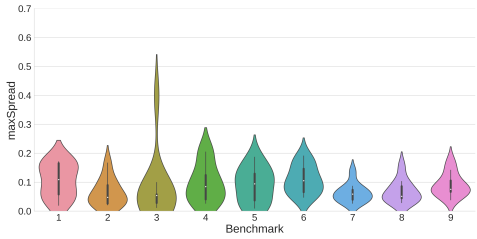

Figure 4.4: RDF4J Version Distribution

In conclusion, none of the methods produces output which is – in all cases – significantly different from the others. Figure 4.5 with maxSpread-display limited to 1.6, shows the four methods next to each other. No striking difference is recognized between them.

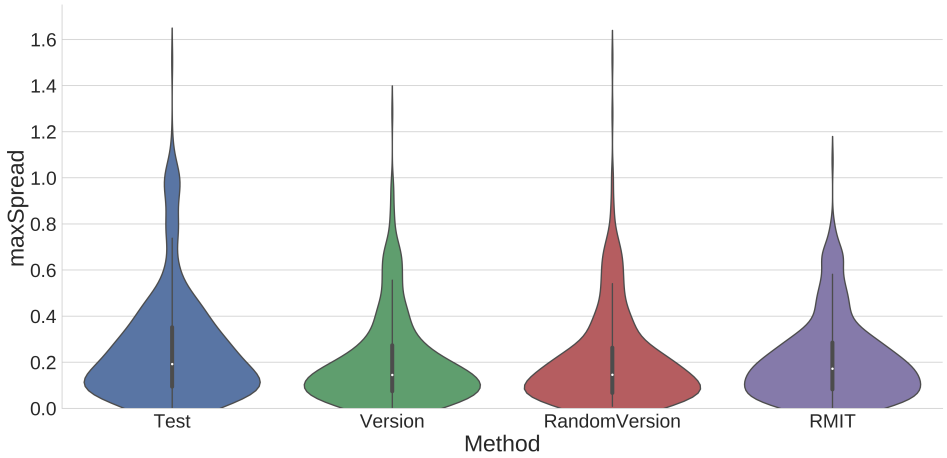

Figure 4.5: Comparison of Distribution-methods over all projects. The *maxSpread*-display is limited to 1.6.

4.3.3 Quality of Methods

For determination of the quality of the methods, this thesis used one-way *Analysis of Variance* (ANOVA) also provided by the Python SciPy-library.[9] ANOVA is another statistical test which compares two or more samples with each other and determines whether the input belongs to the same population. A sample compared by ANOVA consists of the highest maxSpreads for each benchmark within a version. This means that for a project with 16 benchmarks (b1 to b16), and 10 versions, ANOVA compares the following groups:

- **Sample 1:** {maxSpread(b1), maxSpread(b2), ... , maxSpread(b16)} of version 1
- **Sample 2:** {maxSpread(b1), maxSpread(b2), ... , maxSpread(b16)} of version 2
- ...
- **Sample 10:** {maxSpread(b1), maxSpread(b2), ... , maxSpread(b16)} of version 10

Listing 4.1 displays the sample of version *2cfb106* in method Version Range Distribution from project RDF4J. For each benchmark, the listing shows its maxSpread within this version.

```
maxSpread   Benchmark

0.0864411293178  ForwardChainingRDFSInferencerBenchmark.initialize
0.0136667750808  ReasoningBenchmark.forwardChainingSchemaCachingRDFSInferencerSchema
0.0329517240707  ForwardChainingSchemaCachingRDFSInferencerBenchmark.initialize
0.0538099929899  ReasoningBenchmark.forwardChainingSchemaCachingRDFSInferencer
0.204784500179   ReasoningBenchmark.forwardChaining[...]MultipleTransactions
0.114144721003   ReasoningBenchmark.forwardChaining[...]MultipleTransactionsSchema
0.0947683898783  ReasoningBenchmark.forwardChainingRDFSInferencer
0.17166090871    NoReasoningBenchmark.initialize
0.169245298825   ReasoningBenchmark.noReasoning
```

Listing 4.1: ANOVA sample for version *2cfb106* in project RDF4J

In total, there were 10 versions mined per method, so each ANOVA-test was executed with 10 of such samples. Again, the obtained measurement is the p-value and the significance threshold was set at .05. The null hypothesis H_0 assumes that the compared samples belong to the same population, and the examined method produces consistent output over versions. The alternate hypothesis H_1 on the other hand states that the samples are not similar and thus, the method shows inconsistency between versions. The results of the ANOVA-tests are displayed in table 4.7. The p-value gives an indication of the stability of the obtained results: If the p-value lies below .05, the null-hypothesis is rejected in favor of H_1, i.e. the samples differ from each other. Therefore, the results over versions are not stable. Is the p-value bigger, the null-hypothesis is supported and the samples do not statistically differ from each other. The examined method is confirmed to produce measurements which are stable over versions. However, the p-the value itself gives no indication about the *degree* of stability. It can only be said whether the samples are similar or not, and thus, the results consistent over versions or not.

[9] https://docs.scipy.org/doc/scipy-0.19.0/reference/generated/scipy.stats.f_oneway.html

Method	p-value		
	JCTOOLS	protostuff	RDF4J
Random Version Distributon	.036	<.001	<.001
Version Range Distributon	.053	<.001	.308
Test Distributon	.997	.591	.885
RMIT	.936	.812	.901

Table 4.7: The p-values obtained by the ANOVA-tests. Values below .05 imply a rejection of the null-hypothesis and indicate instability over versions.

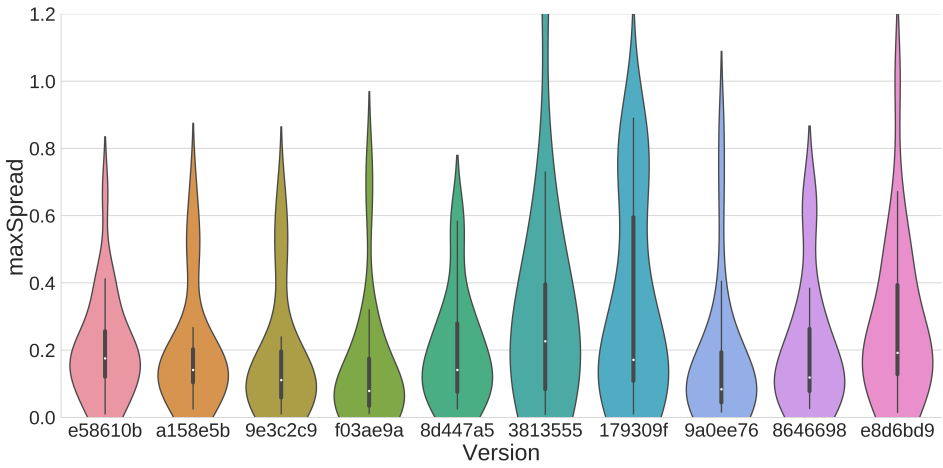

Figure 4.6: JCTOOLS Random Version Distribution with p-value of .036. Values are displayed until *maxSpread* of 1.2.

According to the measurements, it is strongly indicated that Random Version Distribution shows unstable results. For all projects, the p-value lies statistically significant below the threshold and H_0 is rejected. JCTOOLS encounters a p-value of .036, protostuff and RDF4J show a probability of less than .001. Instability is also indicated by Version Distribution in project protostuff. Here, the p-value lies also below .001. In all other ANOVA-tests, H_0 holds. In detail, Test - and RMIT Distribution produce for all projects consistent results over versions. Version Range Distribution obtains for two of the three projects accurate measurements without variation between versions.
Figure 4.6 serves as an illustrating example. Method Random Version Distribution of project JCTOOLS with a p-value of .036 rejects the null hypothesis and therefore encounters instability between versions. This is visible, when comparing the two greenish plots in the middle – version *8d447a5* and *3813555*: Plot *8d447a5* contains benchmarks with a maxSpread of around 0.8 whereas plot *3813555* has notable outliers above the visible threshold.
A more distinct example is given in figure 4.7 where the y-axis is limited from 0.0 to 0.7. Here, there are no distinct outliers but the version-samples were with a p-value smaller than .001 stated as significantly different. Version *da408f7* has for example a maxSpread-range from 0.15 to 0.45 whereas version *2cfb106* covers the range between 0.0 and 0.15.

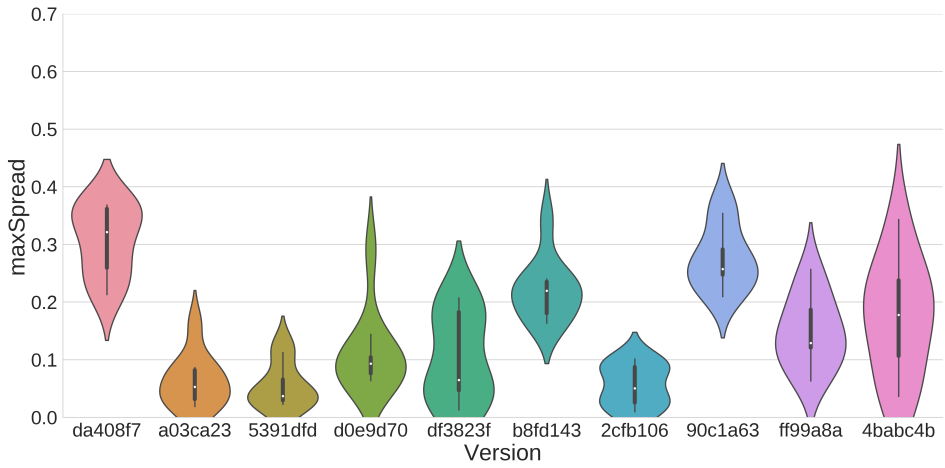

Figure 4.7: RDF4J Random Version Distribution with p-value <.001 which indicates instability over versions. *maxSpread*-values are displayed between 0.0 and 0.7.

Finally, figure 4.8 shows RMIT Distribution in JCTOOLS which lies with a p-value of .936 above the threshold. The null-hypothesis holds and the ANOVA-tests states the method to produce consistent output over versions. The violin plots in the figure confirm this assumption. They are all of equal size and shape. There are no outliers.

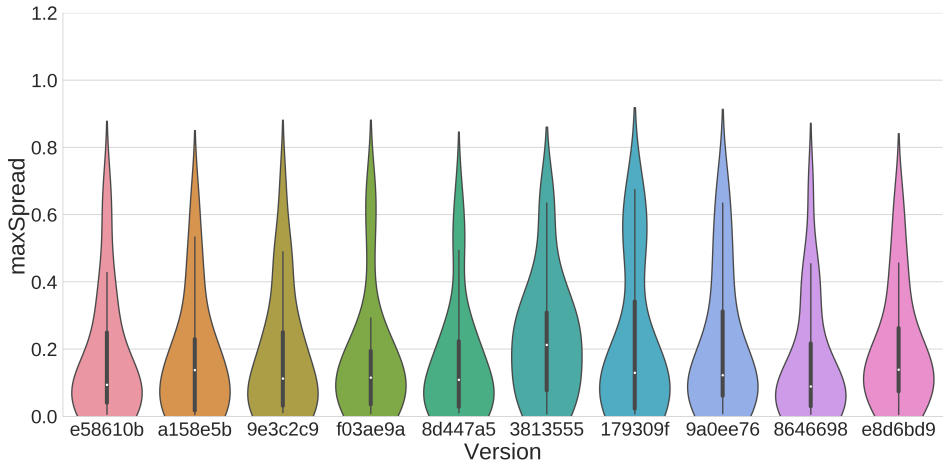

Figure 4.8: JCTOOLS RMIT Distribution with p-value .936. With a p-value above .05, this method produces output which is robust over versions. The y-axis is cut at 1.2.

4.4 Findings

The following section gives a brief recap of the questions examined in this thesis. It concludes with a recommendation for method selection, states potential limitations of the results and suggests future work. Research Question 1 was examined in the previous chapter. It was concerned with the possibilities of dividing a test suite:

RQ 1: *In what ways is it possible to distribute a performance test suite for parallel execution?*

This thesis introduces four methods for division of a performance test suite: Random Version -, Version Range -, Test -, and Randomized Multiple Interleaved Trials (RMIT) Distribution. The first method aims to distribute the work load in a random manner which – if executed in several runs – increases the robustness of the results. For this purpose, from the project to mine, a list of versions is extracted, shuffled and evenly distributed among the computing instances.
The second method – Version Range Distribution – also splits the test suite along the dimension of versions. In contrast to the first approach, the order of versions is kept, and groups of consecutive units are distributed. The aim was to examine, whether preserving the order of versions influences the quality of the measurement.
Distribution by Test is also similar to Random Version Distribution but instead of versions, it randomly distributes the benchmarks of the project. Test Distribution aims to preserve the testing environment during execution since measurements for a benchmark are obtained from only one instance. In Random Version and Version Range Distribution, the benchmark measurements origin from multiple instances.
RMIT Distribution as the fourth method refers to an approach presented by the researchers Abedi and Brecht [AB17]. The aim of this method is to split the test suite along two dimensions (version and test), and randomly generate new pairs which are then distributed among the instances. Likewise, the output should be reliable, robust and repeatable in distributed environments.

RQ 2: *How much time and costs can be saved by executing performance test suites in parallel?*

Parallel execution is always faster than non-parallel execution. Depending on the project size and method, the instances spend different amounts of time in computation. The fastest methods are the ones which distribute versions among instances, i.e. Random Version - and Version Range Distribution. Random Version Distribution is a bit more stable but on average, the two methods are equally fast. In most experiments, the measured durations differ only in a few seconds.
In case, the test suite is splitted by Test Distribution, the experiments run longer since more compilation time is needed for this method. However, the slowest method is RMIT distribution which takes 2 to 3 times longer than Random Version Distribution. Nevertheless, RMIT Distribution experiences the biggest gain factor in comparison of non-parallel with parallel execution: Six instances are over 5 times faster than a single instance. The Version Distribution-methods are 4.78 (Version Range) and 4.93 (Random Version) times faster than a single instance. These methods are roughly 25 % more expensive than non-parallel execution, parallel executed RMIT costs around 15 % more. Test Distribution achieves an average gain of factor 2.49. This method can therefore only halve the time but costs over 140 % more than a single instance.

RQ 3: *Are the results obtained by different distribution techniques similar in terms of quality?*

In comparison of the results between the different versions, RMIT and Test Distribution show for all project p-values above the threshold which is considered as stable. Random Version Distribution lies with p-values .036, <.001, and <.001 for all projects below the significance level of .05. The results between versions for this method strongly vary and are not consistent at all. Version Distribution encounters for project protostuff a value below the threshold, in the other projects, the method performs stable.

After dividing the results into different percentiles, it can be said that 50 % of all obtained measurements for RMIT Distribution show a maxSpread of 0.174, and 99 % one of 0.651. Version Distribution encounters maxSpread measurements of 0.142 for 50 % of the benchmarks and 0.755 for 99 %. There is a relatively big amount of outliers. For Random Version Distribution, maxSpread lies at 0.146 for P50 and 0.690 for P99, respectively. Random Version Distribution also shows an amount of outliers which can be stated as above average. Test Distribution has values of 0.192 for 50 % of the benchmarks and for 99 % 0.831.
None of the examined methods produces output which significantly differs from the other approaches. On average, the results obtained with Version Range Distribution encounter for 99 % of the benchmarks the lowest variability. RMIT distribution however seems to produce measurements closer to each other with less outliers with no statistically confirmed inconsistencies over all versions. In protostuff, Version Range Distribution encounters statistically significant differences between individual versions. Similarly, Random Version Distribution shows a notable but not the highest range of benchmark spread but significant differences between versions for all projects. Finally, Test Distribution is stated to achieve the most deviations in multiple benchmark iterations, but performs stable between the examined versions.

4.4.1 Recommendation

In conclusion, the results suggests to use Version Range Distribution if the aim is to quickly and cheaply achieve benchmark measurements. This method provides output which can vary between versions in some projects but provides on average for 99 % of the benchmarks the lowest maxSpread. Parallel execution is nearly five times faster than experiments on a single instance.
If the aim is on the other hand, to obtain the most robust results, RMIT distribution should be selected. RMIT produces results which are consistent over versions and show the fewest outliers. Nonetheless, it must be noted that RMIT takes twice or even three times the time for execution than e.g. Version Range Distribution. It is therefore the most expensive one.
It is not recommended to use Test Distribution since it produces output which might be stable over versions but has many outliers. Furthermore, Test Distribution can only halve execution time and is the second slowest method. It is also not advisable to choose Random Version Distribution. This method is fast but can show high spreads and provides results of low quality as between different versions a high amount of false-positive regression occurs.

4.4.2 Threats to Validity

Due to the small amount of test data, the statistical expressiveness of the results might be limited. In order to give a more refined statement, more than three projects should be executed over a wider range of benchmarks and versions. The results therefore represent tendencies and should be interpreted as a snapshot.
Furthermore, all experiments have been executed on Google Cloud Engine. It might be possible to obtain different results with another cloud provider.
It is a fact that the choice and design of the distribution methods have relevant influence on the results. There are currently four methods implemented, but other approaches are possible and might be even more appropriate for distributed performance testing. Moreover, the original idea of RMIT was slightly adapted to fit into the context of the distribution algorithm. The method could probably be optimized with further refinement of test suite distribution (e.g. splitting the test iterations) and thus making the trials completely independent from each other. Additionally, it is yet not feasible to identify benchmarks with configurations and count the number of parameters. This information should be taken into account to establish more balanced splits. Test - and

RMIT Distribution potentially suffer from an unbalanced split and worse execution time.
What goes into the same is the composition of the test suites in general: Other projects, benchmarks or versions might result in different output.
Furthermore, the calculation of the overall execution time is limited to a certain extend. For this metric, the time between two status-transmissions was used. Due to potential network latencies, this duration might not exactly correspond to the actual execution time.
Finally, the evaluation of the metrics itself: this bachelor thesis aims to show with the chosen statistical tests a variety of quality attributes. Other tests are possible and might lead to different results.

4.4.3 Future Work

The stated threats to validity motivate some tasks which could be treated in future work. First of all, it would be interesting to analyse the outcomes of other test suites, and such with more than 23 tests and 10 commits to establish more generalizable results. In an extensive evaluation, it might be possible to determine a more precise gain-factor for the implemented methods.
During development, clopper was briefly tested on Amazon EC2 instances. It could therefore be of further interest whether other cloud platforms deliver qualitatively similar results.
Another enhanceable topic of this thesis is the principle of executing performance tests in parallel. As far as my knowledge is correct, there exists no comparable approach which explicitly uses *cloud* instances to execute performance tests.
Furthermore, the implementation itself has potential for future extensions and improvements. The log-file for example, could be extended to provide more transparency about the currently processed unit. Moreover, the state concept could be improved by attaching a time-stamp which enables more precise determination of execution time and thus lowers the influence of network latency. It should also be possible to make the states optional.
As mentioned in chapter 3, there is a troubleshooting-script available but not automatically executed in case of error. This process could therefore be integrated into the existing work flow.
More potential for future work is represented by the distribution algorithms: More efficient ones are likely to exist and other focuses than distribution by versions or tests could be set. In this field, also the recently introduced technique of Randomized Multiple Interleaved Trials should be further examined. It achieved promising results in the experiments and should be tested in different application scenarios.

Chapter 5

Conclusion

Software performance testing is a very important task in the development cycle of applications and services. Regression between versions such as increased response time and lowered throughput can lead to an inappropriate usage of resources, unsatisfied users and eventually, results in a loss of money [CS17]. To make matters worse, performance testing is a tedious process; the test suites take long to execute, but must be repeated several times to obtain expressive results [FJA+10], [HMSZ14], [ABV16]. Additionally, software is changing at a pace which makes it almost impossible to thoroughly test the performance of the whole application before every release [HMSZ14].
This thesis therefore presented a distributed approach which executes a splitted performance test suite in parallel using remote instances. For this purpose, a tool called *clopper* was introduced. clopper is an extension of the framework *hopper* which is designed to mine performance history of software projects [LL17].
In the following, the research questions examined by this thesis are revisited.

RQ 1: *In what ways is it possible to distribute a performance test suite for parallel execution?*

This thesis implemented four different distribution algorithms. As a prerequisite, a collection of JMH-benchmarks over different versions must be available.
The first method *Version Range Distribution* splits the test suite among versions. In detail, it divides the specified series of versions into smaller ranges. The second method *Random Version Distribution* generates random splits of versions. Other than the first method, it does not preserve the order of execution.
Test Distribution is a method which is based on the same procedure, but distributes the different tests of the suite. The last method uses the concept of *Randomized Multiple Interleaved Trials* (RMIT) [AB17]. All possible version-test tuples are created, shuffled and evenly distributed among the instances.
If the available test suite is big enough, it is divided into as many pieces as instances are defined. In case, there is no perfect split possible, some instances will receive a test suite bigger by one version, test or tuple. In case the test suite is not big enough, some instances are released.

RQ 2: *How much time and costs can be saved by executing performance test suites in parallel?*

The logical and confirmed conclusion is that parallel - is always faster than non-parallel execution. However, it is not possible to say that n instances are also n-times faster than a single instance if they process the same workload. For determination of the time saving, the *gain factor* was introduced. The gain factor is the result of dividing non-parallel by parallel execution time. Values bigger than 1.00 correspond to a gain of time, thus, the higher the value, the bigger the gain.
In the conducted experiments, the methods show different factors. Six cloud instances were between 2.49 (Test Distribution) and 5.30 (RMIT) times faster than a single instance with the same

workload. Since the gain factor is smaller than the number of used instances (n), the overall execution costs cannot be reduced: RMIT is with six instances roughly 14.68% more expensive than a single instance, Test Distribution over 140%.

RQ 3: *Are the results obtained by different distribution techniques similar in terms of quality?*

The quality attributes of the methods observed by this thesis are similarity, execution time, overall variability of benchmarks and result consistency over versions.
None of the examined methods produces output which significantly differs from the other approaches. Hence, according to the similarity measurement, no method should be rejected.
RMIT is the slowest but at the same time, the method with the biggest gain factor of 5.30 in comparison from non-parallel to parallel execution. Test Distribution is second slowest and can only halve execution time with a parallel approach. Random Version - and Version Range Distribution are the fastest methods with gain factors of 4.93 and 4.78, respectively.
On average, the results obtained by Version Range Distribution encounter the lowest variability but show for one project statistically significant differences between individual versions. When using Random Version Distribution though, all projects have significant different measurements between versions. Benchmark results also notably vary. The third method, RMIT distribution seems to produce measurements closer to each other with less outliers which are statistically consistent over all versions. Finally, Test Distribution is stated to achieve the most deviations in multiple benchmark iterations but is consistent over the examined versions.

Impact of Findings

Clopper notably lowers the time and eases execution of performance tests. Therefore, it could support developers concluding this cumbersome task in less time. For them, it is recommended to use method Version Range Distribution since this method is very fast. Depending on the project, it produces output which might vary between the mined versions but provides on average benchmarks with a low spread.
The clopper-script is designed for cloud environments but could also be run on a cluster of machines. However, if computation is done by virtual instances, no dedicated machines must be explicitly reserved and resources can be used for other tasks.
Besides reducing execution time, clopper could also help researchers to obtain more robust performance measurements with the application of RMIT distribution. RMIT takes twice or even three times longer than Version Range Distribution but outputs very stable results. For all projects, they are low in variability and consistent over versions.

Bibliography

[AB17] Ali Abedi and Tim Brecht. Conducting repeatable experiments in highly variable cloud computing environments. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 287–292, New York, NY, USA, 2017. ACM.

[ABV16] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. Learning from source code history to identify performance failures. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 37–48, New York, NY, USA, 2016. ACM.

[BMZP14] C. Bezemer, E. Milon, A. Zaidman, and J. Pouwelse. Detecting and analyzing i/o performance regressions. *Journal of Software: Evolution and Process*, 26(12):1193–1212, 2014. JSME-13-0145.R2.

[CH14] Jianhua Cai and Qingchun Hu. Analysis for cloud testing of web application. In *The 2014 2nd International Conference on Systems and Informatics (ICSAI 2014)*, pages 293–297, Nov 2014.

[CS17] Jinfu Chen and Weiyi Shang. An exploratory study of performance regression introducing code changes. ICSME '17, 2017. [Pre-print from an accepted paper of ICSME'17].

[DNM15] Xiaoguang Dai, Boyana Norris, and Allen D. Malony. Autoperf: Workflow support for performance experiments. In *Proceedings of the 2015 Workshop on Challenges in Performance Methods for Software Development*, WOSP '15, pages 11–16, New York, NY, USA, 2015. ACM.

[FJA+10] King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Ying Zou, and Parminder Flora. Mining performance regression testing repositories for automated performance analysis. In *Proceedings of the 2010 10th International Conference on Quality Software*, QSIC '10, pages 32–41, Washington, DC, USA, 2010. IEEE Computer Society.

[FJV+12] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D. Bowers, and Michael M. Swift. More for your money: Exploiting performance heterogeneity in public clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 20:1–20:14, New York, NY, USA, 2012. ACM.

[FP16] Vincenzo Ferme and Cesare Pautasso. Integrating faban with docker for performance benchmarking. In *Proceedings of the 7th ACM/SPEC on International Confer-*

ence on Performance Engineering, ICPE '16, pages 129–130, New York, NY, USA, 2016. ACM.

[GBT11] Jerry Gao, Xiaoying Bai, and Wei-Tek Tsai. Cloud testing- issues, challenges, needs and practices. *Software Engineering : An International Journal (SEIJ)*, 1(1):9–23, September 2011.

[GD13] Deepak Garg and Amitava Datta. Parallel execution of prioritized test cases for regression testing of web applications. In *Proceedings of the Thirty-Sixth Australasian Computer Science Conference - Volume 135*, ACSC '13, pages 61–68, Darlinghurst, Australia, Australia, 2013. Australian Computer Society, Inc.

[Gho14] Sukumar Ghosh. *Distributed Systems: An Algorithmic Approach, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2014.

[GLS11] Joseph Yossi Gil, Keren Lenz, and Yuval Shimron. A microbenchmark case study and lessons learned. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 297–308, New York, NY, USA, 2011. ACM.

[HDS16] Cameron Haight and Federico De Silva. Magic quadrant for application performance monitoring suites. `https://www.gartner.com/doc/reprints?id=1-3M8KIVD&ct=161118&st=sb`, 2016. [Online; accessed 23-June-2017].

[HHF13] Christoph Heger, Jens Happe, and Roozbeh Farahbod. Automated root cause isolation of performance regressions during software development. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 27–38, New York, NY, USA, 2013. ACM.

[HMSZ14] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. Performance regression testing target prioritization via performance risk analysis. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 60–71, New York, NY, USA, 2014. ACM.

[KKSLM14] Vipin Kumar K. S., A. Lallu, and Sheena Mathew. An efficient approach for distributed regression testing of object oriented programs. In *Proceedings of the 2014 International Conference on Interdisciplinary Advances in Applied Computing*, ICONIAAC '14, pages 33:1–33:7, New York, NY, USA, 2014. ACM.

[KWZK16] Johannes Kroß, Felix Willnecker, Thomas Zwickl, and Helmut Krcmar. Pet: Continuous performance evaluation tool. In *Proceedings of the 2Nd International Workshop on Quality-Aware DevOps*, QUDOS 2016, pages 42–43, New York, NY, USA, 2016. ACM.

[LB17] 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*, ICPE '17, pages 373–384, New York, NY, USA, 2017. ACM.

[LC16] Philipp Leitner and Jürgen Cito. Patterns in the chaos; a study of performance variation and predictability in public iaas clouds. *ACM Trans. Internet Technol.*, 16(3):15:1–15:23, April 2016.

[LL17] Christoph Laaber and Philipp Leitner. (h,g)opper: Performance history mining and analysis. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 167–168, New York, NY, USA, 2017. ACM.

[Luo16a] Qi Luo. Automatic performance testing using input-sensitive profiling. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 1139–1141, New York, NY, USA, 2016. ACM.

[Luo16b] Qi Luo. Input-sensitive performance testing. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 1085–1087, New York, NY, USA, 2016. ACM.

[May09] Marissa Mayer. In search of a better, faster, stronger web, 2009. `http://goo.gl/m4fXx`, [Online; accessed 22-June-2017].

[MDH+12] Piyush Mehrotra, Jahed Djomehri, Steve Heistand, Robert Hood, Haoqiang Jin, Arthur Lazanoff, Subhash Saini, and Rupak Biswas. Performance evaluation of amazon ec2 for nasa hpc applications. In *Proceedings of the 3rd Workshop on Scientific Cloud Computing Date*, ScienceCloud '12, pages 41–50, New York, NY, USA, 2012. ACM.

[MFB+07] J. Meier, Carlos Farre, Prashant Bansode, Scott Barber, and Dennis Rea. *Performance Testing Guidance for Web Applications: Patterns & Practices*. Microsoft Press, Redmond, WA, USA, 2007.

[MSWM12] Daniel A. Mayer, Orie Steele, Susanne Wetzel, and Ulrike Meyer. *CaPTIF: Comprehensive Performance TestIng Framework*, pages 55–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[MTHG14] David Maplesden, Ewan Tempero, John Hosking, and John C. Grundy. Performance analysis of object-oriented software. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 662–665, New York, NY, USA, 2014. ACM.

[MTHG16] David Maplesden, Ewan Tempero, John Hosking, and John C. Grundy. A cost/benefit approach to performance analysis. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 15–26, New York, NY, USA, 2016. ACM.

[NAJ+12] Thanh H.D. Nguyen, Bram Adams, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Automated detection of performance regressions using statistical process control techniques. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 299–310, New York, NY, USA, 2012. ACM.

[Nie10] Jakob Nielsen. Website response times, 2010. `https://www.nngroup.com/articles/website-response-times/`, [Online; accessed 22-June-2017].

[RCCB16] Marcelino Rodriguez-Cancio, Benoit Combemale, and Benoit Baudry. Automatic microbenchmark generation to prevent dead code elimination and constant folding. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 132–143, New York, NY, USA, 2016. ACM.

[RDRF06] Ioan Raicu, Catalin Dumitrescu, Matei Ripeanu, and Ian Foster. The design, performance, and use of diperf: An automated distributed performance evaluation framework. *Journal of Grid Computing*, 4(3):287–309, 2006.

[RK16] David Georg Reichelt and Stefan Kühne. Empirical analysis of performance problems on code level. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 117–120, New York, NY, USA, 2016. ACM.

[RKTSR16] Leah Riungu-Kalliosaari, Ossi Taipale, Kari Smolander, and Ita Richardson. Adoption and use of cloud-based testing in practice. *Software Quality Journal*, 24(2):337–364, June 2016.

[SYGM15] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. Comparing and combining test-suite reduction and regression test selection. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 237–247, New York, NY, USA, 2015. ACM.

[VHJG95] John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995.

[ZCTA11] Li Zhang, Yinghui Chen, Fan Tang, and Xiong Ao. Design and implementation of cloud-based performance testing system for web services. In *2011 6th International ICST Conference on Communications and Networking in China (CHINACOM)*, pages 875–880, Aug 2011.

[ZLZY13] Junzan Zhou, Shanping Li, Zhen Zhang, and Zhen Ye. Position paper: Cloud-based performance testing: Issues and challenges. In *Proceedings of the 2013 International Workshop on Hot Topics in Cloud Services*, HotTopiCS '13, pages 55–62, New York, NY, USA, 2013. ACM.