CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

An Extensible Framework for Large-Scale Mining of Performance Microbenchmarks

Master's thesis in Computer Systems and Networks

JOEL ROLLNY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

An Extensible Framework for Large-Scale Mining of Performance Microbenchmarks

JOEL ROLLNY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

An Extensible Framework for Large-Scale Mining of Performance Microbenchmarks

JOEL ROLLNY

Supervisor: Philipp Leitner, Dept. of Computer Science and Engineering
Examiner: Robert Feldt, Dept. of Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2018

An Extensible Framework for Large-Scale Mining of Performance Microbenchmarks

JOEL ROLLNY
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Many performance issues are caused by low-level implementation details, which are also more likely to be the cause of failures in software-intensive systems than erroneous feature implementations. A microbenchmark is a type of performance test that focus on measuring the performance of smaller, yet critical, code sections as opposed to a whole system. It is a complex task to design useful and correct performance microbenchmarks, and the lack of tool support is part of the reason.

A goal of mining software repositories (MSR) research is to motivate the development of tools. To facilitate performing mining studies on a large scale mining frameworks can be used. This study focus on designing and implementing an open-sourced extensible framework for large-scale mining of software performance microbenchmarks. By mining existing microbenchmarks and their evolution researchers and practitioners could gain knowledge valuable for their work on tools facilitating the creation of correct and useful performance microbenchmarks. Thus, the objective is to find out if the created framework can be considered good enough for its purpose.

An action research was conducted where *MicroAnalyzer*, the framework presented in this thesis, was designed and implemented. The study involved eliciting requirements for the framework and creating an architecture allowing mining studies to be highly reproducible. Two mining studies were performed in order to validate the framework.

The results of this study show that *MicroAnalyzer* automates the major parts of the mining process and can be useful for large-scale mining of performance microbenchmarks. Mining studies performed with the framework can be considered highly reproducible.

Keywords: performance, microbenchmarks, mining software repositories, JMH, Go, Java, git.

Acknowledgements

The support and feedback I have gotten from my supervisor at Chalmers University of Technology, Philipp Leitner, have been much appreciated and of great value throughout the writing of this thesis. I would also like to thank my examiner, Robert Feldt, for his feedback.

Joel Rollny, Gothenburg, 12 2018

Contents

List of Figures

List of Tables

1 Introduction

Many performance issues are caused by low-level implementation details, which are also more likely to be the cause of failures in software-intensive systems than erroneous feature implementations [15, 24]. A microbenchmark is a type of performance test that focus on measuring the performance of smaller, yet critical, code sections as opposed to testing a whole system [8]. In this thesis micro implies code on a method level.

The development of microbenchmarks requires expertise [14]. When creating a microbenchmark there are two primary tasks that affects its accuracy. The first is the identification and wrapping of a performance-critical code section into a payload, i.e., an independent program, which is supposed to recreate the execution conditions occurring in the application. The second task is running the payload a sufficient number of times to get an estimate of the execution time.

Wrong conclusions could be drawn from microbenchmarking results due to a number of reasons [16]. The abstraction gap between applications and execution environments may affect a benchmark to produce different, or even contradictory, results on different hardware platforms. In addition, depending on which compiler is being used, various code optimizations could be applied for the same code segment. A compiler may recognize patterns, like dead code, which encourages elimination of parts of the benchmark code, leading to code that runs faster than expected.

There is also a warm-up phase to be aware of when developing microbenchmarks. Sources of influence on measurements need to be recognized and discarded if the goal is to examine sustainable performance [19]. Two sources which influence the warm up are class loading and just-in-time (JIT) compilation. The initial iterations of benchmarked code include dynamic compilation [16]. Later iterations are usually faster since the executed code has been compiled and optimized. Warm-up executions of the benchmarked code could be conducted to reduce the effect of such influences.

Code modifications related to performance are often introduced based on developers assumptions instead of actual performance observations [12]. Such observations could be made if microbenchmarks were implemented. The complexity of designing useful and correct performance microbenchmarks, and in particular the lack of tool support, prevents adoption of microbenchmarking among a wider audience of developers [5, 14].

To enable a wider adoption of microbenchmarking better tools need to be developed, making it less complex to create microbenchmarks. A goal of mining software repositories (MSR) research is to motivate the development of tools [22]. To mine repositories means extracting knowledge from data in software projects. Mining microbenchmarks on a larger scale may allow learning about how developers design their microbenchmarks and what the complex and error-prone parts of that process is. By recognizing such patterns improvements of existing tools could be made and new tools be created. Facilitating the benchmarking process, and meeting the need for tool support, microbenchmarking practices are more likely to be widely adopted.

MSR research is conducted to gain an understanding about empirical aspects of software development [5]. Trends, patterns, and connections can be discovered that may result in suggestions for enhanced software development efficiency, or validation of hypotheses, among other things [2]. Such research is often conducted on projects found on hosting services such as GitHub[1] and Bitbucket[2] because such services contain millions of open-sourced software projects, as well as metadata about the projects' development process [39].

As described by Amann et al. [5] a mining study usually involves several steps such as the collection of raw data, pre-processing of the collected data, and analysis of the pre-processed data. The unstructured form of raw data makes it difficult to analyze, affecting the quality and accuracy of analysis results negatively. Therefore the raw data must be processed into a structured form suitable for analysis. Since pre-processing, as well as analyzing, the data is a time-consuming and complex task, especially when performed at large scale, it needs to be assisted by tools [21]. Even though a number of mining tools exist, these are not always publicly available [30]. And when publicly available they are often difficult to set up and use, restricting reuse of the tools in other mining studies [5]. Also, such software is usually developed specifically for each study in which they are used. Consequently, the scripts and tools available are limited to certain pre-processing tasks and analyses.

Performing mining studies is often a difficult and time-consuming task [40]. Typically only a small number of software repositories are studied due to the challenges of finding, retrieving, pre-processing, and analyzing large amounts of data. It is common that parts of this process is performed manually due to the lack of tool support. A drawback with analyzing a small number of repositories is that the generalizability of the studies are limited [39].

[1]https://github.com/

[2]https://bitbucket.org/

1.1 Purpose of the study

The purpose of this study is to create a mining framework which can be used in MSR studies. In particular mining studies that motivates the development of microbenchmarking tools, even though the framework can be used as a general purpose mining tool.

Performance microbenchmarking has not been studied to a great extent [8, 9]. Common to both studies is that the analyses of the performance tests are done manually. This is because, to the best of my knowledge, there is a lack of tools that automates the mining of microbenchmarks. Therefore, a framework called *MicroAnalyzer*[3] is developed in this thesis. The main goal of the framework is to automate mining tasks in order to simplify the process of making contributions to this research area.

Performing mining studies on microbenchmarking could be beneficial for both researchers and practitioners. Such studies may show what the past microbenchmarking struggles have been, and what the current state of practice is. Knowledge about the obstacles and challenges could be used as a basis for how the future of microbenchmarking tools should be shaped, reducing the barrier of expertise required for microbenchmarking.

This study also intends to investigate to what extent mining studies are reproducible and replicable when performed with *MicroAnalyzer* since this is an important, but lacking, aspect of MSR research. González-Barahona & Robles [1] outline eight elements of interest in the mining process concerning what makes a study reproducible. These elements form the basis for getting an appreciation of the level of reproducibility of studies performed using *MicroAnalyzer*.

1.2 Research questions

MicroAnalyzer is developed in this thesis and in order to validate the framework five research questions are investigated using *MicroAnalyzer*. By answering the research questions the functionality of the framework is demonstrated, as well as its potential and limitations. Thus, the research questions this study intends to answer are:

> **RQ 1:** How many microbenchmarks allow dead code elimination optimizations?
> **RQ 2:** How many microbenchmarks allow constant fold optimizations?
> **RQ 3:** How many microbenchmarks are configured according to published best practices?
> **RQ 4:** How many developers work on microbenchmarks?

[3]https://github.com/MicroAnalyzer/MicroAnalyzer

RQ 5: How often do microbenchmarks change?

The main aim of **RQ1-2** is to confirm that *MicroAnalyzer* can be used for source-code level mining. Dead Code Elimination (DCE) and Constant Folding (CF) are optimizations that the Just-In-Time (JIT) compiler may perform depending on how the microbenchmarks are implemented. Such optimizations can have an effect on microbenchmarking results. DCE and CF was chosen for the research questions because it is considered common that microbenchmark payloads are designed in a way that enables the JIT compiler to perform these two optimizations [14].

RQ3 have the same aim as **RQ1-2** but focus on other parts of the source code, such as annotations instead of expressions and statements. In **RQ3** it is investigated how developers have configured their microbenchmarks since the configuration is important for how viable the measurements are. The results will reflect on how common it may be with unreliable measurements. Performing analyses for **RQ1-3** intends to showcase *MicroAnalyzer* in studies concerning source code mining.

Besides confirming that *MicroAnalyzer* can be used for mining project metadata, the aim of **RQ4-5** is to investigate how common microbenchmarking is among developers, as well as microbenchmark evolution. The results will reflect on the state of microbenchmarking practices. Performing analyses for **RQ4-5** intends to showcase *MicroAnalyzer* in studies concerning project-level mining.

1.3 Outline

The thesis is structured as follows: Chapter 2 introduces microbenchmarking challenges to give the reader an understanding about the background for **RQ1-3**. Chapter 3 discusses related work, followed by Chapter 4 describing the methodology used, as well as the design choices made, for developing *MicroAnalyzer*. The mining studies performed using *MicroAnalyzer*, including data collection and pre-processing, are described in Chapter 5. Discussions about the results, validity threats, and *MicroAnalyzer* in general are given in Chapter 6. Chapter 7 concludes with an outlook on future work.

2

Background

To gain an understanding about performance microbenchmarks the reader is introduced to microbenchmarks and its challenges in sections 2.1 and 2.1.1. Sections 2.1.1.1 and 2.1.1.2 discusses some benchmark designs that enables a compiler to perform the optimizations which constitute **RQ1-2**.

2.1 Performance microbenchmarks

Performance is an important aspect of software systems since it influences the success of the business it is supporting [13, 23]. Some consequences of performance issues, which are more likely to be the cause of failures in software-intensive systems than erroneous feature implementations [15], that arise during production use could be losing customers and revenue [23].

A number of performance test types exist but several of these, such as load testing which validate the performance of the system as a whole, are usually executed in the later stages of development, or even not until after the release [24]. This approach is both expensive and inefficient. In general, the later software issues are discovered the more costly they are to fix. Thus it would be of interest to test the performance of vital parts of a system already in the early stages of development.

The Java platform is widely used for performance-sensitive applications and Java Microbenchmark Harness (JMH[1]) is the dominant microbenchmarking framework used in Java projects on GitHub [8]. JMH handle data collection and microbenchmark execution, and present the measurement results for the user.

2.1.1 Microbenchmarking challenges

There are pitfalls to be aware of when benchmarking Java Virtual Machine (JVM) languages due to JVM features such as JIT compilation, which is when code is compiled during execution rather than prior to execution, and garbage collection

[1]http://openjdk.java.net/projects/code-tools/jmh/

[10, 14]. It is important that warmup iterations are executed on the benchmarked code in order to reach the steady state before doing the actual measurements [16, 19]. The difficulty lies in knowing which number of iterations would be optimal for a specific JIT compiler in order for the warmup to be considered sufficient [16]. A consequence of performing an insufficient warmup is that the initial iterations can be slower than the following iterations, since the later iterations include less compilation, and also because the benchmarked code has already been optimized [10]. Thus in order for the measurements to be correct the developer must be aware of the pitfalls and handle them.

Adding to the complexity of writing correct microbenchmarks are issues such as a benchmark being able to produce contradictory results when executed on different platforms, and that compilers may apply different optimizations to the code [17]. Constant folding and dead code elimination are two examples of such optimizations.

2.1.1.1 Dead code elimination

Code that is considered irrelevant to a program, such as unreachable code or code affecting dead variables, is labeled as dead code and optimized by the JIT compiler [16]. Dead variables are variables that are written to but not read. Removing dead code is an optimization known as dead code elimination (DCE). By performing DCE the running time of programs is reduced since there is less operations to execute.

There is a possibility that code sections may be considered dead under certain circumstances, while not in others. Performing DCE at compile time may pose a problem due to these circumstances not always being known at the time of compilation. Code that would be considered dead in production mode could, for example, be considered the opposite in microbenchmarking mode. Treating live code as dead could result in microbenchmark measurement deviations [14, 16].

A consequence of DCE optimizations is that further optimizations may be allowed since the program structure becomes simplified. An example of when this occur is illustrated in Listing 2.1. The *difference* variable is not used and is thus considered dead code which is why the JIT compiler removes the code on line 5. Doing this removal result in the variables n and i to become dead variables since they are not read from anymore, only written to. They are also eliminated. What happens then is that an empty method is benchmarked. The measurement results would now be misleading from that of measuring the time of declaring two variables, adding them to each other, and assign the sum to a third variable.

The DCE optimization can be avoided by either returning a dead variable from the method, if applicable, which in Listing 2.1 would be the variable *difference*, or by passing the dead variable to a Blackhole via its consume method [14]. Listing 2.2 shows both approaches. A Blackhole is an internal class in JMH and is useful for when there are several dead variables in a method. A dead variable returned from a benchmark method is implicitly consumed by the JMH Blackhole.

```java
@Benchmark
public void subtract() {
    int n = 8;
    int i = 6;
    int difference = n - i; // Result unused
}
```

Listing 2.1: Code example of when performing DCE enable further optimizations resulting in an empty method.

```java
@State(Scope.Thread)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
public class DCE {
    private double e = Math.E;

    @Benchmark
    public void wrong() {
        // This is wrong: result is not used and the
        // entire computation is optimized away.
        Math.exp(e);
    }

    @Benchmark
    public double correct() {
        // This is correct: the result is being used.
        return Math.exp(e);
    }

    @Benchmark
    public void alsoCorrect(Blackhole blackhole) {
        // This is correct: the result is being used.
        blackhole.consume(Math.exp(e));
    }
}
```

Listing 2.2: Code example of a DCE and how to prevent it.

2.1.1.2 Constant folding

When computations can be replaced by constants the JIT compiler may perform an optimization known as constant folding (CF). By performing constant folds all such computations are removed and may result in good performance times being returned from microbenchmarks [14]. These computations are not limited to numbers.

Constant strings and string concatenations are also subject for CF.

A simple example of the CF optimization is illustrated in Listing 2.3. What happens in this example is that the JIT compiler notice that the value of *difference* is based on two constant values 6 and 8, which are both declared inside the method. A constant fold in this case would be that the assignment of integer value 2, as shown in the code comment in Listing 2.3, replaces the code in lines 3, 4 and 5. By not running this code the method may appear to have better performance than it would have if run in production.

```
1 @Benchmark
2 public int subtract() {
3     int n = 8;
4     int i = 6;
5     int difference = n - i;      // int difference = 2;
6 
7     return difference;
8 }
```

Listing 2.3: Code example of a constant folding scenario.

Constant folds can be mitigated by moving variable declarations outside of the method and making them non-final fields of an @State object. These fields should always be read as input and the computation results be based on the input values. Also, the rules for preventing dead code elimination should be followed as well.

```
@State(Scope.Thread)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
public class JMH_ConstantFold {
    private double x = Math.PI;
    private final double y = Math.PI;

    @Benchmark
    public double wrong() {
        // The source is predictable,
        // and computation is foldable.
        return Math.log(Math.PI);
    }

    @Benchmark
    public double alsoWrong() {
        // The source is predictable,
        // and computation is foldable.
        return Math.log(y);
    }

```

```java
    @Benchmark
    public double correct() {
        // The source is not predictable
        // so this is correct.
        return Math.log(x);
    }
}
```

Listing 2.4: Code example of a CF and how to prevent it.

3

Related Work

Published mining studies on software microbenchmarks are summarized in Section 3.1. This is followed by Section 3.2 introducing existing mining tools of which some are used in published mining studies.

3.1 Performance microbenchmarks

Leitner & Bezemer [9] conduct an exploratory study on approximately 100 Java projects from GitHub with the intent to reveal the state of practice of performance testing. They conclude that there seem to be an absence of accepted best practices for performance testing while also being an unpopular task in open source projects. This is based on that the performance tests was created by a small number of developers, the tests being rarely updated, and the tests being only a small part of the test suite. Leitner & Bezemer also conclude that developers appear to struggle implementing performance tests, thus better support should be provided by performance testing frameworks.

Stefan et al., [8] performs a study on Java projects from GitHub using microbenchmarking frameworks. They investigate how common microbenchmarks are compared to functional unit testing. They find out JMH is the dominant microbenchmarking framework and that it seem to be because JMH is considered easier and more useful than the alternatives. It is concluded that microbenchmarks are significantly less common than functional unit tests.

Common to both studies are that several parts of the mining process are performed manually. Leitner & Bezemer [9] use a manual identification process to identify which files contain performance tests in a project. They also study the description on GitHub for the projects in order to classify them. Stefan et al., [8] picks 1000 projects out of approximately 99000 for their study due to manual classification of 99000 projects not being practical.

In another study on microbenchmarks, performed by Laaber & Leitner, the quality of microbenchmark suites is investigated [41]. It is, among other things, tested if the suites discover artificially injected slowdowns. The study objects are five Java projects and five Go projects. They find that the suites have between 16 and 983 microbenchmarks with runtimes ranging from 11 minutes to 8.75 hours. It is shown that the variability of benchmark suites differs depending on the environment. Based on their test setup they conclude that JMH is not a reliable tool for discovering small performance regressions.

3.2 Mining tools

A major contribution of the MSR field is the automation of many of the activities associated with empirical studies in software engineering [20]. The automation permits the repetition of studies on a large number of objects and the ability to verify the generality of many findings. In general, scripts and tools that process and analyze data have been created by researchers in order to automate their studies to a great extent [18, 21]. Those tools implement parts of the method applied in all of their research phases. The number of such tools is low because the effort and time cost involved in creating them is high [5]. Some of the existing mining tools which are not commercially distributed are described in sections 3.2.1 to 3.2.7.

3.2.1 Alitheia Core

Alitheia Core was developed for quality and evolution mining studies on software repositories [33, 34, 35]. Data such as source code, emails, and bug reports is pre-processed into an internal data model used for analyses. Alitheia can parse SVN and Git repositories implemented in Java. A relational database is used to store metadata about pre-processed projects. Access to analysis results is possible both programmatically and through a REST API.

Alitheia has an extensible, service-oriented three-tier architecture. The first tier handles pre-processing of local raw data. The second tier analyzes the pre-processed data. The third tier handles the presentation of results and supports browsing the results via a web interface or through an Eclipse[1] plugin. Alitheia Core can be extended by creating plug-ins as OSGi[2] services. Examples are metric calculating, database schema extending, and data accessing plugins.

A requirement for analyzing a project is a local copy of its source code, mailing list, and bug repository. A user must retrieve this data without the help of Alitheia. When registering the project under study in Alitheia, a pre-processing phase is initiated where the raw data is converted into an internal data model. This model is what the metric plugins work with. Alitheia persists analysis results in either plug-in specific tables or predefined tables.

[1]https://www.eclipse.org/
[2]https://www.osgi.org/

3.2.2 BuCo Reporter

BuCo Reporter[3] is a standalone Java application used to mine both VCSs and bug tracking systems (BTSs) [36]. BuCo can provide reports describing the history of studied repositories by combining information from these sources. The architecture consists of mainly three modules: a VCS module, a BTS module and a reporting module. BuCo Reporter is extensible with connector plugins for BTSs and VCSs, as well as analysis plugins. Examples of analyses that can be performed using BuCo is average lines per revision, and average bug correction time. The analysis results are presented as graphical charts.

Project-level data is pre-processed and stored as a file structure in an XML file. Manual configuration needs to be done when connecting to BTSs and VCSs. It is only possible to connect to and work with one project at a time. If work needs to be done on another project this must be configured between each project. Configuration is also required when generating some of the built-in reports of BuCo. A user of BuCo is recommended to install the BaseX[4] XML database and required to install SourceMonitor[5], which is required for metric generation. Some configuration, such as setting the path to SourceMonitor, is required by a user in order to be able to run BuCo.

3.2.3 OSSMeter

OSSMeter[6] is an extensible cloud-based platform used to monitor and analyze repositories [37]. Users can register and explore quality and activity indicators of repositories. OSSMeter can be extended by creating plug-ins as OSGi services. The platform provides a number of connector plugins responsible for connecting to remote repositories and computing what have changed since recent analyses. This process is done on a daily basis. Existing connectors can be used for SVN and Git repositories. OSSMeter also allow plugins for metrics and issue-tracking systems.

Examples of possible analyses are how actively code is developed, which programming languages are used, or number of open bugs. An analysis result is accessible via a web application which consumes a REST API. This API enables 3rd-party software to access data stored by OSSMeter. Data retrievable from the API consists of project metadata, project metric data, and summaries of metric data. The latter intended for visualization. OSSMeter avoids storing any data locally when possible. Local disk storage is used for temporary data required for the analysis, such as copies of remote software repositories. MongoDB[7] is used to store permanent data.

[3]http://java.uom.gr/buco/
[4]http://basex.org/
[5]http://www.campwoodsw.com/sourcemonitor.html
[6]http://www.ossmeter.org/
[7]https://www.mongodb.com/

3.2.4 Boa

The primary goal of Boa is to perform analyses on the evolution of software repositories on an ultra-large scale [7, 11]. Its three main components are a domain specific language (DSL), a compiler, and a data infrastructure. Queries, written in the DSL, are submitted in Boa's web-based interface. The dataset used consists of Java projects from GitHub and SourceForge, and is updated every two or three months. HBase is used to store all data as HBase allow incremental updates of the existing data. Then flat files are generated from the HBase tables for use in analyses. A user of Boa is restricted to perform analyses on this dataset due to the data infrastructure being closed sourced. It is the infrastructure that is responsible for retrieving and pre-processing repositories.

To perform a mining task using Boa an analysis task must be created in the form of a query. This query is then used as input to the Boa compiler, which is built on the Sizzler compiler. If the query compiles successfully a MapReduce job is created which is executed on a Hadoop cluster. If no error occurs, the resulting output is made available to the user as a text file available for downloading. The cluster makes use of a locally cached copy of the pre-processed repositories when performing the analysis.

3.2.5 SOFAS

SOFAS is based on the idea of software analyses as a service and offer analyses as RESTful web services [3, 4]. As a result interoperability of analysis tools is allowed across platforms and geographical boundaries. These services can be used separately as well as combined to create custom analyses. The analysis results can be retrieved by a user when made available. This approach reduces the need for a user to install and configure external dependencies.

The services supports evolution analyses on CVS, SVN, and Git repositories. SOFAS extract release, commit, and revision information from these repositories. These services require the URL of the repository and valid user credentials in order to work. Examples of possible analyses are who changed which source file, and how many lines have been deleted. SOFAS also has issue tracking services (ITSs) for Bugzilla, Google Code, Trac, and SourceForge. The ITS services extracts problem reports and change requests enabling a user to perform analyses such as the average bug-fixing time and the distribution of bug severity.

3.2.6 LISA

LISA[8] is an open-sourced standalone tool for analyzing software artifacts occurring in repository revisions [44]. The graph framework Signal/Collect[9] is used to load and map the artifacts into a graph representation. Data is read directly from the Git database which means that revisions are not checked out but dealt with in memory. This approach enables parallel parsing of source code. At the moment only Git is supported but LISA can be extended to support other VCSs. Extension points also exist for language parsers and storage methods. At least JRE 8 and sbt[10] version 0.13.12 or higher are required in order to use LISA.

Analyses are performed on a source-code level. In order to execute an analysis a user is required to perform some steps. Projects need to be selected for pre-processing, language mappings created to map the interesting nodes of the parsed data into a graph representation, and the analysis must be formulated. This last step involves creating functions handling graph signals as well as defining a data structure for analysis data. Once an analysis is completed, the analysis results are either persisted in a database or written to a comma-separated value (CSV) file.

3.2.7 BigQuery

Google BigQuery is a cloud-based service that is able to process large datasets fast using the processing power of Google's infrastructure [38, 42]. BigQuery enables SQL-like queries to be executed against terabytes of data in seconds. There are three ways to access the service: a web-based GUI, a REST API, and *bq* which is a CLI. The web-based interface allows browsing, creating tables, running queries, and exporting data. The REST API enables 3rd-party clients making calls to BigQuery.

An analysis task may be performed on multiple publicly available datasets. For example, an analysis task could be performed on a GitHub dataset to find references to StackOverflow[11] questions in the StackOverflow dataset. Loading, reading and exporting data is free but a customer is charged for the information processed. It is free to perform analyses on data up to 1TB per month.

3.3 MSR tool gap

There are a number of characteristics that are desirable for MSR tools to have since their main purpose is to facilitate mining studies. The mining process usually

[8]https://bitbucket.org/sealuzh/lisa
[9]http://uzh.github.io/signal-collect/
[10]https://www.scala-sbt.org/
[11]https://stackoverflow.com/

consists of three main tasks which are retrieving raw data, pre-processing the raw data, and analyzing the pre-processed data [30]. Therefore it is desirable to have these tasks automated as much as possible to provide for efficient software mining. The reproducibility and replicability of mining studies are essential for the validation of the findings [6]. An MSR tool should thus be designed in a way that does not counteract reproducibility and replicability. Another desirable aspect is good performance. For example, if pre-processing and analysis execution times are slow it can become infeasible to perform efficient software mining.

Based on the literature review existing tools have drawbacks and limitations. Users are often expected to do manual work, e.g., collecting raw data, while being restricted to perform a limited number of different analyses. Desirable capabilities of an MSR tool would be automating the mining process while still providing high user control, performing fast pre-processing and analysis tasks on both source code and project-level data, and enabling high reproducibility and replicability of studies. There exists MSR tools having one or several of these characteristics but it does not exist, to my knowledge, one tool that has them all.

Furthermore, MSR tools are usually made for specific purposes thus limiting their usefulness and the different types of potential mining studies [30]. Some tools allow extensions to support more languages, VCSs, and data sources in an attempt to increase their usefulness. To my knowledge no tool exists that specifically focus on performance microbenchmarks.

Boa could be useful in microbenchmark mining studies but the dataset in Boa's infrastructure is limited to Java repositories from SourceForge and GitHub, and does not allow control over the cloning and pre-processing tasks. *MicroAnalyzer* is designed to enable full control over which data sources and repositories to clone and pre-process. This is achieved by using a plug-in architecture and independent modules. As a result performance microbenchmarks from any source written in any language could be used in mining studies.

The execution times of analysis and pre-processing tasks increases with the size of the data used. Boa pre-processes all source code which results in dataset sizes larger than if only microbenchmarks was pre-processed. Using *MicroAnalyzer* it is possible to create datasets containing all source code, but the default behaviour of the framework is to create datasets containing only microbenchmarks. *MicroAnalyzer* automatically identifies files containing microbenchmarks and creates a dataset containing only the source code found in these files. As a result a dataset becomes faster to create as well as analyze.

4

Framework

This chapter describes the methods applied while developing *MicroAnalyzer*, as well as the design and implementation choices of the framework. In Section 4.1 the work flow for developing *MicroAnalyzer* is explained. Architectural choices made to address the most important quality attributes are motivated in Section 4.2, followed by Section 4.3 describing how design and implementation specifics meet important functional requirements of the framework.

4.1 Development methodology

During the initial stage of many software development methods all requirements are solicited and documented [29]. This approach assumes that all requirements can be anticipated early on [28]. Since both technology and business environments often change during a project, even in relatively short projects, methodologies and practices that embrace change were developed [29]. Some of these methods became known as agile methods. Agile development molds process to teams and people, and is about creating and responding to change, as well as reducing the cost of change [27, 28].

Agile approaches recommend short, iterative cycles combined with dynamic prioritization of requirements and feature planning [28]. These iterations should preferably be in the range of two to six weeks, during which adjustments to new information is performed. Thus, the design of the software is an ongoing process, done in smaller chunks, instead of creating all of the design upfront. Also, agile approaches advocates simple solutions so there is less to change and making those changes easier, and continuous testing in order to lower the cost of defect corrections. The development is performed incrementally for faster feedback on the work for which decisions are based upon [27].

Agile methodologies were chosen due to my domain knowledge being very limited at the start of the project. As a consequence early elicited requirements were more likely to change or being removed, and new requirements found, as my domain knowledge increased. Thus, working in an agile fashion allowed me to make design and implementation changes more easily to accommodate requirements changes. An overview of the development process is given in Figure 4.1.

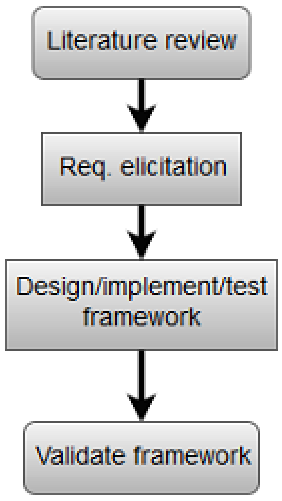


Figure 4.1: *Overview of the development process of MicroAnalyzer.*

Work on the design and implementation of *MicroAnalyzer* was performed iteratively. A couple of requirements were implemented in each iteration, which was ended with a test phase where a dry-run was performed. A subset of the projects under study were tested against the research questions. During this phase my intention was to find what had been missed in the prior design and implementation phases. Smoke tests were primarily used. This type of test focus mainly on assessing if the most significant functionality of a system, or component, work as intended. Findings were then added to the backlog for the next iteration.

4.1.1 Requirements elicitation

Requirements elicitation is concerned with making system developers learn and understand the needs of users and other stakeholders, and is itself a process involving many activities, representing a critical and early but continuous stage in the development of software systems [26]. A substantial part of elicitation involves exploring the problem domain and the requirements that are situated in that domain. The requirements may be spread across many sources, such as documentation, the stakeholders, and other existing systems. All the elicited requirements should be validated against the stakeholders, other systems, each other, and then compared with previously established goals for the system in order to verify that those requirements will provide the necessary functions in order to fulfill the specified objectives of the target system.

Typically the process begins with an informal and incomplete high-level mission statement for the project. This may be represented by a set of fundamental goals, functions, and constraints for the target system, or as an explanation of the problems to be solved [26]. The goal of my requirements elicitation was to get an understanding of the MSR domain and gather requirements for *MicroAnalyzer*.

Two approaches were used for my requirements elicitation. The first was domain analysis. This approach is about examining existing and related applications and documentation in order to capture domain knowledge and gather early requirements [26]. Design documents and published papers describing existing systems used in the MSR domain was reviewed in order to get an understanding of what is required of a system for being efficient in mining software repositories. Then requirements was elicited based on my new understanding of the domain and descriptions of existing systems. After that another approach, called scenarios, was used which describes actions and interactions between a user and a system. Scenarios do not typically consider the internal structure of the system, and require an incremental and interactive approach to their development. It is important when using scenarios to collect all the potential exceptions for each step. Scenarios are useful for understanding and validating requirements, as well as test case development. Some of the elicited requirements are illustrated in Table 4.1.

Table 4.1: *Quality and functional requirements*

Quality Requirements	Functional Requirements
Portability	Clone data repositories
Extensibility	Parse different programming languages
Performance	Pre-process microbenchmarks
Usability	Analyze microbenchmarks
Scalability	Pre-process development history

4.2 Architecture

After performing requirements elicitation it was concluded that the non-functional requirements of most importance for *MicroAnalyzer* were portability, extensibility, performance, usability, and scalability. The resulting component diagram is illustrated in Figure 4.2.

A reason for choosing a plug-in architecture for *MicroAnalyzer* was to facilitate the reproduction of mining studies. Reproducibility is key to validate software mining findings [6]. Studies that can be reproduced can be verified by independent research teams [1]. To be reproducible a study should describe all data, tooling, and configuration settings [5]. According to González-Barahona & Robles [1] a study is reproduced in its whole if its tools, datasets, data sources, and parameters are reused. *MicroAnalyzer* strives to meet these requirements by allowing plug-ins for various data sources, and being run using parameters, among other things.

Mining tools are often limited to run on only specific systems under certain set-

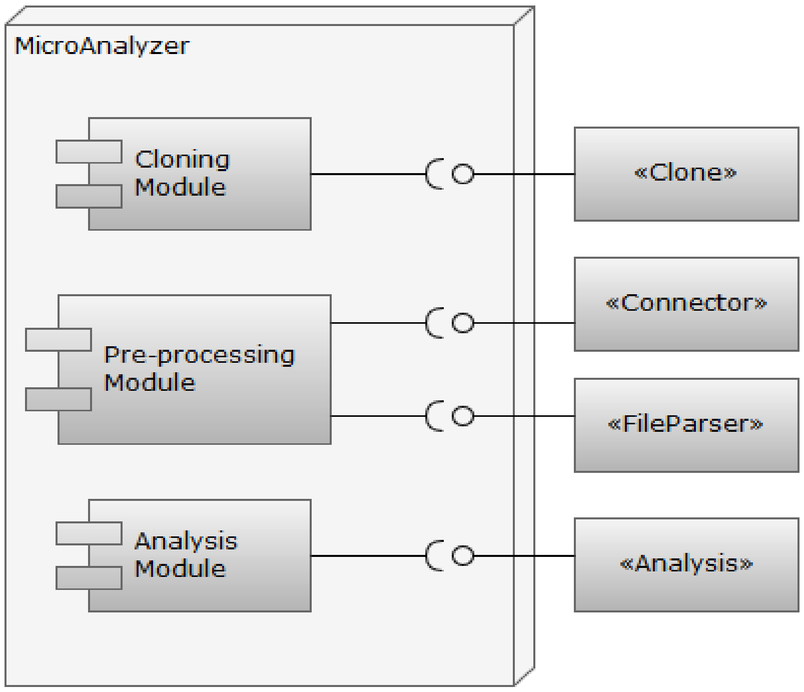


Figure 4.2: *Component diagram illustrating the plug-in architecture of MicroAnalyzer.*

tings [5]. It is also common that these tools are difficult to set up and use [1]. *MicroAnalyzer* should be runnable under any settings on any system, making portability an important quality requirement. Therefore the number of external dependencies was reduced to a minimum. This is also related to the usability requirement because the framework becomes easier to use when no prior installation and configuration of external dependencies are necessary. Portability was a factor why Java was chosen as the implementation language of *MicroAnalyzer* due to its write once, run anywhere (WORA) capabilities. The target system only needs an installation of the Java runtime environment (JRE) in order to be able to use the framework.

Pre-processing software repositories on a large scale may require a significant amount of time depending on the mining tool implementation and the performance of the execution environment. *MicroAnalyzer* is designed with performance in mind. Particularly concerning high throughput of projects, low utilization of computing resources, and short response times for analyses. The framework is also designed to be scalable, that is, capable of pre-processing and analyzing an increasing number of projects without any significant reduction in its performance.

There are many different version control systems (VCS) for software projects, e.g., Git and CVS. Projects can be located on different hosting services, such as GitHub and BitBucket. Projects can be implemented in various programming languages, such as Java and Go. All these possibilities opens up for a number of combinations that could constitute potential mining objects. Therefore it is desirable to make *MicroAnalyzer* extensible so many combinations could be used in mining tasks. Thus, extensibility is considered to be of great importance. Service provider interfaces (SPI) are used to define the interaction between the framework and its plugins.

4.2.1 Presentation layer

One of the requirements for *MicroAnalyzer* was that it should be easy to learn and use. To accomplish this a command line interface (CLI) is used as the frontend, with only a small set of possible parameters. This is in contrast with other mining tools where the learning curve is higher, e.g., Boa where a user must learn Boa's DSL in order to perform analyses on their dataset [7].

Using a CLI for *MicroAnalyzer* increases the framework's portability due to CLIs being more or less mandatory in various operating systems. Boa has a graphical user interface (GUI) as frontend which communicates with Boa's backend located in the cloud [7]. A user of Boa interacts with the GUI via a browser and because the back end is located in the cloud an Internet connection is required. Using a CLI to communicate with a local backend, as is done with *MicroAnalyzer*, can be performed while being offline.

4.2.2 Domain layer

Users of *MicroAnalyzer* are able to perform different mining tasks due to the plug-in architecture of the framework. By creating a plug-in a user can extend the framework to either allow project retrieval from other hosting services, or parse files written in other programming languages, or perform other types of analyses on specific datasets. By making *MicroAnalyzer* extensible there is no need to create new mining tools from scratch for each use case. Since *MicroAnalyzer* is open-sourced anyone can extend the framework with new features, instead of users relying on the developers of the framework to implement desired extensions.

There are three main parts of a mining study. These are data collection, data pre-processing, and data analysis. The domain layer was designed to consist of three independent modules representing these parts. The first module is used to retrieve remote repositories and store them in a local folder named *repositories*. The second module pre-processes the software repositories found in *repositories*. The output from running a pre-processing task using this module is a dataset consisting of structured data. The third module is used to perform analysis tasks on such a dataset. The output from running an analysis task is a text file containing the analysis results.

4.2.2.1 Cloning module

The raw data used in mining studies are typically collected from various remote data sources and stored locally. This is because modifying remote data is usually not permitted without authorized access. Examples of such data sources are GitHub, Google Code[1] and Bitbucket[2].

A cloning module was designed for *MicroAnalyzer* to automate the process of collecting remote data. This module was also designed to be extensible with cloning plugins enabling studies on a diversity of repositories and data sources. A cloning plugin performs the retrieval of remote data and stores it in a local folder.

4.2.2.2 Pre-processing module

When raw data has been stored locally it needs to be transformed into a structured form suitable for analyses. A module was designed for *MicroAnalyzer* to automate the pre-processing of raw data. To meet users need of analyzing projects implemented in various languages *MicroAnalyzer* was designed to be extensible with parser plugins. A parser plugin parses files coded in a certain programming language and converts the parsed code to the framework's internal data model.

To support analyses on the evolution of software projects and files the pre-processing module was also designed to be extensible with connector plugins. A connector plugin is language agnostic and connects to a project's VCS to extract the project's evolution data. Allowing users to extract historical data from various VCSs, such as Git, increases the number of use cases for *MicroAnalyzer*.

4.2.2.3 Analysis module

An analysis module was designed for *MicroAnalyzer* to automate the analysis process. To allow for a wide range of analysis tasks the framework was also designed to be extensible with analysis plugins. By making *MicroAnalyzer* extensible there is less need to create new mining tools for future mining studies. Due to scalability and performance requirements Hadoop[3] MapReduce[4] was chosen to be part of the infrastructure of the analysis module.

[1]https://code.google.com/
[2]https://bitbucket.org/
[3]http://hadoop.apache.org/
[4]https://en.wikipedia.org/wiki/MapReduce

4.2.3 Persistence layer

To increase the portability of *MicroAnalyzer* external dependencies, such as databases, were ruled out. Instead, system-agnostic flat files are used for data storage. The flat files can be copied between different systems, removing the need to install and configure databases. The pre-processing module was designed to create these files and the analysis module designed to use them as input to perform the analyses on. Since the files are used locally the reads and writes can be performed faster than if done over a network.

4.3 Design and Implementation

MicroAnalyzer consists of three modules and a data model. The internal data model is described in Section 4.3.1, followed by a description of the implementation of the framework's modules in Section 4.3.2. How *MicroAnalyzer* identifies files containing microbenchmarks is explained in Section 4.3.3. Section 4.3.4 discusses the table formats chosen for persistence.

4.3.1 Data model

A data model[5] was created for use within *MicroAnalyzer*. Data can be mapped to and from the model. This approach enables the framework to be independent of programming languages due to the use of parser plugins. By parsing and converting data of different programming languages to *MicroAnalyzer*'s data model, the framework can be extended to support any number of languages.

MicroAnalyzer's data model consists of two separate models. One high-level model representing a project, and one low-level representing each changed file in a project's revisions. This separation is done to keep memory usage low when performing analyses. If not, the whole project must fit in the memory, and some projects are extremely large which means that they cannot fit their entire object structure tree in memory at one time. The project object tree is split at the file level by turning each changed file into its own object tree. Once no references exist to such a tree, they are free to be garbage collected. This should solve the problem.

4.3.1.1 Project model

The project model consists of five domain-specific types, illustrated in Table 4.2. The Project type contains metadata about a project, such as which code repositories

[5]https://github.com/MicroAnalyzer/MA-DataModel

that belongs to it and the URL of the project. The reason for making a distinction between a project and a code repository is because some data sources, such as GitHub, makes this separation where several code repositories may belong to one project. A code repository could be imagined as a project's folder [31].

Table 4.2: *Top-level domain-specific types.*

Type	Attributes
Project	id, name, type, repositories, url, created_date, forks, ...
CodeRepository	url, type, revisions
Revision	id, commit_date, committer, files, log
ChangedFile	name, change, type
Person	username, real_name, email

A Project object has references to each of its repositories, and a CodeRepository object in turn provides access to all of the revisions committed into that repository. A Revision type contains information such as the date of the commit and a list of the changed files. Each ChangedFile object references a Person object representing who made changes to that file in that revision. A class diagram illustrating these relationships can be found in Appendix A, and the top-level enumeration types in Appendix B.

4.3.1.2 File model

A file is modeled as a custom abstract syntax tree (AST). Due to being an internal AST of *MicroAnalyzer* this model is applicable for any programming language file mapping. The AST consists of the types found in Table 4.3, and the root of the AST is ASTRoot. For each file one ASTRoot is created. The import statements in a file maps to the *imports* attribute in ASTRoot. Everything between the import keyword and the semicolon is stored as a single string in the *imports* array. A Namespace object contains a file's package name along with any modifiers. Each top-level class, enum or interface is mapped to a Declaration object, where the *type* attribute indicates whether the Declaration was a class, interface, enum, and so on. Since nested classes occur the Declaration object have a list containing nested Declaration objects.

The methods found in a file are mapped into Method objects for the enclosing Declaration. The same applies to fields which are transformed into Variable objects. Statement types represent bodies of code, such as a loop body, or a statement body. These bodies usually contain expressions, if not empty. Expression types correspond to various language-specific details, such as method calls and variable assignments. A model illustrating these relationships can be found in Appendix C, and the enumeration types in Appendix D.

The hierarchical visitor pattern was integrated with the file model due to the pattern enabling hierarchical, as well as conditional, navigation through a file's AST [32].

Table 4.3: *Domain-specific types forming the AST of a changed file in a revision.*

Type	Attributes
ASTRoot	imports, namespaces
Namespace	name, declarations, modifiers
Declaration	name, type, modifiers, fields, methods, parents, ...
Method	name, arguments, return_type, modifiers,...
Statement	type, expressions, condition, statements, updates, ...
Expression	type, literal, method, variable, annotation, ...
Modifier	name, type, visibility, members_and_values, other
Variable	name, type, initializer, modifiers
Type	name, type

This means that when creating an analysis task it is only necessary to navigate through the parts of the AST that is relevant for the analysis. The hierarchical navigation introduces the concept of depth, while the conditional navigation allows skipping branches in the AST. As a result it is both easier to implement analysis tasks as well as enabling better analysis performance, since large parts of the ASTs may be ignored.

4.3.2 Modules

Three independent modules were implemented automating the three main parts of MSR research. Each module support plugins to extend the functionality and number of use cases of *MicroAnalyzer*. Plugins are found automatically by the framework when it boots and scans for plugins in the Java installation's *ext* folder.

4.3.2.1 Cloning module

The cloning module is independent of the pre-processing module because sometimes the repositories under study may already be local to *MicroAnalyzer*. In that case it is unnecessary to copy them from a remote location. The cloned repositories are stored in a local directory named *repositories*, which is created by the framework if it does not already exist. The parameters in Table 4.4 are available for the cloning module. This module is run as shown in Listing 4.1.

Table 4.4: *The available parameters when running the cloning module.*

Parameter	Required	Description
-file	Yes	file containing metadata about repositories to clone
-source	Yes	source of the remote repositories

```
$ java -cp micro-analyzer.jar joelbits.modules.cloning.
```

```
    CloneModule -file <arg> -source <arg>
```

Listing 4.1: Command executed to clone projects from the specified data source. The input file contains metadata about the projects to be cloned.

The *-file* parameter is the name of the JSON file containing metadata about the projects to be cloned. The content in these files, especially the naming of the keys in the key-value pairs, may differ between data sources. GitHub has one set of key names while others, e.g., SourceForge, has their own. Because of this a user of *MicroAnalyzer* must add these name mappings for each data source in an external properties file, in which the framework can look up these mappings when extracting information from the JSON file. The properties file is named *source.mappings* and should be located in the same folder as the framework jar resides in.

The *-source* parameter identifies the data source at which these projects are located. It is used by *MicroAnalyzer* to find any mappings in the properties file matching this source. If such mappings exists, the framework looks up the name of the JSON node containing a project's name and collects all projects' names in a list. This list is used as input to the cloning process. The *-source* parameter also corresponds to the name of the clone plugin and is used by *MicroAnalyzer* to find it. When creating a new clone plugin the class performing the cloning must implement the joelbits.modules.cloning.plugins.spi.Clone interface, as well as override the toString() method with the name of the plugin. Otherwise the framework will not find it.

4.3.2.2 Pre-processing module

This module pre-processes the projects found in the *repositories* directory, located in the same folder as the framework jar is run. The output from running this module is two datasets, described in Section 4.3.4, whose content is adapted to the internal data model of *MicroAnalyzer*. The possible parameters when running this module are displayed in Table 4.5. This module is run as shown in Listing 4.2.

Table 4.5: *The available parameters when running the pre-processing module.*

Parameter	Required	Description
-connector	Yes	used to connect to the repositories VCS
-parser	Yes	parse files coded in specific language
-file	Yes	file containing metadata about repositories to clone
-source	Yes	source of the remote repositories
-dataset	No	will be the name of the created dataset
-all	No	if set all codebase files parsed

```
$ java -cp micro-analyzer.jar joelbits.modules.preprocessing
    .PreProcessorModule -connector <arg> -file <args>
    -parser <arg> -source <arg> -dataset <args> -all
```

Listing 4.2: Command executed to perform pre-processing on projects found in the *repositories* folder.

The same JSON metadata files used as input to the cloning module are used as input to this module. The *-file* parameter corresponds to the name of such a file. The information in these files is extracted the same way as is done in the cloning module. *MicroAnalyzer* looks for mappings in the properties file matching the *-source* parameter. If such mappings exist, the framework looks up the name of JSON nodes containing specific information about a project. This includes number of forks, creation date, and the project URL.

The *-connector* parameter identifies which VCS the projects in the *repositories* folder uses. A VCS contain the development history of a project. *MicroAnalyzer* uses a connector plugin to connect to them and extract evolution data for each repository. The *-connector* parameter corresponds to the name of this plugin. When creating a new connector plugin the class connecting to the VCS must implement the joelbits.modules.preprocessing.plugins.spi.Connector interface, as well as override the toString() method with the name of the plugin.

Projects are implemented in a variety of programming languages. The *-parser* parameter informs *MicroAnalyzer* about the implementation language of interest of the projects in the *repositories* folder. This allows the framework to use the desired parser plugin, which parses files with a specific extension. To add support for another language a new parser plugin must implement the joelbits.modules.preprocessing .plugins.spi.FileParser interface, as well as override the toString() method with the name of the plugin. There is no limitation in which information of the source code that is retained. The implementer of a parser plugin decides what information is mapped from a file to *MicroAnalyzer*'s internal data model.

The *-dataset* parameter is optional. If used, the created dataset will be named after the *-dataset* parameter. If left out, the created dataset will be given a default name. The SequenceFile consisting of project data is named *projects* and the MapFile containing file data is named *files.*

If the *-all* flag is set all source code files, with extensions matching the parser plugin, are parsed. If not set, only files containing microbenchmarks are parsed. The entire revision history of each file is persisted when pre-processing finishes.

4.3.2.3 Analysis module

To make mining studies easier to reproduce and replicate the analysis tasks are made as plugins. Thus it is only necessary to have a copy of the analysis plugin and the dataset used in a study in order to reproduce the study. Replication of studies is essential to compare different mining techniques and their results and can be achieved by, for example, performing an analysis on multiple datasets [3]. *MicroAnalyzer* allows a user to name which datasets the analysis should be performed on. This is achieved by using the optional *-dataset* parameter. The parameter takes a comma-separated list of dataset names. There is no set limit on how many datasets can be combined for an analysis task. The possible parameters when running this module are displayed in Table 4.6. This module is run as shown in Listing 4.3.

Table 4.6: *The available parameters when running the analysis module.*

Parameter	Required	Description
-plugin	Yes	identifies which analysis plugin to use
-analysis	Yes	is the specific analysis to run
-output	Yes	will be the name of the created text file
-dataset	No	names specific dataset(s) subject for analysis

```
$ java -cp micro-analyzer.jar joelbits.modules.analysis.
    AnalysisModule -plugin <arg> -analysis <arg>
    -output <arg> -dataset <args>
```

Listing 4.3: Command executed to perform an analysis on the specified datasets.

The *-plugin* parameter identifies the analysis plugin to use. Each analysis plugin is a MapReduce program for scalability purposes. If run on a distributed cluster a MapReduce program could save time since each map process is executed in parallel. How this applies to *MicroAnalyzer* is illustrated in Appendix E. Each analysis plugin may contain a number of analysis tasks. An analysis task is the actual analysis performed on a dataset. Which analysis task to run is decided by the *-analysis* parameter. When creating a new analysis plugin the joelbits.modules.analysis.plugins.spi.Analysis interface must be implemented, as well as the toString() method overridden with the name of the plugin.

After an analysis task has finished a text file is created by *MicroAnalyzer*. This file contains the analysis results and is named after the *-output* parameter.

4.3.3 Identifying microbenchmark files

The identification of files containing microbenchmarks occur in each file parser. It is the plugin developer who decides how microbenchmarks are identified by implementing the interface method *boolean hasBenchmarks(File file)*. Then a *MicroAnalyzer* pre-processor calls that method on each file when creating a dataset containing only microbenchmark files.

The parser plugin implementation is only relevant when pre-processing raw data. When a microbenchmark dataset has been created analyses can be performed on that dataset regardless of how the microbenchmarks was identified during the pre-processing task. The developer of an analysis plugin decides what data is of interest to analyze. For example, JMH annotates microbenchmarks with @Benchmark and this would be one way of identifying those benchmarks in a dataset.

4.3.4 Persistence

A Hadoop SequenceFile[6] is used for storing the pre-processed projects due to its ease in splitting the input across map tasks. A unique project becomes input to each mapper process. Illustrated in Table 4.7 is the persistence table metadata for those Project objects.

Table 4.7: *The key-value pairs of a Hadoop SequenceFile containing projects. A key is the URL of a project and its value is the binary representation of that Project object.*

Key	**Value**	**Key**	**Value**	**..**	**Key**	**Value**
URL_1	P_1	URL_2	P_2	..	URL_n	P_n

Due to Hadoop MapReduce being part of the analysis module the flat files used for the dataset are of types MapFile[7] and SequenceFile. A Hadoop MapFile is used for storing the file ASTs. The persistence table metadata for the ASTs is illustrated in Table 4.8. When pre-processing is performed a temporary MapFile is created for each project's parsed files to keep memory usage low. After each such file has been written the memory is cleared of their ASTs and next project is pre-processed. When all projects have been processed the temporary datasets are merged into one dataset, which is the output of running the pre-processing module of *MicroAnalyzer*. The temporary files are deleted after the merge. This approach reduces the risk of the Java process running out of memory.

Table 4.8: *The key-value pairs of the SequenceFile found within a Hadoop MapFile, containing file ASTs. A key is the URL of a project combined with a revision ID and a file path, since this combination uniquely identifies a specific version of a file. a value is the binary representation of an AST.*

Key	**Value**	**Key**	**Value**	**..**	**Key**	**Value**
$\text{URL}_1\text{:R}_1\text{:F}_1$	AST_1	$\text{URL}_1\text{:R}_1\text{:F}_2$	AST_2	..	$\text{URL}_n\text{:R}_n\text{:F}_n$	AST_n

Data is stored in its binary form in a SequenceFile. Therefore protocol buffers[8] are used for serialization and deserialization of pre-processed data. Protocol buffers provide fast serialization and deserialization, as well as the serialized data being small in size.

[6]https://hadoop.apache.org/docs/r2.7.3/api/org/apache/hadoop/io/SequenceFile.html
[7]https://hadoop.apache.org/docs/r2.7.3/api/org/apache/hadoop/io/MapFile.html
[8]https://developers.google.com/protocol-buffers/

5

Studies

To demonstrate the potential and limitations of *MicroAnalyzer*, while also validating the framework, two mining studies were performed. A mining study usually consists of the three steps illustrated in Figure 5.1: the collection of raw data, pre-processing of the collected data, and analysis of the pre-processed data.

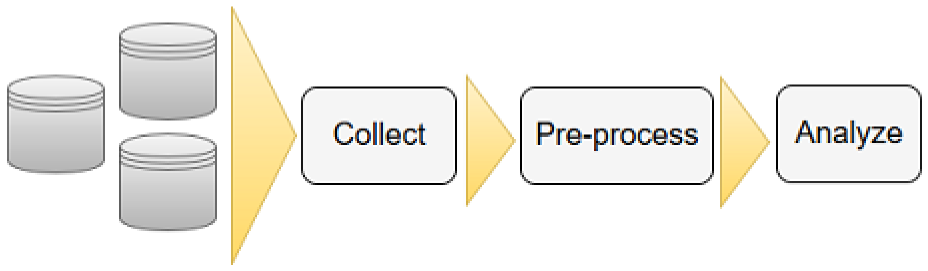


Figure 5.1: *The typical mining study process. First the raw data is collected from a data source, e.g., GitHub. Then the collected data is pre-processed into a structured form which increases the accuracy of the analysis results.*

Using the definition by Amann et al. [5], a mining study usually begins with a research hypothesis and then continue with the pre-processing of the data to be analyzed, to end with an interpretation of the analysis results in order to answer the research questions and hypotheses. The primary reason for doing the mining studies in this thesis was to validate that *MicroAnalyzer* is useful for large-scale mining of software repositories. An overview of how the mining studies were executed is illustrated in Figure 5.2.

The main aim of the mining study described in Section 5.1 is to investigate how common some sources of DCE and CF optimizations are, as well as the state of microbenchmark configurations. Java projects using JMH hosted on GitHub were chosen as study objects because they are the study objects in the study by Rodriguez-Cancio, Combemale, & Baudry [14] where it is stated that developers often design their benchmarks in a way that enables these optimizations. Another reason is because JMH is the most common JVM benchmarking framework and thus allows for a larger number of study objects than the alternatives. This study focus on implementation-level mining which means mining source code ASTs.

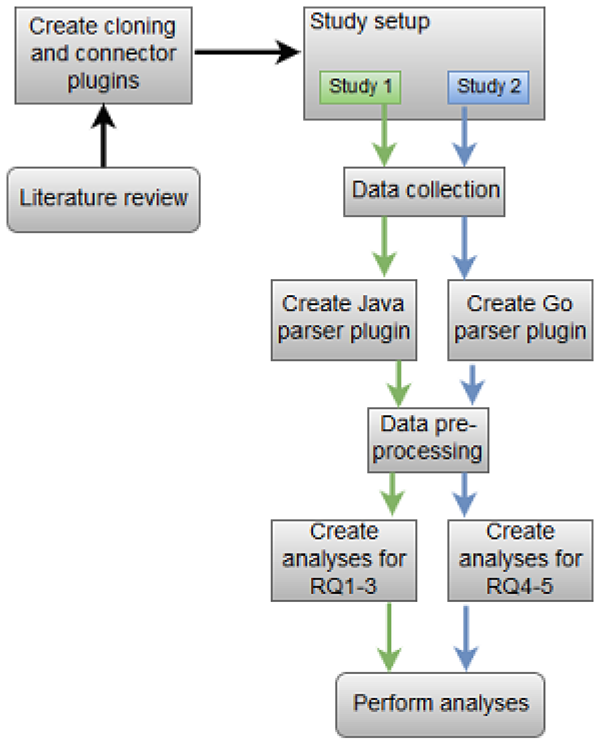


Figure 5.2: *Overview of how the mining studies were carried out.*

The main aim of the mining study described in Section 5.2 is to validate that *MicroAnalyzer* can be used to mine the development history of repositories. This is done by investigating microbenchmarking practices. GitHub repositories implemented in Go containing microbenchmarks was chosen as study objects to showcase the framework plug-in architecture. No external microbenchmarking libraries are of interest since Go includes a testing package for microbenchmarking. This study focus on project-level mining which means mining project related data such as commit information and revision history.

A reason for using GitHub as data source in both studies is due to its vast popularity and millions of open-sourced repositories. Because the hosted repositories are open sourced they are easy to access and use for the study, contrary to closed-source software [2, 25]. Having a larger group of study objects increases the generalizability of the study results [8]. To be able to copy remote repositories from GitHub to a local folder a cloning plugin[1] was created for *MicroAnalyzer*. This local directory is created by the framework and named *repositories*. Since Git is used for all projects on GitHub a Git connector plugin[2] was created to extract the entire development history of the studied repositories.

Measurements were conducted on an Aspire 5741G laptop with an Intel Core i5-

[1]https://github.com/MicroAnalyzer/MA-GitHubCloner
[2]https://github.com/MicroAnalyzer/MA-GitConnector

450M processor, having 2 cores running at 2.4 GHz and 3MB L3 cache. This computer is equipped with 3GB of RAM and had Ubuntu version 16.04.2 LTS and Java version 1.8.0 144 installed. A measurement consists of the time between instantiating the pre-processor and when the data has been persisted.

5.1 Mining study 1: Compiler optimizations and benchmark configurations

To decide if a microbenchmark allows DCE optimizations the implementation of the microbenchmark must meet the following condition: no dead variables in the benchmark body. Other circumstances where dead code occurs, such as unreachable code, is not considered in this study. The analysis is limited to only variables which are unused. The reason for this is that the intention of this study is to showcase *MicroAnalyzer*, not cover every scenario that could result in various optimizations. The analysis approach is to first look through the benchmark body to find any unused declared variables. If such variables are found, the microbenchmark is considered to allow DCE optimizations. To quantify how large part of the microbenchmark population allows DCE optimizations, equation 5.1 is used.

A: All microbenchmarks.
DCE: Microbenchmarks allowing dead code elimination optimizations.

$$\frac{|DCE|}{|A|} \tag{5.1}$$

To decide if a microbenchmark allows CF optimizations the implementation of the microbenchmark must meet the following condition: a final field on a class level of type int, String, or long, is used in a benchmark. These three primitive types were chosen because they are common. Other circumstances allowing constant fold optimizations are not considered in this study. The analysis approach is to first look through the class body for final fields of said types. Then check if any of those fields are used in the benchmark body. To quantify how large part of the microbenchmark population allows CF optimizations, equation 5.2 is used.

A: All microbenchmarks.
CF: Microbenchmarks allowing constant fold optimizations.

$$\frac{|CF|}{|A|} \tag{5.2}$$

The configuration of microbenchmarks is another source affecting measurement results. An analysis task is performed to investigate the current state of microbenchmark configurations. The analysis result is then compared to published best practices [10].

5.1.1 Data collection

BigQuery[3] is a data warehouse containing a large number of datasets, including the data available on GitHub, which can be analyzed interactively. The query in Listing 5.1 was executed on the BigQuery GitHub Java dataset[4] and returned 178 repository names. These repositories had at least one Java file containing the string *import org.openjdk.jmh*, while also having at least three stars, as of February 2017. The reason for choosing this string to identify projects using JMH was due to the fact that the import statement is always present when the JMH library is used. Kalliamvakou et al [2] highlight the fact that many GitHub repositories are personal projects, or copies of other repositories. To reduce the risk of cloning such repositories a repository must have at least three stars.

```sql
SELECT a.repo_name, a.stars
FROM
 [fh-bigquery:github_extracts.2017_repo_stars] a
INNER JOIN
 (SELECT sample_repo_name
  FROM
   [fh-bigquery:github_extracts.contents_java_full_201702]
  WHERE content like '%import␣org.openjdk.jmh%'
  GROUP BY sample_repo_name
 ) AS b
   ON a.repo_name = b.sample_repo_name
   WHERE a.stars > 2
```

Listing 5.1: The query used to retrieve names of GitHub repositories that have at least 3 stars while also containing the 'org.openjdk.jmh' import statement in any source code file.

The Postman[5] software was used to send a GET request to the GitHub API[6] to retrieve the JSON file containing metadata about the 178 repositories. The result of the query in Listing 5.1 were used as URL parameters for the Postman GET request. The response from the API contained metadata about only 166 repositories, which was stored as a JSON file in a local directory. A list of these repositories can be found in Appendix F and statistics about their watchers and stars in Figure 5.3. The reason for the response to contain metadata about fewer repositories could be because the query in Listing 5.1 is performed on a dataset that is more than one

[3]https://bigquery.cloud.google.com
[4]/table/fh-bigquery:github_extracts.contents_java_full_201702
[5]https://www.getpostman.com/
[6]https://api.github.com/search/repositories?q=

year old. The GitHub API interacts with current repositories. Some repositories may have been deleted or moved during the last year.

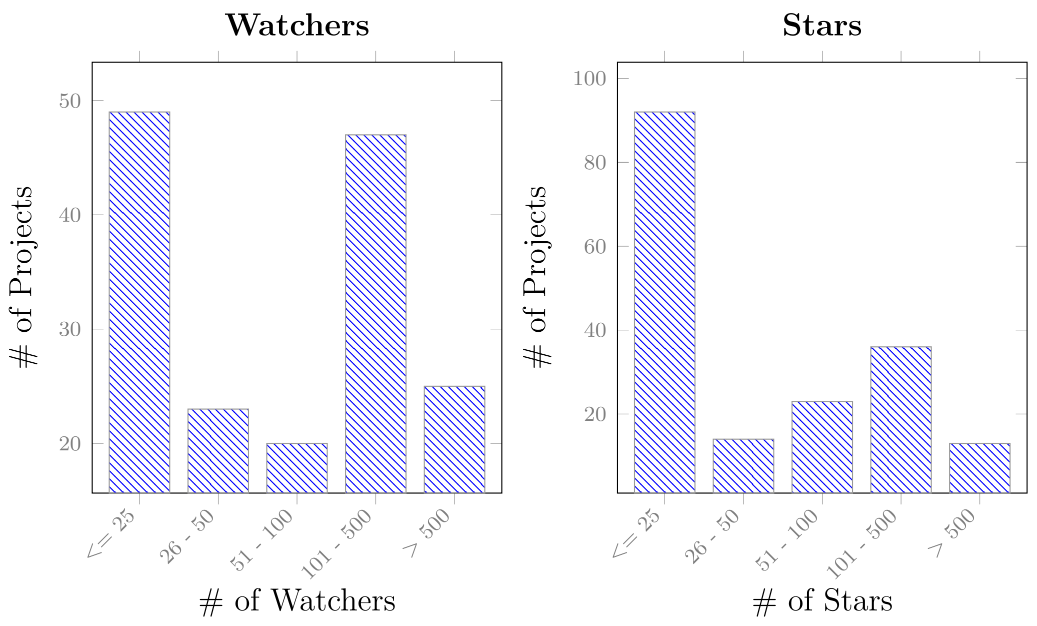


Figure 5.3: *The popularity of the 166 OS Java projects as measured by the number of stars on GitHub and the number of subscribers, known as watchers.*

The repositories named in the downloaded metadata file are hosted on GitHub. The command in Listing 5.2 was executed to retrieve the Java repositories for this study using the clone plugin.

```
$ java -cp micro-analyzer.jar joelbits.modules.cloning.
        CloneModule -file jmh_metadata.json -source github
```

Listing 5.2: Command used to clone Java projects from GitHub.

5.1.2 Pre-processing

A Java parser plugin[7] was created for use in the pre-processing task. The downloaded JSON file containing metadata about Java projects was used as input to the pre-processing module when creating the JMH microbenchmark dataset in Table 5.2. This dataset consists only of files containing microbenchmarks. It is the default behaviour of *MicroAnalyzer* to identify and pre-process only files containing microbenchmarks. Table 5.1 contain statistics about the projects used as input for the pre-processing task. The Java microbenchmark dataset was created by executing the command in Listing 5.3.

[7]https://github.com/MicroAnalyzer/MA-JavaParser

```
$ java −cp micro−analyzer.jar joelbits.modules.preprocessing
        .PreProcessorModule −connector git −file
        jmh_metadata.json −parser java −source github
```

Listing 5.3: Commands executed to perform pre-processing on Java and Go projects.

Table 5.1: *Statistics about the projects used as input for the pre-processing tasks. The size corresponds to the total size of the latest snapshot of all repositories. The number of files corresponds to the total number of files in the most recent snapshot of all repositories.*

Type	*Repositories*	*Size*[*GB*]	*Files*	*Revisions*
Java	166	5.59	121217	211717

Table 5.2: *Statistics about the dataset created for this study. Only files containing microbenchmarks were pre-processed. The duration is the mean value of five measurements performed on the pre-processing execution time.*

Dataset	*Size*[*MB*]	*Parsedfiles*	*Duration*[*s*]	*Duration/file*[*s*]
Java	9.29	6873	3324	0.484

5.1.3 Analyses

By running an analysis plugin *MicroAnalyzer* outputs a text file containing the corresponding analysis results. An analysis plugin[8] was created to answer **RQ1-3**. Within the analysis plugin there are specific analysis tasks created for each research question. All analyses are performed on the Java dataset described in Table 5.2. The analysis tasks for answering the research questions was executed by running the commands in Listing 5.4.

The implementation for the DCE analysis task is given in Listing 5.5. The visitor implementation identifying dead variables is illustrated in Listing 5.6. A visitor is used to traverse ASTs of files in a project. The CF analysis implementation is identical with the DCE analysis except that the visitor implementation identifies class-level final fields used in microbenchmarks. This identification process is shown in Listing 5.7. The analysis task implementation for collecting microbenchmark configurations is illustrated in Listing 5.8 and the visitor implementation identifying the *@Fork*, *@Warmup*, and *@Measurement* annotations is given in Listing 5.9. Class configurations are also extracted since a class-level configuration is used for all microbenchmarks that do not have explicit method-level configurations.

[8]https://github.com/MicroAnalyzer/MA-MicrobenchmarkAnalyses

```
$ java -cp micro-analyzer.jar joelbits.modules.analysis.
    AnalysisModule -plugin thesis -analysis dce
   -output dce.txt
$ java -cp micro-analyzer.jar joelbits.modules.analysis.
    AnalysisModule -plugin thesis -analysis cf
   -output cf.txt
$ java -cp micro-analyzer.jar joelbits.modules.analysis.
    AnalysisModule -plugin thesis -analysis measurement
   -output configurations.txt
```

Listing 5.4: Command used to perform the DCE and CF optimization analyses and the configuration analysis for answering **RQ1-3**.

```java
Set<ASTRoot> benchmarkFiles =
        analysisUtil.latestFileSnapshots(repository);

for (ASTRoot changedFile : benchmarkFiles) {
    if (processedBenchmarkFiles
                .contains(declarationName)) {
        continue;
    }
    processedBenchmarkFiles.add(declarationName);

    visitor.resetAllowDCE();
    visitor.resetBenchmarks();
    changedFile.accept(visitor);
    nrOfBenchmarks += visitor.getNrBenchmarks();
    allowsDCE += visitor.getAllowDCE();
}
```

Listing 5.5: The DCE analysis implementation for answering **RQ1**.

```java
/**
 * Look for dead variables.
 */
private void allowsDCE() {
    List<String> variables =
            new ArrayList<>(variableDeclarations);

    for (String expression : expressions) {
        for (String declaration : variableDeclarations) {
            if (expression.contains(declaration)) {
                variables.remove(declaration);
            }
        }
    }
    if (!variables.isEmpty()) {
        allowDCE++;
    }
}
```

Listing 5.6: The visitor implementation for identifying dead variables.

```java
/**
 * Look for potential CF optimizations.
 */
private void allowsCF() {
    for (String expression : expressions) {
        for (String declaration : finalFields) {
            if (expression.contains(declaration)) {
                allowCF++;
                return;
            }
        }
    }
}
```

Listing 5.7: The visitor implementation for identifying top-level final fields used in benchmarks.

```java
Set<ASTRoot> benchmarkFiles =
        analysisUtil.latestFileSnapshots(repository);

for (ASTRoot changedFile : benchmarkFiles) {
    if (processedBenchmarkFiles
                    .contains(declarationName)) {
        continue;
    }
    processedBenchmarkFiles.add(declarationName);

    BenchmarkMeasurementVisitor visitor =
            new BenchmarkMeasurementVisitor();
    changedFile.accept(visitor);
    writeConfigs(context, declarationName, visitor);
}
```

Listing 5.8: The configuration analysis implementation for **RQ3**.

```java
private void extractBenchmarkConfig(Method method) {
    if (method.getModifiers().stream()
                .anyMatch(this::isBenchmark)) {
        benchmarkConfigurations.
            put(method.getName(), new HashMap<>());
        for (Modifier modifier : method.getModifiers()) {
            if (isConfiguration(modifier
                .getName()) || isBenchmark(modifier)) {
                    benchmarkConfigurations.
                        get(method.getName())
                    .put(modifier.getName(), modifier
                        .getMembersAndValues());
            }
        }
    }
}

private boolean isBenchmark(Modifier modifier) {
    return modifier.getName().equals("Benchmark") &&
        modifier.getType().equals(ModifierType
                                    .ANNOTATION);
}

private void extractClassConfig(Declaration declaration) {
    for (Modifier modifier : declaration.getModifiers()) {
        if (modifier.getType().equals(ModifierType
                .ANNOTATION) && isConfiguration(modifier
                        .getName())) {
            classConfigurations.put(modifier.getName(),
                modifier.getMembersAndValues());
        }
    }
}

private boolean isConfiguration(String configuration) {
    return configuration.equals("Warmup") ||
        configuration.equals("Measurement") ||
            configuration.equals("Fork");
}
```

Listing 5.9: The visitor implementation for identifying benchmarks and their configurations.

5.1.4 Results

To consider the JIT compiler to be allowed to perform a DCE optimization a microbenchmark body had to contain at least one dead variable. Using equation 5.1 the answer to **RQ1** was that 5.87% of the microbenchmarks allow DCE optimizations.

To consider the JIT compiler to be allowed to perform a CF optimization a microbenchmark body had to access a class-level final field of type *int*, *String*, or *long*. Using equation 5.2 the answer to **RQ2** was that 2.53% of the microbenchmarks allow CF optimizations.

Other scenarios enabling these optimizations were not considered in this study. The total number of microbenchmarks allowing these optimizations is given in Table 5.3

Table 5.3: *The first column contains the total number of microbenchmarks analyzed. Only microbenchmarks in the most recent snapshots of the repositories were analyzed.*

Microbenchmarks	*DCE*	*CF*
3320	195	84

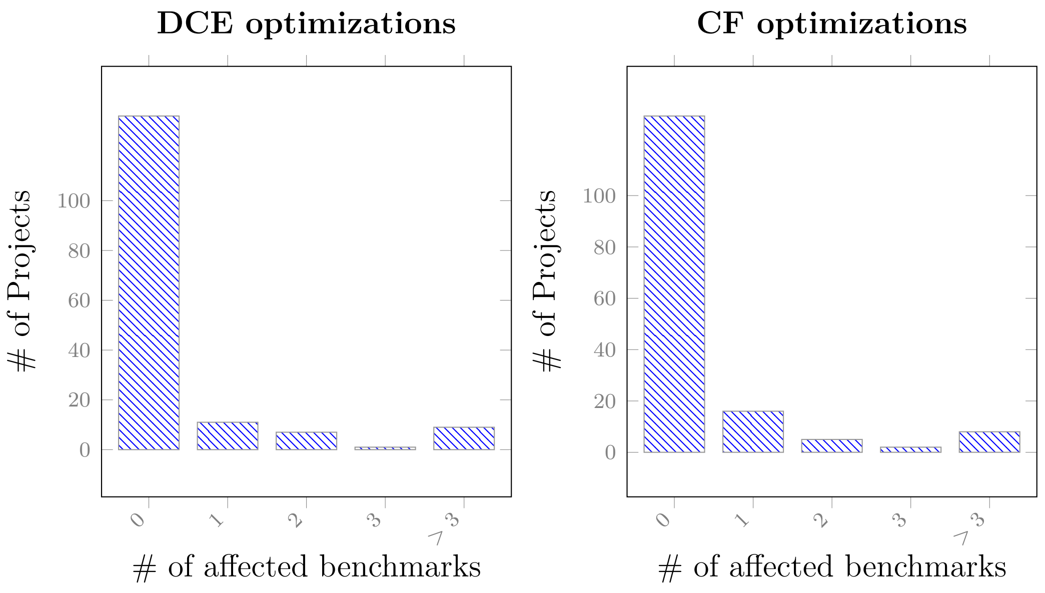


Figure 5.4: *The left diagram illustrates the number of projects that contains microbenchmarks allowing DCE optimizations. The right diagram illustrates the number of projects that contains microbenchmarks allowing CF optimizations.*

The analysis results for **RQ1** and **RQ2** can be found in Appendix G and Appendix H respectively. Only a part of the analysis results are added to the appendices to keep the number of pages down.

As can be seen in Figure 5.4 most projects do not contain microbenchmarks enabling DCE and CF optimizations when considering only the two described cases. In Table 5.4 the ten repositories with the largest number of microbenchmarks allowing DCE optimizations are presented. Out of the 166 repositories 28 contained at least one microbenchmark enabling a DCE optimization. This means that approximately 17% of the studied repositories are affected. Six of the repositories having a benchmark suite of between one and six benchmarks had one benchmark allowing a DCE optimization. In Table 5.5 the ten repositories with the largest number of microbenchmarks allowing CF optimizations are presented. Out of the 166 repositories 31 contained at least one microbenchmark enabling a CF optimization. This means that approximately 19% of the studied repositories are affected. Four of the repositories having a benchmark suite of between one and three benchmarks had one benchmark allowing a CF optimization.

Table 5.4: *The DCE column represents the number of microbenchmarks allowing DCE optimizations. The last column represents the total number of microbenchmark in each repository. Only microbenchmarks in the most recent snapshots of the repositories were analyzed.*

Repository	*DCE*	*Benchmarks*
goldmansachs/gs − collections	52	451
druid − io/druid	41	145
SquidPony/SquidLib	27	143
akarnokd/akarnokd − misc	18	380
udoprog/tiny − async − java	6	16
ReactiveX/RxJava	5	100
apache/logging − log4j2	5	333
JCTools/JCTools	4	92
netty/netty	4	159
arnaudroger/SimpleFlatMapper	3	43

Five repositories are in the top ten for both CF and DCE optimizations. The *goldmansachs/gs-collections* project has most benchmarks allowing both DCE and CF optimizations. The optimizations are mainly centralized to a few projects while CF optimizations have a more even spread in the projects. Overall most projects do not allow any of these specific cases of DCE and CF optimizations, but since many cases are ignored in this study it can be assumed that DCE and CF optimizations are more frequent than these results suggest.

The analysis results for **RQ3**, how many microbenchmarks that are configured according to published best practices, can be found in Appendix I where all configuration combinations occurring less than fifteen times have been filtered out in order to get rid of outliers and noise. Some combinations are configured the same in respect to number of iterations of @Warmup, @Measurement, and @Fork, but are stated as separate combinations in the appendix due to other parameters and time units used. These combinations are considered to be equal since it is the number of iterations that is of interest.

Table 5.5: *The CF column represents the number of microbenchmarks allowing CF optimizations. The last column represents the total number of microbenchmark in each repository. Only microbenchmarks in the most recent snapshots of the repositories were analyzed.*

Repository	*CF*	*Benchmarks*
goldmansachs/gs − collections	12	451
JCTools/JCTools	7	92
apache/logging − log4j2	7	333
udoprog/tiny − async − java	6	16
cache2k/cache2k − benchmark	6	28
netty/netty	6	159
ben − manes/caffeine	4	37
eclipse/eclipselink.runtime	4	119
graalvm/graal − core	3	60
JetBrains/xodus	3	66

There were 3938 microbenchmark configurations in the analysis result. 34 configurations were excluded since they contained variable values declared outside of the annotations. Left in Appendix I are 3407 microbenchmark configurations which means that 531 configurations were filtered out. An empty *@Benchmark()* annotation means that a microbenchmark has no explicit configuration, and thus the default configuration is used. This configuration was at the time of writing ten forks, twenty measurement iterations, and twenty warmup iterations.

Table 5.6: *The number of microbenchmarks for the eight most common configurations. All of them use default time settings which is one second per warmup iteration and one second per measurement iteration.*

Config / #	Warmups	Measurements	Forks
2074	20	20	10
713	5	5	1
81	10	10	1
65	5	10	1
64	10	25	1
58	10	5	10
58	10	10	10
44	3	3	1

As can be seen in Table 5.6 2074 microbenchmarks has a default configuration which corresponds to 60.87% of the microbenchmarks in Appendix I. The second most common configuration occurs on 713 microbenchmarks which is 20.93% of all analyzed microbenchmarks.

5.2 Mining study 2: Microbenchmarking practices

Creating useful and correct performance microbenchmarks is complex and requires expertise [14]. Studying the current state of microbenchmarking practices could give an understanding about aspects such as what the most common obstacles are. To quantify how large part of the studied developer population has worked on microbenchmarks equation 5.3 is used. A developer is labeled a benchmarker if the person have committed changes to any file containing microbenchmarks. The magnitude of the work is not taken into consideration.

A: All developers in the studied projects.
M: Developers having done commits affecting microbenchmark files in their projects.

$$\frac{|M|}{|A|} \tag{5.3}$$

As part of demonstrating the functionality of *MicroAnalyzer* the analysis task for retrieving the values |M| and |A| in equation 5.3 is performed three times on three different datasets. The first analysis is performed on a dataset consisting of repositories implemented in the Go programming language. This is a dataset created specifically for this second mining study to showcase the plugin architecture. The second analysis task is performed on the Java dataset created in the first mining study. Then the analyses results are compared to see if there is any difference between Go and Java projects regarding number of benchmarkers. To showcase that *MicroAnalyzer* can be used to perform analyses on a varying number of datasets an analysis task is performed on both the Go and Java datasets at the same time.

It is also investigated how many microbenchmarks the repositories under study have, and how often these are changed. If a microbenchmark is created but not changed afterwards it is considered to have been modified zero times. This analysis task is done on the Java dataset.

5.2.1 Data collection

The query in Listing 5.10 was executed on the BigQuery GitHub Go dataset[9] and returned 217 repository names. These repositories have at least one Go file containing the string **testing.B*. When this string is a parameter of a Go method it represents a microbenchmark. The reason for choosing repositories with more than twenty stars was to keep the number of projects down, since 217 projects is enough for the demonstrative purposes of this thesis.

[9]fh-bigquery:github_extracts.contents_go

```sql
SELECT a.repo_name, a.stars
FROM
 [fh-bigquery:github_extracts.2017_repo_stars] a
INNER JOIN
  (SELECT sample_repo_name
  FROM
   [fh-bigquery:github_extracts.contents_go]
  WHERE content like '%*testing.B%'
  GROUP BY sample_repo_name
) AS b
  ON a.repo_name = b.sample_repo_name
  WHERE a.stars > 20
```

Listing 5.10: The query used to retrieve names of GitHub repositories that have at least 20 stars while also containing the '*testing.B' string in any Go source code file.

Postman was used to send a GET request to the GitHub API to retrieve a JSON file containing metadata about the 217 Go repositories. The result of the query in Listing 5.10 was used as URL parameters for the Postman GET request. The API response, which was stored as a JSON file in a local directory, contained metadata about only 202 repositories. A list of these repositories can be found in Appendix J and statistics about their watchers and stars in Figure 5.5.

The repositories named in the downloaded metadata file are hosted on GitHub. The clone plugin used for the Java projects was also used to clone these Go projects. The command in Listing 5.11 was executed to copy the remote repositories from GitHub to the *repositories* folder.

5.2.2 Pre-processing

A Go parser plugin[10] was created for use when pre-processing the Go repositories. The downloaded JSON file containing metadata about Go projects was used as input when running the pre-processing module to create the Go microbenchmark dataset in Table 5.8. Table 5.7 contain statistics about the projects used as input to the pre-processing task. The Go microbenchmark dataset was created by executing the command in Listing 5.12. This dataset consists only of files containing microbenchmarks. It is the default behaviour of *MicroAnalyzer* to identify and pre-process only files containing microbenchmarks. In one of the five pre-processing tasks the command in Listing 5.13 was used to give the created dataset a name different than the default so that the dataset could be used for analyses together with the Java dataset created in the first mining study.

[10]https://github.com/MicroAnalyzer/MA-GolangParser

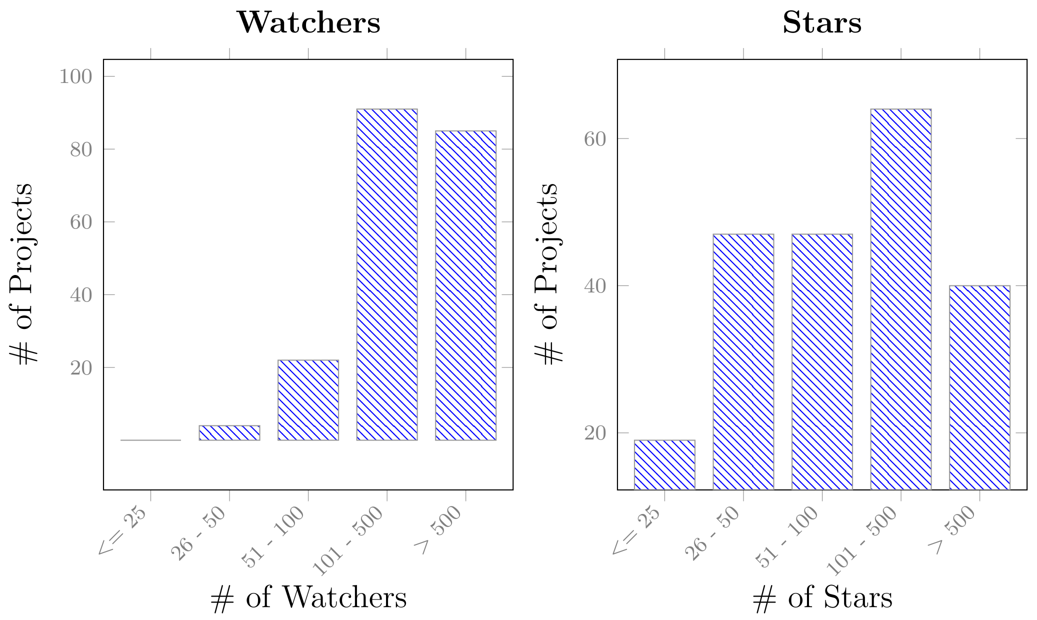


Figure 5.5: *The popularity of the 202 OS Go projects as measured by the number of stars on GitHub and the number of subscribers, known as watchers.*

```
$ java −cp micro−analyzer.jar joelbits.modules.cloning.
      CloneModule −file go_metadata.json −source github
```

Listing 5.11: Command used to clone Go projects from GitHub.

```
$ java −cp micro−analyzer.jar joelbits.modules.preprocessing
      .PreProcessorModule −connector git −file
      golang_metadata.json −parser golang −source github
```

Listing 5.12: Command executed to perform pre-processing on Go projects.

```
$ java −cp micro−analyzer.jar joelbits.modules.preprocessing
      .PreProcessorModule −connector git −file
      go_metadata.json −parser go −source github
      −dataset goDataset
```

Listing 5.13: Command executed to perform pre-processing on Go projects. The created dataset is given a name different than the default. This way the dataset can be used together with other datasets in analysis tasks.

Table 5.7: *Statistics about the projects used as input for the pre-processing task. The size corresponds to the total size in the latest snapshot of all repositories. The number of files corresponds to the total number of files in the most recent snapshot of all repositories.*

Type	*Repositories*	*Size*[*GB*]	*Files*	*Revisions*
Go	202	9.57	128254	309697

Table 5.8: *Statistics about the dataset created for the study. Only files containing microbenchmarks were pre-processed. The duration is the mean time of five pre-processing executions.*

Dataset	*Size*[*MB*]	*Parsedfiles*	*Duration*[*s*]	*Duration/file*[*s*]
Go	11.8	13473	18561.6	1.378

5.2.3 Analyses

Analysis tasks was created for the analysis plugin to answer **RQ4-5**. The Go dataset used in the analyses is described in Table 5.8. The analysis tasks for answering **RQ4-5** were executed by running the commands found in Listing 5.14.

```
$ java -cp micro-analyzer.jar joelbits.modules.analysis.
    AnalysisModule -plugin thesis -analysis benchmarkers
    -output benchmarkers.txt
$ java -cp micro-analyzer.jar joelbits.modules.analysis.
    AnalysisModule -plugin thesis -analysis evolution
    -output evolution.txt
```

Listing 5.14: Commands used to perform analyses with *MicroAnalyzer*.

The analysis task code for retrieving the number of developers that has done some work on microbenchmarks is given in Figure 5.16. Figure 5.17 illustrates the analysis task implementation for answering **RQ5**.

```
$ java -cp micro-analyzer.jar joelbits.modules.analysis.
    AnalysisModule -plugin thesis -analysis benchmarkers
    -output benchmarkers.txt -dataset javaDataset
$ java -cp micro-analyzer.jar joelbits.modules.analysis.
    AnalysisModule -plugin thesis -analysis benchmarkers
    -output benchmarkers.txt -dataset javaDataset,goDataset
```

Listing 5.15: Commands used to perform analyses with *MicroAnalyzer*. The first command performs the analysis task on the Java dataset created in the first mining study. The second command performs the analysis task on both datasets combined.

```java
List<Revision> revisions = repository.getRevisions();
for (Revision revision : revisions) {
    String committer = revision.getCommitter()
                                .getUsername();
    context.write(new Text("committer"),
                            new Text(committer));
    if (!revision.getFiles().isEmpty()) {
        context.write(new Text("benchmarker"),
                            new Text(committer));
    }
}
```

Listing 5.16: The implementation for recognizing how many of the developers have done any work on microbenchmarks.

```java
BenchmarkCountVisitor visitor =
                    new BenchmarkCountVisitor();
for (Revision revision : repository.getRevisions()) {
    List<ASTRoot> benchmarkFiles = analysisUtil
                .allChangedFiles(revision, repository
                                .getUrl());
    for (ASTRoot changedFile : benchmarkFiles) {
        changedFile.accept(visitor);
    }
}
```

Listing 5.17: The implementation for investigating the evolution of microbenchmarks.

5.2.4 Results

The result of performing the first command in Listing 5.14 is given in Table 5.9. Using equation 5.3 the answer to **RQ4** is that 9.82% of the Go developers have performed work affecting microbenchmark files in their projects. This corresponds to a mean value of 2.39 benchmarkers per repository of the 202 Go repositories in the dataset. This number is probably a bit higher than the real number because microbenchmark files can be affected for various reasons even though no real work has been done on the benchmarks. For example, files could be affected if a developer changes the namespace.

Table 5.9: *The total number of developers for all repositories in the Go dataset. Developers that have committed changed affecting microbenchmark files in some way are labeled benchmarkers.*

Dataset	*Developers*	*Benchmarkers*
Go	4886	482

The result of performing the first command in Listing 5.15 is given in Table 5.10. A part of the analysis result is shown in Appendix K to give the reader an idea about how the analysis results look like. Using equation 5.3 the answer to **RQ4** is that 15.09% of the Java developers have performed work affecting microbenchmark files in their projects. This corresponds to a mean value of 1.95 benchmarkers per repository of the 166 Java repositories in the dataset.

Table 5.10: *The total number of developers for all repositories in the Java dataset. Developers that have committed changed affecting microbenchmark files in some way are labeled benchmarkers.*

Dataset	*Developers*	*Benchmarkers*
Java	2140	323

The result of performing the second command in Listing 5.15 is given in Table 5.11. Using equation 5.3 the answer to **RQ4** is that 11.52% of the developers have performed work affecting microbenchmark files in their projects. This corresponds to a mean value of 1.95 benchmarkers per repository of the 368 repositories in the combined datasets. Adding the results in tables 5.9 and 5.10 give 7026 developers and 805 benchmarkers which differs three benchmarkers and 65 developers compared to the result given in Table 5.11.

Table 5.11: *The total number of developers for all repositories in the combined dataset. Developers that have committed changed affecting microbenchmark files in some way are labeled benchmarkers.*

Dataset	*Developers*	*Benchmarkers*
Both	6961	802

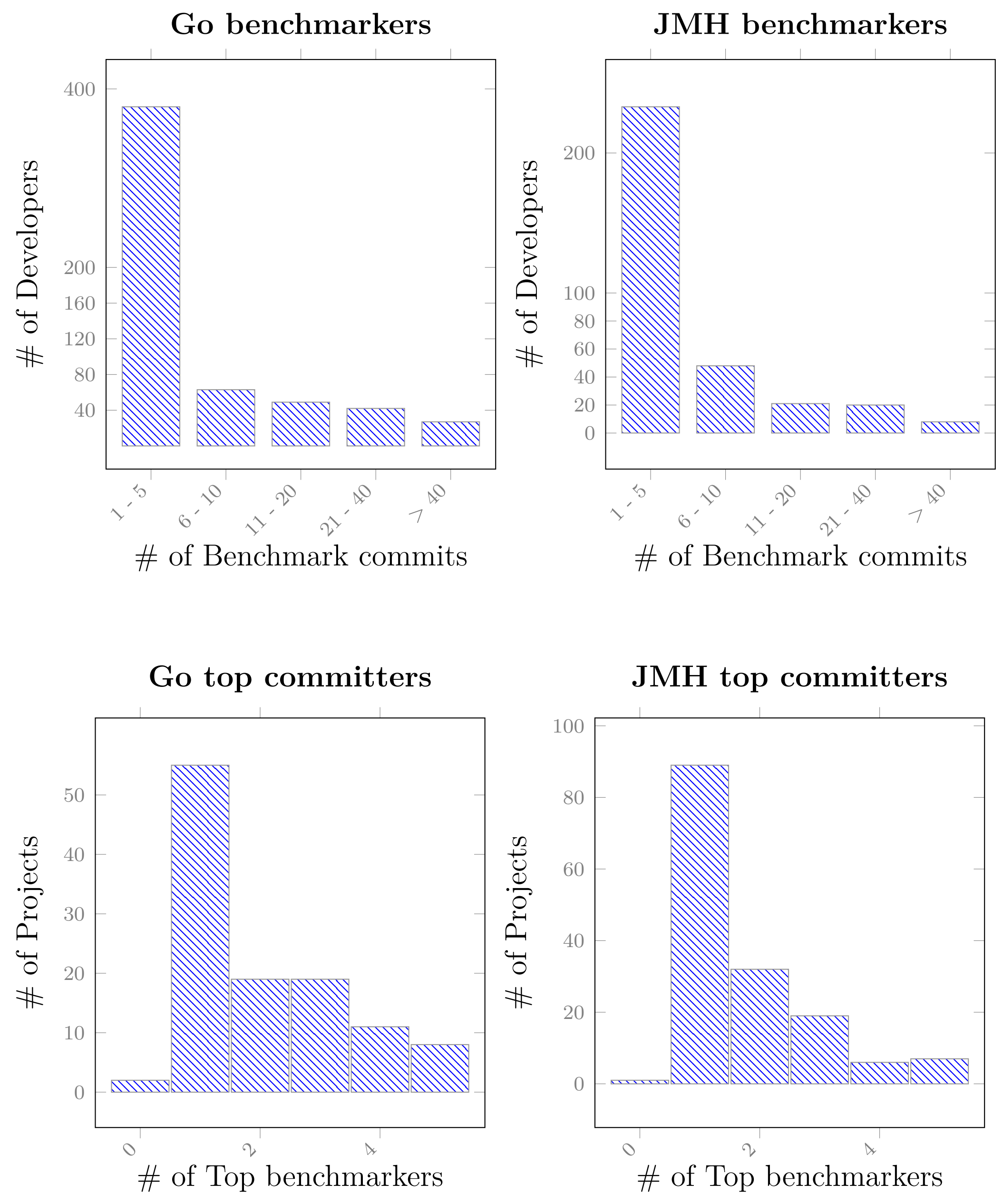


Figure 5.6: *The upper left diagram illustrates the number of commits affecting microbenchmarks per developer of the Go repositories. The upper right diagram illustrates the number of commits affecting microbenchmarks per developer of the Java repositories containing JMH. The lower left diagram illustrates how many benchmarkers there are among the top five committers of each Go project. The lower right diagram illustrates how many benchmarkers there are among the top five committers of each Java project.*

As can be seen in Figure 5.6 there are benchmarkers among the top committers of all Go projects except for two. In approximately twenty of the Go repositories four or all five of the top committers are also benchmarkers. Since some of the Go repositories consists of fewer than five developers all developers of those repositories could be benchmarkers, or most of them. Among the 482 Go benchmarkers in Table 5.9 approximately 400 of them has only performed between one and five commits affecting files containing microbenchmarks. Approximately 70 benchmarkers have performed at least 21 commits affecting files containing microbenchmarks.

Figure 5.7 illustrates the analysis result for **RQ5**. A part of the result is shown in Appendix L to give the reader an idea about how the analysis results look like while keeping the number of pages down. If a microbenchmark has been created but no changes occur to the file containing the microbenchmark afterwards it is considered to have been modified zero times.

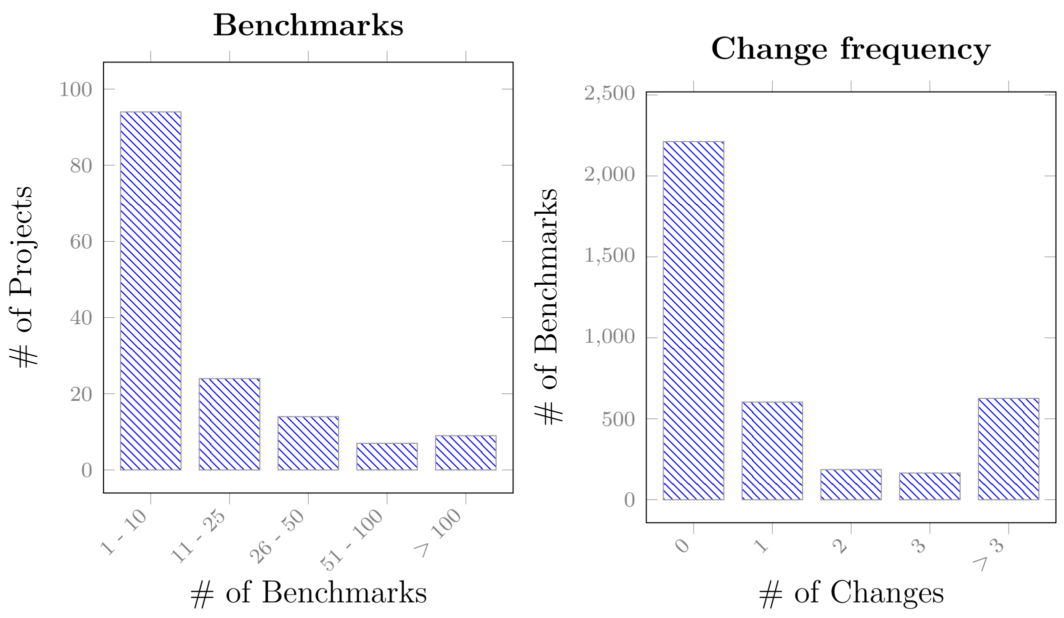


Figure 5.7: *The left diagram illustrates the number of microbenchmarks in the most recent snapshot of all repositories in the Java dataset. The right diagram illustrates the number of modifications of each microbenchmark in the Java dataset.*

As can be seen in Figure 5.7 most projects contain between one and ten microbenchmarks. The smallest number of microbenchmarks in a repository is one and the largest number is 390. The total number of microbenchmarks is 3792 and 2212 of them are never modified. This means that the common case is creating a benchmark and never change it afterwards. When modifications occur it is usually only once.

6 Discussion

This chapter provides a discussion about the findings of the mining tasks performed during the study, followed by a more general discussion about *MicroAnalyzer*. Finally, the validity threats are discussed.

6.1 RQ1-2: How many microbenchmarks allow DCE and CF optimizations?

Optimizations such as DCE and CF can result in microbenchmarks returning misleading performance times. Rodriguez-Cancio, Combemale, & Baudry [14] state that it is common for developers to design their benchmarks in a way that allow the JIT compiler to perform DCE and CF optimizations. Even though there are a number of cases enabling both optimizations only two of them was investigated in this study. As seen in Table 5.3, of the analyzed 3320 microbenchmarks 195 allow DCE optimizations and 84 allow CF optimizations for the investigated cases.

Despite only considering one case for each optimization in the analyses 5.87% of the microbenchmarks allowed DCE optimizations and 2.53% allowed CF optimizations. If all possible cases would have been considered the percentage would probably be substantially higher. Some scenarios, e.g., unreachable code, are likely more difficult for a developer to manually detect than others, like dead variables. This gives the impression of JIT optimizations not being rare in existing microbenchmarks.

It is difficult to say how much the benchmark results are affected by these optimizations without running the microbenchmarks and perform measurements. Therefore it would be suitable to develop a fourth module for *MicroAnalyzer* handling dynamic code analyses. Considering some optimization-enabling scenarios being more difficult to discover than others, maybe tool developers should incorporate functionality that help benchmarkers create good benchmark designs e.g., tool notifying about potential optimizations, or automatic generation of code.

Optimizations such as DCE and CF are not inherently bad. They may provide optimistic results since they enable benchmarks to run faster than the code might do in a production environment, but it depends on the context and how optimistic the measurement results are if they should be prevented. Too optimistic results may cause developers to draw wrong conclusions.

Further investigation would be needed to understand why benchmarks on the JVM are designed in a way that enables JIT optimizations. One reason could be that it is complex to benchmark the JVM and requires a lot of expertise [16]. Another reason could be that developers consider the optimization effects on the measurement results are negligible so they do not put a lot of effort in to counteracting such optimizations. The findings of this study suggests that the statement by Rodriguez-Cancio, Combemale, & Baudry [14] may be correct.

6.2 RQ3: How many microbenchmarks are configured according to published best practices?

Having enough warmup iterations is important for the correctness of microbenchmarking results since it takes time to reach steady-state performance [16, 19]. As illustrated by Georges et al. [10] the number of forks and measurement iterations also has an effect on the microbenchmarking results. According to their findings it seems to be good practice to use the current default configuration for JMH microbenchmarks.

As was shown in Table 5.6 60.87% of the 3407 analyzed microbenchmark configurations used the defaults. The second most common configuration was used by 20.93% of the microbenchmarks and consisted of considerably fewer iterations and forks. Based on published best practices [10] this configuration is not viable for microbenchmark results. It is desirable to have more than ten warmup iterations in order to reach steady state.

Based on the findings presented in Table 5.6 all configuration combinations except the most common has ten or less warmup iterations. While 60.87% of the configurations can be considered to follow best practices 39.13% can not. This implicates that the measurement results from a large set of all microbenchmarks are not reliable. A consequence of misleading results could be that wrong conclusions are drawn [16].

Further investigation would be needed to get an understanding of the reasons why developers use different configurations. One reason could be that they are unaware of pitfalls affecting the measurement results negatively. For example, not knowing how long warmup is needed to reach steady state before performing measurement iterations. Another reason could be that developers configure their microbenchmarks to receive the results they want, ignoring the correctness of the measurement results.

6.3 RQ4: How many developers work on microbenchmarks?

The level of expertise required for designing useful and correct performance microbenchmarks, and the lack of tool support, prevents adoption of microbenchmarking among developers on a larger scale [5, 14]. To get knowledge about how large set of the current developer population has worked with microbenchmarks an investigation was done on the entire development history of Go and Java repositories using microbenchmarks.

By using *MicroAnalyzer* to answer this question the Go repositories in Appendix J was automatically cloned from their remote locations on GitHub to a local directory. The local copies were automatically pre-processed into a dataset which was automatically analyzed together with the Java dataset to show that replication is possible using *MicroAnalyzer*. The analysis results support the findings in [9] about projects usually having only one or a few developers working on microbenchmarks and that these developers seem to be core developers of their projects. They studied 111 Java repositories and found that microbenchmarks are often written once and rarely updated afterwards. My study was on 202 Go and 166 Java repositories and the two upper diagrams in Figure 5.6 indicate that microbenchmarks are written once and rarely updated since most benchmarkers perform only one or a few commits affecting microbenchmarks.

As illustrated in tables 5.9 and 5.10 approximately 9.82% of developers in the analyzed Go dataset, and 15.09% in the Java dataset, has committed changes affecting microbenchmarks at some point. To investigate the extent of their microbenchmarking the diagrams in Figure 5.6 show that most of the benchmarkers only perform a few commits affecting files containing microbenchmarks. It can also be seen that benchmarkers are among the top committers of all Go projects except two. Even though benchmarkers only makes up a small part of all developers in the projects they are most of the time top committers. Thus, benchmarkers seem to be core developers of their projects.

The analysis task was performed on both datasets at the same time to showcase replication using *MicroAnalyzer*. But adding the separate analyses manually gave 7026 developers and 805 benchmarkers which differs compared to the automated result given in Table 5.11. While the difference is small, three benchmarkers and 65 developers, it still indicates that there is a bug in the framework implementation.

As was discovered in sections 5.1.1 and 5.2.1 when identifying Java and Go microbenchmark repositories only a fraction of all Java and Go projects on GitHub contains microbenchmarks. This implicates that the total amount of developers on GitHub working with microbenchmarks is significantly smaller than 9.82% since GitHub hosts millions of Java and Go repositories. While the analysis do not give an answer to why it seems to be unpopular with microbenchmarking among developers it suggests that the current complexity microbenchmarking could be a reason. Another reason could be that developers in many projects are not concerned with performance, or prioritize other tests.

6.4 RQ5: How often do microbenchmarks change?

Only a small number of mining studies have been performed on performance microbenchmarks. Common to those studies are that several parts of the mining process was performed manually. The purpose of this study was to create a framework that automates mining tasks, namely *MicroAnalyzer*. In the process of validating the framework analysis results from published papers was compared to analysis results from using the framework replicating parts of those studies.

It is shown in [9] that the microbenchmarks in the studied 111 Java projects are often written once and rarely updated afterwards. *MicroAnalyzer* was used to perform this analysis on 166 Java repositories. The cloning and pre-processing tasks was not performed since the Java dataset was already created in the first mining study and therefore reused in this analysis. The analysis result for answering **RQ5**, illustrated in Figure 5.7, supports the observation in [9] that microbenchmarks are usually written once and rarely updated.

To explain why it is common that microbenchmarks are created and never modified would require further investigation. One reason could be the complexity of creating correct and useful microbenchmarks. Thus, a developer might settle with only creating a benchmark that seem to show reasonable measurement results.

6.5 *MicroAnalyzer* execution times

Performing mining studies is often a difficult and time-consuming task [40]. Typically only a small number of software repositories are studied due to the challenges of finding, retrieving, pre-processing, and analyzing large amounts of data. To address these challenges *MicroAnalyzer* automates the cloning, pre-processing and analyses. This reduces the difficulty of performing mining studies but it is also important that the execution times of the automated processes are good enough to make the framework usable.

The retrieval process is automated by the cloning module. How fast remote repositories are retrieved depends on the speed of the Internet connection used. Therefore no measurement is done on the execution time of the cloning module.

Five pre-processing tasks were performed on 100 Java repositories to further investigate the pre-processing execution time of *MicroAnalyzer*. This has already been done in Section 5.1.2 for files containing microbenchmarks which showed that such a file took 0.484 seconds to pre-process. The resulting dataset is described in Table 6.1. The total duration indicates that it takes 0.039 seconds to pre-process a file when no distinction is made between regular files and files containing microbenchmarks.

Table 6.1: *Statistics about the projects used as input for the pre-processing tasks. The size corresponds to the total size of the latest snapshot of all repositories. The number of files corresponds to the total number of files parsed. The duration is the mean value of five measurements performed on the pre-processing execution time.*

$Type$	$Repositories$	$Size[GB]$	$Parsedfiles$	$Revisions$	$Duration[s]$
$Java$	100	3.87	511252	149563	19878.8

By comparing the pre-processing time per file it seems that parsing only files containing microbenchmarks shortens the total duration significantly. This is because usually most files in a project are non-microbenchmark files. It can be seen though that the pre-processing time for each microbenchmark file is significantly longer than for each regular file. Parsing a file without making any distinction is probably faster per file since making a distinction requires *MicroAnalyzer* to load and parse each file first to check if it contains any microbenchmarks before deciding if a conversion of the file to the internal data model should be made. Thus, all files will still be parsed, but not mapped and persisted, when performing pre-processing on only microbenchmark files.

Pre-processing 511252 files in 19878.8 seconds when pre-processing all Java files and pre-processing 6873 files in 3324 seconds when pre-processing only Java files containing microbenchmarks could probably be considered reasonable in a mining study context. Especially since a pre-processing task is only required to be performed once. The created dataset can be reused an unlimited number of times for analyses. As can be seen in tables 5.1 and 5.2 that dataset is 9.29MB which is less than 0.2% of the original data.

One goal for *MicroAnalyzer* was to make it suitable for large-scale mining of performance microbenchmarks. Analysis tasks was created to answer the research questions in this study. The execution time of each analysis is given in Table 6.2. As can be seen the analyses on the Java and Go datasets containing 166 and 202 repositories respectively took approximately six seconds including the Hadoop overhead. This is equal to 0.036 seconds per Java repository and 0.030 seconds per Go repository. The analysis of 10 000 repositories would then take approximately 360 seconds. Thus it could be considered that *MicroAnalyzer* is suitable for large-scale mining.

It is difficult to perform a fair comparison to other tools due to tools having different capabilities, restrictions, and feature sets. To put the execution times of *MicroAnalyzer* in perspective they are compared to the execution times of LISA. Some statistics from the study by Alexandru, Panichella, & Gall [44] concerning the pre-processing and analysis of 100 Java projects using the MSR tool LISA is shown in Table 6.3. The total duration consists of tasks such as parsing and analysis. The parsing time for *MicroAnalyzer* includes time for persisting data which is

Table 6.2: *Execution time of each analysis task. Only files containing microbenchmarks were analyzed. The total duration includes overhead such as initiating the Hadoop job. Each duration is the mean value of five analysis task executions.*

$Analysis$	$Duration[s]$	$Totalduration[s]$
$RQ1$	4.175	6.127
$RQ2$	4.147	6.197
$RQ3$	3.886	6.124
$RQ4$	4.312	6.160
$RQ5$	3.943	5.132

excluded from LISA's parsing time. The difference in duration per file and revision is illustrated in Table 6.4. *MicroAnalyzer* performs analyses independently from pre-processing tasks so the analysis duration per file and revision is based on the dataset in Table 5.2 since the analysis tasks were performed on that dataset.

Table 6.3: *Statistics about an analysis task performed with the mining tool LISA.*

$Type$	$Projects$	$Parsedfiles$	$Revisions$	$Duration[s]$	$Parsing[s]$	$Analysis[s]$
$Java$	100	3235852	646261	67380	48300	5700

As illustrated in Table 6.4 the parsing times are approximately twice as long for *MicroAnalyzer* while analysis times are approximately half of LISA's. The reason for the analysis duration per revision being so low for *MicroAnalyzer* is because most revisions does not contain changes to files containing microbenchmarks which means that most revisions are skipped. *MicroAnalyzer* performs sequential parsing which is slower than the parallel parsing done by LISA. A difference between the tools is that performing analysis tasks with *MicroAnalyzer* can be done an unlimited number of times on a pre-processed dataset while pre-processing must be performed each time an analysis task is executed using LISA [44].

Tool / Durations [ms]	***MicroAnalyzer***	***LISA***
Parsing/file	39	15
Parsing/revision	133	75
Analysis/file	0.89	1.76
Analysis/revision	0.03	8.82

Table 6.4: *Comparison between parsing and analysis durations using LISA and MicroAnalyzer. The analysis for **RQ3** was used to calculate the duration per file and revision for MicroAnalyzer.*

6.6 Design impact on study reproducibility

According to Robles [21] mining studies being replicable and reproducible is something that is rare. Tool support is important for efficient software mining and for improving reproducibility and replicability, which is essential in the validation of mining study findings [6].

Due to the importance of mining studies being reproducible and replicable, and considering the low number of studies that actually are, *MicroAnalyzer* was designed with both reproducibility and replicability in mind. González-Barahona & Robles [1] outline eight elements that they consider may have an impact on the reproducibility of studies and therefore could be subject for reuse in new studies. Below each of those elements and how they relate to the *MicroAnalyzer* design is discussed.

1) **Data source.** A data source can be of a number of different types such as hosting services and electronic archives. *MicroAnalyzer* has an architecture enabling plugins that supports retrieval of raw data from any data source. A GitHub repository cloning plugin was created for the mining studies performed in this thesis. This plugin is open sourced and can be reused in any future mining study where repositories located on GitHub needs to be stored locally.
2) **Retrieval methodology.** Often researchers and practitioners must retrieve the data they want to work on from the data sources. This is because in most cases it is not possible to work directly with the data in the sources. GitHub and other services restrict the ability to access and modify their repositories in such a way. *MicroAnalyzer* has a cloning module which automates the retrieval process from various data sources. A JSON file containing information about the repositories to clone is used as input to this module. To make this task fully reproducible a copy of the JSON file, a copy of the cloning plugin, and the parameters used to run the cloning module is required.
3) **Raw dataset.** As a result of said restriction researchers must have a raw dataset which can be used to work on as desired. In this thesis the raw datasets are the Java and Go repositories cloned from GitHub. Since these repositories are stored locally it is possible to work on them as desired. To reproduce a study without having a copy of the raw datasets the retrieval step needs to be performed.
4) **Extraction methodology.** *MicroAnalyzer* has an architecture enabling plugins that supports connecting to any VCS and parsing any programming language. Plugins for Go and Java parsers, as well as a Git connector, was created in this thesis. They are open sourced and can be reused in any future studies. *MicroAnalyzer* has a pre-processing module which automates pre-processing tasks. A JSON file containing information about the locally stored repositories to pre-process is used as input to this module. This is the same file that is used as input to the cloning module. To achieve full reproducibility a copy of the JSON file, a copy of the connector and parser plugins, and the parameters used to run the pre-processing module is required.
5) **Study parameters.** Used to control what parts of the raw data should be

extracted. With *MicroAnalyzer* it is possible to pre-process either all source code files or only files containing microbenchmarks. The flag *-all* is given when all files are of interest for a study. If only files containing microbenchmarks are of interest the flag is left out when running the pre-processing module.

6) **Processed dataset.** The dataset created by running the pre-processing module consists of two flat files. Using flat files as storage facilitates the reproduction of a study since only a copy of the flat files is needed. The dataset must be available for others to acquire if they want to reproduce a study. Otherwise this dataset must be created from scratch by performing all steps from the beginning.
7) **Analysis methodology.** *MicroAnalyzer* has an architecture enabling analysis plugins containing any number of analysis tasks. An open sourced analysis plugin containing analysis tasks answering the research questions was created in this thesis. *MicroAnalyzer* has an analysis module which automates analysis tasks on various data sources. To make an analysis task fully reproducible a copy of the analysis plugin, the dataset, and the parameters for running the analysis module is required. To replicate a study an analyses is allowed to be performed on any number of datasets by adding a comma-separated list of dataset names when running the analysis module.
8) **Results dataset.** The output from running the analysis module is a text file containing the analysis results.

In order to reproduce a mining study a user of *MicroAnalyzer* only needs a copy of the pre-processed dataset, a copy of the analysis plugin, the parameters used to run the analysis module, and the framework jar. Replication of a study can be achieved by performing an analysis on additional datasets. In accordance with the eight elements mentioned by González-Barahona & Robles [1] studies performed using the framework could be considered reproducible and replicable.

For a majority of reviewed papers no tools were found, not even for papers where the authors explicitly state that they have built one [21]. *MicroAnalyzer* is open source which means that the framework is available for the public to use in their mining studies. If widely used the framework could contribute making reproducible and replicable studies less rare.

6.7 Mining tools comparison

Based on the literature review it is a common occurrence in mining studies that various tasks are performed manually, thus restricting the number of studied repositories since manual work is time consuming [21]. To enable studying repositories on a large scale there is a need for tools automating mining tasks.

One of the reasons for developing *MicroAnalyzer* was to fill this gap of tools automating the major parts of the mining process, while at the same time let users keep as much control as possible. As was described in Section 5 the main parts of

Tool / Attribute	***MicroAnalyzer***	***Boa***	***BigQuery***	***LISA***
Purpose	Microbenchmarks	Evolution	Query service	General
Input/Output	file/file	DSL/file	SQL/table	sbt/CSV
Offline/Online	Yes/Yes	No/Yes	No/Yes	Yes/Yes
Open sourced	Yes	Partly	No	Yes
Pre-processing	Yes	No	Partly	Yes
Evolution	Yes	Yes	Partly	Yes
Cloning	Yes	No	No	Yes
Mining level	S/P	S/P	S/P	S

Table 6.5: *Comparison between publicly available mining tools and MicroAnalyzer. The cloning and pre-processing attributes consider when these tasks are controllable by a user. The mining level values S and P corresponds to source code and project respectively.*

the mining process are retrieving raw data, pre-processing the raw data into a structured form, and perform analyses on the structured data. *MicroAnalyzer* consists of three modules each automating a distinct part of the mining process enabling researchers to spend more time on what is relevant for their research.

During the literature review it became clear that existing mining tools are not always publicly available [30]. And when publicly available they are often considered difficult to set up and use [5]. Tools are often limited in their applicability of different types of mining tasks. There is also criticism about the low reproducibility of MSR research [1]. One of the goals with *MicroAnalyzer* was to create a tool that stands out by meeting such requirements. The plug-in architecture increases the number of use cases for *MicroAnalyzer* as well as, together with the use of flat files as data storage, increases the reproducibility of mining studies. *MicroAnalyzer* is compared to BigQuery and some publicly available mining tools in Table 6.5 to highlight their differences.

To my knowledge none of the tools automates the cloning process except for LISA, *MicroAnalyzer*, and to a degree Boa. Boa is actually retrieving, as well as pre-processing, repositories automatically but without letting a user have any control over this process. The dataset used by Boa is located on their cluster and updated on a monthly basis [7]. As a result analyses are limited to Java repositories from SourceForge and GitHub. *MicroAnalyzer* enables full control over which data sources and repositories to clone and pre-process due to its open-sourced plug-in architecture. BigQuery imposes the responsibility of retrieving raw data on the user. Data is uploaded to Google Storage and imported to BigQuery [42].

Client libraries implemented in various languages can be used to call the REST APIs of BigQuery. Since BigQuery is cloud based a user must have an Internet connection in order to use the service. LISA is a standalone tool which can be used to perform analyses offline. If using LISA to clone remote repositories an Internet connection is required. Boa is also cloud based so in order to take advantage of their dataset and infrastructure an Internet connection is required. All parts of the mining process are performed offline by *MicroAnalyzer* making it a standalone mining tool.

Ease of use was considered when deciding to have a CLI as frontend for *MicroAnalyzer*. Thus, a user only has to learn a small number of parameters to run the framework as desired. To execute an analysis on the Boa dataset a query must be written in their DSL. To perform BigQuery analyses queries must be written in an SQL-like syntax either programmatically or via their web-based GUI. Mappings and analyses can be formulated programmatically, e.g., using it as an Eclipse plugin, or visually with the help of the Signal/Collect software.

Boa is partly open sourced. The compiler[1] and the client API[2] are open sourced, but not the infrastructure. It is the infrastructure that is responsible for retrieving and pre-processing Java projects from GitHub and SourceForge. The client API is used to communicate with the Boa infrastructure. *MicroAnalyzer* is open sourced to allow developers and researchers to extend the codebase as they please to enable any type of mining tasks. Documented code that is easy to understand was of high priority when developing the framework.

The internal data model used by mining tools are usually designed for the specific purpose of the tool. This means that during the pre-processing of raw data only certain data relevant for the model are extracted. As a consequence tools become limited in what mining tasks are possible to perform. During the literature review no mining tool specific for performance microbenchmarks was found. It is possible though that some of the existing tools could be used for a number of performance microbenchmark mining tasks if such extensions were created for them. *MicroAnalyzer* enables automatic identification of microbenchmark files and allows creating datasets consisting of only such files to enable faster pre-processing and analysis tasks.

In an attempt to not limit *MicroAnalyzer* too much the data model was designed for source code in general. That way all data in a file is extracted and persisted for analyses. To keep the focus on performance microbenchmarks only files containing benchmarks are parsed and persisted as default. A user of *MicroAnalyzer* must add the *-all* flag to create a dataset consisting of all files. Boa performs fine-grained parsing of source code as well.

6.8 Area of application

In the beginning of this thesis the goal was to make *MicroAnalyzer* designed specifically for mining performance microbenchmarks. During the course of the project

[1]https://github.com/boalang/compiler
[2]https://github.com/boalang/api

it became clear that one of the drawbacks shared by many existing mining tools were their limitation in what could be studied, both in terms of data sources and analysis types. Part of the problem is that the data models used in the tools are coarse grained which means that varying amounts of data are not mapped during pre-processing. As a consequence users have to learn a number of tools in order to be able to perform different types of mining studies.

The process of becoming familiar with mining tools is time consuming and one of the goals of *MicroAnalyzer* became to make the framework more general so that it could be reused in as many different types of mining studies as possible. Being open sourced and having a plug-in architecture enable users to extend *MicroAnalyzer* in order to achieve that goal.

6.9 Threats to validity

Aspects that can affect the validity of this study are discussed in this section. The validity of a study is related to how the study is performed, what is done, and by whom. The trustworthiness of results are denoted by such aspects which makes it important to identify validity threats. Runeson and Höst [43] classifies validity aspects as four categories which are addressed in sections 6.9.1 to 6.9.4.

6.9.1 Internal validity

The internal validity refers to the risk of confounding factors not under investigation having an effect on the observed outcome. If a researcher is unaware of them, or do not realize the extent of their effect, the interpretation of results as well as the conclusion could be different.

The findings in this study are based on results computed with *MicroAnalyzer* for a large number of repositories and rely on the correctness of these computations. The used parser and connector plugins has to correctly interpret the source code and extract desired data. This threat is mitigated by manually inspecting and validating a small fraction of parsed projects during construction of the plugins as well as in the analysis phase. The manual process was done multiple times to mitigate issues related to the human factor.

Factors such as other processes running at the same time as *MicroAnalyzer* competing about resources could affect the running time of a pre-processing or analysis task. To reduce the risk of incorrect execution time measurements each pre-processing and analysis task was performed and measured five times averaging the results.

6.9.2 Construct validity

Construct validity reflects on to what extent something studied represents what was intended to be studied. Threats to this validity concerns the relationship between intention and observation. The correctness of analyses performed with *MicroAnalyzer* was ensured by manually investigating analyzed data and comparing the findings with the automated analysis results. The manual process is error-prone due to the human factor so it was performed multiple times for each analysis to mitigate this issue.

MicroAnalyzer automatically identifies which files contain microbenchmarks. The risk of failing to identify such files is a threat to construct validity. Such identification failures could be a consequence of potential bugs, or microbenchmark files being written in other languages than Java and Go. To reduce the risk of *MicroAnalyzer* not performing this procedure correctly many files were investigated manually to ensure its correctness.

Furthermore, developers can have multiple GitHub profiles. Since it is difficult to decide if multiple profiles are owned by the same person they are treated as separate developers. Hence the analysis results concerning how many developers are working on microbenchmarks may be higher than in reality.

The results of this study is produced using *MicroAnalyzer*. Even though the framework and its plugins have been developed and inspected with great care there is always a risk of bugs. Hopefully such bugs only affect quantitative aspects marginally, and not qualitative aspects at all. The framework and used plugins are publicly available on GitHub.

6.9.3 External validity

The extent to which the outcomes and results of this study are generalizable is referred to as external validity. The risk of a biased and unrepresentative selection of JMH and Go microbenchmark repositories was reduced by making the selection based on popularity. Beyond two programming languages, the sample covers approximately 400 projects of varying sizes and application domains which could increase the generalizability of the findings. By creating parser plugins for other programming languages, or connector plugins for extracting evolution data from VCSs, or clone plugins enabling studies on other data sources, future research could replicate this study to increase the generalizability of the findings.

Only repositories on GitHub was considered. Consequently, it is not assured that the outcomes generalize to projects hosted on other services, such as Bitbucket. The studied projects are open sourced so the results may not extend to industrial projects. The magnitude of microbenchmark usage and the compliance to best practices might be different in closed source environments. Furthermore, since the most popular JMH and Go microbenchmark projects were studied the results cannot be assumed to be the same for less popular projects.

6.9.4 Reliability

Reliability is concerned with how reproducible a study is and how dependent the data and analyses are on a specific researcher. The results should be the same if another researcher conduct a similar study. This study was performed by one author making the results being in the risk of the authors biased subjective point of view. To mitigate reliability threats the data collection, data pre-processing, and analyses are documented and reviewed multiple times to ensure reproducibility of the study. The analysis results, as well as files and datasets used, are available[3] online for replication purposes.

[3]https://github.com/MicroAnalyzer/MA-MicrobenchmarkAnalyses/tree/master/thesis

7 Conclusion

To allow for a wider adoption of microbenchmarking among developers it should be less complex to create correct and useful microbenchmarks. Currently a developer is required to be somewhat of an microbenchmarking expert.

This study consisted of creating a framework, *MicroAnalyzer*, which enables the automation of mining studies. It was an action research where the framework was implemented and the design reflected upon. *MicroAnalyzer* was validated by performing two mining studies whose results were reflected on.

The main purpose of the study was to create an open-sourced extensible framework for large-scale mining of software performance microbenchmarks. As was seen in the existing microbenchmark mining studies [8, 9] there is a need for automation, especially concerning analysis tasks. Further, the study investigated the reproducibility of studies when using *MicroAnalyzer* since this is a rare quality of existing mining tools [21].

The framework is open sourced since many of the existing tools are not available to the public [30], restricting the number of options for researchers and practitioners. When publicly available there is a problem of the tools being difficult to set up and use [5]. The interaction with *MicroAnalyzer* consists of only a CLI with a limited number of parameters, which makes the learning curve less steep.

The results of this study showed that *MicroAnalyzer* can be useful for the automation of large-scale mining. The mining studies can be considered highly reproducible since the framework design meet the reproducibility requirements outlined by González-Barahona & Robles [1] to a high degree. Mining studies could be considered less difficult and time consuming to perform due to the framework automating major parts of the mining process as well as having decent execution times.

MicroAnalyzer is not restricted to mining studies for performance microbenchmarks. Because of its fine grained data model any information in files, as well as on a project level, can be pre-processed and analyzed. Being open sourced and having a plug-in architecture *MicroAnalyzer* enable users to extend the framework for use in other types of mining studies.

Further, this study showed it is common that microbenchmarks have a configuration inferior to what is recommended in published best practices. Also, two of the cases enabling a JIT compiler to perform DCE and CF optimizations were investigated. Taking into consideration that several cases was excluded the results indicated that it is not rare that microbenchmarks are designed in a way that enables optimizations. Based on the inferior configurations and the occurrence of potential optimizations it could be considered that the correctness of existing microbenchmarks is questionable. This does not mean that the microbenchmarks are not usable.

Part of the study focused on the evolution of microbenchmarks. The results showed that microbenchmarks are often created and then never updated afterwards. The extent of microbenchmarking practices was also investigated. Approximately 2.35% of developers in repositories using microbenchmarks worked on them. If all repositories was included the number of benchmarkers would probably be significantly smaller since JMH is the most common benchmarking framework for Java [14]. It was observed that the benchmarkers were among the top committers in their respective projects, indicating that benchmarkers are core developers of their projects. These findings suggest that statements about microbenchmarking being complex and unpopular are still valid [5, 14].

Future work could be extending the framework with a fourth module handling dynamic benchmarking in source code. By using the *-all* flag *MicroAnalyzer* preprocesses all source code. Thus, it is possible to analyze all code in a project. This extends the set of use cases for the framework with, for example, identifying critical sections in source code and give a developer suggestions on how to improve the code to make it perform better.

Bibliography

[1] González-Barahona, J. M., & Robles, G. (2012). On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empirical Software Engineering*, 17(1-2), 75-89.

[2] Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., & Damian, D. (2014, May). The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories* (pp. 92-101). ACM.

[3] Ghezzi, G., & Gall, H. C. (2013, May). Replicating mining studies with SOFAS. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on* (pp. 363-372). IEEE.

[4] Ghezzi, G., & Gall, H. C. (2011, June). Sofas: A lightweight architecture for software analysis as a service. In *Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on* (pp. 93-102). IEEE.

[5] Amann, S., Beyer, S., Kevic, K., & Gall, H. (2015). Software Mining Studies: Goals, Approaches, Artifacts, and Replicability. In *Software Engineering* (pp. 121-158). Springer International Publishing.

[6] Proksch, S., Amann, S., & Mezini, M. (2014, June). Towards standardized evaluation of developer-assistance tools. In *Proceedings of the 4th International Workshop on Recommendation Systems for Software Engineering* (pp. 14-18). ACM.

[7] Dyer, R., Nguyen, H. A., Rajan, H., & Nguyen, T. N. (2015). Boa: Ultra-large-scale software repository and source-code mining. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(1), 7.

[8] Stefan, P., Horky, V., Bulej, L., & Tuma, P. (2017, April). Unit Testing Performance in Java Projects: Are We There Yet?. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (pp. 401-412). ACM.

[9] Leitner, P., & Bezemer, C. P. (2017, April). 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* (pp. 373-384). ACM.

[10] Georges, A., Buytaert, D., & Eeckhout, L. (2007). Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices, 42*(10), 57-76.

[11] Dyer, R., Nguyen, H. A., Rajan, H., & Nguyen, T. N. (2013, May). Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering* (pp. 422-431). IEEE Press.

[12] Bulej, L. (2016, March). Performance Testing in Software Development: Getting the Developers on Board. In *Companion Publication for ACM/SPEC on International Conference on Performance Engineering* (pp. 9-9). ACM.

[13] Ho, C. W., & Williams, L. (2007, February). Developing software performance with the performance refinement and evolution model. In *Proceedings of the 6th International Workshop on Software and Performance* (pp. 133-136). ACM.

[14] Rodriguez-Cancio, M., Combemale, B., & Baudry, B. (2016, August). Automatic microbenchmark generation to prevent dead code elimination and constant folding. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (pp. 132-143). ACM.

[15] Dean, J., & Barroso, L. A. (2013). The tail at scale. *Communications of the ACM, 56*(2), 74-80.

[16] Gil, J. Y., Lenz, K., & Shimron, Y. (2011, October). 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, and VMIL'11* (pp. 297-308). ACM.

[17] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., ... & Hirzel, M. (2006, October). The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices* (Vol. 41, No. 10, pp. 169-190). ACM.

[18] Jovic, A., Brkic, K., & Bogunovic, N. (2014, May). An overview of free software tools for general data mining. In Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2014 37th International Convention on (pp. 1112-1117). IEEE.

[19] Horký, V., Libič, P., Steinhauser, A., & Tůma, P. (2015, January). Dos and don'ts of conducting performance measurements in java. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (pp. 337-340). ACM.

[20] Hassan, A. E. (2008, September). The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.* (pp. 48-57). IEEE.

[21] Robles, G. (2010, May). Replicating MSR: A study of the potential replicability of papers published in the Mining Software Repositories proceedings. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on* (pp. 171-180). IEEE.

[22] Godfrey, M. W., Hassan, A. E., Herbsleb, J., Murphy, G. C., Robillard, M., Devanbu, P., ... & Notkin, D. (2009). Future of mining software archives: A roundtable. *IEEE Software, 26*(1), 67-70.

[23] Heger, C., van Hoorn, A., Mann, M., & Okanović, D. (2017, April). Application performance management: State of the art and challenges for the future. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (pp. 429-432). ACM.

[24] Heger, C., Happe, J., & Farahbod, R. (2013, April). Automated root cause isolation of performance regressions during software development. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering* (pp. 27-38). ACM.

[25] Gousios, G., & Spinellis, D. (2017, May). Mining software engineering data from GitHub. In *Proceedings of the 39th International Conference on Software Engineering Companion* (pp. 501-502). IEEE Press.

[26] Zowghi, D., & Coulin, C. (2005). Requirements elicitation: A survey of techniques, approaches, and tools. In *Engineering and managing software requirements* (pp. 19-46). Springer, Berlin, Heidelberg.

[27] Cockburn, A., & Highsmith, J. (2001). Agile software development, the people factor. *Computer, 34*(11), 131-133.

[28] Highsmith, J., & Cockburn, A. (2001). Agile software development: The business of innovation. *Computer, 34*(9), 120-127.

[29] Williams, L., & Cockburn, A. (2003). Guest Editors' Introduction: Agile Software Development: It's about Feedback and Change. *Computer, 36*(6), 39-43.

[30] Chaturvedi, K. K., Sing, V. B., & Singh, P. (2013, June). Tools in mining software repositories. In *Computational Science and Its Applications (ICCSA), 2013 13th International Conference on* (pp. 89-98). IEEE.

[31] GitHub Help. (n.d.). Retrieved April 13, 2018, from https://help.github.com/articles/github-glossary/#repository

[32] Wiki Wiki Web. (n.d.). Retrieved May 5, 2018, from http://wiki.c2.com/?HierarchicalVisitorPattern

[33] Gousios, G., & Spinellis, D. (2014). Conducting quantitative software engineering studies with Alitheia Core. *Empirical Software Engineering, 19*(4), 885-925.

[34] Gousios, G., & Spinellis, D. (2009, May). A platform for software engineering research. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on* (pp. 31-40). IEEE.

[35] Mitropoulos, D., Gousios, G., & Spinellis, D. (2012, October). Measuring the occurrence of security-related bugs through software evolution. In *Informatics (PCI), 2012 16th Panhellenic Conference on* (pp. 117-122). IEEE.

[36] Ligu, E., Chaikalis, T., & Chatzigeorgiou, A. (2013). BuCo Reporter: Mining Software and Bug Repositories.

[37] Di Ruscio, D., Kolovos, D. S., Korkontzelos, I., Matragkas, N., & Vinju, J. J. (2015, August). Ossmeter: A software measurement platform for automatically analysing open source software projects. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (pp. 970-973). ACM.

[38] Fernandes, S., & Bernardino, J. (2015, July). What is bigquery?. In *Proceedings of the 19th International Database Engineering & Applications Symposium* (pp. 202-203). ACM.

[39] Cosentino, V., Izquierdo, J. L. C., & Cabot, J. (2016, May). Findings from GitHub: methods, datasets and limitations. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on* (pp. 137-141). IEEE.

[40] Dyer, R., Rajan, H., Nguyen, T. N., & Nguyen, H. A. (2015, October). Demonstrating programming language feature mining using Boa. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity* (pp. 13-14). ACM.

[41] Laaber, C., & Leitner, P. (2018). An Evaluation of Open-Source Software Microbenchmark Suites for Continuous Performance Assessment.

[42] Sato, K. (2012). An inside look at google BigQuery, white paper. *Google Inc.*

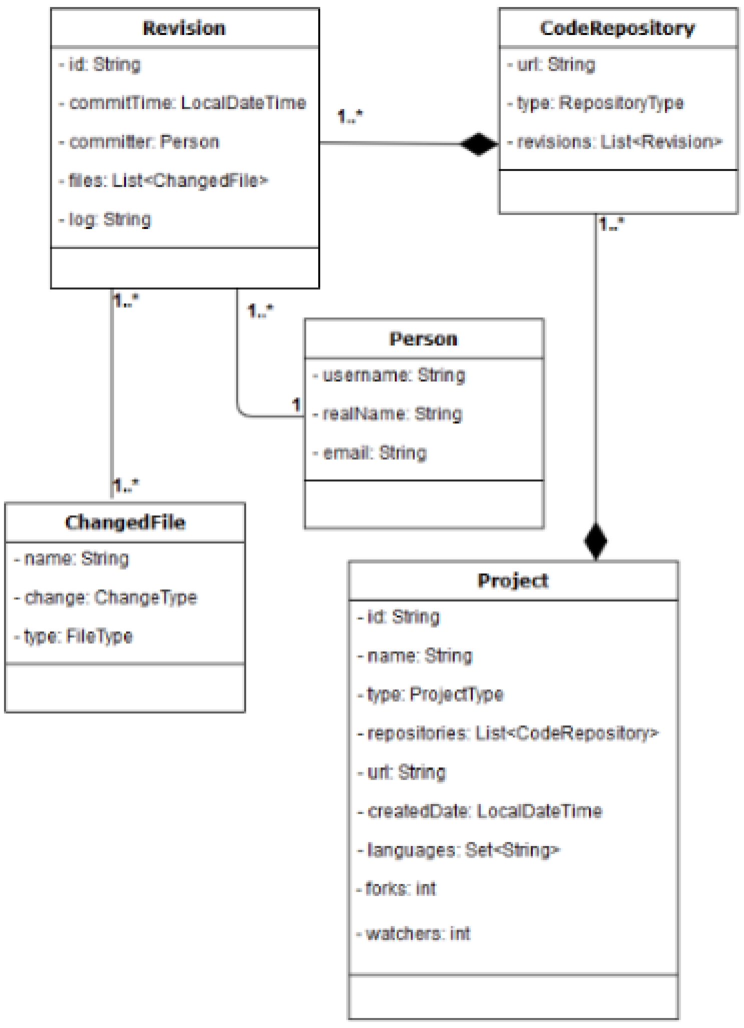
[43] Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering, 14*(2), 131.

[44] Alexandru, C. V., Panichella, S., & Gall, H. C. (2017, February). Reducing redundancies in multi-revision code analysis. In *Software Analysis, Evolution*

and Reengineering (SANER), 2017 IEEE 24th International Conference on (pp. 148-159). IEEE.

A

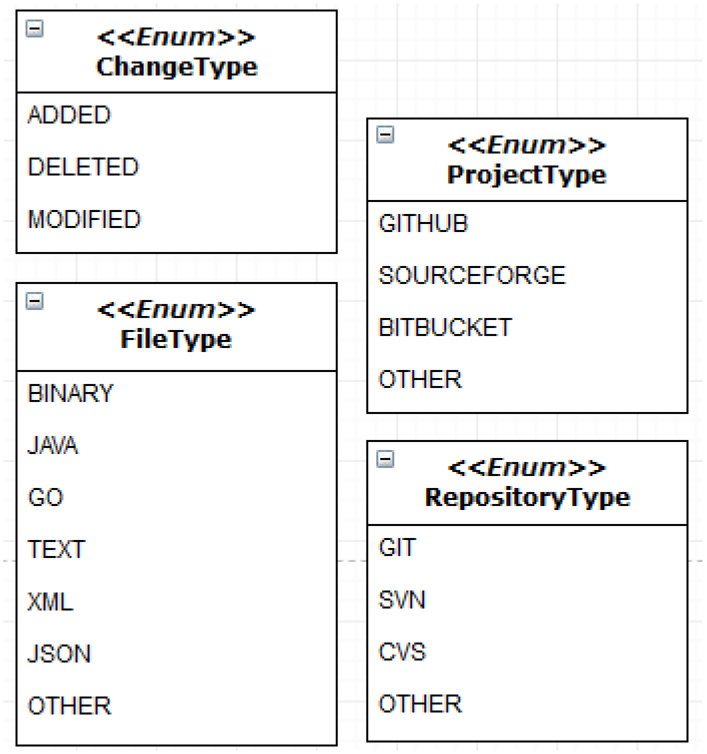
Class diagram of the top-level domain-specific types



B

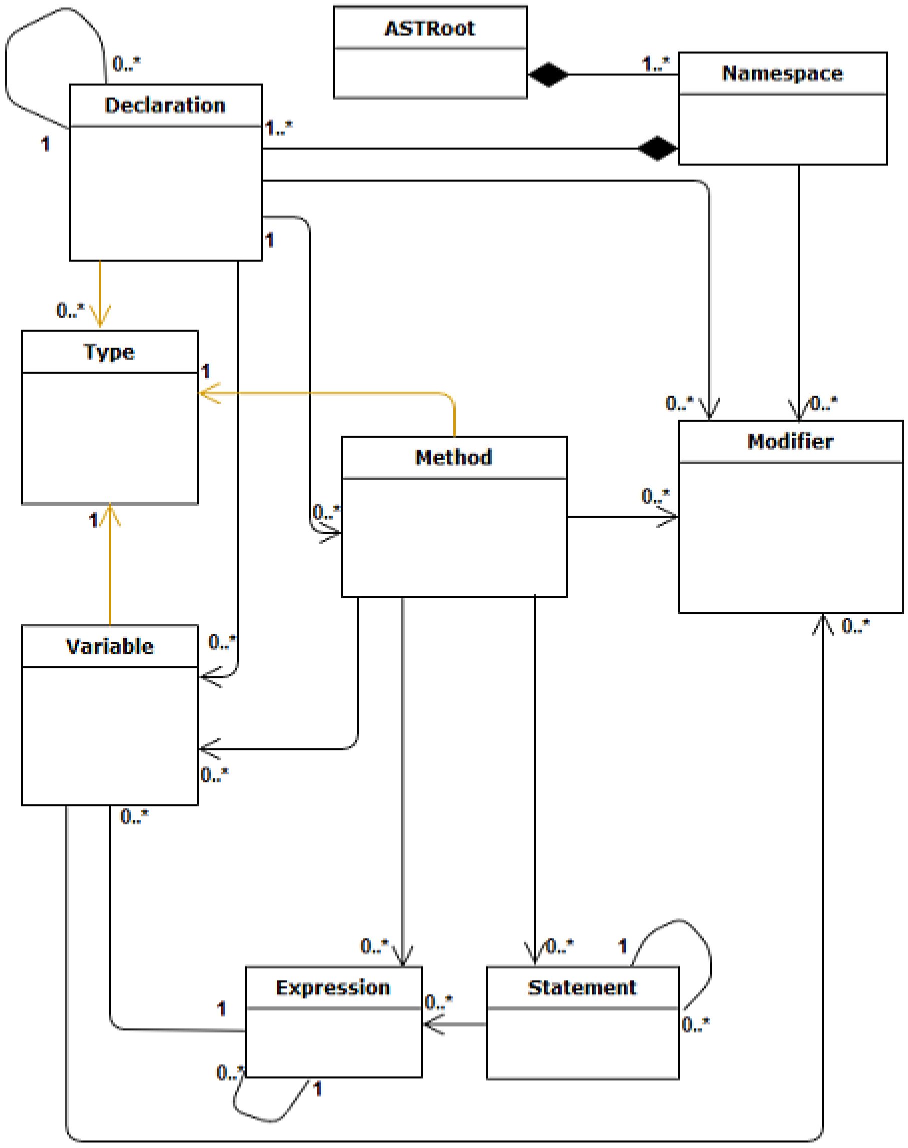
Top-level enumeration types

The ChangeType enumeration shows the three types of file changes used in the data model of *MicroAnalyzer*. The reason for limiting to these three types was to reduce the implementation complexity and at the same time retain the most interesting change reasons. The ProjectType represents where the project is located, while RepositoryType indicates the VCS used. FileType corresponds to source-code file types that can be found in a revision.



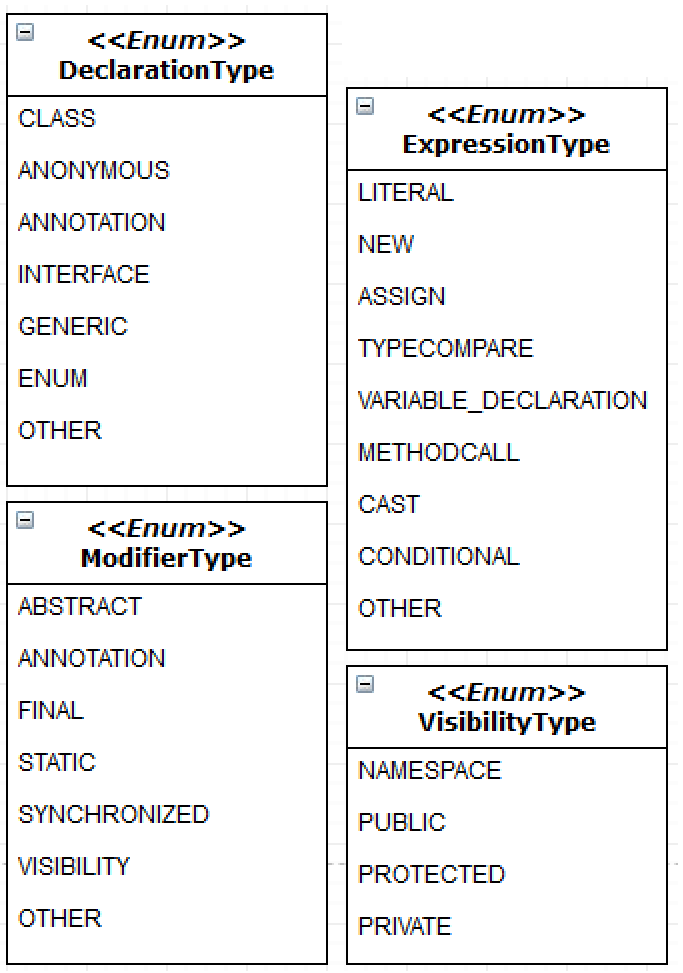
C

Model of the custom AST



D

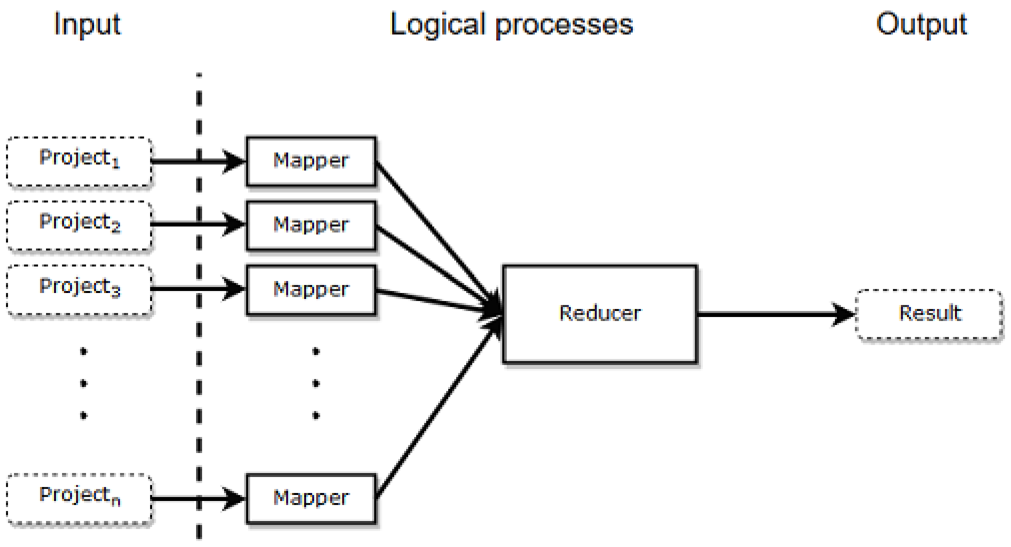
AST enumeration types



E

MapReduce model for input analysis program

Each project from a dataset becomes a single input to a process. The program is instantiated once for each input, where each instantiation is a map process. The map processes send the results of what is being analyzed to a single reducer process, which then aggregates values from all map processes and computes the final values. The final result is written to a text file created in the directory where the *MicroAnalyzer* jar is located.



F

List of repositories in the JMH dataset

Below is a list of the projects that were pre-processed to create the JMH dataset used for analyses. Each project is listed as [user]/[project] so in order to visit a project's page type https://github.com/[user]/[project]

- jenetics/jenetics
- cleberecht/singa
- kno10/reversegeocode
- vongosling/jvm-serializer
- shakamunyi/beam
- benjchristensen/RxJava
- msteindorfer/oopsla15-artifact
- vladimir-bukhtoyarov/rolling-metrics
- Domo42/saga-lib
- maxdemarzi/graph_processing
- relayrides/pushy
- fakemongo/fongo
- x-stream/xstream
- tools4j/decimal4j
- junkdog/artemis-odb
- pietermartin/sqlg
- cryptomator/cryptolib
- Netflix/spectator
- lmdbjava/benchmarks
- vlsi/compactmap
- dmart28/reveno
- benalexau/hash-bench
- akarnokd/ixjava
- Devexperts/aprof
- davidmoten/rtree
- apache/directory-kerby
- spotify/apollo
- opendaylight/yangtools
- square/okio
- biboudis/jmh-profilers
- crate/crate
- terma/fast-select
- graalvm/truffle
- AdoptOpenJDK/vmbenchmarks
- aol/cyclops-react
- robinst/autolink-java
- batterseapower/btreemap
- json-iterator/java
- snazy/ohc
- openzipkin/zipkin
- nitsanw/jmh-samples
- akarnokd/akarnokd-misc
- apache/logging-log4j2
- vladimir-bukhtoyarov/bucket4j
- almondtools/stringbench
- graalvm/graal-core
- ssaarela/javersion
- smoketurner/notification
- vsonnier/hppcrt
- awood/crl-stream
- davidmoten/rxjava2-extras
- TridentSDK/Trident
- pgjdbc/pgjdbc
- jchambers/jvptree
- RoaringBitmap/RoaringBitmap

- carrotsearch/hppc
- druid-io/druid
- arteam/100-Java-Concurrency-questions
- JetBrains-Research/viktor
- foursquare/fsqio
- SquidPony/SquidLib
- Vedenin/java_in_examples
- willhaben/blog
- goldmansachs/gs-collections
- davidmoten/geo
- arienkock/parallelism-benchmarks
- alexeyr/pcg-java
- firebase/netty
- arnaudroger/SimpleFlatMapper
- apptik/jus
- larusba/neo4j-jdbc
- Mojang/blixtser
- graalvm/mx
- peptos/traffic-shm
- eventsourcing/es4j
- promeG/TinyPinyin
- winklerm/phreak-examples
- belliottsmith/injector
- netzwerg/paleo
- JetBrains/xodus
- allegro/json-avro-converter
- apptik/RHub
- reactor/reactive-streams-commons
- uber/tchannel-java
- eclipse/eclipselink.runtime
- smoketurner/uploader
- udoprog/tiny-async-java
- netty/netty
- paritytrading/parity
- ktoso/sbt-jmh
- nielsbasjes/yauaa
- tudorv91/SparkJNI
- FIXTradingCommunity/silverflash
- cryptomator/siv-mode
- rmpestano/cukedoctor
- twitter/hpack
- real-logic/benchmarks
- danielnorberg/rut
- spring-cloud/spring-cloud-sleuth
- GumTreeDiff/gumtree
- HubSpot/jinjava
- vladimirdolzhenko/gflogger
- platan/idea-gradle-dependencies-formatter
- MeteoGroup/jbrotli
- linkedin/pinot
- easymock/objenesis
- mikvor/hashmapTest
- Squarespace/template-compiler
- eseifert/gral
- tobijdc/PRNG-Performance
- Techcable/Event4J
- marklogic/marklogic-jena
- mp911de/logstash-gelf
- trautonen/logback-ext
- ReactiveX/RxJava
- h2oai/h2o-3
- automenta/narchy
- neowu/core-ng-project
- akarnokd/RxJava2Extensions
- opendaylight/controller
- johnlcox/motif
- airlift/aircompressor
- stagemonitor/stagemonitor
- mbosecke/template-benchmark
- huxi/sulky
- bramp/unsafe
- dimagi/commcare-core
- openzipkin/zipkin-reporter-java
- orfjackal/cqrs-hotel
- neo4j-contrib/neo4j-jdbc
- cache2k/cache2k-benchmark
- requery/requery
- atlassian/commonmark-java
- grpc/grpc-java
- Barteks2x/CubicChunks
- NICTA/javallier
- DanielThomas/spring-boot-starter-netty
- rxin/jvm-unsafe-utils
- gvsmirnov/java-perv
- vsch/flexmark-java
- Mercateo/rest-schemagen
- aNNiMON/Own-Programming-Language-Tutorial
- bingoohuang/westcache

- spotify/sparkey-java
- protobufel/protobuf-el
- google/conscrypt
- davidmoten/audio-recognition
- blynkkk/blynk-server
- RobWin/javaslang-circuitbreaker
- dropbox/dropbox-sdk-java
- komiya-atsushi/fast-rng-java
- davidmoten/rxjava-extras
- davidmoten/rtree-3d
- TridentSDK/TridentSDK
- ReactiveSocket/reactivesocket-java
- debop/debop4k
- fabienrenaud/java-json-benchmark
- Graylog2/graylog-plugin-pipeline-processor
- spotify/completable-futures
- pinterest/jbender
- Vanilla-Java/Microservices
- kpavlov/fixio
- burmanm/gorilla-tsc
- presidentio/test-data-generator
- titorenko/quick-csv-streamer
- shunfei/indexr
- JCTools/JCTools
- airlift/airlift
- scalecube/scalecube
- jgrapht/jgrapht
- lucwillems/JavaLinuxNet
- real-logic/simple-binary-encoding
- glowroot/glowroot
- junkdog/entity-system-benchmarks
- ben-manes/caffeine

G

Part of the analysis result for RQ1

First comes the repository name, followed by [microbenchmarks allowing DCE optimizations]/[number of microbenchmarks].

Benchmarks: 3294 optimizations: 195

- RxJava 5/100
- glowroot 0/5
- SparkJNI 0/2
- java_in_examples 2/49
- graylog-plugin-pipeline-processor 0/10
- java 0/2
- Trident 0/8
- simple-binary-encoding 0/4
- entity-system-benchmarks 0/12
- gumtree 0/1
- zipkin 0/54
- test-data-generator 0/8
- ixjava 0/11
- jvm-unsafe-utils 0/3
- spring-boot-starter-netty 0/7
- apollo 0/1
- SquidLib 27/143
- reveno 0/1
- spring-cloud-sleuth 0/15
- conscrypt 0/7
- decimal4j 0/102
- idea-gradle-dependencies-formatter 0/0
- okio 0/18
- debop4k 0/2
- directory-kerby 0/2
- xstream 0/18
- requery 0/2
- java-perv 1/37
- jmh-samples 0/19
- flexmark-java 0/4
- logstash-gelf 0/16
- logback-ext 0/2
- yangtools 0/6
- SimpleFlatMapper 3/43
- RoaringBitmap 0/203
- Microservices 0/1
- bucket4j 0/9
- PRNG-Performance 0/18
- graph_processing 0/7
- jvm-serializer 0/5
- gorilla-tsc 0/4
- jvptree 0/8
- grpc-java 1/34
- uploader 0/1
- jbrotli 0/5
- jenetics 1/43
- crl-stream 1/4
- javersion 0/8
- akarnokd-misc 18/380
- cryptolib 2/8
- hash-bench 0/3
- pinot 0/31
- controller 0/12
- fast-rng-java 0/9
- RHub 0/14
- xodus 1/66

- gral 0/1
- scalecube 0/10
- viktor 0/27
- completable-futures 0/4
- crate 0/30
- sqlg 2/8
- rtree 0/42
- hpack 1/2
- jinjava 2/4
- saga-lib 0/10
- narchy 0/2
- quick-csv-streamer 0/5
- spectator 0/38
- pcg-java 0/4
- beam 0/13
- 100-Java-Concurrency-questions 0/5
- compactmap 0/9
- indexr 2/27
- pgjdbc 0/35
- Own-Programming-Language-Tutorial 0/2
- cache2k-benchmark 0/28
- blog 0/5
- template-benchmark 0/8
- cqrs-hotel 0/6
- rut 0/5
- notification 0/6
- btreemap 0/6
- hashmapTest 0/1
- westcache 2/2
- JCTools 4/92
- neo4j-jdbc 0/3
- h2o-3 0/18
- motif 0/4
- yauaa 0/15
- benchmarks 0/27
- zipkin-reporter-java 0/10

H

Part of the analysis result for RQ2

First comes the repository name, followed by [microbenchmarks allowing CF optimizations]/[number of microbenchmarks].

Benchmarks: 3294 optimizations: 84

- RxJava 0/100
- Microservices 0/1
- bucket4j 0/9
- PRNG-Performance 0/8
- graph_processing 0/7
- jvm-serializer 0/5
- gorilla-tsc 0/4
- jvptree 0/8
- grpc-java 0/34
- uploader 0/1
- jbrotli 0/5
- jenetics 1/43
- crl-stream 0/4
- javersion 0/8
- akarnokd-misc 0/380
- cryptolib 0/8
- hash-bench 1/3
- pinot 2/31
- controller 0/12
- fast-rng-java 0/9
- RHub 0/14
- xodus 3/66
- gral 0/1
- scalecube 0/10
- viktor 0/27
- completable-futures 0/4
- crate 1/30
- sqlg 0/8
- rtree 0/42
- jinjava 0/4
- saga-lib 0/10
- quick-csv-streamer 0/5
- spectator 1/38
- pcg-java 0/4
- beam 0/13
- 100-Java-Concurrency-questions 0/5
- compactmap 0/9
- indexr 0/27
- pgjdbc 0/35
- Own-Programming-Language-Tutorial 0/2
- cache2k-benchmark 6/28
- blog 0/5
- template-benchmark 0/8
- cqrs-hotel 0/6
- rut 0/5
- notification 0/6
- btreemap 0/6
- JCTools 7/92
- neo4j-jdbc 0/3
- h2o-3 0/18
- motif 0/4
- yauaa 0/15
- benchmarks 0/27
- zipkin-reporter-java 0/10

I

Analysis result of JMH microbenchmark configurations

Below is the exact result found in the text file created by the analysis plugin used for analyzing JMH microbenchmark configurations. After each configuration combination is the number of occurrences of that combination.

@Fork(value 1,jvmArgs "-Xmx3g", "-XX:CompileThreshold=1") @Measurement(time 10,iterations 1) @Warmup(time 5,iterations 1) 18

@Fork(1) @Measurement(iterations 10) @Warmup(iterations 10) 15

@Fork(value 1,jvmArgsPrepend "-Xmx128m") @Measurement(iterations 10,time 1,timeUnit TimeUnit.SECONDS) @Warmup(iterations 10,time 1,timeUnit TimeUnit.SECONDS) 20

@Fork(value 1) @Measurement(iterations 5,time 2,timeUnit TimeUnit.SECONDS) @Warmup(iterations 5) 23

@Fork(1) @Measurement(iterations 4) @Warmup(iterations 6) 26

@Benchmark() @Measurement(iterations 10) @Warmup(iterations 20) 18

@Fork(1) @Measurement(iterations 5,time 5) @Warmup(iterations 3,time 2) 27

@Fork(1) @Measurement(iterations 10,time 1,timeUnit TimeUnit.SECONDS) @Warmup(iterations 5,time 1,timeUnit TimeUnit.SECONDS) 26

@Fork(1) @Measurement(iterations 3) @Warmup(iterations 3) 44

@Fork(1) @Measurement(iterations 5) @Warmup(iterations 5) 75

@Measurement(iterations 10) @Warmup(iterations 5) 21

@Fork(value 1) @Measurement(iterations 5,time 5,timeUnit TimeUnit.SECONDS) @Warmup(iterations 5) 47

@Fork(1) @Measurement(iterations 5,time 1,timeUnit TimeUnit.SECONDS)

@Warmup(iterations 5) 20

@Fork(1) @Measurement(iterations 5,time 1,timeUnit TimeUnit.SECONDS)
@Warmup(iterations 5,time 1,timeUnit TimeUnit.SECONDS) 56

@Fork(value 1) @Measurement(iterations 5,time 1,timeUnit TimeUnit.SECONDS)
@Warmup(iterations 5) 498

@Benchmark() @Fork(1) @Measurement(iterations 10) @Warmup(iterations 10) 21

@Fork(value 1) @Measurement(iterations 10) @Warmup(iterations 10) 25

@Measurement(iterations 10,time 1,timeUnit TimeUnit.SECONDS)
@Warmup(iterations 10,time 1,timeUnit TimeUnit.SECONDS) 58

@Fork(value 1) @Measurement(iterations 30) @Warmup(iterations 15) 32

@Benchmark() 2074

@Fork(2) @Measurement(iterations 5) @Warmup(iterations 1) 22

@Fork(1) @Measurement(iterations 10) @Warmup(iterations 20) 15

@Benchmark() @Warmup(iterations 20) 18

@Fork(3) @Measurement(iterations 5,time 1) @Warmup(iterations 10,time 1) 58

@Fork(value 1) @Measurement(iterations 25) @Warmup(iterations 10) 64

@Benchmark() @Fork(1) @Measurement(iterations 5) @Warmup(iterations 5) 64

@Fork(1) 22

J

List of repositories in the Go dataset

Below is a list of the projects that were pre-processed to create the Go dataset used for analyses. Each project is listed as [user]/[project] so in order to visit a project's page type https://github.com/[user]/[project]

- tsuna/gohbase
- couchbase/go-couchbase
- mikespook/gorbac
- blevesearch/bleve
- gonet2/wordfilter
- mdlayher/ethernet
- ory-am/hydra
- jcla1/gisp
- ikawaha/kagome
- andyleap/gencode
- nanopack/mist
- jaredfolkins/badactor
- mjibson/goread
- go-playground/form
- klauspost/dedup
- cockroachdb/cockroach
- marcusolsson/goddd
- gorilla/context
- ethersphere/go-ethereum
- micro/go-micro
- korobool/nlp4go
- arnauddri/algorithms
- anacrolix/utp
- go-ini/ini
- google/kythe
- corestoreio/csfw
- grafov/m3u8
- tobegit3hub/seagull
- dsnet/compress
- bogem/id3v2
- nsqio/nsq
- brianvoe/gofakeit
- Maratyszcza/PeachPy
- eurie-inc/echo-sample
- coreos/etcd
- restic/restic
- google/uuid
- flynn/flynn
- atlassian/gostatsd
- jolestar/go-commons-pool
- rogpeppe/godef
- google/badwolf
- cweill/gotests
- YoungPioneers/blog4go
- pingcap/tidb
- zerobotlabs/relax
- apex/log
- remind101/empire
- asciinema/asciinema
- omniscale/imposm3
- tendermint/go-p2p
- djherbis/buffer
- coreos/coreos-kubernetes
- valyala/quicktemplate
- rs/xlog
- mjibson/go-dsp

- tidwall/tile38
- muthu-r/horcrux
- mailru/easyjson
- absolute8511/nsq
- mwitkow/go-flagz
- mozilla/mig
- asdine/storm
- eBay/fabio
- miekg/dns
- youtube/doorman
- hashicorp/go-memdb
- rfjakob/gocryptfs
- taironas/tinygraphs
- ipfs/go-ipfs
- sensorbee/sensorbee
- smancke/guble
- fujiwara/fluent-agent-hydra
- cgrates/cgrates
- golang-commonmark/markdown
- danjac/podbaby
- skip2/go-qrcode
- albrow/zoom
- tsuru/planb
- robfig/revel
- tdewolff/minify
- gambol99/keycloak-proxy
- dgraph-io/dgraph
- Workiva/go-datastructures
- julienschmidt/go-http-routing-benchmark
- jjneely/buckytools
- alanctgardner/gogen-avro
- hailocab/go-geoindex
- tvdburgt/go-argon2
- klauspost/compress
- Psiphon-Labs/psiphon-tunnel-core
- nats-io/gnatsd
- mrekucci/epi
- nathanielc/morgoth
- ryanchapman/go-any-proxy
- couchbase/sync_gateway
- influxdata/kapacitor
- micromdm/micromdm
- klauspost/pgzip
- StabbyCutyou/buffstreams
- coreos/dbtester
- gcc-mirror/gcc
- montanaflynn/stats
- Go-zh/go
- elodina/go_kafka_client
- go-pg/pg
- lunixbochs/struc
- sajari/docconv
- guregu/dynamo
- looplab/eventhorizon
- square/inspect
- siddontang/xcodis
- pierrre/imageserver
- fabric8io/gofabric8
- litl/shuttle
- golang/gofrontend
- ziutek/mymysql
- antha-lang/antha
- jpillora/cloud-torrent
- gocraft/health
- trivago/gollum
- jkl1337/go-chromath
- smartystreets/go-disruptor
- h2non/gentleman
- google/gopacket
- cpmech/gosl
- paulmach/go.geo
- decred/dcrd
- digitalocean/doctl
- square/squalor
- goshawkdb/server
- jackc/pgx
- Shopify/ejson
- robertkrimen/otto
- uber-go/zap
- stargrave/govpn
- valyala/ybc
- sourcegraph/appdash
- bmatsuo/lmdb-go
- golang/go
- rana/ora
- GianlucaGuarini/go-observable
- gobwas/glob
- SebastiaanKlippert/go-wkhtmltopdf
- olivere/elastic
- valyala/fasthttp

- zhenjl/encoding
- nf/sigourney
- flynn/go-tuf
- go-interpreter/ssainterp
- getlantern/lantern
- patrickmn/go-cache
- miekg/coredns
- chihaya/chihaya
- weaveworks/scope
- polydawn/repeatr
- tendermint/tendermint
- eris-ltd/eris-db
- Comcast/rulio
- golang/exp
- biogo/biogo
- Lafeng/deblocus
- go-playground/pool
- cyanly/gotrade
- gocraft/dbr
- EngoEngine/engo
- kavu/go_reuseport
- proullon/ramsql
- exercism/xgo
- buger/jsonparser
- aerospike/aerospike-client-go
- vladimirvivien/automi
- Masterminds/go-in-practice
- mgutz/dat
- soundcloud/roshi
- gin-gonic/gin
- Phillipmartin/gopassivedns
- centrifugal/centrifugo
- yahoo/coname
- blampe/goat
- mesosphere/mesos-dns
- derekparker/trie
- zond/god
- cznic/ql
- vmihailenco/msgpack
- opentracing/basictracer-go
- tylertreat/BoomFilters
- cloudflare/golibs
- square/go-jose
- riscv/riscv-go
- prataprc/goparsec
- ulikunitz/xz
- tideland/golib
- uber/tchannel-go
- xjdrew/kone
- go-kit/kit
- revel/revel
- skynetservices/skydns
- go-gl/mathgl
- 01org/ciao
- uber/ringpop-go
- ibuildthecloud/systemd-docker
- cosiner/gohper
- nelhage/gojit
- hashicorp/nomad
- gocql/gocql
- eleme/banshee
- hyperhq/hypercli
- mat/besticon
- SoftwareDefinedBuildings/btrdb
- mission-liao/dingo
- rlmcpherson/s3gof3r
- codetainerapp/codetainer
- GoogleCloudPlatform/cloudsql-proxy
- ljfranklin/terraform-resource
- Knetic/govaluate
- tcolgate/mp3

K

Subset of the analysis result for RQ4 on the Java dataset

Below is part of the analysis result for RQ4. Only a small part of the analysis result is shown to keep the number of pages down but at the same time give the reader an understanding of how the analysis results look like. The ratio given for the top five committers are their respective [benchmark commits]/[all commits]. The list in the end for each repository consisting of numbers represent the number of commits affecting microbenchmark files for each benchmarker in the repository.

committers: 2140
benchmarkers: 323

caffeine has 1 benchmarkers out of 18 committers
Top 5 committers of caffeine are
caffeine Ben Manes: 51/982
caffeine gilga1983: 0/12
caffeine GitHub: 0/5
caffeine Julian Vassev: 0/3
caffeine Ken Dombeck: 0/2
Commits per developer affecting microbenchmark files in caffeine are 51

RxJava has 1 benchmarkers out of 126 committers
Top 5 committers of RxJava are
RxJava Ben Christensen: 0/2047
RxJava David Karnok: 0/575
RxJava akarnokd: 0/510
RxJava GitHub: 2/365
RxJava zsxwing: 0/274
Commits per developer affecting microbenchmark files in RxJava are 2

JCTools has 11 benchmarkers out of 32 committers
Top 5 committers of JCTools are
JCTools nitsanw: 52/500

JCTools Doug Lawrie: 4/63
JCTools GitHub: 7/44
JCTools Richard Warburton: 1/28
JCTools Nitsan Wakart: 0/19
Commits per developer affecting microbenchmark files in JCTools are 52,4,1,7,1,1,3,1,3,1,1

logging-log4j2 has 12 benchmarkers out of 32 committers
Top 5 committers of logging-log4j2 are
logging-log4j2 Gary Gregory: 28/2062
logging-log4j2 rpopma: 169/1928
logging-log4j2 Gary D. Gregory: 1/1463
logging-log4j2 Ralph Goers: 27/1224
logging-log4j2 ggregory: 27/1049
Commits per developer affecting microbenchmark files in logging-log4j2 are 1,27,1,1,27,169,1,28,1,33,6,32

directory-kerby has 6 benchmarkers out of 20 committers
Top 5 committers of directory-kerby are
directory-kerby Drankye: 3/467
directory-kerby plusplusjiajia: 4/357
directory-kerby Colm O hEigeartaigh: 1/251
directory-kerby Kai Zheng: 1/124
directory-kerby Emmanuel Lécharny: 2/62
Commits per developer affecting microbenchmark files in directory-kerby are 2,3,1,1,4,1

L

The analysis result for RQ5 on the RxJava repository

Below is part of the analysis result for RQ5. Only RxJava is shown to keep the number of pages down but at the same time give the reader an understanding of how the analysis results look like. Listed is the names of microbenchmarks and the number of times the files containing them has been modified since the microbenchmarks was created. A zero means that the microbenchmark has only been created and not modified after that.

In RxJava there are 97 microbenchmarks

flatMapTwoNestedSync has been modified 0 times
mergeNSyncStreamsOfN has been modified 0 times
bpRangeMapJust has been modified 0 times
bpRangeMapRange has been modified 0 times
subscribeOnSingle has been modified 1 times
singleJust has been modified 0 times
obsMaybe has been modified 0 times
subject1 has been modified 0 times
observeOnFlowable has been modified 1 times
unbounded1 has been modified 0 times
subscribeOnFlowable has been modified 1 times
flatMapXRange has been modified 0 times
subscribeOnObservable has been modified 1 times
internal has been modified 0 times
maybe has been modified 0 times
pipelineObservable has been modified 1 times
flowMaybe has been modified 0 times
flowFlatMapCompletable0 has been modified 0 times
range has been modified 0 times
rangeSync has been modified 0 times
groupBy has been modified 0 times
observableInner has been modified 0 times
nbpRangeMapRange has been modified 0 times
observeOnCompletable has been modified 1 times

```
observableBlockingFirst has been modified 0 times
rangeFlatMap has been modified 0 times
rangeObservableFlatMap has been modified 0 times
obsFlatMapSingleAsObs1 has been modified 0 times
flowFlatMapSingle1 has been modified 0 times
subscribeOnMaybe has been modified 1 times
single has been modified 0 times
flowFlatMapIterableAsFlow0 has been modified 0 times
pipelineSingle has been modified 0 times
mergeTwoAsyncStreamsOfN has been modified 0 times
flowSingle has been modified 0 times
flowFlatMapIterableAsFlow1 has been modified 0 times
flowable has been modified 3 times
observable has been modified 3 times
subscribeOnCompletable has been modified 1 times
flowFlatMapIterable1 has been modified 0 times
bpRange has been modified 0 times
flowFlatMapIterable0 has been modified 0 times
flowFlatMapMaybe1 has been modified 0 times
obsFlatMapMaybeAsObs0 has been modified 0 times
obsSingle has been modified 0 times
obsFlatMapMaybe0 has been modified 0 times
flowFlatMapMaybe0 has been modified 0 times
obsFlatMapObservable0 has been modified 0 times
obsFlatMapMaybe1 has been modified 0 times
obsFlatMapObservable1 has been modified 0 times
flowFlatMapCompletableAsFlow0 has been modified 0 times
pipelineCompletable has been modified 1 times
oneStreamOfNthatMergesIn1 has been modified 0 times
obsFlatMapMaybeAsObs1 has been modified 0 times
observeOnMaybe has been modified 1 times
flowFlatMapSingleAsFlow1 has been modified 0 times
flowFlatMapMaybeAsFlow1 has been modified 0 times
flowFlatMapMaybeAsFlow0 has been modified 0 times
rangeObservableFlatMapJust has been modified 0 times
observableBlockingLast has been modified 0 times
rangeObservable has been modified 0 times
flatMap has been modified 0 times
obsFlatMapCompletableAsObs0 has been modified 0 times
obsFlatMapIterable0 has been modified 0 times
external has been modified 0 times
flatMapIntPassthruAsync has been modified 0 times
obsFlatMapIterable1 has been modified 0 times
unbounded1k has been modified 0 times
mergeNAsyncStreamsOfN has been modified 0 times
bounded1k has been modified 0 times
```

```
pipelineMaybe has been modified 1 times
unbounded1m has been modified 0 times
bounded1m has been modified 0 times
merge1SyncStreamOfN has been modified 0 times
flowableBlockingLast has been modified 0 times
obsFlatMapIterableAsObs0 has been modified 0 times
flowFlatMapFlowable1 has been modified 0 times
obsFlatMapCompletable0 has been modified 0 times
flatMapIntPassthruSync has been modified 0 times
parallel has been modified 0 times
nbpRange has been modified 0 times
flowFlatMapFlowable0 has been modified 0 times
nbpRangeMapJust has been modified 0 times
obsFlatMapIterableAsObs1 has been modified 0 times
bounded1 has been modified 0 times
subject1m has been modified 0 times
obsFlatMapSingle1 has been modified 0 times
observeOnObservable has been modified 1 times
pipelineFlowable has been modified 1 times
mergeNSyncStreamsOf1 has been modified 0 times
flowableInner has been modified 0 times
subject1k has been modified 0 times
rangeFlatMapJust has been modified 0 times
observeOnSingle has been modified 1 times
flowableBlockingFirst has been modified 0 times
completable has been modified 0 times
singleJustMapJust has been modified 0 times
```

M

Abbreviations

MSR - Mining Software Repositories
JMH - Java Microbenchmark Harness
AST - Abstract Syntax Tree
VCS - Version Control System
DCE - Dead Code Elimination
DSL - Domain Specific Language
JRE - Java Runtime Environment
SPI - Service Provider Interface
CSV - Comma-Separated Value
BTS - Bug Tracking System
JIT - Just-In-Time
CF - Constant Folding