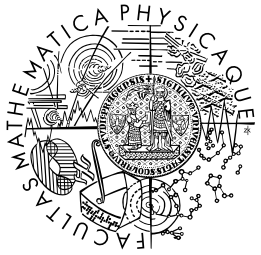


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Petr Stefan

Java Performance Testing For The Masses

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	prof. Ing. Petr Tůma, Dr.
Study programme:	Computer Science
Study branch:	ISS

Prague 2018

I declare that I carried out all work on this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague on May 8, 2018 Bc. Petr Stefan

Title: Java Performance Testing For The Masses

Author: Bc. Petr Stefan

Department: Department of Distributed and Dependable Systems

Supervisor: prof. Ing. Petr Tůma, Dr., Department of Distributed and Dependable Systems

Abstract: Java is a major platform for performance sensitive applications. Unit testing of functionality has already become a common practice in software development; however, the amount of projects employing performance tests is substantially lower. A comprehensive study in combination with a short survey among developers is made in order to examine the current situation in open-source projects written in Java. Results show that suitable tools for measurements exist, but they are hard to use or the outputs are difficult to understand. To improve the situation in favor of performance evaluation a set of user friendly tools for collecting, comparing and visualizing the data is designed, implemented, and verified on a sample Java project.

Keywords: Java, performance, SPL, JMH, unit testing, software development process

I would first like to thank my thesis supervisor prof. Petr Tůma, who supported the research with aspiring guidance, numerous valuable ideas and comments and a good deal of constructive criticism. I greatly appreciate his involvement that resulted in publishing the part of the thesis as a conference paper.

I would also like to express my very profound gratitude to my parents and to my girlfriend for supporting me throughout my study. Finally, I highly value all the efforts of my professors and my Alma Mater for all the knowledge I hope to use and evolve in my professional life. Thank you.

Contents

Introduction

Java is a major platform for performance-sensitive applications. Among major examples are frameworks for distributed computing – Hadoop[1] and Flink[2], or container servers – Tomcat[3] and Glassfish[4]. Maintaining good performance over time is therefore often important. One way to achieve this is continuous performance testing.

However, performance testing of Java programs is far from trivial, because the characteristics of the underlying runtime make the performance behavior more complex than that of languages compiled to native code of the target CPU architecture [1]. Common factors affecting such behavior are the nondeterminism of the garbage collector, thread scheduling, JIT (Just-In-Time) compilation or VM (Virtual Machine) optimizations based on time sampling of the running code. To avoid incorrect conclusions about performance, a suitable testing methodology with statistically rigorous data processing must be used.

Common wisdom of software development is that systematic unit testing of the source code helps reach and maintain high quality of the application. This statement is validated by some of the major software companies [2] as well as by analysis of known failures caused by insufficient testing [3]. There is evidence that earlier testing can reduce costs to repair the discovered flaws [4].

A similar statement holds for the performance testing of the applications [5]. Early testing and fine tuning the applications often leads to massive performance gains, but anecdotal evidence suggests that performance testing is not as common as unit testing of functionality. The adoption rate of the supporting tools and frameworks is not as high as we expect despite the fact that many exists for over a decade.

Performance testing can be done on several levels of granularity, including evaluation of architectural performance models [6], system performance testing and (micro)benchmarking. This work is interested only in the benchmarking since the benchmarking frameworks are similar to the frameworks for unit testing of functionality. This could lead to a better understanding of the tools by the developers that already use the unit testing frameworks.

The term "microbenchmark" is nowadays overused and its meaning is a little blurred. We go along with Ehliar and Liu [7] who define microbenchmarks as performance indicators of typical tasks for a given application. Usually it is a specific part of a larger application, but in a few situations it can also be a standalone application itself.

Since the performance testing tools are not much used globally, we want to understand the causes of this state and try to overcome the obstacles to help the developers with faster adoption of these tools. Papers [8] and [9] give a general overview of the topic. The main problems seems to be more expensive test execution (in terms of time), issues with testing automation, complicated setup of the tools and lack of knowledge for data interpretation.

[1] `http://hadoop.apache.org/`
[2] `https://flink.apache.org/`
[3] `http://tomcat.apache.org/`
[4] `https://javaee.github.io/glassfish/`

These problems point to a list of topics that need to be addressed in this thesis. We set four goals in order to provide a complete performance testing assistance for the developers. Fulfilling the goals involve solving problems in the following activities:

- writing performance tests,
- gathering and storing the data,
- automated evaluation of performance changes in the data,
- visualizing the data.

To confirm that real projects have matching issues with the presented goals, a study mining Java open-source code repositories on GitHub is provided. A supplementary developer survey gives us their ideas about testing, but as shown by Daka and Fraser [10], the responses are not necessarily accurate.

We believe that most of the goals can be addressed by developing a set of simple-to-use tools to make performance testing of Java projects more user friendly and hopefully more popular for the software developers. This set of tools includes a build system plugin for running the tests and saving their outputs, a tool for computing statistical differences between the performance data to find regressions, and a tool for interactive visualization of the measured data to give the users a hands-on experience with the actual performance of the application.

To confirm that the proposed tools are usable in real world applications and actually bring an improvement in the testing workflow, we have chosen a sample project and used the tools to verify that using the tools during the development could potentially find hidden issues or find existing issues sooner than without any tool support.

The thesis is structured as follows. In Chapter 1 we present the state-of-the-art tools for unit testing of functionality and performance testing and their usage in open-source projects hosted on GitHub. Together with the supplementary developer survey, we find possible problems in the performance testing workflow and design a framework to help reduce them. The end of the chapter addresses the related work. Chapter 2 presents the design and implementation choices for our framework. Chapter 3 presents the practical validation of the framework on a sample Java project. The conclusion summarizes our results and discusses the future development and adoption process of the presented framework.

Appendix A shows the transcript of the form used for our developer survey. The user documentation of the presented framework is in Appendix B. Note that this part of the thesis does not follow the strict rules of academic text and instead uses more common style suitable for writing the user documentation. Appendix C provides the API documentation for the interaction between the evaluation and the visualization components. Appendix D contains a description of the demo prepared for the reviewer of this thesis.

1. Analysis

To improve the process of performance testing by microbenchmarks, we want to understand the present testing workflow with its strengths and weaknesses first. The data about contemporary and past workflow patterns can be obtained in several ways:

- scanning sources of open-source projects,
- surveying open-source developers,
- surveying software companies,
- analyzing questions about performance on developer websites.

Open-source projects allow everyone to view their source code and investigate the used frameworks in detail. This allows us to collect the relevant data about how the performance testing frameworks are used in many applications. The major platform for open-source projects is GitHub[1], but GitLab[2], BitBucket[3] or SourceForge[4] also have a significant user base. Our choice is to use the biggest platform available. GitHub provides a web API to find projects matching given criteria, so projects using the Java language can be programmatically discovered and saved for further processing.

The results of a developer survey depend on the willingness of the developers to answer the questionnaire. A crucial part is to target only those developers who made or altered some of the performance tests. They can provide sensible results about how they are testing the performance, how are the results processed, why were the used tools chosen, etc. The questionnaire needs to be fairly short in order not to take too much time from the developers. We made a small survey to complement our knowledge about the open-source testing and to support the results from the GitHub projects.

In case of a company survey, we do not believe that we could entice more than a few software corporations to provide meaningful responses with quality results to meet the demands of this text. Similar imprecise results are expected from analyzing questions on Q&A websites such as StackOverflow[5], where the quality differs extensively as shown by paper [11]. Thus these two types of analysis are not pursued further.

A serious question concerns the validity of our results on a wider scale if proprietary code is omitted. We guess that the proprietary code could be slightly more tested, but it is hard to get real evidence for comparison. Open-source software development has some specifics [12] compared to commercial closed-source development. The developer team of an open-source project is mostly distributed, without personal contact, and the contributors select the features they want to work on depending on their preferences and skills. They also have more reviewed code, better modularity, faster releases of minor versions and more unified coding standards. Traditional development tends to happen in centralized

[1] `https://github.com/`
[2] `https://about.gitlab.com/`
[3] `https://bitbucket.org/product`
[4] `https://sourceforge.net/`
[5] `https://stackoverflow.com/`

teams lead by the project managers with precise design, good documentation or a fixed schedule of releases. There is no data for direct comparison of overall performance for these two types of project development.

To understand the concept of modern tools for unit and performance testing, the representatives of the most used frameworks in the industry are presented with brief description and a usage example.

1.1 Unit Testing of Functionality

The most important aspect of every program is its correct functionality. Programs giving wrong results or crashing could be very harmful in some cases, not just useless. Unit testing can prevent these situations by taking every piece of code and testing its outputs for common and corner cases of inputs. The advantage of this approach is that the tests are executed for every new change in the code, so the bugs introduced in these changes are found instantly. Unit testing became popular and formed a standard for most projects.

With the unit testing movement other projects appear to help developers keep code correctness in mind. Most notably there are the continuous integration (CI) servers, which usually build the code after every push to a remote repository and execute the test code. The responsible developer is notified about failures and further actions can take place as well, for example automatic deployment is blocked. Popular representatives are Travis CI[6], AppVeyor[7] or Jenkins CI Server[8].

Only two existing unit testing frameworks for Java from the list became more popular – TestNG and JUnit. The following sections give a brief overview of both of them.

1.1.1 JUnit

JUnit[9] is probably the most popular unit testing framework ever. The original idea of xUnit frameworks comes from SUnit (Smalltalk unit testing framework), but the most popular representative is the Java version. Many other frameworks even for different programming languages are inspired by JUnit and try to follow a similar user interface. The tests are defined by a simple annotation, `@Test`, and in many cases this is the only requirement for the test. Inside each test, a block of code is executed and the results are checked against the expected values using `assertX` functions (`assertTrue`, `assertEquals`, etc.).

Usually, the JUnit framework is supported by the IDE, where there are buttons to run one or all tests and the outputs are displayed directly. Command-line use is also possible. The very active development of this project since 2000 resulted in version 5 in November 2017. An example test is in Listing 1.1.

[6] `https://travis-ci.org/`
[7] `https://www.appveyor.com/`
[8] `https://jenkins.io/`
[9] `http://junit.org/`

```java
class StandardTests {
  @BeforeAll
  static void initAll() {}

  @BeforeEach
  void init() {}

  @Test
  void succeedingTest() {}

  @Test
  void failingTest() {
    fail("a failing test");
  }

  @Test
  @Disabled("for demonstration purposes")
  void skippedTest() { /* not executed */ }

  @AfterEach
  void tearDown() {}

  @AfterAll
  static void tearDownAll() {}
}
```

Listing 1.1: JUnit example test

1.1.2 TestNG

TestNG[10] is a unit testing framework heavily inspired by JUnit and extended with support for additional features like functional testing, integration testing or end-to-end testing. The testing process can be started from plugins for popular IDEs, providing visual results, or directly from the ANT, Maven and Gradle command-line build systems. The default output formats are HTML and XML, but newer versions provide the Reporter API for third-party output generators as well. TestNG is still in active development which started in 2010.

The unit tests are configured by multiple annotations. An example testing class is in Listing 1.2.

1.2 Practice of Unit Testing

There are several practices for writing and managing the unit tests. For all of them an initial setup of testing framework of a choice is required, but both presented frameworks follows simple steps to integrate themselves into the variety of Java build systems. The next thing is to write some code and the corresponding unit tests. There are two main methods when to write the actual tests.

The test-driven development (TDD) means that tests are written prior to application code, which can help to design the interfaces as well as achieve high coverage of the unit tests. The other way is to write the tests after the application

[10] http://testng.org/doc/

```java
public class SimpleTest {
  @BeforeClass
  public void setUp() {
    // code that will be invoked when
    // this test is instantiated
  }

  @Test(groups = { "fast" })
  public void aFastTest() {
    System.out.println("Fast test");
  }

  @Test(groups = { "slow" })
  public void aSlowTest() {
    System.out.println("Slow test");
  }
}
```

Listing 1.2: TestNG example test

code which seems to have no obvious benefits, but it is the option used more in our opinion.

When both the application code and the tests are ready, they have to be evaluated. Commonly used examples are numerous IDE integration plugins which provide the interface for running the tests and a visual representation of the results. Another popular way is to use the remote CI server, where the code is regularly sent (for instance after push to version system) and evaluated in a text console. In case of failure, the reports are delivered to the author of the last change via email.

These two methods can be combined so the developer usually checks the correct functionality of his/her changes before a commit, and the CI server verifies the correctness after the push to the remote repository. The complete execution of all unit tests of functionality should take at most a few minutes.

At this point the testing environment and the evaluation workflow are set and every new code change is processed in the same way as described above. To sum up, the testing pipeline consists of writing the test and the actual code (in arbitrary order), testing own changes locally and pushing them to the remote repository, which may trigger a check in the CI server.

1.3 Performance Testing

Practical performance of Java programs can be observed in two different ways – by monitoring the running application in a real deployment with real workload, or by writing synthetic tests for smaller pieces of code and monitoring predefined criteria (execution time, consumed memory, etc.). The first method gets more real results [13], because it tracks the whole application running in its target environment working on real data. The measurements can run for a long time, even for the whole application runtime, but on the downside a performance problem may not be easy to track down to the code. On the other hand microbenchmark-

ing results tend to be more artificial, so drawing conclusions from them has to be done with care.

The described monitoring scenario relies on profiler tools. The JVM provides support for such tools with the Java Management Extensions (JMX)[11]. This technology provides information about performance and resource consumption on both local and remote applications running on the Java platform. Some of the popular tools are JConsole[12] (a graphical tool included in the Java Development Kit (JDK)), JProfiler[13] or VisualVM[14]. The profilers are often paid and target the enterprise environment. This kind of tools is more suitable for monitoring server applications with long uptime, than for other application types.

Synthetic microbenchmarks are an alternative to profiling. They target only small pieces of code at a time, running in a loop on predefined data sets. This approach helps tracking performance during the development process when each code change can be immediately tested by the corresponding microbenchmark and every little difference is captured. However, the testing data is often different from the real world production data and the workload could be different as well, so it is possible that the results do not fully correspond with the real performance of the application.

This thesis explores only microbenchmarking because the tests run relatively short time, thus they can be run more often during the development process. They are often similar to the unit tests and also they tend to give more stable results than profiling. A description of some of the most popular microbenchmarking frameworks follows.

1.3.1 Caliper

Caliper[15] is a tool focused on Java microbenchmarks being developed at Google since 2008. It provides an integration with an online results archive, which can help users with interpreting the results, as well serve as a guide describing the best practices for the measurements. Android support allows running the tests directly on mobile devices, but it may not work in the latest version of Caliper. The simplest Caliper benchmark according to the official web tutorial is in Listing 1.3.

```
1 public static class Benchmark1 {
2   @Benchmark
3   void timeNanoTime(int reps) {
4     for (int i = 0; i < reps; i++) {
5       System.nanoTime();
6     }
7   }
8 }
```

Listing 1.3: Caliper example test

[11] http://www.oracle.com/technetwork/articles/java/javamanagement-140525.html
[12] http://openjdk.java.net/tools/svc/jconsole/
[13] https://www.ej-technologies.com/products/jprofiler/overview.html
[14] https://visualvm.github.io/
[15] https://github.com/google/caliper/wiki/ProjectHome

The usual way to include Caliper into a project is through a Maven dependency. After compilation, the measurements are started by the provided `com.google.caliper.runner.CaliperMain` main class. The results are automatically uploaded to the web archive and saved to the local filesystem in JSON format. The overall measurement time is controlled by the framework itself. The Maven dependency snippet for Caliper is in Listing 1.4.

```
<dependency>
  <groupId>com.google.caliper</groupId>
  <artifactId>caliper</artifactId>
  <version>v1.0-beta-2</version>
</dependency>
```

Listing 1.4: Caliper Maven dependency

1.3.2 ContiPerf

ContiPerf[16] is a utility for writing performance tests alike regular unit tests. The performance tests are extensions of the JUnit framework tests and the same code can be used for both purposes. Supported annotations can define measurement properties and expectations for the results. An HTML report with static data visualization is generated for every invocation of the measurements and the raw values can be exported in the CSV format. The utility was developed between 2010 and 2015. There is no evidence of any activity since then.

A simple ContiPerf benchmark is in Listing 1.5. The tool can be easily integrated with the Maven build system. The dependency snippet for executing the framework at the *test* target is in Listing 1.6.

```
public class SimpleTest {
  @Rule
  public ContiPerfRule i = new ContiPerfRule();

  @Test
  @PerfTest(invocations = 1000, threads = 20)
  @Required(max = 1200, average = 250)
  public void sleepAWhile() throws Exception {
    Thread.sleep(100);
  }
}
```

Listing 1.5: ContiPerf example test

1.3.3 Japex

Japex[17] is another tool for writing microbenchmarks with a philosophy similar to the JUnit framework. But Japex is not a direct extension of JUnit. The test suite configuration is defined in an XML file, the output includes HTML

[16] `http://databene.org/contiperf`
[17] `https://github.com/kohsuke/japex`

```xml
<dependency>
  <groupId>org.databene</groupId>
  <artifactId>contiperf</artifactId>
  <version>2.4.3</version>
  <scope>test</scope>
</dependency>
```

Listing 1.6: ContiPerf Maven dependency

reports with charts and timestamped XML results. For each tested algorithm, a separate Java "driver class" has to exist including the code with the control methods such as `prepare`, `warmup`, `run`, or `finish`. This tool was developed in 2005 to 2011 and now seems to be abandoned. Even the project website `https://japex.dev.java.net/` is left inaccessible.

A sample configuration for a test suite with two algorithms (`XDriver`, `YDriver`) and two test cases (with different arguments) is presented in Listing 1.7. The tests can be executed by the Maven plugin, which can be added as a dependency to the `pom.xml` file as shown in Listing 1.8.

```xml
<testSuite name="Sample Test Suite" xmlns="http://www.sun.com/
    japex/testSuite">
  <param name="libraryDir" value="lib"/>
  <param name="japex.classPath" value="${libDir}/classes"/>
  <param name="japex.warmupTime" value="10"/>
  <param name="japex.runTime" value="10"/>

  <driver name="XDriver">
    <param name="Description" value="Driver for X parser"/>
    <param name="japex.DriverClass" value="com.foo.XDriver"/>
  </driver>
  <driver name="YDriver">
    <param name="Description" value="Driver for Y parser"/>
    <param name="japex.driverClass" value="com.foo.YDriver"/>
  </driver>

  <testCase name="file1.xml">
    <param name="japex.inputFile" value="data/file1.xml"/>
  </testCase>
  <testCase name="file2.xml">
    <param name="japex.inputFile" value="data/file2.xml"/>
  </testCase>
</testSuite>
```

Listing 1.7: Japex example test configuration

1.3.4 JMH

JMH[18] is a Java harness for writing and analysing microbenchmarks. It is a part of the OpenJDK project and is supported by the Oracle company. The performance tests are written like JUnit tests, only a different set of annotations is used. A sample benchmark is shown in Listing 1.9.

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

```
<dependency>
  <groupId>com.sun.japex</groupId>
  <artifactId>japex-maven-plugin</artifactId>
  <version>1.2.4</version>
  <scope>test</scope>
</dependency>
```

Listing 1.8: Japex Maven dependency

```
public class Benchmarks {
  @Benchmark
  @BenchmarkMode(Mode.Throughput)
  @OutputTimeUnit(TimeUnit.SECONDS)
  public void measureThrgpt() throws InterruptedException {
    TimeUnit.MILLISECONDS.sleep(100);
  }
}
```

Listing 1.9: JMH example test

The test configuration is highly customizable using annotations or command-line arguments at runtime. A recommended benchmarking project is created from the JMH Maven archetype. This procedure sets up proper dependencies and allows building a self-contained JAR bundle with all of the tests and the JMH code. This could be useful for measurements on a separate computer.

The results can be saved in a human-readable text or in one of the machine readable formats, CSV, SCSV, and JSON. Third-party data from optional JVM profilers can be included as secondary metrics into the results. It is possible to use a custom main method and a Java API to tweak the framework, but the flexibility and simple uniform usage would be lost. This project started in 2013 and has been actively developed since. A community of users created many plugins for Gradle, Jenkins, IntelliJ IDEA, and other projects.

1.3.5 JUnitPerf

JUnitPerf[19] is an extension of the JUnit 3 testing framework which uses the decorator pattern to extend the original API. Unit testing methods are executed as timed performance tests through a wrapper class defining other properties. This encapsulation allows using the same code for both unit and performance testing. However, it does not support important features like warmup or multithreaded testing. The project was developed in 2008 to 2010 and now is not maintained. The suggested alternative is Caliper.

The project is compiled using Apache Ant. The installation instructions suggest downloading a compiled JAR file and placing it on the classpath. Each test suite has a separate main method for execution. The results are printed to the standard output in human-readable format. An example timed test is shown in Listing 1.10.

[19] https://github.com/clarkware/junitperf

```
1 public class ExampleTimedTest {
2   public static Test suite() {
3     long maxElapsedTime = 1000;
4     Test testCase =
5         new ExampleTestCase("testOneSecondResponse");
6     Test timedTest = new TimedTest(testCase, maxElapsedTime);
7     return timedTest;
8   }
9
10  public static void main(String[] args) {
11    junit.textui.TestRunner.run(suite());
12  }
13 }
```

Listing 1.10: JUnitPerf example test

1.4 Practice of Performance Testing

Compared to unit testing of functionality, there are no established procedures that describe the common workflow of performance testing. We collect a large survey to overcome the lack of data, and create our own fresh statistics from number of existing projects.

The results presented in this section were originally presented as the conference paper [14] as joint effort of all its authors.

1.4.1 GitHub Statistics

To evaluate usage of the introduced tools, we made a Python crawler script for GitHub. It uses the GitHub Search API v3[20] through the PyGithub library[21]. For further processing, a list of all Java repositories was created. To eliminate the many nearly empty repositories (excluding forks gives approximately 2.4 million repositories), we perform a query searching for at least two forks and nonempty size. Because the API is limited to return a maximum of one thousand items at a time, we use a binary search algorithm based on forks, stars and project size. A total of 99 019 such repositories was found.

There is a threat that the repository list might not be complete. Limitation of the API allows only ten requests per minute, so the overall crawling takes about one day in which new repositories may be created or existing repositories deleted. Also, there is no way to ensure the completeness of the list, because the partial response indicator flag is set to a negative answer when the results are incomplete or the results are complete and the search query timed out. However, small differences from the exact state are not important for our summarizing study.

The original study was run in 2016. Since then, we have attempted to update the results, however, the GitHub Search API gives us wrong results compared to our previous findings. We have also tried to analyze the GitHub Archive[22] collecting metadata of all repositories in The Google Cloud, but we managed to only

[20] https://developer.github.com/v3/search/
[21] https://github.com/PyGithub/PyGithub
[22] https://www.gharchive.org/

get 387 720 Java repositories (excluding forks), which is an order of magnitude less than the first search. We are not able to find the cause of the observed divergence, so we present the data from our first search, which are the most complete we could get.

Every repository from the list was cloned (about 3 TB of data) and analyzed for usage patterns of the presented tools and frameworks. The Java sources were parsed and checked for imported packages and used annotations, listed in Table 1.1. This approach gives reliable results, because importing the package is required for using each of the frameworks. Although obscure practices could invoke tests without import statements, the number of users of these techniques is estimated as very low. Also, importing the package does not necessarily mean that the project is using proper benchmarking, but it is unlikely that there are many projects with this kind of unused dependency. The number of tests in each project is calculated by counting the used test markers of the framework.

Framework	Base package	Test marker
JUnit	`org.junit`	`@Test`
TestNG	`org.testng`	`@Test`
Caliper	`com.google.caliper`	`@Benchmark`
ContiPerf	`org.databene.contiperf`	`@PerfTest`
Japex	`com.sun.japex`	`JapexDriverBase`
JMH	`org.openjdk.jmh`	`@Benchmark`, `@GenerateMicroBenchmark`
JUnitPerf	`com.clarkware.junitperf`	`TimedTest`, `LoadTest`

Table 1.1: Packages and annotations used to detect the unit and performance testing frameworks. Each framework is detected by the presence of the base package, one test marker counts as presence of one test.

Table 1.2 gives an overview of the repositories that use one of the surveyed frameworks. The data shows that the most popular framework is JMH, but still with only 278 projects, which means that the framework is very rare. Because other performance frameworks have at least one order of magnitude less projects, it is sensible to further focus on JMH as the most used representative. On the other hand, unit testing of functionality is quite popular as about a third of all projects adopted one of the two presented frameworks.

In addition to the established benchmarking tools, we searched for a proprietary method of benchmarking. Proprietary benchmarking means custom time measurements of a piece of code where the system clock is queried twice in one block of code and the difference of these values is calculated. However, these hits are just potential performance tests where further human classification is required to filter out false positives.

When speaking of proprietary benchmarking, a manual classification of all projects is not feasible and an automated script cannot categorize the reason of querying the clock, so a random set of 1000 projects was picked and manually examined. From this set, 332 projects are querying the system clock using one of the functions `System.nanoTime()`, `System.currentTimeMillis()` and variants of `ThreadMXBean.get*Time()`. To reduce the error ratio, the classification was

Framework	Repositories	Relative usage
JUnit 4	30871	31.177 %
TestNG	2053	2.073 %
Caliper	12	0.012 %
ContiPerf	17	0.017 %
Japex	52	0.053 %
JMH	278	0.281 %
JUnitPerf	11	0.011 %
Total	99019	100 %

Table 1.2: Java test framework usage on GitHub.

made independently by two people who assigned each project a category. Then, similar categories from both people were merged (about 70% of their answers matched). The results are presented in Table 1.3. Proprietary benchmarking is used in about 3.4% of all the examined projects, which is low on global scale, but also an order of magnitude more than the JMH framework. On top of that, other categories may be connected to tracking performance for example in logs, but this cannot be considered as unit testing of performance.

Usage	Count	99 % CI
Timeout handling	125	99 – 154
Logging of durations	133	107 – 163
Querying calendar time	122	97 – 151
Event scheduling, GUI	89	67 – 115
Randomization, unique naming	65	47 – 88
Proprietary benchmarking	34	21 – 52
Custom timer infrastructure	29	17 – 46

Table 1.3: Usage of `System.nanoTime` and `System.currentTimeMillis`. Projects that span multiple categories are counted multiple times.

The main problem of proprietary benchmarking is that it is done in non-uniform way and often badly since the Java runtime needs some time for instance to optimize the code. This implies that using a dedicated framework is always a better choice. Since JMH is the only one with non-trivial user base, we examine it in more detail. Figure 1.1 shows the version of JMH used by each project, Figure 1.2 shows the adoption speed of new releases for all projects using JMH. The graphs illustrate the tendency to use newer project versions, but the adoption speed is not very good including for very active projects.

Figure 1.3 compares the number of tests per project for JUnit and JMH. The first unit tests are created just at the project beginning and new ones are added during the whole lifetime, with slightly more speed towards the beginning. This is expected because young projects create more new (core) functionality that needs to be tested. However, the first performance tests are seen much later and then the curve follows a similar shape as for unit testing. This means that even for

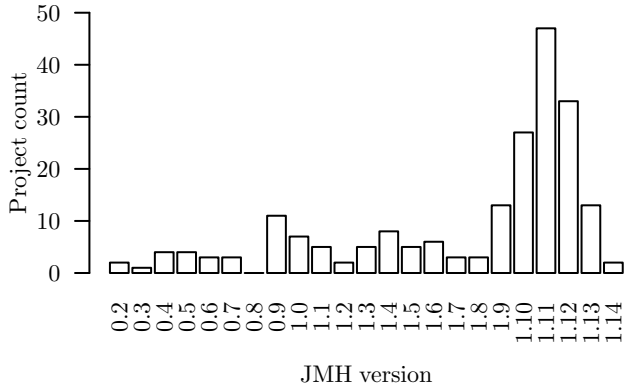


Figure 1.1: JMH versions used by projects. Versions 1.11, 1.12, 1.13 and 1.14 were released on January, April, July and September 2016, respectively.

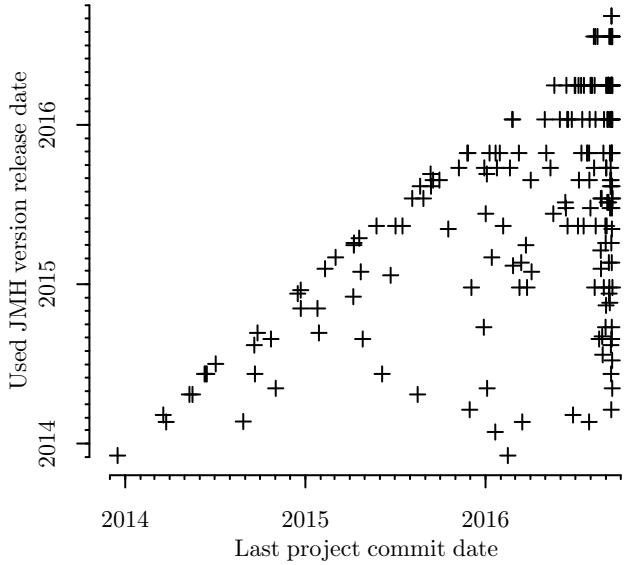


Figure 1.2: JMH version adoption delay. Each project is represented by one point, projects near the diagonal use the most current JMH version, the further from the diagonal, the more obsolete the JMH version used.

projects with performance concerns, early testing is not established habbit and correctness is more favored than performance in early development stages.

To see the categories of projects that are using the JMH benchmarking framework, we label each project with exactly one category based on the project README or its documentation. Similar categories are grouped together to give reasonable results. Table 1.4 presents the top categories with more than 10 members. 66 projects are below this limit and 30 analyzed projects are excluded because their purpose was only to compare other projects with each other.

To sum up the GitHub survey, our premise that unit testing is much more established in open-source projects than performance testing is confirmed. The JMH framework as the biggest representative of performance testing has a hundred times fewer users than the representatives of the unit testing frameworks. Performance tests are written later than unit tests and the adoption of newer versions of the framework also has a significant delay. Proprietary benchmarking

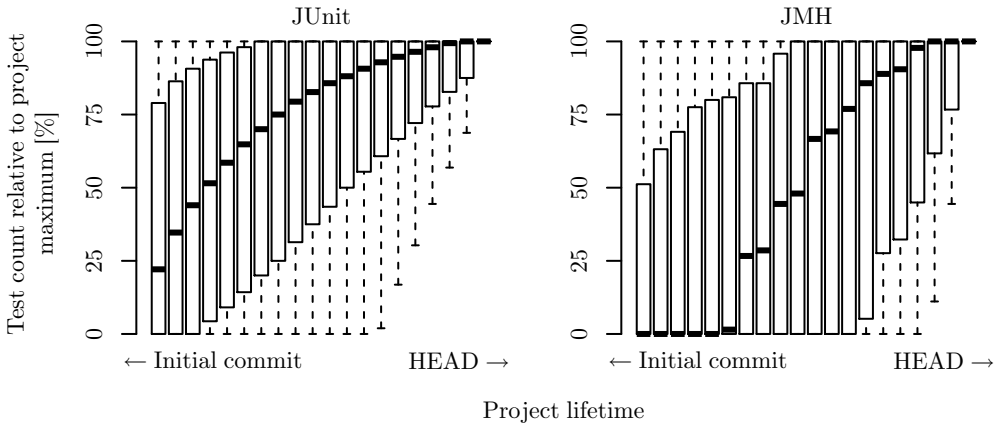


Figure 1.3: Test count during project lifetime. Both axes are normalized, the project lifetime (X axis) stretches from the initial commit to the `HEAD` commit of each project, the test count (Y axis) ranges from zero to maximum number of test cases across the history of each project. Outliers are not shown, whiskers are at 1.5 IQR.

Category	Count
Database (ORM, SQL . . .)	33
Tutorials and examples	30
Networking and distributed systems	29
Algorithms	27
Data structures	22
Object serialization, parsers (XML, JSON, . . .)	22
Web frameworks or plugins	18

Table 1.4: JMH benchmark classification. Categories created through hierarchical clustering, small categories not shown.

is not used much and can be incorrectly implemented, so it is generally better to use the JMH framework. More in-depth information about the running times of the performance tests and the distribution of the performance changes across code versions are presented in [14].

1.4.2 Developer Survey

Despite the quality of the state-of-the-art microbenchmarking tools, the GitHub results confirmed our hypothesis about their rare usage. We come up with several possible reasons why the performance unit testing tools are not very popular in the open-source developer community:

- the target projects are not performance critical,
- the testing tools are hard to setup and configure,
- the developers do not have enough time to write, maintain and regularly run the tests,

- there is no obvious place where to run the tests, especially when a project is developed in a team and CI servers are not designed to run performance measurements,
- the developers do not know how to interpret the test data,
- the performance is tested in other ways.

Many projects are small single-purpose applications working only on limited sizes of input data with short total execution times. These kind of projects are not performance critical, but their users insist on correct functionality, which is one of the many reasons for the big gap between popularity of functionality and performance unit testing.

The performance testing tools can be hard to configure. Possible causes are overly universal design of the tools, lack of community and documentation support and ignorance of the testing principles by the developers. The previous list of popular tools showed that basic configuration should not be an issue, but some advanced scenarios can be different.

Open source developers focus mostly on the implementation correct functionality in the whole software development process. Performance tests are mostly not fun to write because they do not bring immediate results visible to end users. It implies that other feature requests and fixing reported bugs have often higher priority for the developers.

One of the hardest part of the performance evaluation process is to correctly interpret the results. Raw data of one measurement give only a limited view on the performance, because there is no baseline to compare them with. Instead, useful results come from comparing two or more versions against each other. Since each version has more data samples to reduce side effects of other processes in the target system, statistical methods for comparing sets of samples are used. This could be difficult to do correctly, especially for people not interested in statistics and math in general.

To support our speculations, we did a short developer survey. We have identified developers who contributed to performance tests in projects using the most popular microbenchmarking framework, JMH. These people already have some knowledge about the performance tests, so the questions can be better targeted than if the form is sent to regular Java developers. Also, the number of such developers is reasonably low to manually process the supplementary textual answers. The complete survey form is attached as Appendix A.

However, it is important to remember that the answers come only from users of one particular performance framework and thus they might not be fully accurate for performance microbenchmarking in general. We are aware of that, but the survey was already done for our paper [14] and extending it is hard due to a time delay. Resending a similar form to the same developers again would result in imprecise data because they would mostly not respond to the second form, so we decided to use the original answers.

1.4.3 Survey Results

The developer survey was sent to 483 developers of 278 relevant projects using the dominant performance framework, JMH. The developers were chosen based on the fact that they edited code for at least one performance test. 111 completed

forms were returned, 78 of those allowed publishing complete results, 26 allowed publishing only a summary and 7 disallow any form of publishing.

The popularity of the JMH framework comes from the trust in results (72%), active maintenance (60%) and good documentation (40%) of the project. This corresponds to legitimate demands on every used dependency – the project should quickly adopt new techniques and correct old bugs, have clear usage with detailed description and most importantly the users have to trust the outputs of the software. Build system integration demands are surprisingly low at 33%, which can indicate that the developers are willing to invest some time to initially set up the framework in their build workflow.

JMH is mainly used for benchmarking own projects (80%), followed by comparing alternative applications (64%) and examining the performance of external code (37%). This corresponds to the significant amount of tutorial and example projects that allow to get relevant information about performance of software alternatives and make the right decision during the development.

In a discussion about how often the performance tests should be run, almost half of the developers were inclined to run them after every commit (47%), followed by those who want to run them only on every release (37%). Additional comments from the developer survey also support the fact, that the second approach can save a significant amount of time. However, only 42% of the developers run performance tests on a regular basis (commit or release).

The performance tests are often more complex and harder to implement than the regular unit tests. 64% of developers update the performance tests only when fixing performance issues and a minority of 28% maintains the performance tests on regular basis. These numbers show that performance testing is frequently done after some issues are pointed out instead of preventing them by early testing. The previous observation implies that the tests are mostly used only for fixing particular issues, because regular testing to prevent repeating the old issues is not well established.

One of the biggest issues about performance benchmarking is processing the data. Most developers (77%) perform manual processing and just 13% of them use automated plotting with manual checks. A fully automated evaluation process was mentioned in negligible 6% of the answers. 80% of all developers claim that they acted on the performance testing results, however most of the actions are just design decisions (the tests are comparing alternatives) and not fixing discovered performance regressions as primarily expected. This implies that in current situation the performance tests are used more to guide design choices than to keep good performance of a project.

To conclude the survey results, the demands of the performance test developers are summarized. A significant part (61%) think that automated evaluation would result in more and better performance testing, exactly half of the developers want better build system integration and about a third (31%) feel that making the performance tests should be simpler. 27% of the developers mention issues with their spare time or budget. These obstacles line up with the previous findings and should be addressed to improve the testing workflow and extend the number of projects that use performance testing.

1.5 Goals Revisited

The analysis of the current state in Java microbenchmarking confirmed that some tools for the performance measurement exist but they are not widely used. An active development with the support of a bigger organization to warrant some future stability and trustworthiness seems to be the major key for success. Simplicity and time or financial budget of the developers are another important criteria. To extend the number of actively testing projects, a set of easy to use tools has to be available and the overall public awareness about performance testing and the supporting tools has to be raised. This thesis focuses on developing the tools.

Several core use-cases which are used as a base for our design decissions follow from the analysis:

1. The performance tests need to be configured to produce correct and stable results. One of the possible ways is to increase the number of test cycles and then visually find the initial amount of values to skip. This requires a data visualizer with scatter plot showing the data run separately.

2. Developers in a small team with no testing infrastructure write performance tests only in a spare time or when they discover a performance problem. Due to limited resources they run the benchmarks against the released versions of the projects only, and then they visually inspect the performance in a data visualizer.

3. A team with a great care about performance runs the tests after every push to their code repository. A dedicated server performs the measurements, stores the resulting data and automatically tries to find performance drops between the last and some of the older measurements. When a problem is found the developers are notified by email. At the same time, the server can schedule more measurements of any version when it concludes that there are not enough data for statistically significant answer.

4. The project managers are not familiar with the details of performance testing; however, they understand the concepts and want to follow the progress compared to previous states of the project. This means that they only care about visualization of the performance data, so the visualizer has to be easy to run, and has to provide overview graphs over multiple versions.

Some of the goals defined in the introduction (writing performance tests, gathering and storing the data, automated evaluation of slowdowns over the data and visualizing the data) are almost completely covered by the existing tools, others are hard to achieve without extensive modifications of the current software. But, a simple toolset or a framework that integrates all of the parts together to use as one complete package is missing. Our solution attempts to fill this hole in the Java performance testing tools. The goals defined in the introduction are revisited to reflect the model use-cases:

- the developers write performance tests in a dedicated testing framework,
- we provide a build system integration plugin for test execution, storing the data, and generally for preparing the testing environment,

- we provide a tool for automated evaluation of the performance changes depending on (pre)defined performance criteria,
- we provide an interactive graphical data visualization tool supporting multiple versions of the data and various graph types.

We focus on a modular design to allow usage of the evaluator or the visualizer as a standalone application with other ways to prepare the inputs for them. Big effort is exerted into making each part simple and easily understandable. The whole solution targets at smaller open-source projects where good performance is important, but we hope that the same tool can be used as a starting point for much bigger projects where some customization is likely to happen (dedicated testing hardware in multiple configurations, different kinds of statistics, etc.).

The designation of the tool to small open-source teams means that most of the measurements are performed on personal computers with a regular OS installation (including GUI, web browser in the background, enabled HyperThreading, etc.), so the data can have higher noise level than measurements in the lab environment and statistical processing needs to cope with these conditions. Differences in performance across versions are presented as a simple answer with the possibility to examine the results further.

The data visualization component is designed to be flexible independent on the different types of data, multiplatform and interactive. Hands-on experience with the visualized data is faster and more understandable than text description. Finally, integration into a build system seems important even though the developer survey shows only a little interest. A suitable build system is chosen based on the used components but the integration into a different system should be straightforward.

The proposed set of tools reflects our findings and the answers from the developer survey. Using the established projects as dependencies can give credibility to our project and can help to extend the performance testing to the masses. The simple design is the best for common usage by most of the projects, but some of them may need more modifications. Although customization is possible with a set of configuration values, some highly advanced changes would require the code editing. However, an open-source nature of our solution makes customization of the project fairly easy. Detailed design and implementation choices are described in Chapter 2, the user documentation is attached as Appendix B.

1.6 Related Work

Unit and performance testing are established activities to control software quality. Studies [15] and [16] shows reduction of functional bugs when unit testing is used, while papers [8] and [9] report similar results for early performance testing and a broader overview of the topic challenges. Multiple studies inspect the testing practices including the developer opinions. Many companies have issues with test automation as reported by [17] or [18] in context of regression testing. Importance of reasonable setup effort is shown by [19].

DevOps principles are becoming very popular. Multiple studies [20, 21, 22] show that accepting these principles and integrating them into the software development process increases the performance awareness. This statement is proven

also by [23] where early integration of automated testing on the Jenkins server helps detecting performance regressions earlier.

Study [24] summarizes the actual state of performance testing in the Java open-source projects, where manual classification of 111 projects from GitHub is performed. The results correspond with the data collected and presented in this thesis and [14]. The topic of statistical evaluation of the results is covered by paper [25] where a formalism for comparison of test methods and versions is defined. Paper [26] examines visualization of performance data where the results of the tests are rendered into documentation.

A complete performance testing framework (not only) for the Java microservices is designed in [27]. The results are promising, but missing the visualization part and focus only on the HTTP endpoints makes this framework hardly usable for non-web projects. A similar tool for the Flask Python framework exists [28], including the version control system (VCS) integration and a web dashboard presenting the collected data.

A different approach to automated performance testing is provided by instrumentation based tools such as Kieker [29], but the need for complicated environment with stable test fixtures is not suitable for simple and small projects. To create a stable environment for testing, a tool like DataMill [30] can be used. It can eliminate data bias caused by binary link order, process environment size and many more platform dependent factors.

2. Tools Design

From the analysis we have the goals for a new framework helping developers with performance testing from the start to the end. These goals cover the main steps of performance testing – writing and running the tests, storing the performance measurement data, testing slowdowns and speedups across the saved measurements and visualizing them. We have a suitable testing framework, but data evaluation and visualization is not covered by the existing tools enough, as also reported by the developers. A good user experience is a priority, so our framework is designed with attention to the user interface as well as setting the appropriate default configuration values.

Using suitable existing tools for the subtasks is preferred, however, their connection and interoperability needs to be improved. The main task for this part of the thesis is to find useful open-source software projects and implement the tools for testing and evaluating performance on top of them.

2.1 Performance Tests

For performance testingi, we use The JMH framework. The main reasons for this choice are evident from our previous GitHub and developer survey. JMH is the most used Java microbenchmarking tool in open-source projects at GitHub and there is no evidence that other open-source repository hostings may report substantially different numbers. Another benefit of JMH is that some developers are already using it to write performance tests and other performance and unit testing frameworks are used similarly compared to JMH.

The choice of the JMH framework is connected to the supported build system. The only officially supported build system of the JMH framework is Maven, but there is also a community supported Gradle build plugin. Our GitHub survey shows that 223 of the JMH projects are build with Maven and another 52 projects use Gradle. This implies that Maven is an eligible build system for our framework, however, we left the possibility of using a different build system in the future open, mainly because of increasing popularity of Gradle.

Creating new projects and adding performance tests directly follows the JMH standards. A new project is generated from the Maven archetype, for example the command for generating a new project in test folder is shown in Figure 2.1.

```
mvn archetype:generate \
        -DinteractiveMode=false \
        -DarchetypeGroupId=org.openjdk.jmh \
        -DarchetypeArtifactId=jmh-java-benchmark-archetype \
        -DgroupId=org.sample \
        -DartifactId=test \
        -Dversion=1.0
```

Figure 2.1: Maven archetype for creating JMH enabled project.

No special modifications are required for writing actual performance tests, so the numerous official examples[1] can be used as a decent study document. A summary of the general rules follows:

- Choose what to test. Performance tests should have a well defined scope, so the results have an exact meaning and the developers can act on their basis.
- Test precisely. Make sure the testing code is not optimized out by the compiler. The testing frameworks provides methods to deal with this situation, for example the `Blackhole` object of JMH.
- Set up the tests. Each test needs a good configuration to reflect actual performance. It is important to find a balance between the reasonable length of testing and the number of obtained measurements in this time window to allow us making statistically significant conclusions. A high enough warmup time should filter out initial set of imprecise values.

The completed tests are built and run with the following commands:

```
$ cd test/
$ mvn clean install
$ java -jar target/benchmarks.jar
```

The `benchmarks.jar` file is a self-contained executable JAR archive which holds the benchmarks and all the essential JMH code. The entrypoint is provided by the framework and allows runtime changes of the test properties (such as the number of iterations), output format or the use of tests which are executed. JMH supports specifying a custom main method for the JAR, however, this leads to variance between projects, lack of standard command-line interface and it does not bring considerable benefits for most projects. For our use-case the default main method is preferred.

Each of the performance tests needs to be configured. The test is executed for a longer period of time (i.e., half an hour is advised) without skipping warmup, and the right warmup configuration is determined from the results. Since each test is unique, this process should be repeated on a per-test basis and the final configuration should be set for each test using annotations attached to the test method. Another option is to establish a common configuration for all tests and pass it as a command-line argument to the compiled archive, but this approach does not scale well for all but the smallest projects. The final configuration is valid on that exact testing computer and cannot be used elsewhere without prior validation.

2.2 Storing Data

JMH provides several machine-readable output formats for saving performance data – plain text, CSV, SCSV, JSON, LaTeX. The JSON format is meant to

[1] `http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/`

contain a full data dump about the measurement, other output formats contain only partial information. We therefore use the JSON output format.

There are several benchmarking modes in JMH – throughput, average time, sample time and single shot. Each of the modes has its application and is meant to be used in different benchmarking scenarios. Our investigation shows that the sample time output values are just averages of the actual captured values. After a discussion with the lead developer, Aleksey Shipilev, we have prepared a patch for extending the JSON output for this mode. The patch was merged to upstream as revision `2e5a7761ce12`[2] at 16 August 2016 and released in JMH 1.14. The sample time benchmarking mode produces a histogram of randomly sampled values taken during the test execution. This can rapidly generate a large number of values where a significant portion are duplicates. Since JMH does not preserve the time order of the samples, we put a histogram of the values (pairs of values and their quantities) into the output file. This format reduces the file size compared to unfolding the histogram values.

The JMH performance data is obtained as text files containing the JSON formatted data. To design a suitable long term data store, some basic requirements have to be satisfied:

- the store needs to be easy to setup, preferably without any configuration,
- the store should be bound to one project and allow moving/sharing data between developers,
- the data has to be inserted from JSON files,
- reading the data has to be possible with version and test granularity,
- reading the data has to be fast enough for interactive work with the visualizer,
- it has to be possible to append new measurement data to the existing ones.

We can think of several possible storages. Prior to deciding a suitable alternative, we estimate the data size. A mid-size project can have about 200 benchmarks (the most in our survey is 470). We guess that such a project gets 5 commits per day on average. Each measurement of this size can produce a JSON file about 1.5 MB large. For per commit measurement mode, this is about 1800 files with 2.5 GB of raw data per year. However the old data can be archived in compressed format to reduce the required disk space.

With this size we are able to store the data in raw JSON files in a simple directory structure. Databases like MySQL would provide better reading performance, but the data need to be converted and inserted from the JSON files and the MySQL database requires non-trivial configuration as it runs as a separate service. A more suitable alternative to MySQL database is SQLite, which is a one file database with a library engine and no configuration.

We choose a simple directory structure with raw JSON files inside the project root as the easiest option for our project. It requires no configuration and the data is saved in a native JSON format, from which third-party processing tools can benefit. The same arguments apply against other data stores like RRD or MongoDB. The performance of reading the data files is sufficient, because the visualizer mostly work with less than a hundred of test versions at once. More

[2] `http://hg.openjdk.java.net/code-tools/jmh/rev/2e5a7761ce12`

displayed versions make the graph items very small, so it is better to avoid it. Such amount of data can be processed in less than a couple of seconds on modern hardware, which is acceptable. The proposed directory structure is shown in Figure 2.2.

```
${projectRoot}/
├── measurements/
│   └── data/
│       ├── v-1426436027-1d78747356c4e1699f9bac3d14b38e509c36147d/
│       │   ├── 1520002713.json
│       │   └── 1520031458.json
│       └── v-1426436819-997c5c3930a6e8bb61a2efc8eebfcfd10fac4474-dirty/
│           └── 1520078025.json
├── src/
├── target/
├── ...
└── pom.xml
```

Figure 2.2: Proposed directory structure for storing JMH measurements alongside the the project sources.

The presented directory structure follows a standard structure of the Maven projects. Along default entries there is a new directory, `measurements/`, which contains the data for our tools. We do not use the `target/` directory because it can be cleaned anytime and Maven structure permits metadata in top level directories. The `data/` subfolder holds the data from executing the JMH benchmarks, where the measurements follow a specific naming convention. It is not required to follow these rules if the metadata are not needed. Every JSON file inside `data/` represents a single version with a single measurement and every directory containing the JSON files represents a version (using the directory name) with possibly multiple measurements inside.

The naming convention is the same for both files and directories in `data/` with the exception that files have the `.json` suffix. The schema is in Figure 2.1. The names of the measurement files in the version directory are just UNIX timestamps of the measurements with the `.json` suffix.

Executing the benchmark measurements and saving the results in the specified format needs to be automated, which implies the need for a build system integration plugin. We created such a plugin, *jmh_spl-maven-plugin*, in the *cz.cuni.mff.d3s.spl* package. The plugin is available from the Maven Central Repository and is ready for use from any project. Figure 2.3 shows goals of the plugin and their connection to the Maven build lifecycle.

For executing the JMH benchmarks, there is the *data_saver* goal of the plugin. It binds itself to the *verify* phase of the build process so its execution is triggered by the `mvn clean verify` command of following like `install` or `deploy`. The *test* phase of the build process is not used because the plugin needs the JMH `benchmarks.jar` to be already generated. Its main purpose is to run the `benchmarks.jar` with additional arguments to save the outputs in a correct format in the right location. Basically it just executes the

Name Section	Meaning
`v`	Common prefix, idea is that the identifier should start with letter (not number)
`1426436819`	Unix creation timestamp of the version, usually found in VCS (Git)
`997c5c...ac4474`	Identifier of the version, usually Git commit hash or Git tag
`dirty`	Optional flag that the measurement was performed on code under development and may not reflect any version in VCS

Table 2.1: The description of the naming conventions for the stored measurement versions. The identifier the table is describing is `v-1426436819-997c5c3930a6e8bb61a2efc8eebfcfd10fac4474-dirty`.

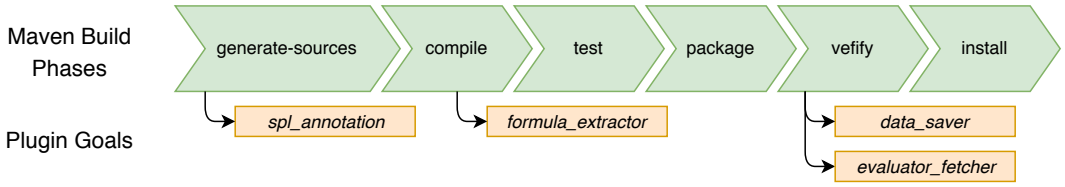


Figure 2.3: Important parts of Maven build lifecycle and connection of *jmh_spl-maven-plugin* goals to the build phases.

```
$ java -jar target/benchmarks.jar -rf json -rff \
  ./measurements/data/v-1426436819-997c5c3930a6e\
  8bb61a2efc8eebfcfd10fac4474-dirty/1520078025.json
```

command, optionally exteded by additional arguments. These can be specified permanently in the project configuration or temporarily as the command-line arguments. Basic plugin setting is shown in Listing 2.1.

When the measurements are executed from a project directory containing Git VCS files, the commit hash and timestamp of the `HEAD` commit are automatically obtained from there. If Git is not accessible and the version is not specified on the command-line, the current timestamp with the *default* identifier are used to construct the version name and the timestamp is also used as the identifier of the measurement.

The described plugin meets our demands for easy build system integration. It requires only a few lines added to the `pom.xml` file at project setup and then it just works via the standard Maven command. The data is stored in an easy to understand and native way for JMH, so third-party tools can access them. Git integration is also very useful because most contemporary open-source projects use Git as their VCS.

```xml
<plugin>
  <groupId>cz.cuni.mff.d3s.spl</groupId>
  <artifactId>jmh_spl-maven-plugin</artifactId>
  <version>1.0.4</version>
  <executions>
    <execution>
      <goals>
        <goal>data_saver</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Listing 2.1: Maven dependency configuration of the presented plugin for saving the performance results.

2.3 Testing Performance Changes

One of the primary objectives of our framework is comparison of performance data, which informs the developers if their changes improve performance or bring regressions. The data can contain some amount of noise so statistical methods need to be used to provide accurate (statistically significant) results.

We use an existing tool, *spl-evaluation-java*[3], as a base for statistical processing and we have extended it to support the JMH output format of the data. Also, we have moved the build process of the tool to Maven to comply with JMH and our *jmh_spl-maven-plugin*. The tool relies on the Stochastic Performance Logic (SPL) [25] formalism for data processing, resulting in a ternary answer – *"yes, the formula evaluates as correct"*, *"no, there is no evidence that the formula evaluates as correct"* and *"there is not enough data to provide a statistically significant answer"*.

The default implementation uses the Welch t-test [31] interpretation, but there are more interpretations which can be potentially used like Mann-Whitney u-test [32]. The formulas are evaluated on 0.95 significance level.

The complete SPL formalism is very complex, but our use-cases uses only a subset of the formulas. These formulas describe assumptions about performance of two versions, for example that version 1.0.0 of the code is not slower than version 0.9.3. Supported syntax of the formulas is described in Extended Backus-Naur Form[4] in Figure 2.4.

In JMH one benchmark can be executed multiple times with different measurement modes and with different runtime parameters. To distinguish between such benchmarks in formula evaluation, the arguments are encoded into the benchmark name. The schema is a fully qualified domain name, the benchmark mode and the list of parameters in a `name=value` format, all of the sections concatenated with the `@` symbol into a single string.

Each formula is valid for one benchmark and two of its already measured versions, so there is no obvious place where to write the formulas. Our solution allows a number of possible places:

[3] `https://github.com/D-iii-S/spl-evaluation-java`

[4] `https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form`

```
SPLFormula = AtomicFormula
           | "(" SPLFormula ")"
           | SPLFormula OP SPLFormula ;
OP = "<" | ">" | "&&" | "||" | "=>" ;
AtomicFormula = Identifier | Number ;
Identifier = Letter { Letter | "." | "-" | "_" } ;
Number = "0"
       | NonZeroIntNumber { IntNumber }
       | [ { IntNumber } ] "." { IntNumber } ;
IntNumber = "0" | NonZeroIntNumber ;
NonZeroIntNumber = "1" | "2" | "3" | "4" | "5" | "6" | "7"
                 | "8" | "9" ;
Letter = "A" | "B" | "C" | "D" | "E" | "F" | "G"
       | "H" | "I" | "J" | "K" | "L" | "M" | "N"
       | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
       | "V" | "W" | "X" | "Y" | "Z" | "a" | "b"
       | "c" | "d" | "e" | "f" | "g" | "h" | "i"
       | "j" | "k" | "l" | "m" | "n" | "o" | "p"
       | "q" | "r" | "s" | "t" | "u" | "v" | "w"
       | "x" | "y" | "z" ;
```

Figure 2.4: Structure of SPL formulas described in EBNF.

- an annotation for the benchmark method with an exact specification of the versions,
- an annotation for the benchmark method with placeholder versions, which are bound to real versions dynamically before the formula evaluation,
- same as two previous bullets, but the annotation is specified before the class so it is valid for all benchmarks in that class,
- a runtime command-line argument per benchmark or wildcard for all of the benchmarks together.

For processing annotations, there are two issues that need to be solved – providing the annotations to the project and parsing them from the code to provide the formulas for evaluation. Both issues can be easily achieved by extending the Maven plugin with the relevant goals.

The first goal, *spl_annotation*, provides an `@SPLFormula` annotation, which takes an SPL formula string as the only argument. This goal is bound to the *generate-sources* lifecycle phase, which precedes compilation when the annotation has to be available. The second goal is *formula_extractor*, which is executed in the end of the *compile* lifecycle phase of the Maven build process. It reads the `/META-INF/BenchmarkList` file with a list of benchmarks generated during the build of the JMH enabled project and parses the file using the JMH API, gets annotations for each of them and finally writes them to the `/META-INF/SPLFormulas` file. In the *package* phase of the build, this file is included into the final JAR package.

We solve the distribution of the SPL evaluation engine JAR to the users with another goal in the Maven plugin. The goal is *evaluator_fetcher* and it downloads the *spl-evaluation-java* JAR from the Maven Central Repository into the `measurements/` directory in the project root. The default placement is shown in Figure 2.5, but the version of the tool and target path can be configured to fit specific needs.

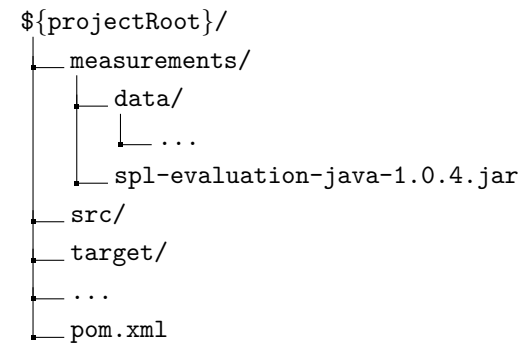


Figure 2.5: Proposed position of `spl-evaluation-java` archive in project directory structure.

Usage of the evaluator tool is described in detail in the user documentation in Appendix B. The basic pattern is to tell the tool where to get the corresponding SPL formulas, the data directory is usually left with default value. Both options are configured by the command-line arguments. Then there are two main operation modes – evaluate the formulas or just check if all formula versions are available in the data and print the answers. The latter mode could be primarily useful for automatic evaluation with on demand execution of measurements.

To sum up, we provide a statistical tool for measured performance data comparison and we keep usage simple for the end users. The simplest scenario of getting the tool and evaluating an SPL formula requires only adding one goal (*evaluator_fetcher*) to *jmh_spl-maven-plugin* in the `pom.xml` file, running Maven and then executing one JAR file with one command-line argument specifying the formula. An open issue is integration of the evaluation itself into Maven. A standalone tool without direct Maven integration is chosen because of better portability (the `measurements/` directory can be taken to another computer separately and contains all possible data and tools), as well as easier manipulation with the dynamic command-line arguments.

2.4 Data Visualization

An important feature for comparing data is their visualization. Graphs provide a lot of information in an easy to understand form. For performance measurements, the graphs can help set the initial configuration of the benchmarks as well as find the size of performance changes across multiple versions. The visualizer helps users to act based on the SPL evaluation tool suggestions.

We find several attempts to create a data visualizer for the JMH performance data. The first one is a web application at `https://nilskp.github.io/jmh-charts/` using non-standard features which break compatibility with some browsers. The second visualizer is a standalone web application at `http://jmh.morethan.io/` with an optional Jenkins plugin. This project is more suitable for the purpose of this thesis, however, it is missing some of the key features following from our model use-cases – interactive comparison of multiple versions and different runs of a single version. Also, both applications require manual loading of the JSON files to the browser which is against our principle of user-

friendly interface. Thus, we choose to write a new visualizer instead of performing extensive modifications of the existing software.

Writing a graphical user interface (UI, GUI) is not easy. For almost every programming language there are numerous frameworks that help the developers with the UI, for example Java Swing[5], Python Tkinter[6] or Qt widgets[7]. From a variety of possibilities we choose writing the visualizer as a dynamic web page. Some of the biggest benefits are that the interface is easy to design and develop, it is multiplatform and requires no additional tools installed (we believe that a web browser is a base part of every end user system). This choice follows the trend of popular projects like Visual Studio Code[8], which are written in JavaScript and packed into desktop apps using the Electron framework[9].

Choosing JavaScript for frontend interface directly affects the architecture of the solution. Performance data is presented from JavaScript running in a web browser, which cannot directly read files on the local filesystem. A common way to bypass this restriction is to create a separate server with the purpose of reading the data and serving them to the browser using the HTTP protocol. From many interfaces designed for HTTP transfer (XML-RPC, SOAP, ...), Representational State Transfer (REST) is used for its high overall popularity in web applications and good support in many libraries.

The interface between the client and the server part is designed and documented to allow different visualizing frontends fetch performance data from our server. One of the most popular tools for this is Swagger[10]. The open-source tool turns a simple YAML description of the API into a documentation in multiple formats and also allows an automatic generation of the server and client stubs for many different frameworks. In our case the API design is pretty straightforward, because it just needs to allow fetching data with test and version granularity. This leads to three endpoints which are described in detail in Appendix C:

- get the available tests,
- for a selected test get the available versions,
- for a pair of test and version get performance data.

A similar alternative is to structure the endpoints differently, i.e., first get a list of all available versions and then for a selected version get the list of available tests. However neither of the alternatives has a significant benefit so we choose the first presented version, because we believe that loading more versions for one test is a more common pattern.

2.4.1 Client

The client JavaScript frontend is written in the React framework v16[11]. It is a single page application using The EcmaScript 6 standard. Data is fetched using

[5] `https://docs.oracle.com/javase/6/docs/api/javax/swing/package-summary.html`
[6] `https://docs.python.org/3.6/library/tkinter.html`
[7] `https://doc.qt.io/qt-5.10/qtwidgets-index.html`
[8] `https://code.visualstudio.com/`
[9] `https://electronjs.org/`
[10] `https://swagger.io/`
[11] `https://reactjs.org/`

the `fetch` API and stored in the state of the root component of the application. A common pattern for storing data in JavaScript applications is to use a shared storage (for example Redux[12]), but for smaller projects it is easier to use the state of the main component and avoid a set of possibly big dependencies. The app is built on top of Create React App[13] and uses Yarn for managing dependencies.

The key component is a JavaScript graphing library. Required features are support of multiple graph types including line plots, box plots and histograms, modern look, React integration and permissive licence. We choose Plotly.js[14] as a suitable candidate with appealing user experience. The React integration is not seamless, but a blog post on the Plotly Academy site[15] is very helpful to set up the project. Other tested third-party React components for graph plotting turn out to be lacking some of the important features of the library.

From the design perspective, the page is split into three sections. The left one is for browsing through the available tests and versions, and the right one is horizontally split into a big upper section containing the graph of the loaded values with controls and a small lower section containing the list of loaded tests or versions. This structure follows the contemporary standards for user interface. A basic user interface overview is in Figure 2.6, detailed information is in the user documentation in Appendix B.

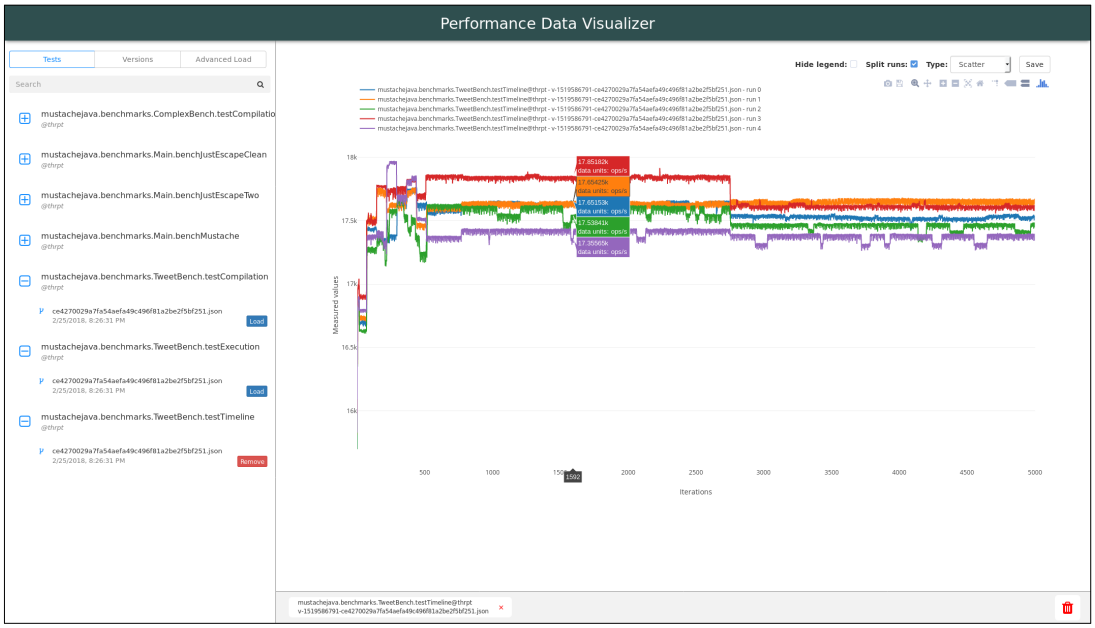


Figure 2.6: User web interface of performance data visualizer.

The displayed data can have miscellaneous formats, so the metadata (title, axis labels, etc.) cannot be set in advance because they depend on the actual state of the application. It is possible to show the complete test metadata in a table including the units, which are also displayed in tooltip triggered by mouse hover over the data when available. The axis labels are determined from the actual graph type. This characteristic is emphasized in user documentation as well.

[12] `https://redux.js.org/`
[13] `https://github.com/facebook/create-react-app`
[14] `https://github.com/plotly/plotly.js/`
[15] `https://academy.plot.ly/react/3-with-plotly/`

The app can be configured with an `.env` file in the project root. The only used variable is the `REACT_APP_API_BASE`, which points to `http://localhost:42000` by default. For normal usage without debugging symbols, the app needs to be compiled in production mode. The build process creates minified and optimized versions of JavaScript and cascade style files including all of their dependencies. The process also transpiles the EcmaScript 6 code to a standard supported by the common web browsers. After successful compilation by the `yarn build` command the results are in the `build/` directory. The content of this directory can be served by any web server supporting static files.

2.4.2 Server

The server side component of the visualizer has to serve the performance data as a REST service. Because the *spl-evaluation-java* project already has a parser for the JMH data files, it is natural that serving the data is responsibility of this component as well. Also, data visualization fits the purpose of this component, which can be described as helping users to understand the data.

The server stub is generated from the Swagger API definition. From a list of Java servers we choose msf4j[16] for its relatively small size and painless integration into our project. JMH data is read and parsed with the existing methods and one new command-line switch allows to start the API server instead of evaluating the SPL formulas.

When a server is available, it is possible to serve the client files as well. This solution makes the evaluator a compact standalone tool with all capabilities for evaluation and visualization of the data. Also, our Maven plugin has a goal that fetches the evaluator, so this additional integration requires no more configuration. It implies that there is a second service in the msf4j server which serves the files generated from the build process of the client part of the visualizer. These files are included in the *spl-evaluation-java* project as resources, which are bundled into the resulting JAR archive.

These two web services run on a common network port, with the default configuration set to 42000. They bind all available network interfaces, but the usual usage would be through the localhost. To simplify the process of running the visualizer, the `http://localhost:42000/` URI is opened in the default web browser when the user graphical interface is available. Also, the URI is printed to the console by the msf4j framework.

The described process creates a self contained bundle for offline usage. Packing this bundle into the existing component does not increase the complexity for users, but brings new functionality with only minor adjustments. The Maven plugin creates the `measurements/` directory in the project root, which contains both the data and the tools necessary for evaluation and possibly also visualization of the data, so this folder can be copied to a different location while preserving all functionality.

[16] `https://github.com/wso2/msf4j`

3. Practical Validation

We want to assess practical usability of the presented testing framework in a real world project. Because the framework is new and no project is using it yet we do a custom study where the abilities of the framework are validated on a sample Java project. We prepare a set of experiments to help us with evaluation of the framework usability. The experiments are similar to the model use-cases presented in Section 1.5.

A project of choice is a string templating engine *mustache.java*[1]. It is an established project with more than one thousand commits and 42 releases (on 12.2.2018) which gives us a plenty of data to examine multiple different usage scenarios. Also there is an effort to implement some performance tests, but sadly just using the proprietary method, sometimes also with connection to the JUnit tests. This way of performance testing is not ideal as was discussed earlier in Section 1. However, there are a few recent JMH performance tests as well which can be directly used as a base for our framework integration.

For execution of benchmarks we use a dedicated server. The hardware and software properties of the server are described in Table 3.1.

Server	Supermicro SYS-5038MR-H8TRF
CPU	8 core Intel Xeon E5-2620 v4 @ 2.10GHz with disabled Hyper-threading
RAM	4× 16 GB DDR4-2133
Disk	Seagate 1TB HDD, 7200RPM, 128MB Cache
OS	Fedora 25 Server
Java	OpenJDK 1.8.0_144

Table 3.1: Hardware and software properties of used testing computer.

3.1 Ease of Integration

The first experiment focuses on integrating our framework into *mustache.java* to allow easy test execution and visualization of the collected data. Primary evaluation metrics are complexity to achieve a similar setup to the original execution of the performance tests and a complexity to evaluate SPL formula with the existing data or display the data in the visualizer. These metrics are expressed in number of added lines to the project build configuration file and the number of additional options to commands invoking requested actions. Secondary metrics of the experiment evaluation include discussion about integration into other common and non-standard project setups.

Original project uses a Maven module for the JMH tests, so the build and the execution configurations are located in the module `pom.xml` file. The test execution is configured using the *exec-maven-plugin* in the *test* phase of the build process. Whole configuration includes one plugin with 28 lines. Similar configuration with our plugin *jmh_spl-maven-plugin* takes only 18 lines, but we save

[1] `https://github.com/spullara/mustache.java`

5 lines on argument representation. Direct comparison is shown in Figure 3.1. Both configurations require adding one plugin section of similar length to the project build file so the configuration complexity is even despite the fact that *jmh_spl-maven-plugin* configuration is slightly shorter and the plugin is slightly more powerful.

```xml
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <version>1.6.0</version>
  <executions>
    <execution>
      <phase>test</phase>
      <goals>
        <goal>exec</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <executable>java</executable>
    <arguments>
      <argument>-Xmx1000m</argument>
      <argument>-classpath</argument>
      <classpath />
      <argument>org.openjdk.jmh.Main</argument>
      <argument>-f</argument>
      <argument>2</argument>
      <argument>-wi</argument>
      <argument>5</argument>
      <argument>-i</argument>
      <argument>10</argument>
    </arguments>
  </configuration>
</plugin>
```

```xml
<plugin>
  <groupId>cz.cuni.mff.d3s.spl</groupId>
  <artifactId>jmh_spl-maven-plugin</artifactId>
  <version>1.0.4</version>
  <executions>
    <execution>
      <goals>
        <goal>spl_annotation</goal>
        <goal>evaluator_fetcher</goal>
        <goal>data_saver</goal>
        <goal>formula_extractor</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <additionalOpts>-f 2 -wi 5 -i 10</additionalOpts>
  </configuration>
</plugin>
```

Figure 3.1: Comparison of Maven configuration allowing performance tests execution. Left snippet represents traditional way for JMH benchmarks taken from the *mustache.java* project, right snippet represents usage of our plugin for the same setup.

Both presented plugins allow to run the tests with the same arguments, but *jmh_spl-maven-plugin* also saves the measurement results in the JSON format, provides and parses SPL formulas from annotations to the resulting JAR and fetches evaluator JAR near the results. Since SPL formulas are not considered at this time, the configuration could be simplified by omitting *spl_annotation* and *formula_extractor* goals.

To judge the complexity of formula evaluation and visualizer execution we suppose that some measurement are already done using our plugin. This means that the performance data are saved in `measurements/data/` directory inside the project root and the evaluator JAR is downloaded into the `measurements/` directory itself. Default configuration of the evaluator is aware of this structure, so SPL formulas can be evaluated only with a one argument specifying either the formulas or their location. Running the visualizer needs the additional argument `-S` which causes opening the local measurements in the default web browser. This implies that number of required arguments is very low and there is no much space for further optimization.

Previous text describes integration with JMH enabled project using Maven modules. Projects that are not already using JMH should follow its integration guidelines first. Because the easiest way of creating a JMH project is from the Maven archetype, it is reasonable to have a separate Maven module for performance tests in the original project. Multi-module setups help with code organisation for bigger projects, but they can be overly complicated for smaller projects.

The only difference between these setups is in position of the `pom.xml` file where is the plugin configuration, nevertheless the plugin is compatible with both variants.

The plugin is written to be compatible with other Maven plugins, however plugins modifying the self-containing JAR or data directory can prevent smooth integration. Such conflicts have to be resolved manually by changing configuration of one or both non-compatible plugins.

CI servers are popular for unit tests execution, but usually they are not suitable for performance testing. They often use virtualization technologies like Docker and there is no control about the server hardware configuration and utilization. In this case we recommend to disable execution of the performance testing with the command-line argument. Performance testing on CI server is reasonable only on dedicated hardware with controlled load and with possibility to keep the measured data, for example on a shared volume. The plugin is then configured to save the data into that location.

General conclusion for integration of our evaluation framework is that the build file settings are similar to the traditional integration of JMH with custom target for tests execution, but our plugin is slightly shorter and provides more functionality. The executions of formula evaluator and data visualizer are very easy using few arguments with clear syntax. The framework is also well prepared to the integration into non-standard project setups.

3.2 Test Configuration

The next experiment targets configuration of performance tests. Performance of Java application can change over time when JIT compiles and optimizes pieces of the bytecode, which makes it necessary to skip some initial measurements until the performance of the application stabilizes. The JMH framework is aware of this and allows the developers to configure the number of skipped iterations or the initial time period when measurement results are not saved. This configuration can be global or separate per test.

The right setup depends on many factors and one of the best ways to find it out is to run each test longer and find a spot from which the data values are similar. The execution length depends on more factors like the actual source code, size of the test, target results precision, etc. but sensible values are usually from several minutes up to one hour. The goal of this experiment is to find out if our framework can help the developers to create suitable configuration for a test and then compare such values with the original configuration provided by the developers of *mustache.java* who were using only the plain JMH framework.

Because the procedure is the same for all tests, only one representative is chosen – `mustachejava.benchmarks.TweetBench.testTimeline` in commit `8877027`. Optimal results are discussed afterwards to cover various warmup behavior and sketch conclusions drawn from them.

After integration of our framework an extended measurement with no skipped warmup iterations is collected. Figure 3.2 shows performance characteristic of this test with no warmup iterations and total of 5000 iterations per run. In this case the measurement can be started using the `mvn clean install -Ddata_saver.additional_options="-f 5 -wi 0 -i 5000"` command. This runs for about seven hours, so each run takes more than an hour.

Figure 3.2: Performance characteristic of `testTimeline` benchmark in 5 forks per 5000 iterations with no skipped warmup iterations.

The visualization style with one line per benchmark run clearly shows that after the first couple of values is significantly slower than about the last third of values. This means that the default configuration with only 5 warmup iterations is definitely not enough. For optimal benchmark there is a point from where the values have only small differences so the measurements of such values has very tight confidence interval. Wrongly designed benchmark produces unstable values for whole execution time, so no number of skipped warmup iterations improve the width of resulting confidence interval.

The results from Figure 3.2 show that performance changes occur near iteration 2750 which is after about 40 minutes from the start of the execution. However, this time is too long for regular performance testing. Relatively stable results are available after 520 iterations, but this seems to be quite long time as well, so a sensible value of warmup iterations is about 75. The results from this point are not fully stable, but at least they are close to median of the values (17 599 in this case).

Each benchmark have to be examined individually, because its behaviour cannot be determined in advance. Goal is to find a balance between execution time of the benchmark and precision (width of confidence interval) of the results. Time and iteration based warmup settings are evenly good in our view, so which one is used depends on consideration of the author. Multiple benchmarks can share a configuration, but the warmup needs to be set as maximum warmup value across them which can lengthen overall testing time. Also each testing computer has different settings, for example a sensible value of warmup iterations for the same benchmark on our other computer is 30.

The right warmup setting can change in development process, so we occasionally recommend to verify the actual configuration. More advanced setup can save all values and then determine and trim the warmup programmatically in

evaluator component, however this approach is more space consuming and our evaluator engine is not fully prepared for this yet.

The conclusion for this experiment is that finding the right warmup settings is easier when there is a plot of the data. The original configuration used by the project developers does not use a long enough warmup, and we believe that using our framework would lead to better configuration of the test from its very beginning. Finally we summarize a few advices about benchmark configuration.

3.3 Regressions in Released Versions

The execution of performance tests is expensive, and the testing habits like frequency or expected coverage taken from functional unit testing therefore do not necessarily apply. One way to deal with this situation is to run regularly only those tests that monitor the overall performance of the application. In case of *mustache.java*, a good example is a test rendering a complex template. Only when a regression is found, additional measurements of specific microbenchmarks can be made to track down the root cause of the observed slowdown.

This experiment covers the first half of the process, that is, finding big regressions in the performance of the whole application. All released versions (42 tagged Git commits) are chosen as measurement points. Since there is no suitable existing test, a new JMH benchmark for rendering large and complex template was written and evaluated against each version. The result of this experiment is a list of versions that show a significant drop in performance compared against their predecessor, backed by the plots from the visualization tool. The comparison of performance is done both by evaluating SPL formulas and visually by inspecting plots of all versions.

The new JMH test for rendering a mustache template is designed to be complex and large enough to use most of the project features in suitable size. From the total of 42 versions of *mustache.java*, only 36 are buildable with the new test using the current tools on the testing computer. The failed ones are mostly the oldest available versions. From the 36 buildable versions, 35 give nonempty performance results, which are then used for further processing.

Our task is to find performance regressions, that is, versions that are slower than their previous version. In long term testing process it is advisable to sometimes compare versions that are not adjacent. This evaluation can detect small regressions that are not statistically significant when comparing only adjacent versions. The SPL formula for two version comparison is simply `newVersion < oldVersion`. The test is measuring throughput, hence higher value means faster code. When this formula evaluates as `true`, we have a statistically significant result that the newer version is slower that the older one. If the formula would be changed to `newVersion > oldVersion`, we would get an indication when the new version is faster, but an evaluation of the formula to `false` would be interpreted as there being no evidence that the new code is faster – it could be slower or it could be as fast as the old version.

When evaluating a batch of formulas, it is impractical to write them as annotations into the code, therefore a different method is used. From the three possibilities (command-line formulas, file with formulas or annotation meta formulas with a mapping file to real versions), command-line formulas

are used. For evaluating all the formulas we write a script that executes the test with all pairs of versions `newVersion` and `oldVersion` under consideration using the `java -jar spl-evaluation-java-1.0.4.jar -c '*:newVersion < oldVersion'` command.

Having 35 data versions means there are 34 SPL formulas for evaluation. The results show that exactly half of the formulas holds and the other half fails (meaning that there was not enough statistical evidence to reject the opposite formula). Versions that are statistically slower than their predecessor are 0.7.3, 0.7.4, 0.7.5, 0.7.9, 0.8.2, 0.8.4, 0.8.8, 0.8.9, 0.8.10, 0.8.12, 0.8.15, 0.8.17, 0.9.0, 0.9.1, 0.9.2, 0.9.3 and 0.9.5.

The formulas are only concerned with statistical significance, not with size of effect – at this point, we therefore do not see how much the performance changed between the stated versions. It may be a significant problem or just a minor slowdown caused by newly added features. The graph presenting the performance of all of the versions can be shown just by giving the `-S` command-line option to the *spl-evaluation-java* instead of `-c <formulas>`. Multiple versions can be loaded using the *Advanced Load* dialog which switches to box plot visualization by default. The output is shown in Figure 3.3.

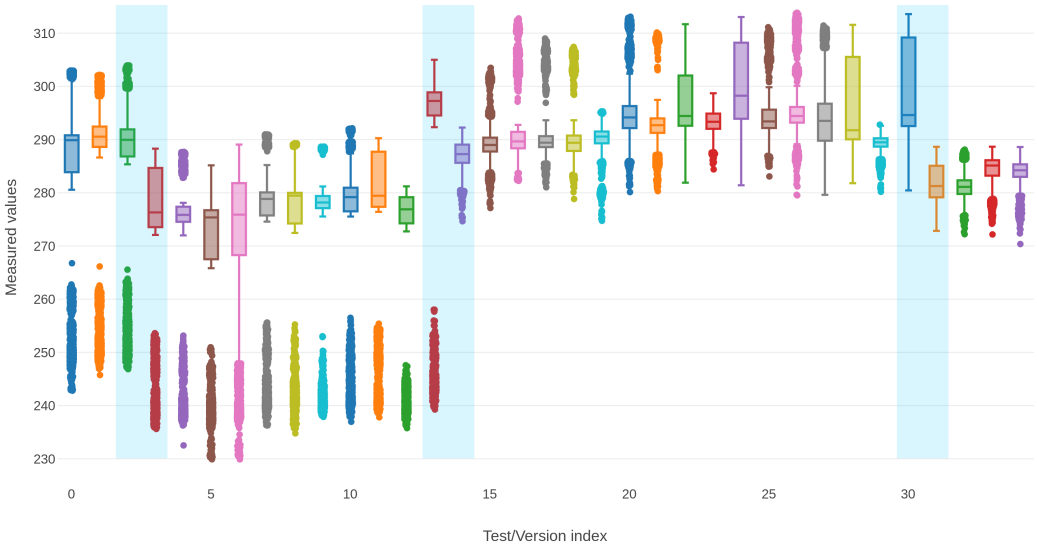


Figure 3.3: Performance characteristic of all versions sorted by release date. The leftmost version is 0.7.0, the rightmost is 0.9.5. Three performance drops are highlighted – slower versions are 0.7.3, 0.8.4 and 0.9.2. Legend is hidden to keep the graph uncluttered.

The graph shows, that there are three bigger performance drops in history of the project. The other changes are relatively small, for example the difference between 0.9.4 and 0.9.5 (the last two versions) which can be caused only as side effect of bigger code base or something not directly connected to a concrete commit. On the other side the marked bigger changes have to have direct causes which should be traced in Git history. An interesting observation is higher number of outliers in the early versions compared to more recent measurements. A scatter

plot of one such version shows that the measurement values frequently alternate between two clusters. This behavior changes in version 0.8.4 (purple box at index 14) – from there onward the data are more stable.

The results of this test show that the SPL evaluation engine can easily provide alerts and point to a specific version which introduces a performance regression. This could be automated in the testing pipeline of a project or simply executed on demand by hand. One thing to improve is the evaluation of multiple formulas across the performance data where an explicit invocation of the evaluation tool for each of the formulas is needed. This requires writing a simple `for`-loop by hand which could be somehow eliminated by the tool itself. However, a command-line interface for specifying the versions, their ordering and SPL formulas per test needs to be designed to fit easy-to-use interface of the tool.

The visualizing part of the tool does a useful work especially where all of the version can be loaded and explored with only a few clicks. The tool runs fast enough even after loading measurements for several tens of versions together into one graph. This overview gives a practical insight of the performance history over time. The discovered issues can examined in zoom or separately in different graphs. A nice feature is also possibility to save graphs in SVG vector format. To sum up, we find the visualizer to be a remarkably useful tool for understanding performance characteristics of any project.

3.4 Tracking Down A Performance Issue

In the previous section, we have found several performance changes. We know the overall performance of the complete project, however, we have no data about the reasons behind the changes. This last experiment follows the previous one by trying to determine what changes exactly are responsible for the observed behavior. The results are judged by the complexity of the investigation steps, including data collection and evaluation. A discussion about the quality of the project benchmarks is also a part of the results.

3.4.1 Small Change Investigation

To localize the changes, more performance data with finer granularity are needed. That implies testing performance of different pieces of the project on every change in the code. We use JMH benchmarks provided by the project to track performance between versions 0.9.2 and 0.9.3. This change is reported as statistically significant slowdown by the evaluator, but the main boxplot shows that it is not one of the three biggest performance drops. However, those changes are not covered with JMH tests which are initially available since version 0.9.1. Our choice can be a nice study example because the selected version range is reasonably tight and there are several commits with attempts to fix one particular bug.

Our investigation proceeds within an eleven-commit window, including the release and merge commits. This effectively means that the performance changes could be caused by one of only five commits, however for the completeness of our results we focus on all eleven. The commits originate in two separate branches, but both merge commits are fast-forward type so there is no interleaving and the

commits can be examined in a sequential manner. The version topology from Git is shown in Listing 3.1.

```
* | 07349e4 (tag: mustache.java-0.9.3) [maven-release-plugin] prepare release mustache.java-0.9.3
* |   eca08ca Merge pull request #173 from samfraser/bug_172-Comments-not-valid-in-ExtendCode-block
|\ \
| |/
|/|
| * caa3a34 BUG-172: fix issue if there is a~comment in an ExtendCode block of a~mustache template,
| |   the ExtendCode.init method fails with a~IllegalArgumentException.
| * 16550d3 BUG-172: fix issue if there is a~comment in an ExtendCode block of a~mustache template,
| |   the ExtendCode.init method fails with a~IllegalArgumentException.
| * 936ba69 BUG-172: fix issue if there is a~comment in an ExtendCode block of a~mustache template,
| |   the ExtendCode.init method fails with a~IllegalArgumentException.
| * 6d7225c BUG-172: fix issue if there is a~comment in an ExtendCode block of a~mustache template,
|/    the ExtendCode.init method fails with a~IllegalArgumentException.
*   8ac71b7 Merge pull request #170 from samfraser/bug_169-partials-in-JARs-not-found
|\
| * e788eb3 BUG-169: fixed an issue with partials referenced using absolute path rather than relative
| |   path not being found if they are in JAR files if the path contained a~double forward slash.
| |   This was due to the ClasspathResolver.java putting a~forward slash at the end of the resourceRoot
| |   in the constructor, so when a~resource was passed in with a~forward slash already appended to
|/    the front, the resulting path contains a~double forward slash.
* ac716e7 Updated to 0.9.2 release
* d883bbe [maven-release-plugin] prepare for next development iteration
* a14af1d (tag: mustache.java-0.9.2) [maven-release-plugin] prepare release mustache.java-0.9.2
```

Listing 3.1: Git topology of *mustache.java* project between versions 0.9.2 and 0.9.3.

We have checked out the project at each commit in selected range and add *jmh_spl-maven-plugin* to the build file. Then we build the project and execute all of the six benchmarks using standard Maven command. To avoid previously identified insufficient warmup, the values for each test are altered to use 5 forks, 100 warmup iterations and 100 measurement iterations. These values are identified as sufficient for all of the tests and have an acceptable duration for regular performance measurements.

The first thing to try is the evaluation of SPL formulas testing the performance drop separately for each test across all versions. The linear character of the commits easily allows comparing two adjacent versions each time. The results in Table 3.2 show that many formulas are evaluated as complying for some tests. The examples of these cases appear also in the first, the second and the last formula where the performance of the project remains the same, because these commits do not change the actual code. This emphasizes that a positive formula evaluation result does not necessarily mean any real performance change.

Performance of `benchJustEscapeClean` and `benchJustEscapeTwo` remains the same because no commit in observed range changes `HtmlEscaper` class or any of its dependencies. This statement holds for performance of the first benchmark, the second one indicates some variability of the results in the visualizer. A deeper look into the individual versions shows that some of the forks are very stable and others run slowly and unevenly. Our investigation shows that this is primarily caused by empty `NullWriter` implementation which is affected with code optimizations performed by the virtual machine. Our implementation using `Blackhole` object from JMH evince slower, but much more stable results.

Another benchmark `benchMustache` renders a simple string template containing one loop. The results are quite stable, but every fork evince different values. This is also caused by wrong implementation of `NullWriter` object. Overall performance of the benchmark is similar for all versions except the one from the commit that changed only the project readme.

The remaining three benchmarks target the performance of template compilation and rendering of two small but non-trivial templates. The compilation

	benchJustEscapeClean	benchJustEscapeTwo	benchMustache	testCompilation	testExecution	testTimeline
v-1465329679-a14af1d421ccd6bd17a4dfcec63bac1f6d9096f4 > v-1465329685-d883bbeb200bb50ddd12f9ee1b2238794b401b54	V	C	V	C	V	V
v-1465329685-d883bbeb200bb50ddd12f9ee1b2238794b401b54 > v-1465337643-ac716e792e4183e33574cc4e9a81f6558dc07c5c	C	V	C	V	V	V
v-1465337643-ac716e792e4183e33574cc4e9a81f6558dc07c5c > v-1467639938-e788eb372f9329a9d07b1603c51e8a9aad776663	V	V	V	V	C	C
v-1467639938-e788eb372f9329a9d07b1603c51e8a9aad776663 > v-1467827803-8ac71b72a840a7debc676f1093b399030e25f8aa	V	C	C	V	C	V
v-1467827803-8ac71b72a840a7debc676f1093b399030e25f8aa > v-1467890766-6d7225c1e7c3b77c3a0f4441936de85769ac17e1	C	V	V	C	V	C
v-1467890766-6d7225c1e7c3b77c3a0f4441936de85769ac17e1 > v-1467894596-936ba693697601f2f0f857111402c31c6aaf7eeb	V	V	C	V	V	V
v-1467894596-936ba693697601f2f0f857111402c31c6aaf7eeb > v-1467896002-16550d3aeef037ccfb3bd684f8bbd80906eb635a	C	C	C	C	C	V
v-1467896002-16550d3aeef037ccfb3bd684f8bbd80906eb635a > v-1467896461-caa3a347148999ced14bdf72279edf4f568e61c3	V	C	V	V	V	C
v-1467896461-caa3a347148999ced14bdf72279edf4f568e61c3 > v-1468509942-eca08ca2bd58898310d302e63da185a568aee81a	V	C	C	V	C	C
v-1468509942-eca08ca2bd58898310d302e63da185a568aee81a > v-1468950957-07349e45399a160a75a8482f578347b8816b035b	C	V	C	V	C	V

Table 3.2: Evaluation results of SPL formulas comparing commits between versions 0.9.2 and 0.9.3. The letter `C` stands for a positive result (complies), the letter `V` stands for a negative result (violates).

benchmark exhibits similar performance in all versions. The performance of the rendering benchmark is a bit different for each of the versions, but a deeper look into the scatter plot shows stable fork results with disparate medians for each invocation. This behaviour is similar to the `benchJustEscapeTwo` benchmark which use the `NullWriter` class as well.

The observed commits attempt to fix two reported issues, #169 and #172 from GitHub. The first change fixes loading of JAR resources with absolute paths. It could have a small performance impact on template rendering, however there is no evidence of a bigger performance change. The second issue fixes a problem with including subtemplates containing comments. There is one commit with the new logic and three commits with debugging outputs and overall code polishing. In this case there is no significant performance change as well.

Fixing both of the issues comes with new unit tests, but no performance tests. It corresponds to the state of performance tests in this project as well as general state in the open-source world. Performance tests of this project could be extended to cover more of the code and to produce more stable data. We find two serious issues with them – wrong implementation of the `NullWriter` and wrong warmup configuration. To sum up we do not find any evidence of performance change in observed range although SPL formula evaluator provides

statistically significant proof of performance drop. It means that SPL formulas can detect really small performance changes which are caused by inaccuracy of the measurement itself and we highly recommend to verify size of the change prior deeper investigation.

3.4.2 Insight Into Bigger Performance Drops

In previous chapter we found three bigger performance drops between versions 0.7.2 – 0.7.3, 0.8.3 – 0.8.4 and 0.8.18 – 0.9.2. These version ranges are not covered with JMH performance tests, so we cannot use the technique from the previous section out of the box. Basically we have several options how to get insight into performance of these versions:

- backport JMH tests from `HEAD` commit into each commit between the version ranges, add JMH dependency to the project and measure all of the commits,
- similar to previous, but use our new complex JMH test, which can be backported to each commit more easily,
- try to manually find the issue and then just verify our assumption with the complex JMH test.

We decided to go with the last option because it is the most suitable one for projects which do not perform regular testing of every commit. Getting relevant performance data at once would lead to possibly a few days of nonstop measurements which is not acceptable. Selected solution could save a lot of time to project developers, especially when they have a good insight into the code or an idea what could cause the performance issue. If this approach fails then measurement and evaluation of all of the commits could easily follow. Results of this experiment are number of positive commit hits and a time how long we investigate each issue.

Version 0.7.3

First we look into the performance issue between released versions 0.7.2 and 0.7.3. This development iteration have 53 commits and some of the commit messages indicates, that they may change performance. We label a few candidates for measurement – commit `bc489c3` which talks about a small performance regression in its commit message, commit `85d4a25` which removes a concurrency and brings more correctness and commit `7060cb5` which may bring back some of the lost performance. This set of measurements is small enough to give us results in reasonable time, so the developer can continue his work to evaluate the data and possibly try to fix the performance issue.

However, we are not able to build these commits because of differences in `java.lang.CharSequence` class in Java 1.6 and 1.8. This means that we cannot verify the performance of these commits which implies that we cannot determine the root of performance issue in version 0.7.3 of *mustache.java*. Our investigation take about three quarters of an hour altogether.

Version 0.8.4

Development cycle between versions 0.8.3 and 0.8.4 takes 33 commits. We identified a few candidates for negative performance change – commit `07e0d32` which adds `BaseObjectHandler` object extending a hierarchy of objects using reflection by one level, commit `a9a008b` adding earlier flushing of output buffer and commit `06b01f0` enabling escaping of input text by default. We are not able to compile these commits with our current setup, so we cannot verify our assumptions. We spend about half an hour investigating this performance drop.

Version 0.9.2

The last of bigger performance drops is between versions 0.8.18 and 0.9.2, however version 0.8.18 is just a minor fix after release of 0.9.X versions and in not merged to master branch, so we use version 0.9.1 which also evince a noticeable performance drop. In this development iteration there are 38 commits.

There are many commits which modifies the benchmarks. There are initial JMH tests which go through a rapid development so their results would not give us comparable values. The commit `4e0b75d` adds more text symbols which need to be escaped, but commit `1f35caf` do an overall optimization of the escaping, so we think that they do not cause any slowdown. Our candidates for causing the performance issue are commits `657fd7a` and `9d74cc9` which together implement using custom immutable `ArrayList` class and use it in core components of the project. These commits are just before 0.9.2 release, so to prove our hypothesis we measure preceding commit `4e6e677` and the release commit `a14af1d`.

The measurements show that our assumptions are correct. The latter commit is significantly slower than the first one and size of the change matches the size of change between versions 0.9.1 and 0.9.2. The measurements are presented in Figure 3.4. Results of our experiment implies the we successfully locate the performance issue with only two measurements which noticeably improves the time needed get useful details. The other option is to measure all commits in observed range which would take more than four hours. For comparison, we work on locating the performance change for about two hours.

Generally the performance issues are in versions that contains above average number of commits from the previous release. Usually many of the commits are irrelevant for performance changes, so there are only a few candidates for further measurement. Investigating only these changes can improve overall time to fix the issue or can provide a larger time window to allow deeper inspection of the code. SPL evaluation engine and performance data visualizer helps to easily understand the data, especially when executing more than one test.

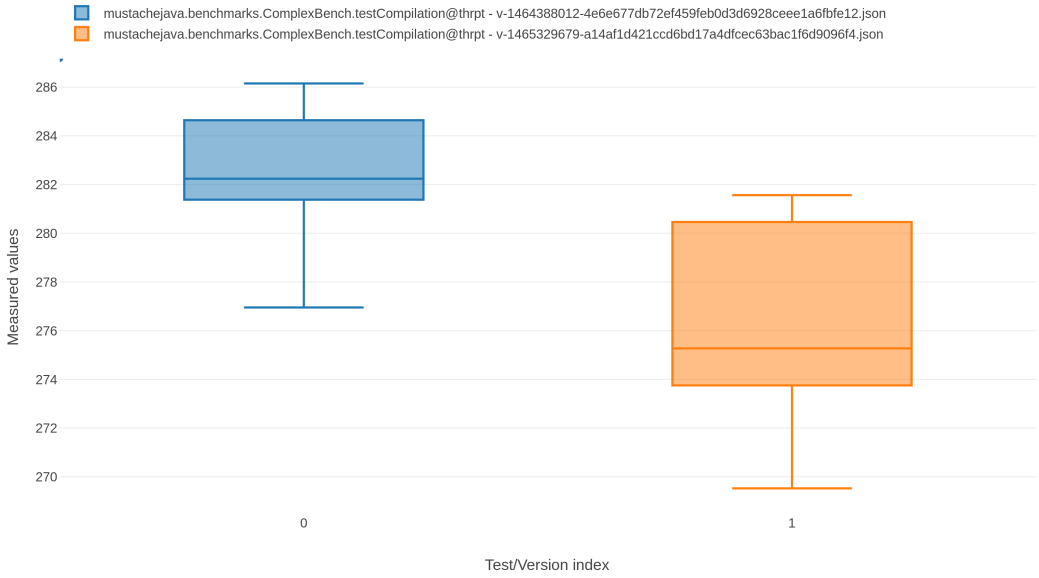


Figure 3.4: Performance characteristic of commits `4e6e677` and `a14af1d` of the *mustache.java* project. The measurements are in the throughput mode in `ops/s` units.

Conclusion

We performed a comprehensive study of Java open-source projects with focus on performance unit testing in the thesis. The results proved our expectation that the unit testing of functionality is far more popular than the performance testing. Despite the fact that suitable (micro)benchmarking tools exist, only a negligible part of the projects is using any. More projects depend on custom proprietary benchmarking, which often reveal poor quality.

A complementary developer survey provides an overview about the performance testing workflow and gives us some ideas for the requested improvements. Performance tests are often written only after a performance issue is discovered, they are used just for fixing the issue and then they are no longer maintained. A systematic testing is carried out by less than a half of addressed developers.

Major issues of performance unit tests are their bigger writing complexity compared to unit tests of functionality and more difficult interpretation of the results. To help eliminate the last issue, we created a custom tool aiming at both the automatic and the manual interpretations of the results for small and mid-sized projects with no custom testing infrastructure.

Performance tests themselves are left to be written in the JMH framework, which is the only framework with a non-trivial usage numbers in the explored set of projects. The framework is a part of OpenJDK and has an increasing user base giving the project good credibility. The interpretation of the data is a responsibility of the SPL framework, which validates formal performance assumptions through statistical methods. Data visualization is handled by an interactive web application using the React framework and the Plotly.js graphing library.

The final solution (one tool for both the formula evaluation and the visualization plus the Maven plugin for integrating it with JMH-enabled projects) is tested against a sample open-source project. The integration of the solution into the selected project proved to be very easy and straightforward, same as was running JMH benchmarks via the Maven plugin and using the visualizer to display the data. The SPL formula evaluator turns into a useful tool for comparison of well designed benchmark data; however, it can produce false alarms for unsteady benchmarks. This should be improved in the future development of the tool. In general, the solution fulfils our expectations to extensively reduce the complexity of data processing for the developers of Java projects.

Coming back to the statement in the title of this thesis, we hope that our tool improves the expansion of performance awareness in Java projects; however, one can hardly expect massive popularity anytime soon. An adoption of regular performance testing is a slow process where the developers need to get used to the tools and the way of processing the results gradually. Doubtless, it is a pleasure to contribute to the movement towards making better and faster software.

Bibliography

[1] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: 10.1145/1297027.1297033.

[2] D. G. Feitelson, E. Frachtenberg, and K. L. Beck. Development and Deployment at Facebook. *IEEE Internet Computing*, 17(4):8–17, July 2013. ISSN 1089-7801. doi: 10.1109/MIC.2013.25.

[3] Paul Brebner. Thoughts on the ABS Census website crash on census night (9 August 2016). `http://www.performance-assurance.com.au/thoughts-on-the-abs-census-website-crash-on-census-night-9-august-2016/`.

[4] John D. McGregor. Test early, test often. *The Journal of Object Technology*, 6(4):7, 2007. ISSN 1660-1769. doi: 10.5381/jot.2007.6.4.c1.

[5] V. K. Myalapalli and S. Geloth. High performance JAVA programming. In *2015 International Conference on Pervasive Computing (ICPC)*, pages 1–6, January 2015. doi: 10.1109/PERVASIVE.2015.7087004.

[6] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Trans. Softw. Eng.*, 30(5):295–310, May 2004. ISSN 0098-5589. doi: 10.1109/TSE.2004.9.

[7] Andreas Ehliar and Dake Liu. Benchmarking network processors. In *Swedish System-on-Chip Conference*, 2004.

[8] Elaine J. Weyuker and Filippos I. Vokolos. Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study. *IEEE Trans. Softw. Eng.*, 26(12):1147–1156, December 2000. ISSN 0098-5589. doi: 10.1109/32.888628.

[9] Murray Woodside, Greg Franks, and Dorina C. Petriu. The Future of Software Performance Engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 171–187, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 978-0-7695-2829-8. doi: 10.1109/FOSE.2007.32.

[10] Ermira Daka and Gordon Fraser. A Survey on Unit Testing Practices and Problems. In *Proceedings of the 2014 IEEE 25th International Symposium on Software Reliability Engineering*, ISSRE '14, pages 201–211, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-6033-0. doi: 10.1109/ISSRE.2014.11.

[11] Antoaneta Baltadzhieva and Grzegorz Chrupa\la. Question Quality in Community Question Answering Forums: A Survey. *SIGKDD Explor. Newsl.*, 17 (1):8–13, September 2015. ISSN 1931-0145. doi: 10.1145/2830544.2830547.

[12] John Wilmar Castro Llanos and Silvia Teresita Acuña Castillo. Differences Between Traditional and Open Source Development Activities. In *Proceedings of the 13th International Conference on Product-Focused Software Process Improvement*, PROFES'12, pages 131–144, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-31062-1. doi: 10.1007/978-3-642-31063-8_11.

[13] G. Kick, C. Decker, and P. Duffin. Caliper: Micro-benchmarking library for Java. `https://github.com/google/caliper`, February 2018.

[14] Petr Stefan, Vojtech Horky, Lubomir Bulej, and Petr Tuma. Unit Testing Performance in Java Projects: Are We There Yet? In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 401–412, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4404-3. doi: 10.1145/3030207.3030226.

[15] Nachiappan Nagappan, E. Michael Maximilien, Thirumalesh Bhat, and Laurie Williams. Realizing Quality Improvement Through Test Driven Development: Results and Experiences of Four Industrial Teams. *Empirical Softw. Engg.*, 13(3):289–302, June 2008. ISSN 1382-3256. doi: 10.1007/s10664-008-9062-z.

[16] Boby George and Laurie Williams. An Initial Investigation of Test Driven Development in Industry. In *Proceedings of the 2003 ACM Symposium on Applied Computing*, SAC '03, pages 1135–1139, New York, NY, USA, 2003. ACM. ISBN 978-1-58113-624-1. doi: 10.1145/952532.952753.

[17] Per Runeson. A Survey of Unit Testing Practices. *IEEE Software*, 23(4): 22–29, July 2006. ISSN 0740-7459. doi: 10.1109/MS.2006.91.

[18] Emelie Engström and Per Runeson. A Qualitative Survey of Regression Testing Practices. In *Proceedings of the 11th International Conference on Product-Focused Software Process Improvement*, PROFES'10, pages 3–16, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 978-3-642-13791-4. doi: 10.1007/978-3-642-13792-1_3.

[19] M. Greiler, A. van Deursen, and M. A. Storey. Test confessions: A study of testing practices for plug-in systems. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 244–254, June 2012. doi: 10.1109/ICSE.2012.6227189.

[20] Wolfgang Gottesheim. Challenges, Benefits and Best Practices of Performance Focused DevOps. In *Proceedings of the 4th International Workshop on Large-Scale Testing*, LT '15, pages 3–3, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3337-5. doi: 10.1145/2693182.2693187.

[21] Johannes Kroß, Felix Willnecker, Thomas Zwickl, and Helmut Krcmar. PET: Continuous Performance Evaluation Tool. In *Proceedings of the 2Nd International Workshop on Quality-Aware DevOps*, QUDOS 2016, pages 42–43, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4411-1. doi: 10.1145/2945408.2945418.

[22] Jürgen Walter, André van Hoorn, Heiko Koziolek, Dusan Okanovic, and Samuel Kounev. Asking "What"?, Automating the "How"?: The Vision of Declarative Performance Engineering. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 91–94, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4080-9. doi: 10.1145/2851553.2858662.

[23] Jan Waller, Nils C. Ehmke, and Wilhelm Hasselbring. Including Performance Benchmarks into Continuous Integration to Enable DevOps. *SIGSOFT Softw. Eng. Notes*, 40(2):1–4, April 2015. ISSN 0163-5948. doi: 10.1145/2735399.2735416.

[24] Philipp Leitner and Cor-Paul Bezemer. An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 373–384, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4404-3. doi: 10.1145/3030207.3030213.

[25] Lubomír Bulej, Tomáš Bureš, Jaroslav Keznikl, Alena Koubková, Andrej Podzimek, and Petr Tůma. Capturing Performance Assumptions Using Stochastic Performance Logic. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 311–322, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1202-8. doi: 10.1145/2188286.2188345.

[26] Vojtěch Horký, Peter Libič, Lukáš Marek, Antonin Steinhauser, and Petr Tůma. Utilizing Performance Unit Tests To Increase Performance Awareness. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 289–300, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3248-4. doi: 10.1145/2668930.2688051.

[27] André de Camargo, Ivan Salvadori, Ronaldo dos Santos Mello, and Frank Siqueira. An Architecture to Automate Performance Tests on Microservices. In *Proceedings of the 18th International Conference on Information Integration and Web-Based Applications and Services*, iiWAS '16, pages 422–429, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4807-2. doi: 10.1145/3011141.3011179.

[28] P. Vogel, T. Klooster, V. Andrikopoulos, and M. Lungu. A Low-Effort Analytics Platform for Visualizing Evolving Flask-Based Python Web Services. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 109–113, September 2017. doi: 10.1109/VISSOFT.2017.13.

[29] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 247–248, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1202-8. doi: 10.1145/2188286.2188326.

[30] Augusto Born de Oliveira, Jean-Christophe Petkovich, Thomas Reidemeister, and Sebastian Fischmeister. DataMill: Rigorous Performance Evaluation Made Easy. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 137–148, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1636-1. doi: 10.1145/2479871.2479892.

[31] B. L. Welch. The Generalization of 'Student's' Problem When Several Different Population Varlances Are Involved. *Biometrika*, 34(1-2):28–35, 1947. doi: 10.1093/biomet/34.1-2.28.

[32] H. B. Mann and D. R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, 18(1):50–60, March 1947. ISSN 0003-4851, 2168-8990. doi: 10.1214/aoms/1177730491.

List of Abbreviations

CI ..... Continuous Integration

GPG ..... GNU Privacy Guard

IDE ..... Integrated Development Environment

IQR ..... Interquartile Range

JAR ..... Java Archive

JIT ..... Just-In-Time (Compilation)

JMH ..... Java Microbenchmark Harness

JVM ..... Java Virtual Machine

OS ..... Operating System

Q&A ..... Questions and Answers

REST ..... Representational State Transfer

RRD ..... Round Robin Database

SCSV ..... Semicolon Separated Values

SOAP ..... Simple Object Access Protocol

SPL ..... Stochastic Performance Logic

VCS ..... Version Control System

VM ..... Virtual Machine

A. JMH Usage Survey

1. For what purpose do you use JMH in your projects?

   Check all that apply.

   - Benchmarks to learn performance of system or libraries
   - Benchmarks to learn performance of project code
   - Comparative benchmarks between alternatives
   - Other: ...

2. Why did you choose JMH over other frameworks?

   Check all that apply.

   - Well documented
   - Actively maintained
   - Trust the results most
   - Easy integration with build system
   - Do not know other similar tools
   - Other: ...

3. Do you run the JMH benchmarks regularly?

   Either as part of continuous integration or similar technique or as part of your quality assurance process.

   Mark only one.

   - Yes
   - No
   - Other: ...

4. Are the benchmark results processed and evaluated automatically?

   Mark only one.

   - Yes (automatically processed and anomalies reported)
   - Yes (automatically plotted and manually inspected)
   - No (only manual inspection)
   - Other: ...

5. How systematically in general do you test performance in your project?

   Mark only one.

   - 1 (Rigorous and regular testing)
   - 2
   - 3

- 4
- 5 (Only incidental testing)

6. Did you ever act on performance test results?

   Mark only one.

   - Yes, I find regressions or improvements regularly
   - Yes, I make design decisions based on test results
   - Yes, but the tests report interesting results only rarely
   - No, never found any bug with the performance tests
   - Other: ...

7. What level of performance testing would you want in the development process?

   Mark only one.

   - I would like to have regular performance tests run with every commit
   - I would like to run performance tests before every release
   - I prefer user reports to own performance tests
   - Other: ...

8. What would help you utilize performance tests more in your development process?

   Check all that apply.

   - Simpler performance test implementation environment
   - Tighter integration with build infrastructure
   - More automated result evaluation
   - More time or budget to run the tests
   - Other: ...

9. When do you update (add, delete, refactor) your performance tests?

   Mark only one.

   - Only when fixing or evaluating performance issues
   - Regularly, same as any other code
   - Just before a release is due
   - Other: ...

10. Anything else

    Space for your comments on JMH, performance testing or any other thoughts that crossed your mind when completing this survey.

11. Using your answers

    Mark only one, required.

- I agree with using my answers in both an overall summary and for possible anonymous quoting
- I agree with using my answers in an overall summary but do not quote me
- I only agree with private use of my answers

12. Contact

    Fill in your e-mail address if you wish to be notified of the results. This mail will not be used for any other purpose.

B. User Documentation

The purpose of our framework is to help Java developers with performance testing. The solution provides these main features:

- run JMH benchmarks through Maven,
- store benchmark results in standard structure and format, using Git metadata when possible,
- evaluate SPL formulas against saved data to find statistically significant performance changes,
- interactively visualize the data in a web browser.

Basic knowledge of benchmarking and JMH is a prerequisite for understanding the following text. The expected skills also include the basics of Java and Maven build system. For running our examples and compiling the tools, we expect Java (version at least 1.8.0) and Maven (version at least 3.5.0) installed if not specified otherwise.

B.1 Quick Start

This section provides a simple tutorial that covers the creation of a new project, integration with Maven, benchmarking and processing the data. More elaborated descriptions of the individual steps can be found in one of the following sections, dedicated to individual components.

Set up a new JMH project

First, we generate a new JMH project from the Maven archetype:

```
$ mvn archetype:generate \
      -DinteractiveMode=false \
      -DarchetypeGroupId=org.openjdk.jmh \
      -DarchetypeArtifactId=jmh-java-benchmark-archetype \
      -DgroupId=org.sample \
      -DartifactId=test \
      -Dversion=1.0
```

The project is initialized in the `test/` folder, which becomes our new working directory. To write a sample benchmark, replace the code of the `testMethod()` from the `src/main/java/org/sample/MyBenchmark.java` file with the new content (ignoring imports for simplicity):

```
1 @Benchmark
2 @BenchmarkMode(Mode.Throughput)
3 @OutputTimeUnit(TimeUnit.SECONDS)
4 public void measureThroughput() throws InterruptedException {
5   TimeUnit.MILLISECONDS.sleep(100);
6 }
```

Now we create a local Git repository and commit the initial state:

```
$ git init
$ git add .
$ git commit -m "Initial version"
```

Add the Maven plugin and run the benchmark

Now, we add our custom Maven plugin into the project. To achieve this, add the following snippet into the `<plugins>` section of the `pom.xml` file:

```
<plugin>
  <groupId>cz.cuni.mff.d3s.spl</groupId>
  <artifactId>jmh_spl-maven-plugin</artifactId>
  <version>1.0.4</version>
  <executions>
    <execution>
      <goals>
        <goal>evaluator_fetcher</goal>
        <goal>data_saver</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

After this, we can start benchmarking simply by executing the `mvn clean install` command. When it is finished, a new directory called `measurements/` is created in the project directory, containing one JAR file, and, in the `data/` subdirectory, a version directory with a single JSON data file – for example:

```
./data/v-1522435282-3308f81b97820f826bc522c8bbbb60a1fe10965c-
dirty/1522435657.json
```

Reduce code performance and run the benchmark again

To simulate a performance regression in the tested code, we modify the benchmark to sleep a longer period of time. The new benchmark method body is `TimeUnit.MILLISECONDS.sleep(120);`. Then we commit the changes and run the benchmark again:

```
$ git commit -a -m "Worse performance"
$ mvn clean install
```

Evaluate a SPL formula to find regression

Our benchmark is set to throughput mode, so higher values mean better performance. Generally, it is desired that newer versions are not slower than older ones. This can be translated to a SPL formula saying *"older version is faster than newer version"* and a positive answer means that a performance problem is found.

The formula can be evaluated calling the *spl-evaluation-java* tool with the formula as an argument:

```
$ java -jar measurements/spl-evaluation-java-1.0.4.jar -c \
  "org.sample.MyBenchmark.measureThroughput: \
  v-1522435282-3308f81b97820f826bc522c8bbbb60a1fe10965c-dirty > \
  V-1522437046-5105cd589ec92684d6f7ca27d477b9aba17e17e7"

Benchmark: org.sample.MyBenchmark.measureThroughput@thrpt
    formula:  v-1522435282-3308f81b97820f826bc522c8bbbb60a1fe\
              10965c-dirty > v-1522437046-5105cd589ec92684d6f\
              7ca27d477b9aba17e17e7
    result: COMPLIES
```

The result is positive, the performance drop is statistically confirmed.

Run the visualizer and browse the data

Now we want to display the data to visually inspect the size of the performance change. The visualizer is started with this command:

```
$ java -jar measurements/spl-evaluation-java-1.0.4.jar -S
```

The default web browser is opened when graphics environment is available. The data can be loaded into the graph from the left menu or via the *Advanced Load* button.

The example shows a complete testing workflow on a simple project, but the usage pattern is similar even for much bigger projects. For existing projects it is critical to integrate JMH framework, after that using presented Maven plugin is a matter of couple minutes.

B.2 Maven Plugin

Our Maven plugin provides an easy integration of the benchmarks based on JMH with our SPL evaluation engine and data visualizer. It is available from the Maven Central Repository with groupId *cz.cuni.mff.d3s.spl* and artifactId *jmh_spl-maven-plugin*, hence it can be automatically fetched during the Maven build process.

The plugin has four goals, each enabling different functionality. They can be used together or in an arbitrary combination to suit specific needs of the target project. Each goal can be specified as a separate `<execution>` section in `pom.xml` or all of them can share one common section with multiple `<goal>` items.

The following code snippet shows the base template which is extended by the `<execution>` section(s) of the used plugin goals.

```xml
<plugin>
  <groupId>cz.cuni.mff.d3s.spl</groupId>
  <artifactId>jmh_spl-maven-plugin</artifactId>
  <version>1.0.4</version>
  <executions>
    ...
  </executions>
</plugin>
```

B.2.1 The Data Saver Goal

The *data_saver* goal handles the saving of the measured data generated from the `benchmarks.jar` in JSON format into a configured directory with a custom revision identifier. In the default configuration, this goal is executed in the Maven *verify* phase, because that follows the *package* phase where the `benchmarks.jar` file is created.

The following snippet fits into the base plugin configuration template and enables the *data_saver* goal:

```xml
<execution>
  <id>Generate data</id>
  <goals>
    <goal>data_saver</goal>
  </goals>
</execution>
```

There are several configuration options modifying the behaviour of this plugin goal. These options can be specified from the command-line with a property identifier or from the `pom.xml` file. The properties start with the `data_saver.` prefix which is ommited in the table.

pom.xml	property	default value	description
revisionID	revision_id	`v-<timestamp>-<id>` where `<id>` is `default` or `HEAD` revision from Git and `<timestamp>` current or `HEAD` unix timestamp	Identifier of current revision used to construct data path. Existing data for this revision id will remain untouched.
jmhJar	benchmarks_jar	`${project.build.directory}/${uberjar.name}.jar`	Path to benchmarks.jar generated by the JMH build.
resultPath	result_path	`${project.basedir}/measurements/data`	Path to the directory where are measured data in JSON format stored. Will be created if absent.
additionalOpts	additional_options	`""`	Arguments that will be passed directly to the `benchmarks.jar` while executing. For example `"-v SILENT -foe true"`.
skip	skip	`false`	Skip goal execution.

Multiple measurements with the same version identifier are possible, the identifier determines the directory name, the actual measurements are stored as files with names derived from the current timestamp. For Git enabled projects the version identifier is determined from the repository metadata. Executing measurements of only some of the tests is possible via the `additional_options` property where fully qualified test name is given as the last argument. For more information, refer to the JMH documentation.

Examples of the command-line and the `pom.xml` configuration usage:

```
$ mvn clean install -Ddata_saver.revision_id=ver3
```

```
<configuration>
  <jmhJar>
    ${project.build.directory}/${uberjar.name}.jar
  </jmhJar>
  <resultPath>${project.basedir}/measurements</resultPath>
</configuration>
```

B.2.2 The SPL Annotation Goal

The SPL annotation goal is used to install the `@SPLFormula` annotation, which is used for attaching SPL formulas to benchmarks. It can be attached to a benchmark method or to a class, affecting all of its methods. The default Maven execution phase is *generate-sources*. The generated `SPLFormula.java` file is copied to the `target/generated-sources/spl_annotations/cz/cuni/mff/d3s/spl` directory. The compilation `CLASSPATH` is altered to include this directory. This plugin requires no configuration.

The `pom.xml` snippet for including this target in the base plugin configuration template:

```
<execution>
  <id>Provide SPL annotation</id>
  <goals>
    <goal>spl_annotation</goal>
  </goals>
</execution>
```

B.2.3 The SPL Extractor Goal

The SPL extractor goal finds the `@SPLFormula` annotations in source code files, parses them and saves the formulas into the `META-INF/SPLFormulas` file, which is included in the final `benchmark.jar`. The default Maven execution phase is *compile*. The file format is line based – `benchmark_name:SPL_formula`. This plugin requires no configuration.

The `pom.xml` snippet for including this target in the base plugin configuration template:

```
<execution>
  <id>Parse SPL annotation from sources</id>
  <goals>
    <goal>formula_extractor</goal>
  </goals>
</execution>
```

B.2.4 The Evaluator Fetcher Goal

The evaluator fetcher goal downloads the *spl-evaluation-java*[1] JAR from the Maven Central Repository and saves it into the directory containing the performance data (`measurements/` by default). The JAR contains command-line application providing the SPL formula evaluation engine and the performance data visualizer. The default Maven build cycle is *verify*.

The `pom.xml` snippet for including this target in the base plugin configuration template:

```
<execution>
  <id>Fetch evaluator</id>
  <goals>
    <goal>evaluator_fetcher</goal>
  </goals>
</execution>
```

The configuration properties of this goal are described in the following table. The properties start with the `data_saver.` prefix which is ommited in the table.

pom.xml	property	default value	description
evaluatorVersion	version	1.0.4	Version of the downloaded JAR.
evaluatorSaveDir	save_dir	`${project.basedir}/measurements`	The directory where the JAR is stored. Created if absent.

B.2.5 Compilation

The plugin can be compiled from source by executing the `mvn clean install` command from the cloned Git repository. The compiled JAR file is in the `target/` directory of the project. Note that correct GPG configuration of Maven may be required to create signed builds.

B.3 SPL Formula Evaluator

The SPL formula evaluator project implements a tool for evaluating SPL formulas against performance data. This tool is used to determine whether performance assumptions on a set of collected measurement data are correct.

[1] `http://repo1.maven.org/maven2/cz/cuni/mff/d3s/spl/spl-evaluation-java/1.0.4/spl-evaluation-java-1.0.4.jar`

B.3.1 Console Interface

The evaluator is managed through a console interface with several command-line options. A quick help is printed on errors (for example when no argument is given):

```
usage: spl-evaluation-java [-c <formulas>] [-d <data_directory>] [-f
       <formula_file>] [-j <benchmarks.jar>] [-p] [-r <mapping_file>] [-S]
Evaluate measured data against SPL formulas.
 -c,--commandline-formulas <formulas>   Read SPL formulas from command-line
                                        arguments.
 -d,--data-dir <data_directory>         Path to directory with measured
                                        data.
 -f,--file-formulas <formula_file>      Read SPL formulas from text file.
 -j,--jar-formulas <benchmarks.jar>     Read SPL formulas from JAR file.
 -p,--print-unknown                     Print unknown revisions and exit.
 -r,--revision-mapping <mapping_file>   Mapping of formula revisions to
                                        file names.
 -S,--server                            Run API server and visualizer
```

Each SPL formula is bound to one benchmark. The formulas can be specified in three ways – inside the `benchmarks.jar` package (collected from the source code annotations), in a text file, or directly on command-line. The formulas are parsed in this order, latter formula has higher priority and overrides any previous benchmark formula.

- `benchmarks.jar` – formulas are saved in the `META-INF/SPLFormulas` file, formatted as `benchmark_name:formula`. This file is generated using the Maven plugin (*jmh_spl-maven-plugin*) from the `@SPLFormula` annotations.
- text file – a simple text file formatted as `benchmark_name:formula`
- command-line arguments – list of pairs `benchmark_name:formula` separated by space. Note that enclosing each value into quotes may be required because of shell argument parsing. To denote all all benchmarks, use `*` as the benchmark name.

The **data-dir** option specifies the path to main directory with measurement data. It can be omitted for default settings, which make the tool look for the `data/` directory in the same base path as the JAR itself. The **print-unknown** option enables a mode that print versions mentioned in SPL formulas but not present in the measured data or transitively in the custom mapping of revisions. No evaluation is performed. The **revision-mapping** option reads a plain text file with colon-separated values of custom revision and actual measured data revision name. Each line contains one mapping, empty rows or rows starting with the `#` are ignored. The **server** option runs a web server for the API and the visualizer, listening on address `http://0.0.0.0:42000`. `Index.html` is the main locator for the visualizer, the API endpoints are on their URLs according to the API documentation.

Example console usage:

```
$ java -jar spl-evaluation-java.jar -d ./demo-data/jmh \
  -r /tmp/mapping.txt -j ./target/benchmarks.jar \
  -c "cz.stdin.ps.MyBenchmark.testMethod:last < ver2" \
  "cz.stdin.ps.MyBenchmark.methodTwo:ver1 < base"
```

B.3.2 Library

This project can also be used as a library, providing the SPL evaluation engine functionality over arbitrary data. Examples of how to use it this way are located in the `src/demo/` directory, where is located a separate `pom.xml` file to build them. Note that the parent project have to be installed in the local Maven repository prior to building the demo (using the `mvn clean install` command).

Two prepared scenarios can be run directly from Maven:

- `mvn exec:java@regression-tester`
- `mvn exec:java@sensitivity-comparison`

B.3.3 Documentation

The reference documentation can be generated using the `mvn clean site` command. The JavaDoc-generated html provides code documentation, basic information about usage of the spl evaluation engine, unit test reports and other project metadata. The documentation is generated in the `target/site/` directory.

The SPL formalism is originally created for performance unit testing, more information is available at the SPL for Java page[2] of the Department of Distributed and Dependable Systems (Faculty of Mathematics and Physics, Charles University in Prague)[3].

B.3.4 Compilation

A standalone compilation is triggered with the `mvn clean package` command, for compilation with JAR creation use the `mvn clean install` command. All of the generated files are stored in the `target/` directory, including the main `spl-evaluation-java-1.0.4.jar` file. The attached unit tests can be compiled and executed by running the `mvn test` command.

The data visualizer functionality needs the compiled files of the performance data visualizer project in the `src/main/resources/cz/cuni/mff/d3s/spl/visualization` directory before building the SPL evaluation engine. Linux systems can use the `build_visualizer.sh` script from the project root that install the visualizer files into the project resources (all the necessary tools for building the visualizer need to be installed otherwise the build will fail).

B.4 Perf Data Visualizer

The Perf Data Visualizer is a multiplatform visualizer of performance data designed to support various data formats. It uses a REST API to fetch the list of measurements, their metadata and their actual values. The tool is written in JavaScript using the ECMAScript 6 specification and the React framework. The graph visualization engine comes from the Plotly.js project.

The visualizer project is normally a part of the *spl-evaluation-java* package, but it can also be used as a standalone web application. It should work in most modern web browsers, the tested ones are Firefox 59 and Chrome 65.

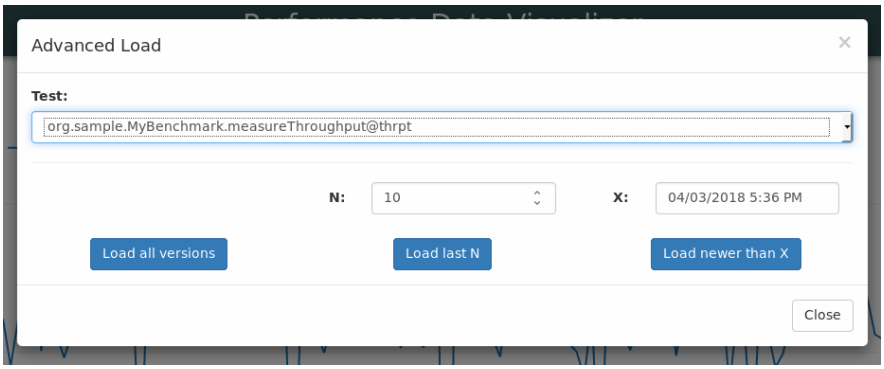
[2] `http://d3s.mff.cuni.cz/software/spl-java`

[3] `http://d3s.mff.cuni.cz/`

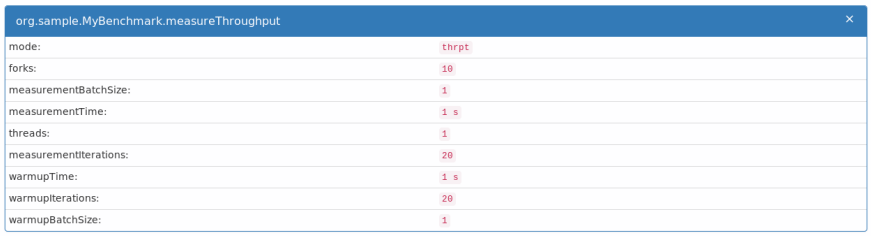
B.4.1 Interface and Functions

The visualizer is a single page web application designed for desktop usage. The structure of the interface is shown in the following picture:

1. Button toolbar. These buttons serve to switch the display mode of box 2 and initiate advanced loading of multiple data versions. The advanced load button shows a modal dialog (shown in the next picture) with a select box for the benchmark name and three columns for three different loading styles – all available versions, the last N versions where N is a given integer, and versions newer than X where X is a given date.



2. A searchable list of tests or versions. The default representation shows the test items, with each item being expandable to show the sublist of available versions for that test. When the display is switched to the versions mode, the sublist for each version contains the available tests. Each sublist item contains a button to load or remove the associated data for visualization. Clicking on a top-level list item in the tests display mode shows a table containing the test metadata above box 4 (an example metadata table is in the following picture).



3. Custom graph controls. This set of controls provides additional configuration for graph visualization. The following controls are available:

   - **Max bins** (only for histogram mode) – The maximum count of histogram bins. This value is used in the algorithm calculating the actual number of bins, a change in this value therefore may not be immediately reflected in the graph.
   - **Hide legend** – A check box for hiding the legend, useful when many versions are loaded.
   - **Split runs** – A check box for rendering all benchmark runs over each other instead of concatenating them.
   - **Type** – A select box for the graph type. Currently three options are available: a scatter plot, a histogram and a box plot.
   - **Save** – A button for saving the rendered graph in a vector format (SVG).

4. Graphing area. This area is rendered by the Plotly.js graphing library with all of its features. It provides interactive zooming, moving the data, showing values on hover and more.

5. Loaded data manager. This box shows the currently loaded and visualized data. When more than three versions are loaded, the box is scrollable. Each of the versions can be unloaded by clicking the red cross on its right side, the trash bin icon near the right border removes all loaded versions.

B.4.2 Graphing Area

The Perf Data Visualizer is designed to be a suitable tool for various applications of visualizing performance data. The interactive nature of the tool implies that the graph cannot easily have all of the formal requisites like a title, exact labels with units, etc. The axis labels can be deduced from the graph type, but for example the units of measurements are missing, because every loaded test or version can use different units. This problem is solved by displaying the units for each data sample separately in a box on mouse hover.

The scatter plot shows the measured values, preserving the order of measurement. The X axis shows the sequential order of the measured value, the Y axis gives the value itself. This is the default graph type.

The histogram shows the measured values grouped by the count of their occurence. The X axis shows the measured values, the Y axis indicates the number of occurrences of that value (or value range) in the input data. The style depends on the number of bins rendered. This mode is useful mostly for JMH sample time mode, where the order of the samples is not preserved.

The box plots are intended for comparison of multiple versions of the same benchmark. The X axis shows individual tests or versions, the Y axis plots the measured values. For each test or version, a box shows multiple statistics about the data:

- the middle line represents the median
- the top and bottom sides of the rectangle represent the 1^{st} and the 3^{rd} quartile
- the whiskers (solid vertical lines extending from the rectangle) represent the interquartile range, the top whisker is the largest value under Q3 + 1.5 * IQR, the bottom whisker is the lowest value over Q1 – 1.5 * IQR
- any data outside the whisker range is considered to be outliers, represented by dots in the graph

B.4.3 Compilation

The project is managed by the Yarn dependency manager[4] and bootstrapped with Create React App[5]. The compilation process consists of several steps:

1. Installing dependencies through the `yarn install` command,
2. Editing the `.env` file and setting the `REACT_APP_API_BASE` variable to point to the API base URL, and
3. Building the production files with the `yarn build` command.

The output files are generated in the `build/` directory and can be served by any web server as static files.

When developing the app, a debug mode server is started with the `yarn start` command. A custom web server listens on `http://localhost:3000`. The unit tests can be started with the `yarn test` command.

B.5 Practical Tips

We present some tips and recommentations for the measurements and usage of the framework in general in this section. They are certainly not valid for all possible use-cases, so each of the tips has to be considered before its application to the target project.

1. Generate benchmarking project from the JMH Maven archetype. Not all of the JMH dependencies are obvious, so use a Maven module when you add benchmarks to an existing project.

[4] `https://yarnpkg.com/lang/en/`

[5] `https://github.com/facebook/create-react-app`

2. When you desire to check the performance of a single benchmark in the middle of the development process you can specify its full name as the last argument in additional options passed to JMH (for example `-Ddata_saver.additional_options="org.package.MyBenchmark"` from the command-line). Multiple measurements are saved as the same version, but you can compare them as separate runs from the visualizer. Another option is to use different result paths or revision ids for such temporary measurements.

3. The `measurements/` directory inside the project root is not meant to be versioned. It contains the JAR of our evaluator and the possibly big performance data. Since the data are bound to the exact computer configuration the only reason for sharing them is the visualization. For this, we recommend to set a server containing the data and running a webserver proxying the requests to the local visualizer. Note, that this server should not be used for the measurements because the webserver may negatively affect the precision of the measurements.

4. The unit testing of functionality and the performance testing are different. With the former we try to cover the code as much as possible to avoid non-tested spots with potential bugs. Adding more performance tests highly affects the overall testing time and brings more data that are hard to correctly evaluate. We recommend to regularly run just a few of high-level tests and make more detailed measurement only when an issue is found.

C. Visualizer API

This is API documentation of Performance Data Visualizer.

Version: 1.0.0

BasePath:/

MIT license

C.1 Methods

- `GET /tests`
- `GET /tests/{testId}/revisions`
- `GET /tests/{testId}/revisions/{revisionId}/data`

C.2 Description

`GET /tests`

List of all available measurements (benchmarks)

Get list of all measurements which have at least one data version in a configured datastore. Each item contains required and optional metadata (for example all metadata from JMH output, such as number of iterations or benchmarking mode). Optional metadata are served as key-value pairs.

Return type

array[Test]

Example data

Content-Type: application/json

```
[ {
  "metadata" : {
    "key" : "metadata"
  },
  "name" : "cz.stdin.ps.SampleTest.testMethod",
  "id" : "id"
}, {
  "metadata" : {
    "key" : "metadata"
  },
  "name" : "cz.stdin.ps.SampleTest.testMethod",
  "id" : "id"
} ]
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses

200 Success

500 Internal server error

`GET /tests/{testId}/revisions`

List of different measurement versions in specific test

Returns list of different measurements for specific test. The versions are returned from oldest to newest according to version timestamp. If the timestamps matches, resulting order is from lexicographical comparison of version ids.

Path parameters

- **testId** (required) – ID of test (one of `id` fields from `/tests` endpoint)

Return type

array[Version]

Example data

Content-Type: application/json

```
[ {
  "id" : "1501159669-7649a1c363f58f732b0503130ea93f0ef0719e15",
  "timestamp" : 1501159669
}, {
  "id" : "1501159669-7649a1c363f58f732b0503130ea93f0ef0719e15",
  "timestamp" : 1501159669
} ]
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses

200 Success

404 Test not found

500 Internal server error

`GET /tests/{testId}/revisions/{revisionId}/data`

Get data for specific test and version

Path parameters

- **testId** (required) – ID of test (one of `id` fields from `/tests` endpoint)
- **revisionId** (required) – ID of measurement version (one of the values returned from `/tests/${testId}/revisions` endpoint)

Return type

Data

Example data

Content-Type: application/json

```
{
  "data" : [ [ 18268.275199999567, 18268.275199999567 ],
             [ 18268.275199999567, 18268.275199999567 ] ],
  "units" : "ns/op"
}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses

200 Success Data

404 Version or test not found

500 Internal server error

C.3 Models

`Data`

- **units** String – Units of values (the same for all)
- **data** (optional) array[array[Double]] – Values separated by distinct measurements (for example JMH forks – different JVM instances).

`Test`

- **id** String – Unique identifier of this test case
- **name** String – Name of test case, usually full method name
- **metadata** (optional) map[String, String] – Optional test metadata (execution parameters, etc.)

`Version`

- **id** String – Unique version identifier, usually creation time of measured commit. The used format is Unix timestamp (number of seconds elapsed since January 1, 1970). If the timestamp is unknown, 0 is returned.
- **timestamp** Long – Version timestamp, usually creation time of measured commit. Used format is Unix timestamp (number of seconds elapsed since January 1, 1970). If the timestamp is unknown, 0 is returned.

D. Reviewer Demo

This appendix presents a quick starting guide for the reviewer of this thesis. The electronic attachment of the thesis contains a demonstration JMH project in a local Git repository. The project already have some benchmarks and a few performance measurements which can be immediately examined. We require Java at least 1.8 and Maven at least 3.5.0 to be installed.

1. Download the electronic attachment of the thesis, unzip it and open a terminal inside the `reviewer_demo/` directory.
2. Run the performance data visualizer and browse through the data. From the terminal run:

   ```
   $ java -jar measurements/spl-evaluation-java-1.0.4.jar -S
   ```

   The visualizer opens in the default web browser at `http://localhost:42000/index.html`. We recommend to compare all versions of the `measureDoubleLog` benchmark together and then add any version of the `measureSingleLog`. The change is caused by wrong implementation of the first benchmark which is fixed in the `8e89c2e` commit.
3. When finished, stop the visualizer server by hitting `Ctrl + C` in the terminal.
4. The `benchmarks.jar` file contains SPL formulas for the benchmarks. We can print them and find if any of the versions are missing. From the terminal run:

   ```
   $ java -jar measurements/spl-evaluation-java-1.0.4.jar \
     -j target/benchmarks.jar -p
   ```

   The result is that the formulas contain version `other` which is unknown in the measured data.
5. Bind the `other` version to a specific measurement and evaluate the formulas. The version mapping is prepared in the file `measurements/version_mapping.txt` file, so the evaluation is started from the terminal as follows:

   ```
   $ java -jar measurements/spl-evaluation-java-1.0.4.jar \
     -j target/benchmarks.jar -r measurements/version_mapping.txt
   ```

   The formulas are now successfully evaluated. Two of them violate, one complies.
6. Run the measurements with a custom version identifier:

   ```
   $ mvn clean install -Ddata_saver.revision_id=reviewerTest
   ```

   Omiting the argument would result into another measurement for the last Git version (requires Git to be installed).
7. These are the basics. More details are in the user documentation and the text of the thesis.