Performance Testing Support in a Continuous Integration Infrastructure

Sören Henning

Kiel University
Department of Computer Science
24098 Kiel, Germany

July 16, 2017

Abstract. The application of *continuous integration* allows an agile software development process by automating the build process, so it is nowadays seen as a good practice. However, if the build process is automatized, also the testing has to be automatized, to ensure quality and to detect faults while integrating. One possible quality characteristic of a software that could be checked is its performance.
In this paper, we propose a performance testing framework for Java that executes performance tests by using microbenchmarks. It uses the Java benchmarking toolkit JMH and can test for machine-dependent performance assertions. This framework can be included into the continuous integration server Jenkins, so performance tests will be executed automatically during the build process. We conducted an feasibility evaluation of this approach by applying it to the continuous integration infrastructure of the Pipe-and-Filter framework TeeTime.

1 Introduction

Continuous integration [3, 6] describes a practice in software engineering where the work of several developers or teams is integrated daily or multiple times per day. This facilitates an agile development process and enables the continuous delivery of software artifacts. When speaking about continuous integration, one normally refers to an automated process that is executed, for example, after a commit to the source code repository or daily at a specific time.

A continuous integration server typically tries to build, i.e., compile and link, the software. Moreover, it is common practice to automatically execute tests and perform further analyses in order to ensure that no faulty version of a software is released.

Whereas unit testing [10] is an established method to verify functional quality characteristics that is also used a lot in practice, automated performance testing is done only rarely. This is often due to the fact that performance testing is a time and resources consuming task. However, for many software such as web-based applications, performance is a highly relevant attribute as it is often business-critical. In this paper we present an approach how performance tests can be executed automatically in a continuous integration infrastructure for Java-based software.

In the following, we first state out the foundations of Java performance measurements in Section 2. Moreover, we present the technologies our approach and its evaluation relies on. After that, we describe how and why we use microbenchmarks for performance tests in Section 3. In Section 4 we explain how our proposed performance test framework works in detail, followed by a demonstration of how we integrated it into a continuous integration tool in Section 5. Afterwards, we evaluate our approach in Section 6 by applying it to the continuous integration infrastructure of a performance-sensitive Java framework. Finally, we summarize this work in Section 7 and show which points future work may address.

2 Foundations

2.1 Performance Measurements in Java

According to Eusgeld, Freiling, and Reussner [5], performance is the time behavior and the resource efficiency of a software. The ISO/IEC 25000 standard [11] defines this as one of six quality characteristics of software. In this paper, we only consider the execution time of Java code segments concerning performance.

As described by Georges, Buytaert, and Eeckhout [7], Blackburn et al. [2], and Horký et al. [9], the performance of Java program sections can differ significantly from run to run. The Java Virtual Machine (JVM) loads classes on demand when they are used the first time. Since class loading takes time, the program section that first executes this code is slower at this time. Another even more important reason is the Just-In-Time (JIT) compiler that dynamically recompiles the Java bytecode based on gained knowledge about the execution behavior of the program. Thus, it can optimize program sections that are executed frequently. To gain statistically significant measurement results, these aspects have to be considered. Thus, for conclusive results, it is appropriate to execute measurements multiple times: At first, to warm up the JVM and, subsequently, to calculate a mean value.

When measuring the performance of small program sections with sample inputs or states, as we propose it for performance tests, one has to take further aspects into account. Compilers (applies not only the JVM) perform lots of optimizations such as *dead code elimination* or *constant folding*.

2.2 The Java Microbenchmarking Harness (JMH)

Microbenchmarks are performance measurements of *small* code segments, whereby the size is not clearly defined. The Java Microbenchmarking Harness (JMH) [14] is a tool for declaring and executing such benchmarks for programming languages targeting the JVM.

Benchmarks are specially annotated Java methods that contain the program code that is supposed to be executed. With further annotations, one can configure the execution setting. Possible configuration parameters are, for instance, the duration of the warm up period, the number of measurements, or the number of JVM forks. Also one can choose between different modes that describe what JMH measures. This is, for example, how often a benchmark is executed per second or the average execution time of this benchmark. However, even though JHM handles a lot of configuration, it does not free the developer from writing meaningful benchmarks.

Typically, JMH benchmarks are defined in particular classes, separated from the actual program code. These benchmark classes enable more complex benchmarks that have states or interact with each other.

JHM provides a so called *Runner* that executes these benchmarks under consideration of the given configuration and measures the execution time. Based on this, it calculates a so called *score* in a settable unit, for example, the execution time directly or the throughput. Afterwards, JMH returns that score along with additional statistics.

In our performance testing framework, tests are defined and executed by means of JMH as described in Section 3.

2.3 The Continuous Integration Server Jenkins

Jenkins [12] is an open source web-based software for continuous integration. It can be set up for multiple projects that are built and tested independently from each other. In a common configuration, the build process is triggered by an external event. For instance, this can be a change to the source code, a manual user input, or a specific point in time. Afterwards, Jenkins checks out the source code (e.g., from a version control system) and successively executes one or multiple build actions such as compiling and testing. These build actions are typically sequences of command line instructions. A build can either succeed or fail, depending on the responses of the individual build steps. More specifically, a build is successful if and only if all build steps execute without reporting a failure.

Jenkins provides a graphical user interface that is accessible via a web browser. It allows to configure the build process and also to manually start it. Moreover, this interface shows the history of all builds along with information about whether they were successful or not. For each build, Jenkins also shows the console output that is printed during the build process. The single build steps usually use the console to display information to the user.

Furthermore, Jenkins functionality can be extended with plugins. The plot plugin [4] enables displaying charts in Jenkins' web interface based on data from CSV tables. We use Jenkins together with its plot plugin to display the course of performance measurements.

2.4 The Pipe-and-Filter Framework TeeTime

The Pipes-and-Filters pattern [15] is an architectural pattern for systems that process a stream of data. The individual processing steps are performed in components that are called filters. These filters can be connected by pipes, so that objects can be sent through these pipes and pass filter by filter [8].

TeeTime [16, 8] is a Java framework for developing software systems that are based on the Pipes-and-Filters pattern. It contains the basic entities *stages*, *ports*, *pipes*, and *configurations*. *Stages* are equivalent to the filters in the pattern. The framework provides multiple abstract stage types that can be extended at will. At its execution, a stage reads an object from its *input ports*, processes it in a defined way, and sends it to its *output ports*. In addition, the framework provides a number of predefined stages. A pipe connects an output port of one stage with an input port of another stage. In a configuration, stages could be defined and their ports connected. The framework takes care of creating the right pipes between the ports.

The framework we present in Section 4 is implemented with TeeTime. Moreover, we use TeeTime's continuous integration infrastructure as a case study in Section 6.

3 Using Benchmarks for Performance Tests

Unit tests consist of a sequence of program instructions and assertions about the behavior that must hold at certain points of this instructions. Typically, there is exactly one assertion for each test that is checked as the last statement of a test. We want something similar to this for performance tests and, therefore, decided that performance tests consist of a sequence of instructions and assertions about the performance of these instructions.

In our proposed framework, the instructions part of a test is a JMH benchmark. Hence, we do not have to implement an own mechanism to measure the execution time of tests and, also, do not have to launch the different steps of a reasonable test execution such as multiple measurement runs, decoupled JVM warm up runs, or JVM forks. Moreover, test developers can utilize the features of JMH to avoid compiler optimizations and, thus, write meaningful performance tests.

JHM benchmarks are often already used in Java projects for performance experiments or tests that have to be checked manually. This leads to another reason for choosing JMH since these benchmarks can directly be transformed to performance tests containing assertions that can be checked automatically.

A reasonable choice of benchmark parameters depends on the benchmark itself and on available resources. Therefore, it would not make sense to let our framework set these parameters by itself, so we decided to give the test developer the responsibility to write statistically significant tests.

Whereas for unit tests it is clearly determinable whether a test was successful or not, for performance tests this decision is more challenging. This is due to the fact that execution times may differ from run to run (as describes in Section 2) and that they also depend on the executing machine. To deal with the issue of slightly varying execution times on the same machine, assertions are not defined as fixed values but instead as intervals. Thus, tests are successful if their execution time is within the bounds of their assertion interval. A sole definition of upper bounds is often not sufficient since also a sudden improvement in performance can be an indicator of unintended behavior. However, if a lower bound is actually unnecessary, it simply can be set to zero. Different execution times on different machines are handled by individual assertions that must be defined for each machine that executes the tests.

In contrast to unit tests, assertions for performance tests may change from time to time, for instance, when implementations of algorithms are replaced by more efficient ones. In this case, the assertions have to be adjusted appropriately.

4 Proposal for a Performance Testing Framework

We have developed a performance testing framework for Java called RadarGun[1]. One can execute it in two different ways: First, as a command line program and, second, it provides a Java API. RadarGun can be seen as an encapsulated function that obtains a set of performance tests, i.e., benchmarks and associated performance assertion bounds, as input and outputs information about every test whether it was successful or not and, additionally, the actual value.

The actual procedure corresponds to a pipeline-like processing based on the following steps: Execution of the passed benchmarks, comparison with their corresponding assertion, and finally generation of reports or further actions. For this reason, we decided to implement the proposed framework with the Pipe-And-Filter framework TeeTime. This enables an encapsulating of the individual processing steps and an abstraction from the data exchange and the pipeline's execution. Further advantages are a more flexible expandability and a higher exchangeability of processing steps.

Figure 1 gives an overview of RadarGun's Pipe-and-Filter architecture. The *Benchmark Runner* is a producer stage that creates the initial elements for this configuration by running the desired benchmarks. Afterwards, it forwards the benchmark results to the *Results Comparator*. This stage compares each of them with its corresponding assertion. The result of this comparison is forwarded to a *Distributor* that broadcasts it to the stages *Results Printer*, *CSV Exporter*, and *Exit on Fail Stage*. For each of these stages it is configurable whether it should be used or not. The Results Printer outputs the test result on the system's console, the CSV Exporter stores it in a CSV table file, and the Exit on Fail Stage aborts the whole program execution as soon as one test fails. In TeeTime every configuration is a stage itself. Therefore, RadarGun's Pipe-and-Filter configuration exposes one of the distributor's output ports. In this way, one can extend RadarGun with custom stages.

[1] `https://github.com/SoerenHenning/RadarGun`

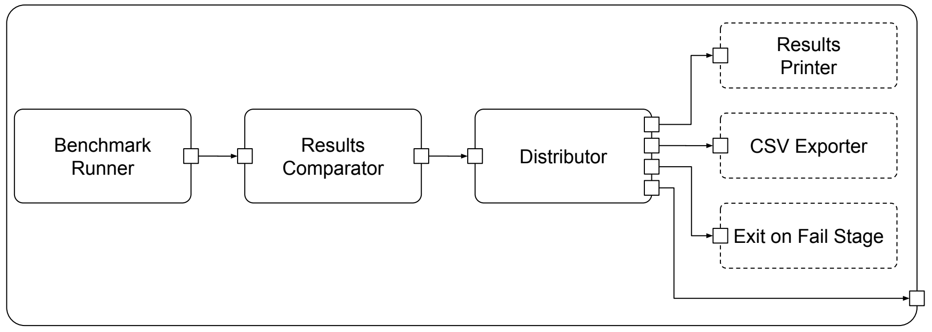

Fig. 1: Overview of RadarGun's Pipe-and-Filter architecture

4.1 Execution of JMH Benchmarks

The *Benchmark Runner* stage can be seen as a wrapper around JMH. It executes a given set of JMH benchmarks using JMH's `Runner` class. If not specified more precisely, it executes all benchmarks found by JMH in the current Java classpath. However, when RadarGun's Java API is used, a custom JMH `Runner` object can be passed to define more precisely the benchmarks to run. Moreover, one can configure whether the default JMH output should be printed to the console. After the JMH runner finished its execution the Benchmark Runner stage catches the benchmark results and forwards them to next processing stage.

4.2 Comparison with Predefined Assertions

As explained in Section 3, performance assertions have to be defined for each machine individually that executes these performance tests. Therefore, we have to decide how the executing machine will be identified and where the assertions are defined.

Machine Identification A clear identification of machines is difficult. On the one hand, from a more theoretical point of view, since it is difficult to find a definition for the equality or similarity of machines. On the other hand, even if we had such a definition, it would probably state something like: Two machines are equal if and only if all their components and configurations are equal. Then, this would be difficult to implement as it is technically problematic to read out all components and configurations of a system and to compare them.

Thus, we have opted for a more practical approach by making the process of machine identification configurable and exchangeable. RadarGun defines the Java interface *Machine Identifier* (see Listing 1, `MachineIdentifier`) that potential identifiers have to implement. The `testMachine()` method of an identifier supplies a boolean value indicating whether the executing machine matches this identifier or not. In this way, miscellaneous identifiers can be implemented that differ in accuracy.

One good option to clearly identify machines is, for instance, taking advantage of the way how they are identified in a network, for example, with their MAC or IP address. A problem with this is that computers can have multiple network interfaces and, in particular, none. Moreover, an identification by the IP address is only possible if it is statically specified. Most operating systems have the concept of something like a computer name. This enables another option to identify a machine. We have implemented several identifiers, which are outlined in the following.

Network Address Identifier This identifier is constructed with an array of network addresses which can either be IP addresses or host names. Its test method returns true if the executing machine has one of those host names or IP addresses.

Mac Address Identifier This identifier behaves similar to the Network Address Identifier. However, it tests for one or more MAC addresses.

Windows Computername Identifier Also this identifier behaves similar to the Network Address Identifier. However, it tests for one or more Windows computer names. Thus, it can only be used on Windows machines.

Wildcard Identifier The test method of this identifier returns true for all machines. It can be used when no machine distinction is necessary, for instance, if there is only one machine.

Dismiss Identifier The test method of this identifier returns false for all machines. In our implementation, we use it for realizing the Null Object pattern [13].

By using the described identifier interface, it is possible to extend RadarGun by further identifiers in a later step or to use it with own custom identifiers.

In order to load identifiers dynamically, as described later, they must provide a constructor that accepts an arbitrary number of strings as arguments. In the example of the MAC address identifier these are one or multiple MAC addresses for which the executing system will be tested.

Location of Test Assertions One possibility of defining assertions could be to do this directly in the benchmarks. For example, this could be done with Java annotations like the other configuration options of JMH benchmarks. However, this implies several disadvantages. For a high number of machines, declaring all

Listing 1: The interface for machine identifiers

```
public interface MachineIdentifier {

   public boolean testMachine();

}
```

assertions in the benchmarks themselves would lead to long and overcomplicated test cases. When a new machine is added, one have to modify all tests. This is especially challenging if tests are executed on machines that are not under control of the source code developers. Assume, for example, someone wants to build an open source software project in which development he was not involved. If he now wants to execute performance tests, he has to modify the source code for all tests (at least in a local copy) to define his own machine specific assertions. Linked to this, when new test cases are defined, it can be cumbersome to obtain an overview which machine exists at all. Furthermore, the test execution instructions would not be longer plain JMH benchmarks when using this method.

For this reasons, we decided to separate the benchmarks from the assertions entirely. First, there are the benchmarks that are plain JMH benchmarks and, second, there are assertions that are located in separate files.

Declaration of Test Assertions Assertions are defined in files in the YAML data serialization standard [1]. We selected YAML since its syntax is designed to be human-readable. A YAML block (start declared by three hyphens) always describes a set of assertions for a certain machine. As defined by the YAML standard, multiple blocks can be listed in one file. Also multiple YAML blocks can describe the same or overlapping machines.

Listing 2 shows an example of such a YAML file containing one block. Firstly, it defines the machine by an identifier class (Line 2) and the parameters by which it will be created (Line 3). Afterwards, all assertions are declared by the fully qualified name of the benchmark and the lower and upper bound for the permitted benchmarks result, i.e., the JMH score (Lines 4-7).

Comparison of Actual Score with Test Assertions RadarGun can be executed with a set of multiple YAML files or directories from which is selects all containing YAML files. These files cannot be located in the file system only, but also in the current classpath. Thus, one can place the test assertions at the same place as the benchmarks, for instance, in the same Git repository. In this way, the declaration of assertions is flexible and adaptable for different use cases or workflows. For each found definition of assertions, the framework creates a

Listing 2: Example of a YAML file that declares assertions

```
1 ---
2 identifier: MacAddressIdentifier
3 parameters: [01:23:45:67:89:AB]
4 tests:
5   myproject.benchmark.MyFirstBenchmark.run: [70, 90]
6   myproject.benchmark.MySecondBenchmark.run: [6.4, 6.7]
7   myproject.benchmark.MyThirdBenchmark.run: [1300, 1400]
```

Java object representing the defined machine identifier. Then it checks whether it matches the executing machine and, if so, collects the declared assertions.

When now the Results Comparator processes benchmark results, it compares the actual value with the lower and upper bound and creates a proper test result object. The possible test results are:

1. The benchmark result value is within the assertion bounds.
2. The benchmark result value is greater than the assertion's upper bound.
3. The benchmark result value is less than the assertion's lower bound.
4. There is no assertion for this test.

After a test result object for one these types is created, it is forwarded to the succeeding stages in the Pipe-and-Filter configuration.

4.3 Further Processing of Test Results

In a last step, RadarGun can handle the test results in different ways. Therefore, it connects corresponding stages to its Pipe-and-Filter configuration. These stages behave independent from each other so a user can specify the desired actions. Moreover, this is extendable for further actions, for instance, export to other formats.

The Results Printer outputs the result for every test on the system's standard output stream or a different one if configured. The actual output is one line per test that states the result status, i.e., failed, successful, or not executed, the benchmarks name, the test result's value, and its assertion. This output is activated per default since RadarGun is typically executed from the command line.

The CSV Exporter exports every test result to a separate CSV table. This provides a simple approach to keep track of the test executions. Besides the actual measured value, it stores also the assertion bounds as they can change between test executions. This stage appends new values always to the end of the table or, if that does not exist, creates a new one. As suggested in Section 5, Jenkins can use these CSV tables to generate a visualization of the test results.

The Exit on Fail Stage terminates RadarGun with an exit code of -1 if at least one test has failed. This is the common way to signal a continuous integration tool that the build process should fail. For this stage, we provide two different implementations. The first one terminates as soon as the first test fails. Hence, other tests will not be analyzed anymore. The second one, waits until all tests have been analyzed and terminates RadarGun if one or more of them has failed. Which of the two implementations is more appropriate depends on the use case.

5 Executing RadarGun by a Continuous Integration Tool

RadarGun is developed with the intention to be easily includable into a continuous integration process. In the following, we describe the integration into the

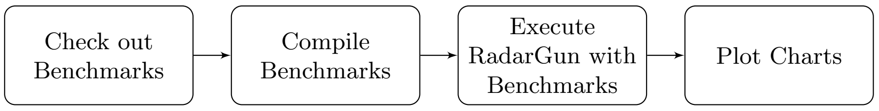

Fig. 2: Steps of an automatic execution of performance tests with RadarGun

continuous integration server Jenkins as an example. Although not demonstrated here, this approach might be similar for other continuous integration tools.

Besides the execution of performance tests, further desired functionalities are the output of test results, a generation of charts that plot the course of test results and perhaps bounds, and the ability to abort the build process to avoid releasing versions with performance issues.

These requirements lead to a sequence of actions as visualized in Figure 2. First of all, Jenkins has to check out the JMH project containing the benchmarks. Afterwards it compiles them using Apache Maven. RadarGun executes these tests in the next step, produces an output to Jenkins' console, and creates or updates the CSV tables. For this purpose, RadarGun has to be configured accordingly via command line options. In the last step, the Jenkins Plot Plugin loads these CSV tables and creates a chart for each of them. For this purpose, in Jenkins' configuration a separate plot has to be generated for each test. RadarGun provides an automatically abort of the build process if one or more tests fail which can be very useful. However, the Plot Plugin can only update plots in *after build action*. Thus, the plots will not b updated and therefore will not show the failing builds.

6 Feasibility Evaluation

The Pipe-and-Filter framework TeeTime (Section 2.4) has a completely automated build infrastructure[2]. On a daily basis, a Jenkins server builds a so called *snapshot* from the current version of the source code. Afterwards, Jenkins checks the quality of this build with various tools and, on success, releases it.

High performance is a key feature of TeeTime, so an automatic detection of performance changes between builds is particularly important. Since good performance results are shown and published [17], deviations in performance may make these results invalid. Therefore, integrating automatic performance tests is a way to detect this.

In a feasibility evaluation, we show how automated performance testing with RadarGun can be added to a continuous integration infrastructure by the example of TeeTime. Since TeeTime already provides JMH benchmarks[3], we can transform them to RadarGun tests.

[2] `https://build.se.informatik.uni-kiel.de/jenkins/view/TeeTime`
[3] `https://build.se.informatik.uni-kiel.de/teetime/teetime-benchmark`

6.1 Methodology and Test Scenarios

Jenkins builds and publishes the current version of TeeTime daily at a specific time in the project *teetime-nightly-release*. We created a new project called *teetime-nightly-performance-test* for automatically executing TeeTime's performance tests. The start of this project is triggered after the build of *teetime-nightly-release* has finished.

TeeTime currently provides three benchmarks. Each of them measures the performance of a different way to detect termination signals when they are received from pipes. We convert these benchmarks to performance tests by adding assertions for each of them. These assertions are declared in a YAML file located in a separate branch of the TeeTime benchmarks Git project. To determine reasonable assertion bounds we executed the tests several times and examined the range of the execution times.

We set up the nightly performance test project as describes in Section 5. Jenkins first loads the source code of the TeeTime benchmarks projects, where we also have defined the assertions. Then, it compiles the benchmarks and, afterwards, Jenkins loads the latest version of RadarGun and executes it with the compiled benchmarks. For every test, we define an after build action that plots the resulting values for this test.

For our feasibility evaluation, we only consider one benchmark since the benchmarks do not differ in their methodology. Therefore, we selected the `Port2PortBenchmark`. To simulate deviations in performance we intentionally decelerate the benchmarks by using JMH's blackhole[4]. It burns CPU cycles according to the given workload value, in our case by a value of 10. Before starting the actual evaluation, we executed the test multiple times and observed that the execution time in all runs is between 30 nanoseconds per operation (ns/op) and 35 ns/op. Thus, we define these values as the assertion for this test. In the following, we describe the scenarios we analyzed in our feasibility evaluation.

S1: Result within bounds First, we evaluate the case where the measured execution time matches the specified assertion. This means the measured value is greater or equal than the lower assertion bound and lower or equal than the upper bound.

For a correct behavior, we expect that Jenkins produces a console output, which confirms that the test was successful. In addition, it should also print the test's name, its assertion bounds, and the actual execution time including further statistical information provided by JHM. Furthermore, we expect that the Jenkins Plot Plugin updates the chart for this benchmark by adding new values on the x axis. These values should be the actual measurement, the upper, and the lower assertion bound. We also expect that they correspond with the values that are printed to the console. Thus, the measured value has to be between the values for the lower and upper bound.

[4] `org.openjdk.jmh.infra.Blackhole.consumeCPU(long tokens)`

S2: Result lower than lower bound In a second scenario, we evaluate the case that the execution time is lower than the lower assertion bound. The benchmark was therefore faster than expected. To simulate this behavior we remove the previously set deceleration.

In this scenario, we expect that Jenkins produces a console output that indicates that this test has failed. In addition, further information as described in scenario S1 should be displayed. Also the chart should be extended by a new entry that displays the actual value, the lower, and the upper bound. These values should correspond to the values displayed on the console. The new chart entry for the actual value should be below the entry for the lower bound.

S3: Result greater than greater bound The third evaluation scenario is that the benchmark is slower than expected and hence its execution time is greater than the upper assertion bound. To simulate this behavior we increased the deceleration from a value of 10 to a value of 15.

As in scenario S2, we expect a console output that displays the fail of this test and also the additional information. Also in this scenario we expect that the chart is extended by a new entry that displays the actual value as well as the lower and the upper bound for this test. These values should corresponds with the values displayed on the console. Since in this case the measured value is greater than it should be, we expect that it exceeds the bounds.

In total, we perform 20 builds, whereby we first execute scenario S1 ten times, than S2 one time, than S1 again six times, than S3 one time, and afterwards S1 two times. In this way, we expect to obtain a realistic plot. Since scenario S1 is executed 18 times in total, we evaluate its output exemplary by the first build. Since we conducted some experiments before the actual evaluation, the counter of the considered builds starts at #59 and thus runs to #78.

All evaluation scenarios are executed at the build server of the Software Engineering Group at Kiel University with Jenkins version 2.64. The JHM benchmarks are built with Apache Maven 3.2.3 and RadarGun is executed by Oracle Java 1.8.0_40.

6.2 Results and Discussion

Figure 3 shows the chart that is generated by the Jenkins Plot Plugin. It contains the actual execution time (called *score*, red color) as well as the lower assertions bound (green) and the upper assertion bound (blue) in ns/op in relation to the build number. The chart displays the last 20 builds from build #59 to build #78.

The values for the lower and the upper bound are constant at 30 ns/op and 35 ns/op, respectively. This corresponds to the defined assertions and, thus, to the expected behavior. For most builds, the score value fluctuates slightly but stays between the lower bound and the upper bound. At build #69 it exceeds the upper bound and reaches approximately 40 ns/op whereas at build #76 it

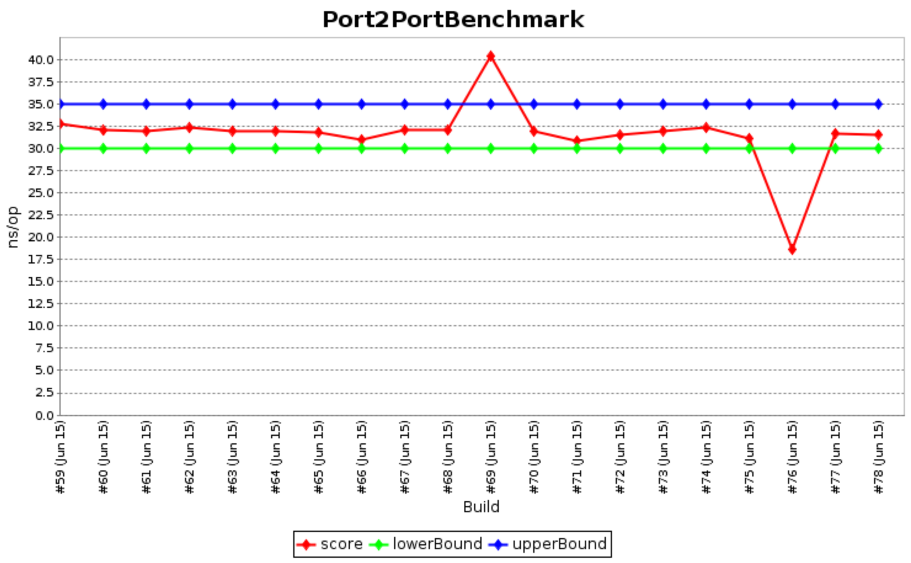

Fig. 3: The chart generated by Jenkins in our feasibility evaluation

falls to approximately 18 ns/op. Also this is expected as we executed scenario S2 and S3 in this builds and scenario S1 in all the others.

Listing 3 shows RadarGun's output on Jenkins' console after the test in build #59 was executed. First it states that this test was successful, followed by the test name. After that, the actual score containing additional statistical information is printed. Finally, the assertion bounds are displayed in parentheses. They comply with the values declared in the YAML file. This build executes scenario S1 which represents a successful test. In summary, the console output is in accordance with our expectations.

Listing 3: RadarGun console output for build #59

```
[SUCCESSFULL] teetime.benchmark.Port2PortBenchmark.queue Score: 32.777
    ±(99.9% 1.982 ns/op (Bounds: [30.0, 35.0])
```

The output for build #69 is shown in Listing 4. Its schema corresponds to the output for build #59. However, this time the measured score exceeds the assertions, so the test is consequently labeled as failed. This meets our expectations since in this build scenario S2 was executed.

Listing 4: RadarGun console output for build #69

```
[FAILED] teetime.benchmark.Port2PortBenchmark.queue Score: 40.386 ±(99.9%
    1.637 ns/op (Bounds: [30.0, 35.0])
```

Listing 5 shows the output after build #76. Here again, the schema corresponds to the output schema of build #59. This time, the score undercuts the assertions and the test is also labeled as failed. In this build, scenario S3 was executed so the output is as desired.

Listing 5: RadarGun console output for build #76

```
[FAILED] teetime.benchmark.Port2PortBenchmark.queue Score: 18.613 ±(99.9%)
    4.536 ns/op (Bounds: [30.0, 35.0])
```

For all of the three evaluated scenarios, the values from the console output, i.e., score, lower bound, and upper bound, comply with the values printed in the plot. Moreover, they equal the values that are set in the assertions declaration and output by JHM. Thus, we conclude that our approach works as desired.

6.3 Threats to Validity

We evaluated our approach with only one TeeTime benchmark. However, JMH provides further test methods and configuration parameters, which we do not have used. For instance, this is measuring the throughput instead of the execution time. To increase the validity, we also have to evaluate other types of benchmarks.

Moreover, our evaluation was only executed with one hardware and software environment. Thus, we cannot guarantee that our approach is also feasible in other environment. An execution on different systems would increase the validity.

7 Conclusions

In this paper, we presented an approach of how performance testing can be included into a continuous integration infrastructure. Hence, we pointed out necessary requirements for a performance testing framework and, afterwards, presented an implementation called RadarGun.

This performance testing framework utilizes JMH to execute benchmarks and extends it by providing the ability to define assertions for them. RadarGun can be integrated in the build process with an continuous integration tool, to execute those performance tests automatically, e.g., before publishing releases, every day, or after every commit. To show the feasibility of our approach, we applied it to the open source project TeeTime, which has high performance requirements.

We designed RadarGun primarily to write and execute performance tests for Java programs. However, since we basically rely on JHM, it is likely that RadarGun is also applicable for other emerging JVM-based languages such as Kotlin or Scala. This may be analyzed in future works.

The manual configuration of the Jenkins Plot Plugin can be cumbersome for a high number of performance tests. Thus, a RadarGun Jenkins plugin is currently under research. It could automatically generate visualizations for all tests. Apart from this, the plugin could also generate interactive charts instead of the current static ones to allow zooming and moving through the build history. As there seems to be no way to update the chart after a failed build, this may also be addressed by it.

References

[1] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. *YAML Ain't Markup Language (YAML) (tm) Version 1.2*. Tech. rep. YAML.org, Sept. 2009.

[2] Stephen M. Blackburn et al. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *Proceedings of the OOPSLA*. 2006.

[3] G. Booch. *Object-oriented Analysis and Design with Applications*. Benjamin/Cummings series in object-oriented software engineering. Benjamin/Cummings Publishing Company, 1994.

[4] Nigel Daley and Eric Nielsen. *Jenkins Plot Plugin*. Accessed: 2017-06-12. 2017. URL: `https://plugins.jenkins.io/plot`.

[5] Irene Eusgeld, Felix C. Freiling, and Ralf Reussner, eds. *Dependability Metrics: Advanced Lectures*. Springer-Verlag, 2008.

[6] Martin Fowler and Matthew Foemmel. *Continuous integration*. Accessed: 2017-06-18. 2006. URL: `https://www.thoughtworks.com/continuous-integration`.

[7] Andy Georges, Dries Buytaert, and Lieven Eeckhout. "Statistically Rigorous Java Performance Evaluation". In: *Proceedings of the OOPSLA*. 2007.

[8] Sören Henning. "Visualization of Performance Anomalies with Kieker". Bachelor's Thesis. Kiel University, Sept. 2016.

[9] Vojtěch Horký et al. "DOs and DON'Ts of Conducting Performance Measurements in Java". In: *Proceedings of the ICPE*. 2015.

[10] Institute of Electrical and Electronics Engineers. "IEEE Standard for Software Unit Testing". In: *ANSI/IEEE Std 1008-1987* (1986).

[11] International Organization for Standardization. "Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE". In: *ISO/IEC 25000:2014* (Mar. 2014).

[12] Kohsuke Kawaguchi. *Jenkins*. Accessed: 2017-06-12. 2011. URL: `https://jenkins.io`.

[13] Robert C. Martin, Dirk Riehle, and Frank Buschmann, eds. *Pattern Languages of Program Design 3*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[14] OpenJDK. *Java Microbenchmarking Harness*. Accessed: 2017-06-12. 2017. URL: `http://openjdk.java.net/projects/code-tools/jmh`.

[15] M. Shaw. "Larger Scale Systems Require Higher-level Abstractions". In: *SIGSOFT Softw. Eng. Notes* (Apr. 1989).

[16] Christian Wulf, Wilhelm Hasselbring, and Johannes Ohlemacher. "Parallel and Generic Pipe-and-Filter Architectures with TeeTime". In: *International Conference on Software Architecture (ICSA) 2017*. Apr. 2017.

[17] Christian Wulf, Christian Claus Wiechmann, and Wilhelm Hasselbring. "Increasing the Throughput of Pipe-and-Filter Architectures by Integrating the Task Farm Parallelization Pattern". In: *Proceedings of the CBSE*. Apr. 2016.