RadarGun: Toward a Performance Testing Framework

Sören Henning, Christian Wulf, and Wilhelm Hasselbring
Software Engineering Group, Kiel University, Germany
(sh,chw,wha)@informatik.uni-kiel.de

Abstract

We present requirements on a performance testing framework to distinguish it from a functional testing framework and a benchmarking framework. Based on these requirements, we propose such a performance testing framework for Java, called *RadarGun*. RadarGun can be included into a continuous integration server, such as Jenkins, so that performance tests are executed automatically during the build process. We conducted a feasibility evaluation of this approach by applying it to the continuous integration infrastructure of the Pipe-and-Filter framework TeeTime.

1 Introduction

The performance of Java program sections can differ significantly from run to run [1, 3]. One cause is the on-demand class loader of the Java Virtual Machine (JVM). The first access on a particular class takes more time than all successive ones. Another cause is the Just-In-Time compiler that recompiles and optimizes the Java bytecode at runtime based on gained knowledge about the execution behavior. Hereby, the JVM decides non-deterministically which of its many optimization strategies it should apply.

Hence, it is necessary to execute and measure the program section of interest multiple times: At first, to warm up the JVM and, subsequently, to calculate a mean value. Moreover, the JVM as a whole needs to be executed repeatedly to reduce the influence of its non-deterministic choice of optimization strategies.

Functional testing frameworks like JUnit[1] and Citrus[2] do not provide sufficient support for measuring performance in this way. Likewise, benchmark frameworks such as JMH [6] and MooBench [4] are not sufficient since they lack support for automatically comparing and evaluating the measurement results.

With this paper, we make the following three contributions: First, we define requirements on a performance testing framework in order to distinguish it from a (unit/integration) testing framework and a benchmark framework. Second, we present a prototype implementation of such a performance testing framework, called *RadarGun*. Third, we evaluate RadarGun by applying it within the continuous integration environment of the TeeTime project [7].

[1] http://junit.org
[2] http://www.citrusframework.org

2 Requirements

A performance testing framework should fulfill the following four requirements: (1) The framework must be able to *automatically and repeatedly* execute the JVM and the program section of interest to gain statistically significant and stable measurement results. (2) The framework must be able to *automatically aggregate* measurements to a single, representative measurement score, e.g., the minimum, the median, the maximum, or the average. This measurement score serves as the base for the subsequent comparison with the expected execution time. (3) The framework must provide an assertion mechanism which checks whether the measurement score is within a time *interval*. Besides the influences caused by the JVM (see Section 1), there are additional influences, such as the operating system's scheduler, which can cause small, but unavoidable variations in the final measurement score. Note that a sole definition of an upper bound, like in JUnit, is often not sufficient. A sudden, unintended improvement in performance can also be an indicator of an erroneous behavior. For example, a method can return prematurely due to an invalid argument exception and thus takes less time than usual. Hence, a reasonable assertion for a performance test must define a time interval instead of a single time value. (4) The framework must automatically differentiate between different machines on which the measurements are collected. The measurement score of a particular performance test highly depends on the actual executing machine so that the expected score must be specified and loaded in a *machine-dependent* way.

3 The RadarGun Framework

We developed RadarGun[3], an open-source performance testing framework for Java, which meets all of the requirements from Section 2. Its main idea is to execute JMH [6] benchmarks and compare the results with predefined performance assertions associated with the machine which executed the benchmarks.

By using JMH, we do not have to implement a test execution engine in order to perform multiple runs of the benchmarks. JMH automatically performs multiple JVM forks, warm-up runs, and actual measure-

[3] https://github.com/SoerenHenning/RadarGun

ment runs. Moreover, we do not have to implement an own mechanism to measure and to aggregate the execution times of performance tests. JMH automatically computes all relevant measurement metrics mentioned in Section 2. Finally, JMH benchmarks are often already used in Java projects for performance experiments. These benchmarks can thus be used by RadarGun without any modification.

Different execution times on different machines are handled by individual assertions that must be defined for each machine executing the tests. These assertions are specified in a text file. After executing a set of benchmarks, RadarGun automatically loads the assertions associated with the underlying machine from this text file and checks whether the results lie within the specified time intervals. In this way, tests are evaluated as successful or failed, respectively. In Section 4, we describe this approach in more detail.

RadarGun can be used in two different ways: either via the command line or via its Java API. In both cases, it obtains a set of performance tests as input and returns information about the results of the tests as output. Internally, it consists of a Pipe-and-Filter architecture with one filter for each of the three processing steps mentioned above: executing the passed benchmarks, comparing them with their corresponding assertions, and generating the report of the results. Listing 1 shows an example console output for three runs of the performance test named `teetime.benchmark.Port2PortBenchmark.queue`.

```
[SUCCESSFUL]
teetime.benchmark.Port2PortBenchmark.queue
Expected score interval: [30.0, 35.0]
Actual score: 32.777 ± 1.982 ns/op (ci=99.9%)

[FAILED]
teetime.benchmark.Port2PortBenchmark.queue
Expected score interval: [30.0, 35.0]
Actual score: 40.386 ± 1.637 ns/op  (ci=99.9%)

[FAILED]
teetime.benchmark.Port2PortBenchmark.queue
Expected score interval: [30.0, 35.0]
Actual score: 18.613 ± 4.536 ns/op  (ci=99.9%)
```

Listing 1: RadarGun console output for build #59, #69, and #76, respectively.

4 Performance Assertions

Since assertions for performance tests depend on the machine which executes these tests, we first describe how RadarGun identifies and distinguishes different machines. Subsequently, we describe where and how performance assertions are specified.

Machine-Dependent Assertions A clear identification of machines is difficult. On the one hand, from a theoretical point of view, it is difficult to find a definition for 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 of their components and configurations are equal. Then, a corresponding implementation would be difficult as it is technically challenging to read out all components and configurations of a system and to compare them. Note also that when using virtual machines, details of the associated hardware resources are often not fully disclosed.

For this reason, we have opted for a more practical approach by making the process of machine identification configurable and exchangeable. RadarGun defines a Java interface `MachineIdentifier` which potential identifiers have to implement. The single method `testMachine()` 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. We provide two machine identifiers with RadarGun: one based on the network address and one based on the MAC address. By using the described identifier interface, it is possible to extend RadarGun by further, custom identifiers.

Location of Assertions We decided to fully separate the benchmarks from the assertions so that existing benchmarks can directly be used as performance tests without any modifications. Moreover, benchmarks therefore do not need to be recompiled if we change an assertion or add assertions for new machines. Finally, this approach also increases the maintenance for performance testers because all assertions for a single machine are located at a central position.

Format of Assertions Assertions are defined in a text file which conforms to the YAML data serialization standard[4]. We selected this file format since its syntax is designed to be human-readable. A YAML file consists of multiple YAML blocks where each block, beginning with three hyphens, describes a set of assertions for a certain machine. Listing 2 shows an example of such a YAML file containing one block. It defines the machine by an identifier class (Line 2) and associated constructor parameters (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 (Line 4-7).

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

Listing 2: Example of a YAML file that declares assertions

5 Feasibility Evaluation

The Pipe-and-Filter framework TeeTime [7] has a fully automated build infrastructure[5]. On a daily basis, a Jenkins server builds a *snapshot* version from the

[4] `http://www.yaml.org/spec/1.2/spec.html`

[5] `https://build.se.informatik.uni-kiel.de/jenkins/view/TeeTime`

latest version of the source code. To ensure its high performance shown in [5], TeeTime already provides JMH benchmarks[6]. In this section, we show that and how RadarGun can utilize these benchmarks in order to integrate performance tests into TeeTime's existing build environment. Especially, we show how RadarGun visualizes passed and failed performance tests.

Methodology and Test Scenarios Currently, TeeTime provides three benchmarks. Each of them measures the performance of a different way to read elements and termination signals from a pipe. We converted these benchmarks to performance tests by adding assertions for each of them in a corresponding YAML file. Afterwards, we created a new Jenkins project which automatically executes these tests with RadarGun when a new build of TeeTime has finished.

In this paper, 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 feature. It burns CPU cycles according to the given workload value (in our case: 10). Afterwards, we executed the benchmark multiple times in order to identify a valid lower and upper bound for a successful test. We measured an average score in all runs between 30 and 35 nanoseconds per operation (ns/op).

We evaluated three different scenarios with this benchmark. In the first scenario (S1), we consider a run whose average score lies within the assertion bounds. This test should pass successfully. In the second and third scenario (S2 and S3), we consider a run whose average score is above the upper assertion bound and, respectively, below the lower assertion bound. Thus, both tests should fail.

In total, we performed 20 builds which trigger the following scenarios: ten times S1, one time S2, six times S1 again, one time S3, and finally two times S1 again. In this way, we expect to obtain a realistic plot. Since S1 is executed 18 times in total, we evaluate its output representatively by the first build.

We executed all of the evaluation scenarios on a non-virtual machine (Intel Xeon E5620, 16 GB RAM) with Jenkins 2.64 and Oracle's Java virtual machine 1.8.0_40. The benchmarks were written with JHM 1.9.2 and are built with Maven 3.2.3.

Results Figure 1 shows the result chart generated by Jenkins. It illustrates the individual measurement scores of the 20 builds (red color) as well as the associated lower (green) and upper (blue) assertion bound.

The values for the lower and the upper bound are constant at 30 and 35 ns/op, respectively. This corresponds to the predefined assertions. For most builds, the score fluctuates slightly, but stays within the assertion bounds. At build #69, the score exceeds

[6] `https://build.se.informatik.uni-kiel.de/teetime/teetime-benchmark`

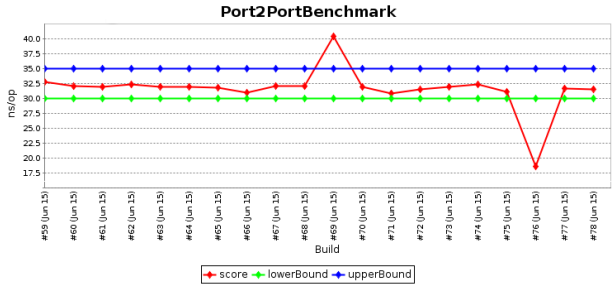

Figure 1: The result chart generated by Jenkins

the upper bound and reaches approximately 40 ns/op whereas at build #76, it goes down to approximately 18 ns/op. These results correspond to our expectations for S1-S3. Moreover, the values from the console output (see examples in Listing 1) perfectly comply with the values displayed in the chart. Thus, we conclude that our approach works as desired.

6 Conclusions

In this paper, we identify requirements for a performance testing framework and present a corresponding prototype implementation, called *RadarGun*. Performance tests are based on JMH benchmark enriched with machine-dependent time interval assertions. We showed that RadarGun can be integrated into the build process with a continuous integration tool to execute those performance tests automatically.

As future work, we plan to integrate RadarGun into Kieker's [2] build infrastructure and to develop an improved, interactive Jenkins plot plugin that eases the usability of performance testing charts.

References

[1] A. Georges, D. Buytaert, and L. Eeckhout. "Statistically Rigorous Java Performance Evaluation". In: *Proceedings of the OOPSLA*. 2007.

[2] A. Van Hoorn, J. Waller, and W. Hasselbring. "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis". In: *Proc. of the ICPE*. 2012.

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

[4] J. Waller, N. C. Ehmke, and W. Hasselbring. "Including Performance Benchmarks into Continuous Integration to Enable DevOps". In: *SIGSOFT Softw. Eng. Notes* 40.2 (2015), pp. 1–4.

[5] C. Wulf, C. C. Wiechmann, and W. Hasselbring. "Increasing the Throughput of Pipe-and-Filter Architectures by Integrating the Task Farm Parallelization Pattern". In: *Proc. of CBSE*. 2016.

[6] OpenJDK. *Java Microbenchmarking Harness*. `http://openjdk.java.net/projects/code-tools/jmh`. 2017.

[7] C. Wulf, W. Hasselbring, and J. Ohlemacher. "Parallel and Generic Pipe-and-Filter Architectures with TeeTime". In: *Proc. of ICSA*. 2017.