(h|g)opper: Performance History Mining and Analysis

Christoph Laaber
Department of Informatics
University of Zurich
Switzerland
laaber@ifi.uzh.ch

Philipp Leitner
Department of Informatics
University of Zurich
Switzerland
leitner@ifi.uzh.ch

ABSTRACT

Performance changes of software systems, and especially performance regressions, have a tremendous impact on users of that system. Historical data can help developers to reason about how performance has changed over the course of a software's lifetime. In this demo paper we present two tools: *hopper* to mine historical performance metrics based on benchmarks and unit tests, and *gopper* to analyse the data with respect to performance changes.

1. INTRODUCTION

Performance is an important factor when building and maintaining software systems. Therefore performance changes - either regressions or improvements - are of interest to software developers. Nevertheless performance changes, in particular regressions, are often not recognized for a long time [2]. While application performance management tools such as NewRelic[1] provide monitoring capabilities of deployed systems, performance testing presents the developer early feedback about a system's performance. However, performance testing is not as well understood among developers as unit testing is [3]. Statistical methods such as t-tests and change point analysis provide the means for detecting performance changes. A different approach to looking at performance test results is applying a cost model to detect potential performance regressions [1].

Reasoning about performance changes over a software's lifetime might give developers insight into which kind of changes led to a particular change. Ideally developers can deduct which changes can possibly introduce performance changes and use that information to their advantage.

This demo paper introduces two command-line tools for mining historical performance data based on tests and analysing the acquired results with respect to performance changes. Both, *hopper*[2] and *gopper*[3] are available online.

[1]https://newrelic.com
[2]https://github.com/sealuzh/hopper
[3]https://github.com/sealuzh/gopper

ICPE'17 April 22-26, 2017, L'Aquila, Italy

ACM ISBN 978-1-4503-4404-3/17/04.
DOI: http://dx.doi.org/10.1145/3030207.3053662

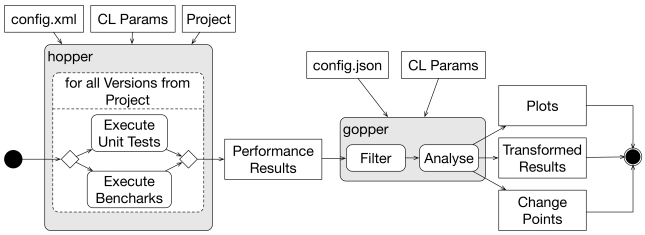

Figure 1: Sample Workflow

2. MINING PERFORMANCE DATA

Figure 1 depicts an example workflow for mining and analysing historical performance data, similar in idea to the methods used in previous research [1]. The first stage - on the left in Figure 1 - of the presented tool chain is *hopper*, a tool to extract historical performance data of Java software projects. *hopper* (i) walks through the history of a project version by version, (ii) retrieves performance metrics by executing tests for each version, and (iii) stores these results in a file. The supported types of applications are described along three dimension: build-system, version-type, and test-type. Maven (MVN)-based projects are supported with version-types based on git commits and Maven versions. Gradle-based projects are only supported with git commit version-types. *hopper* can currently execute two types of tests: Java Microbenchmarking Harness (JMH) performance benchmarks[4], and JUnit tests.

Input.

As input, *hopper* takes a configuration file, a set of parameters and a project (as part of the config-file). The config-file defines the name and location of the project, test-type-specific parameters and which version range the mining should be applied to. Program parameters specify config and output file path, which test-type to execute, which version-type and which build-system to use. In addition more fine-grained options are available, such as (i) whether versions that did not change executable code are skipped, (ii) execution of every x^{th} version, (iii) whether the versions are incrementally build, and (iv) if only a subset of all tests should be executed.

The bash command in Listing 1 executes the unit tests of protostuff [5] for every 5^{th} chronological git version. A config-file similar to the one in Listing 2 is required. It declares a version range from `277eded` to `5bbf909`. The tag `execs` specifies the number of test executions per version. For executing benchmarks, the tags `jmh_root` for the JMH tests

[4]http://openjdk.java.net/projects/code-tools/jmh/
[5]https://github.com/protostuff/protostuff

Listing 1: *hopper* Bash Command

```
python ./Hopper.py -f /tmp/config.xml \
    -o /tmp/results.cvs -t unit -b commits \
    --skip 5
```

Listing 2: *hopper* Config-File

```
<historian type="MvnCommitWalker">
  <project name="protostuff" dir="/tmp/ps">
    <junit><execs>20</execs></junit>
    <versions>
      <start>277eded</start>
      <end>5bbf909</end>
    </versions>
  </project>
</historian>
```

and `jmh_arguments` for the JMH command line arguments (e.g. warmup/measurememt, forks) are required.

Output.

`hopper` saves the metrics obtained from the test executions to a CSV-file. Each line represents the information of a single execution, consisting of the version of the software (e.g. the commit hash), the executed test and the measured performance metric (e.g. throughput or runtime).

3. ANALYSING PERFORMANCE DATA

The second stage - on the right in Figure 1 - deals with data transformation and analysis with respect to performance changes.

Subprograms.

gopper currently implements six subprograms: (i) `merge` - merging multiple performance metric files into a single file, (ii) `filter` - filtering test results by minimum mean value, minimum median value and/or minimum versions, (iii) `analyse` - analysing test results for performance changes with either Bayesian Change Point Analysis or Welch's t-test, (iv) `toCangePoints` - transforming test results to change points, (v) `plot` - plotting test result data with (if available) their change points, and (vi) `save` - saving the transformations and change points to files.

Input.

Input to `gopper` is a config-file and a non-empty list of subprograms. The config-file defines: (i) `"In"` - the input file(s), (ii) `"Out"` - the output files for the transformed test results, identified change points and printed plots, (iii) `"Analyse"` - the analysis function and their parameters for analysing performance changes between consecutive project versions, and (iv) `Transform` - the transformation functions and their parameters for filtering test results.

Listings 4 and 3 show a simple example that use the result from stage 1 (see Figure 1: Performance Results), filter those where the mean value is below 0.01, calculate the change points, plot the test results (see Figure 2), and save the transformed test results and change points.

Output.

Three kinds of output are supported by *gopper*: plots, test results and change points. Plots are either time series with a single performance value per version (e.g. unit test execution times), or box plots with multiple values per version (e.g. throughput benchmarks). The plot in Figure 2 shows the performance results for one particular unit test. On the y-axis we see the execution time in seconds and the x-axis shows the different versions from older (left side) to newer (right side). The analysis function (t-test) discovered a change point between version 714e618 and 1dfc05f, which

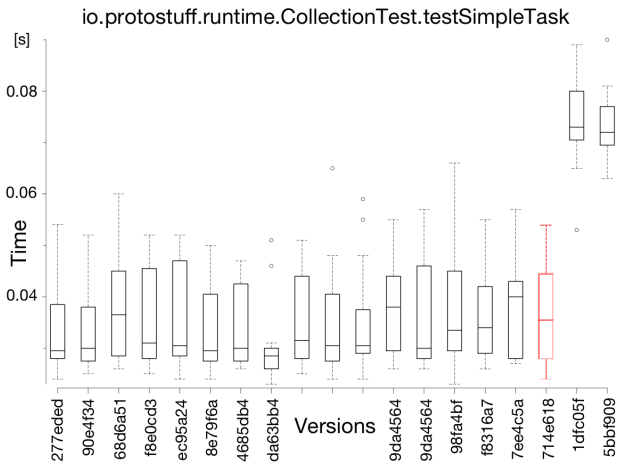

Figure 2: Unit Test Execution Plot with one Change Point

Listing 3: *gopper* Bash Command

```
gopper -c /tmp/config.json \
       filter analyse toChangePoints plot save
```

Listing 4: *gopper* Config-File

```
{ "In": ["/tmp/results.csv"],
  "Out": {
    "TestResults": ["/tmp/results_altered.csv"],
    "ChangePoints": ["/tmp/cps.json"],
    "Plot": "/tmp/plots" },
  "Analyse": {
    "Name": "ttest",
    "Params": [0.99, true] },
  "Transform": [{
    "Name": "minMean",
    "Params": [0.01] }]}
```

is marked in red in the plot. Furthermore `gopper` supports saving the transformed input file in the same format (CSV) as `hopper` and a JSON file that shows the change points detected with the corresponding tests it occurred in.

4. CONCLUSION AND FUTURE WORK

hopper calculates performance metrics based on repeated execution of tests over the history of a system. As a second step *gopper* takes that data, analyses it, and plots the results. For the future, we want to add Infrastructure-as-Code (e.g. Docker, Chef) scripts to ease the installation process. The robustness of the statistical tests needs some additional work, such that reported change points are more reliable. Moreover we want to focus on reproducible results in virtual environments, such that parallelization of test executions on e.g. cloud instances is possible.

5. REFERENCES

[1] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. "Learning from Source Code History to Identify Performance Failures". In: *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering (ICPE)*. 2016, pp. 37–48.

[2] Christoph Heger, Jens Happe, and Roozbeh Farahbod. "Automated Root Cause Isolation of Performance Regressions during Software Development". In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE)*. 2013, pp. 27–38.

[3] 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 International Conference on Performance Engineering (ICPE)*. 2017.