Bachelor Thesis

August 12, 2019

Effects of source code properties on variability of software microbenchmarks written in Go

Mikael Basmaci

of Istanbul, Turkey (15-721-244)

supervised by

Prof. Dr. Harald C. Gall

Christoph Laaber

University of Zurich^UZH

s.e.a.l. software evolution & architecture lab

Bachelor Thesis

Effects of source code properties on variability of software microbenchmarks written in Go

Mikael Basmaci

University of Zurich UZH

s.e.a.l. software evolution & architecture lab

Bachelor Thesis

Author: Mikael Basmaci, mikael.basmaci@uzh.ch

URL: <url if available>

Project period: 25.03.2019 - 25.09.2019

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

Here comes the acknowledgements.

Abstract

Here comes the abstract.

Zusammenfassung

Here comes the summary.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Creating bug-free and stable software was one of the main goals of software engineering for many years. Stakeholders of a software system could probably be easily satisfied if the system did what it promised to do. Although this does not imply that performance was not an important aspect of software, it was still not the number one priority of the time.

With the evolution of software engineering over the years, and the constantly advancing technology that makes software engineering possible, the importance of performance has grown. Having a performant system nowadays is not only the wish of customers or stakeholders, but also one of the goals of software developers when developing software. A better performance is important for many reasons. To give some examples: With better performance, a data driven application will load the data faster, a calculation will result quicker, or an interactive software will be more responsive. All these examples show that more or same amount of work can be done in less time, with better performance.

The usual way of testing software against bugs has been usage of test suites that include unit tests, which test specific parts of the software to assess their completeness. These are complemented with integration and end-to-end tests, which go into a deeper level. Shortly, integration tests help assess the completeness of multiple software modules combined, and end-to-end tests report whether the whole software works without bugs. For the performance testing of a software however, there used to be no specific way of doing until the occurrence of performance testing frameworks, which have been getting more and more popular in the recent years. With early adoption of these frameworks in the development phase of a project, performance regressions can be detected and taken care of pre release.

There are different kinds of performance testing found in the literature. One of them is stress testing, which tests the software system for the stability while putting it under extreme workloads to detect its breaking point. The longer the system holds, the more stable the system is. Another type of performance test is load testing, which tests the software system with high loads to find out about performance bottlenecks. These types of performance tests are useful for when one wants to assess the general performance of a complete software system before it goes into production. Testing with a deeper level of granularity involves the software microbenchmarks, with which one can test modular functions of a software system and measure their performance. This thesis is oriented to this kind of performance tests.

Rest of this thesis is structured as follows: I give an introduction to related topics and work about this thesis in the **Background and Related Work** section. In the **Methodology** section, I explain the three steps I take to do the analytic part of this thesis and elaborate on the threads to the validity of the methodology. I present the results of these steps in the **Results** section and present some ideas for the future work. Next comes the **Discussion** section where I discuss about the methodology and the results of this thesis. I conclude with the **Conclusion** section.

Chapter 2

Background and Related Work

2.1 Background

2.1.1 Software microbenchmarks

- Different kinds of benchmarks found on literature
- What are software microbenchmarks
- How are they used
- Why are they used

Different kinds of benchmarks are found in the literature. While sometimes benchmarks refer to a standardization of a measure, other benchmarks assess the performance of the underlying system. For example, there are different benchmark programs to measure the performance of hardware such as Central Processing Unit(CPU), to be able to compare their performance with others of its kind. Another example can be the benchmark function of a game, which gives the average FPS score of the game run on a hardware.

Software developers can analyze the performance of parts of their software with the so called software microbenchmarks, which can be defined as unit tests for performance of a software. These benchmarks, unlike other benchmarks that test a whole system for the performance, test only a small fraction of the software, such as one or multiple little functions. Executing microbenchmarks for a defined time period, one can sample an average execution time of the benchmark, as well as the data points as the results of multiple iterations.

Depending on the underlying testing framework, developers can choose to run a benchmark for a limited time to see how many iterations can happen in this pre defined time, or also choose to run the benchmark for a specific amount of iterations to see how long it takes the functions within the benchmark to result in average. Such frameworks are especially useful for when determining whether a new introduced feature in the system causes performance regressions. Furthermore, these regressions can be detected early whilst still being in the development stage of the software, and according changes to the code can be scheduled before the changes go to production, which ensures the stability for the performance in the evolution of the software.

2.1.2 Benchmark variability

- Explaining data points as the result of iterations
- Defining term "benchmark variability"

- Factors that play a role in the variability of benchmarks
- Why is it important to find out / predict the variability causes

When a benchmark is run, it executes the underlying code for a certain time period over and over, giving two data types as a result: The first one can be explained as a collection of execution times for each iteration, whereas the second one is the average of this collection, hence, the average execution time of the benchmark. Execution times of a benchmark are usually expressed in nanoseconds. In this thesis, the term benchmark variability refers to the variability of a benchmark's resulting execution times. It is basically computed by measuring how far the single executions fall from the average value. In this manner, if the executions are in the close neighborhood of the average value, the benchmark is considered as stable and it's predicted that another execution would take same or similar amount of nanoseconds as the average value. If however, the points are distributed on a broader scale, it means that the benchmark is rather unstable, i.e., it cannot be predicted how long another execution of the benchmark will take.

The variability of a benchmark may depend on a lot of factors such as the execution platform, the hardware the benchmarks are executed on, or, even the programming language of the software itself. For example, the same benchmark can deliver different average values on different CPUs, as one CPU may perform much more operations in a second than the other one. Similarly, the same benchmark can have a different average for the execution in a personal computer than the average in a cloud-based machine. A factor that may play a role in the variability of a benchmark can be the source code properties that the benchmark has. To this end, a source code property can affect the result of a benchmark. In this thesis, my aim is to find whether there is a correlation between the source code and the result of a benchmark.

Executing performance tests of a software is usually a long, time consuming process that can take up to days to finish. If a developer can rely on the microbenchmarks of the software, this can help reduce the time spent on performance tests for the whole project. When testing the software for performance via microbenchmarks, the aim of the developer is to have the highest possible coverage with the minimal effort. For this, the minimal benchmark suite is needed, which should cover all or the most of the functionalities in the software. As the size of the software grows, the benchmark suite grows as well, and it gets harder to keep track of the tested and not tested functions. To this manner, the importance of predicting variability causes dives in. To keep the size of the benchmark suite minimal while having a good coverage, developers might not need testing all the functionalities. That's why, it's a good idea to predict which parts of the source code should be tested by predicting the variability of the according benchmark. With this, developers can be supported to write better benchmarks.

One of the ways to predict the variability might be through analyzing source code properties of the software. This can on one hand help the developers identify the cause of slowdowns in a newer version for example, on the other hand help them understand which source code property affects the results in which way.

The goal of this thesis is to study the variability of results of performance tests on a large scale with many projects written in Go. For this, the reason for the high/low distribution of benchmark results of these Go Projects are investigated based on their source code properties such as File IO, HTTP Calls, Loops, User IO and Memory Management, which can affect the performance rigorously.

2.1.3 Go

- Some information about the programming language
- How is a benchmark defined in Go language, with examples and how can we execute these?
- Data points acquired from a previous study from Laaber et al.
- Research questions R1 R2

Go is a statically typed, compiled programming language, designed at Google and was released in November 2009. Some of its innovational features include built-in data structures for advanced concurrent programming, Goroutines as lightweight processes and built-in performance benchmarking/testing libraries. While still being quite a new language among other languages, Go is the forth most active programming language in Github, and third-most highly paid language globally, according to Stack Overflow Developer Survey 2019. Furthermore, it is effectively used in the industry and has a growing community.

Go comes with a great built-in package for testing and benchmarking, which makes it easier for developers to unit test their code while still developing, or measure the performance of their functions. To create a benchmark function, the only thing one has to write is a function beginning with the name "Benchmark" and give a parameter *testing.B. The collection of all the benchmarks within a project build its benchmarking suite. Go also supports these built-in packages with a CLI flag called "test", with which a developer can execute all the tests or benchmarks of a project on a single command.

2.2 Related Work

Here comes the literature research about benchmark variability. While some papers look at the benchmark variability causes, some of them try to predict the performance regressions based on different versions of a software by analyzing different commits and mining source code.

To the best of my knowledge, there exist no study analyzing benchmark variability of software microbechmarks written in Go by trying to find a correlation with the used source code metrics in these benchmarks.

[LL18]

Chapter 3

Methodology

In this section, I present the methodologies I followed to get to the results. As the project consists of 3 main steps, these steps are explained respectively.

3.1 Calculating variabilities

First part of the study involved doing a qualitative analysis by analyzing the given data set. From this data set, I firstly had to extract the valid projects, i. e. choose the projects that have a positive average benchmark score. Secondly, I calculated some metrics for each of benchmark that indicate the variability. The results follow in the first point of Results section.

3.1.1 Data set

The starting point was a data set that was acquired by Laaber et. al from a previous study? (Ask for details)

Out of 482 projects, 230 of the projects qualified because they had a score bigger than -1 in the main file of data set. ...
This includes the categorization of benchmarks of 228 projects written in Go by looking at the total executions, the mean, coefficient of variation and relative confidence interval width with 95% and 99% confidence intervals. Secondly, we introduce the methodology we used to extract source code properties from in the first place chosen benchmarks. Thirdly, comparing the source code properties with the variabilities of benchmarks by using regression models to achieve the results.

3.1.2 Metrics as the variability indicators

For calculating the variabilities of benchmarks, I oriented myself at 3 main metrics. First metric is "Coefficient of Variation" (CV), which is a metric used often in statistics to calculate variability of a data set. ...

3.1.3 Python coding

How I coded the calculations and created the visualizations.
Which library I used for creating the visualization?
How I used the pa-tool to get the RCIW95 and RCIW99 values

3.2 Extracting source code properties

Second part of the study was to extract the source code properties of benchmarks written in Go. This also had 2 parts. Results of this part are to be found in second part of the Results section.

3.2.1 Decision on source code properties

Which steps I took at deciding for the source code properties

3.2.2 Golang coding

- How I coded in Go to extract the properties? (Also get into detail about AST's in Go and giving code examples)
- Which libraries or other tools I used (Callgraph tool, cyclomatic complexity tool) ?

3.3 Finding correlations

Third and last part of the study was to find correlations between source code properties of benchmarks and their variability. In this last part, I used a regression model on some chosen benchmarks to find correlations. Results of this part are to be found in the third part of Results section.

3.3.1 First analysis - Second analysis

- Which dependent and independent variables do I have for the comparison?

3.3.2 Regression model

- Which regression model I used for the results, how did I get the correlations results?

3.4 Threats to the validity

There are threats to the validity of this study.

3.4.1 Number of projects

There are probably thousands of projects written with Go. These can be with or without benchmarks, however, I was limited to a number of projects for the outcome of this study and the number might not reflect the truth.

3.4.2 Chosen properties

I presented my chosen properties for this analysis, however, are these enough for this study? There may be other metrics that might be very relevant to this study, which I haven't discovered whilst searching.

3.4.3 Analysis of the unknown

Since the analysis is made statically for the second part of the methodology, this can be a threat to the validity since we don't actually know which nodes in the cyclomatic complexity are visited.

Chapter 4

Results and Future Work

In this section, I present the results from the 3 parts of this study.

4.1 Variabilities of benchmarks

After calculating the variability metrics of benchmarks from the projects, I categorized the variabilities in 10 percent buckets starting from 0 o 100 percent. All the benchmarks that have a variability higher than 100 percent fall into one final bucket. ...

4.2 Source code properties

For every project that I was able to analyze by using the tool introduced in second part of methodology section, I was able to collect the source code properties. CSV? ...

4.3 Correlation between variabilities and source code properties

- Yes
- No
- Not really?
- Can't really say

4.4 Future Work

What could be done with the results in this thesis in the future?

Chapter 5

Discussion

In this section, I discuss the results that I obtained and how relevant these are.

5.1 Chosen properties

Do the chosen properties make sense?
Why did I choose these properties and how could these affect the benchmark variability?

5.2 Static analysis

This study involved doing a static analysis for the source code of project written in Go. What pros and cons does this have? Why doing a dynamic analysis would make more sense?

5.3 Size of data set

Coming from 482 open source projects written in Go, down to 228 that I could analyze. Is the size of data set small or big enough to acknowledge the results?

Chapter 6

Conclusion

I finally conclude with what I have done with this project: How I started, which steps I took and which results I achieved.

Bibliography

[LL18] Christoph Laaber and Philipp Leitner. An evaluation of open-source software microbenchmark suites for continuous performance assessment. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 119–130, New York, NY, USA, 2018. ACM.