Bachelor Thesis

September 2, 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

Performance is one of the compelling qualities of a software system [4] and it shapes the way developers build elements of the software. It is affected by lots of factors such as the software itself, the operating system the software is run on, the middleware, the hardware or even the underlying communication networks [4]. 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. Being an important software quality factor, it can also play a big role on the profitableness of a company [5] [1].

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. Till late 2000s, there has not been a lot of research on the field of software performance testing [6]. With the evolution of software engineering over the years, and the constantly advancing technology that drives software engineering further, the importance of performance has grown. However, the primary problems encountered in the field of software engineering are often related to the performance regressions [6].

A performance regression is to be found when a new version of the software gives a worse user experience in terms of performance, such as having longer response times, or consuming extra resources such as RAM or CPU while giving the same user experience [5]. Solutions to these kinds of problems include supply of more hardware, which comes costly and is not applicable for large software systems [5] or finding out the regression causes in the software by testing.

To assess the performance of the system and improve it, Software Performance Engineering (SPE) activites are needed [4]. There are two general approaches found in the literature: first one is called measurement-based SPE, which stands for experimental performance calculations made on the software to report about the performance, second one is called model-based, which refers to creating performance models while developing to meet the performance requirements [4].

There are different kinds of performance testing belonging to measurement-based SPE 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. They are, however, quite complex in terms of setups, manual configurations and mostly important, having long execution times [5]. This complexness makes it hard to find performance regressions as fast as possible, causing delay in Continuos Deployment (CD) architectures [7].

There exist another type of performance testing, which is with the so called software microbenchmarks. These are adopted as performance evaluation strategy for rather small and library-like projects [8] and can test modular functions of a software system and measure their performance [1]. With early adoption of these tests, developers can find out about performance regressions whilst developing, and find solutions before they become bigger performance issues, which one would first realize at the time of analyzing results of bigger performance tests such as stress or load tests. A challenge with testing using microbenchmarks is the non-deterministicity of the results, which means a certain variability exist for the results of a benchmark executed [7].

In this thesis, I am studying the variability of results of performance tests on a large scale with many projects written in Go. For this, I am investigating the reason for the high/low distribution of benchmark results of these Go Projects based on their source code properties such as signature properties, body related features, other source code metrics, and usage of Go's standard libraries which can affect the performance rigorously.

To this end, I want to answer the following research questions:

- **RQ1: How variable are microbenchmark results of Go projects?**
- **RQ2: Which source code properties contribute to the stability of benchmark results and how?**

My hypothesis for the the first research question is: I expect the benchmarks to be quite variable, having a normalized distribution between 1% to 100% of variation scores. To answer my first research question, I get all the projects benchmarks, calculate their variability with different measurement metrics such as Coefficient of Variation (CV) and Relative Confidence Interval Width (RCIW) and report the variabilities of benchmarks for 228 projects, having in total 4589 benchmarks.

My hypothesis for the second research question is: I expect to see significant effects of body related features and Go's standard library usages to the variation of benchmark results. In particular, I suppose that cyclomatic complexity, file IO, http calls and loops have a direct impact on the variability. For the second research question, I download all the projects from Github, fetch their dependencies for the according commit, from which the benchmark results are from. Then I run my parser program which collects the source code properties from all the functions and make a callgraph analysis for the benchmarks to collect all the visited functions' properties, resulting into properties of the benchmark.

For the "how" part of RQ2, I do a correlation analysis with the collected properties of each benchmark. [final is yet to come]

Rest of this thesis is structured as follows: I give an introduction to related topics and background about this thesis in the 2. Chapter. In the 3. Chapter, 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 4. Chapter. Next comes the 5. Chapter where I discuss about the methodology and the results of this thesis and present some ideas for the future work. I conclude with the 6. Chapter.

Chapter 2

Background and Related Work

2.1 Background

In this subsection, I give a detailed background information about software microbenchmarks, their variability, the programming language Go and the way microbenchmarks are implemented and used in Go projects.

2.1.1 Software microbenchmarks

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 are found either as microbenchmarks or performance unit tests in the literature [1]. These two terms differ in the way they test the underlying code for performance: While microbenchmarks report performance counters (e.g., execution time, throughput, latency etc.) for the different iterations of a trial, performance unit tests act like unit tests in functional testing, reporting if a pre set performance criteria has been reached at the runtime of the test [1]. Some studies such as [9] [10] investigate the usage of performance unit tests, while sometimes other studies such as [7] [11] work with microbenchmarks. In this thesis, I work with microbenchmarks and in the rest of this thesis I use the term performance test interchangable with testing with microbenchmarks.

Microbenchmarks usually test for only a small fraction of the software, such as one or multiple little functions, to assess their performance. A microbenchmark can for instance test the performance of a newly introduced data structure, an implemented algorithm, or even the concurrency performance of functions [1]. As in functional testing, microbenchmarking also comes with testing frameworks that allow developers to create function pools, which are often also called microbenchmark suites (in contrast to unit test suites in functional testing). As an example, for Java there exist Java Microbenchmarking Harness (JMH), which is similar to JUnit, but is implemented to help developers with the creation of microbenchmarks [12]. Using this framework, developers can put a "Benchmark" annotation to their microbenchmark functions and test them [12]. Figure 2.1 from Costa et. al's paper [1] illustrates the workflow of JMH's microbenchmark executions

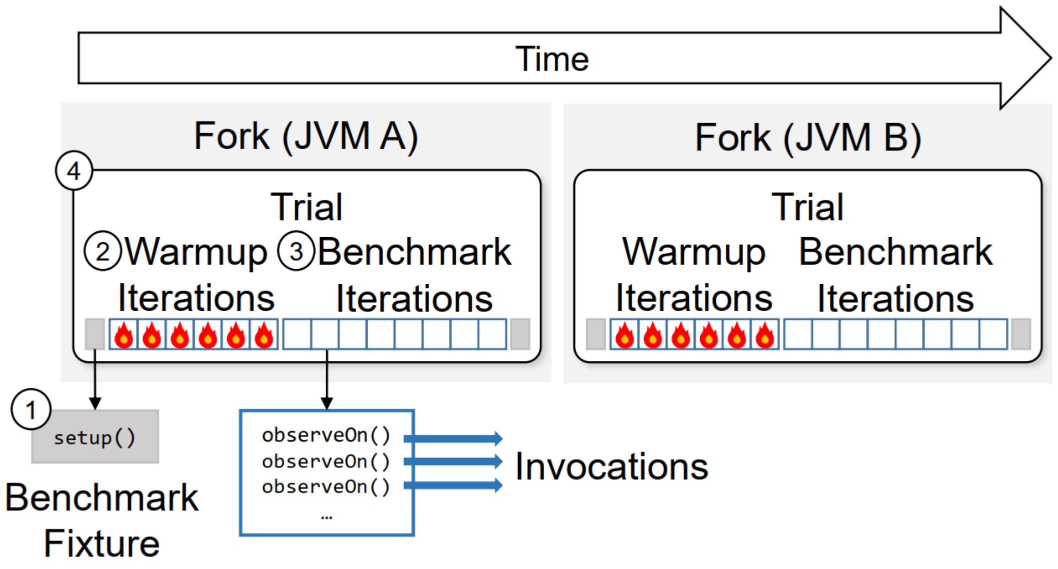


Figure 2.1: Execution of a microbenchmark in JMH, taken from [1].

for the ***observeOn*** method. For each test in JMH, one or more forks can be generated that run individual Java Virtual Machine (JVM)s. Inside a fork, a trial is executed (4), which consists of an optional setup phase (1), warmup iterations (2) and benchmark iterations (3). In the setup phase, the environment for the benchmark is initialized, which is followed by warmup iterations that run the benchmark to bring the system into a steady state but do not save their performance counters, and finally the effective benchmark iterations are executed. By default, both of iteration types invoke the microbenchmark method as many times as possible for 1 second and benchmark iterations record the results. Finally, it is up to the developer to save or visualize the output, which summarizes the performance counters for each benchmark found in the microbenchmark suite. Frameworks like JMH [12] allow developers to parametrize the execution of microbenchmarks for configuring amount of iterations per fork, amount of warmup/benchmark iterations etc.

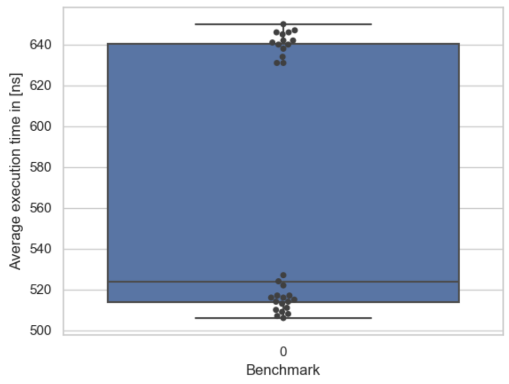
Depending on the underlying testing framework, developers can choose to run a benchmark for a limited time to see how many iterations can happen in this pre defined time, or also choose to run the benchmark for a specific amount of iterations to see how long it takes the functions within the benchmark 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 Microbenchmark variability

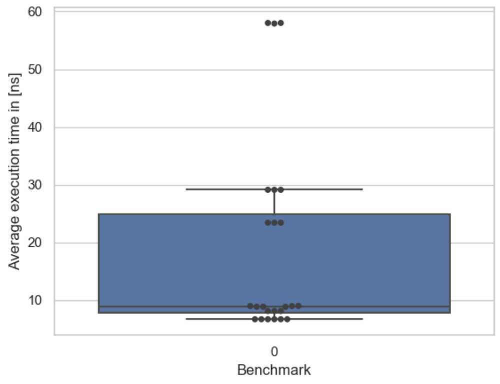
Unless otherwise configured, when a microbenchmark is run, it executes the underlying code for a certain time period (e.g., 1s) over and over for each benchmark iteration and returns a collection of average execution times for each benchmark iteration [8]. Execution times of a benchmark are usually expressed in nanoseconds [8] [1]. In this thesis, the term microbenchmark variability refers to the variability of a microbenchmark's resulting collection of execution times. It is basically computed by measuring how far the single execution times fall from the average execution time. In this manner, if the executions times 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.

There are 2 main metrics that I use to calculate the variability of a microbenchmark in this thesis. The first one is called **coefficient of variation (CV)**, which is also known as the relative standard derivation [8]. "CV is a statistical measure for dispersion among a population of values" [8], and in this thesis I use CV on the performance counters (i.e. average execution times in nanoseconds) of a microbenchmark's benchmark iterations to calculate the variation of a microbenchmark. CV is a measurement that is used to determine the variability in pervious studies as well [8] [13]. Additionally, I calculate the **relative confidence interval width (RCIW)** with 95 and 99 percent confidence intervals of each microbenchmark and refer to them as RCIW95 and RCIW99 respectively. "The RCIW describes the estimated spread of the population's CV, as computed by statistical simulation" [8].

Regarding calculations of these metrics: CV is calculated by dividing the standard deviation of a dataset by the mean of the dataset. For RCIW, another methodology is followed, which is explained as bootstrapping with hierarchical random resampling and is also used in Laaber et al.'s paper [8] [14] [15] . The reason to follow this methodology resides in the non-normalized performance measurements [8]. Bootstrapping with randomly resampling refers to randomly sampling data points from a dataset, and in this thesis I use this technique with 10.000 bootstraps on the execution times of each microbenchmark to generate a collection of normalized mean values out of them, which I can then use to create the 95 and 99 percent relative confidence intervals.



(a) Execution results of a microbenchmark with a RCIW99 of 10.28.



(b) Execution results of a microbenchmark with a RCIW99 of 64.18.

Figure 2.2: Example for a low and high variance microbenchmark.

To illustrate a low and high variable microbenchmark: Figure 2.2a shows the execution times of the benchmark *BenchmarkAddSafe* from **deckarep/golang-set** project [16]. As the box plot shows, the points are all within the whiskers and there are no outliers. Also, the distribution of the points vary from 506 to 650 nanoseconds and it has a 99 percent confidence interval width (RCIW99) score of 10.28. In Figure 2.2b, the benchmark *BenchmarkParseUintHex* from **segmentio/objconv** project [17] is shown with its execution times. This benchmark's average execution times vary from 6.68 nanoseconds to 57.9 nanoseconds, having 3 outliers. This time, RCIW99 score is 64.18. The score in RCIW99 is an estimator for the variability, hence, the higher this score, the higher variability the benchmark has.

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 microbenchmark itself [18]. 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 time in a personal computer than the average in a cloud-based machine, or in different cloud-based machines compared to each other [8]. Some other factors include the concurrency, I/O latencies, virtualization etc.
[8].
Variability is a crucial factor of a benchmark, because it has an impact on the stability and reliability of it's results [7]. It is important to have stable and reliable benchmarks, because then the developers can rely on the results of the microbenchmarks when they test parts of the software and find the regression causes with a higher trust. To this end, it's essential to predict the variability of microbenchmarks. There has been studies trying to predict the variability of performance tests in public Infrastructure-as-a-Service (IaaS) clouds by comparing aspects such as hardware heterogeneity, multi-tenancy and control over optimizations [13], or by comparing the outcomes of forks, trials and iterations in terms of variability [8]. One way 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.

Executing usual performance tests (e.g., stress tests, load tests etc.) of a software is usually a long, time consuming process that can take up to days to finish [7]. 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 microbenchmark 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 benchmarks based on source code. With this, developers can be supported to write better benchmarks by for example being alerted to which new functions should have priorization in testing for performance.

In this thesis, my aim is to find whether there is a correlation between the source code and the variability of a benchmark. For this, I analyze source code properties of microbenchmarks written in Go and use them as dependent variables of a statistical equation, where the independent variable is the variability of the benchmark in RCIW99. The results can be used to understand the correlation between source code and variability, as well as be used to predict the variability of a benchmark for further applications.

2.1.3 Go

Go is a statically typed, compiled programming language, designed at Google and was released in November 2009 [19]. 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 [20]. 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 [21]. 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**. These functions are typically defined in files ending with the **_test.go** suffix, and the collection of all the benchmarks within a project build its microbenchmarking suite. Listing 2.1 shows the example benchmark *BenchmarkHash* from project **ironsmile/nedomi** [2].

```go
func BenchmarkHash(b *testing.B) {
   var m = buildMapHash(id, count)
   for i := 0; b.N > i; i++ {
      if _, ok := m[first.Hash()]; !ok {
         b.Fail()
      }
      if _, ok := m[middle.Hash()]; !ok {
         b.Fail()
      }
      if _, ok := m[last.Hash()]; !ok {
         b.Fail()
      }
   }
}
```

Listing 2.1: An example benchmark from [2].

Go also supports these built-in packages with a CLI flag called "test", with which a developer can execute all the tests of a project on a single command. Executing all benchmarks takes an additional flag "bench", which executes all the benchmarks found in the current directory(package) of a project. An example run of a microbenchmark suite from **tidwall/buntdb** [3] can be seen in Listing 2.2. The first column is the name of the benchmark, the second column is the amount of executions and the third column is the execution time per execution [21].

```
mikael@mikael-VirtualBox:~/Desktop/BenchmarkProjects/tidwall/buntdb/src-
/github.com/tidwall/buntdb$ go test -bench=.
goos: linux
goarch: amd64
pkg: github.com/tidwall/buntdb
Benchmark_Set_Persist_Random_1-4   300000 4467 ns/op
Benchmark_Set_Persist_Random_10-4   1000000 2094 ns/op
Benchmark_Set_Persist_Random_100-4   1000000 1690 ns/op
```

```
Benchmark_Set_Persist_Sequential_1-4  500000 3614 ns/op
Benchmark_Set_Persist_Sequential_10-4  1000000 1388 ns/op
Benchmark_Set_Persist_Sequential_100-4  1000000 1209 ns/op
Benchmark_Set_NoPersist_Random_1-4  1000000 1774 ns/op
Benchmark_Set_NoPersist_Random_10-4  1000000 1135 ns/op
Benchmark_Set_NoPersist_Random_100-4  1000000 1107 ns/op
Benchmark_Set_NoPersist_Sequential_1-4  1000000 1821 ns/op
Benchmark_Set_NoPersist_Sequential_10-4  2000000 744 ns/op
Benchmark_Set_NoPersist_Sequential_100-4  2000000 746 ns/op
Benchmark_Get_1-4  2000000 808 ns/op
Benchmark_Get_10-4  3000000 544 ns/op
Benchmark_Get_100-4  3000000 525 ns/op
Benchmark_Ascend_1-4  5000000 317 ns/op
Benchmark_Ascend_10-4  2000000 594 ns/op
Benchmark_Ascend_100-4  500000 3034 ns/op
Benchmark_Ascend_1000-4  50000 27257 ns/op
Benchmark_Ascend_10000-4  5000 271873 ns/op
Benchmark_Descend_1-4  5000000 304 ns/op
Benchmark_Descend_10-4  3000000 565 ns/op
Benchmark_Descend_100-4  500000 3065 ns/op
Benchmark_Descend_1000-4  50000 27362 ns/op
Benchmark_Descend_10000-4  5000 275479 ns/op
PASS
ok  github.com/tidwall/buntdb 71.590s
```

Listing 2.2: Example microbenchmark suite execution from [3].

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.

Bunu duzelt At the time of writing this thesis, most of the existing work is on the research of performance unit testing in Java ecosystem. [9] [7]

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. To shortly illustrate, Figure 3.1 show the steps along with the programming/scripting languages involved in this thesis. In the first step, I analyze given dataset of microbenchmark results with help of Python and create a .csv file containing all the individual microbenchmarks along with their mean, CV, RCIW95 and RCIW99 values. In the second step, I download all the projects that have an entry in the previous .csv file and run a parser tool written in Go that collects source code properties for each of the benchmarks found in the projects. In the final step, I do a correlation analysis using the variabilities of individual benchmarks as independent variables and their properties as dependent variables. The outcomes of each step are presented in the 4. Section.

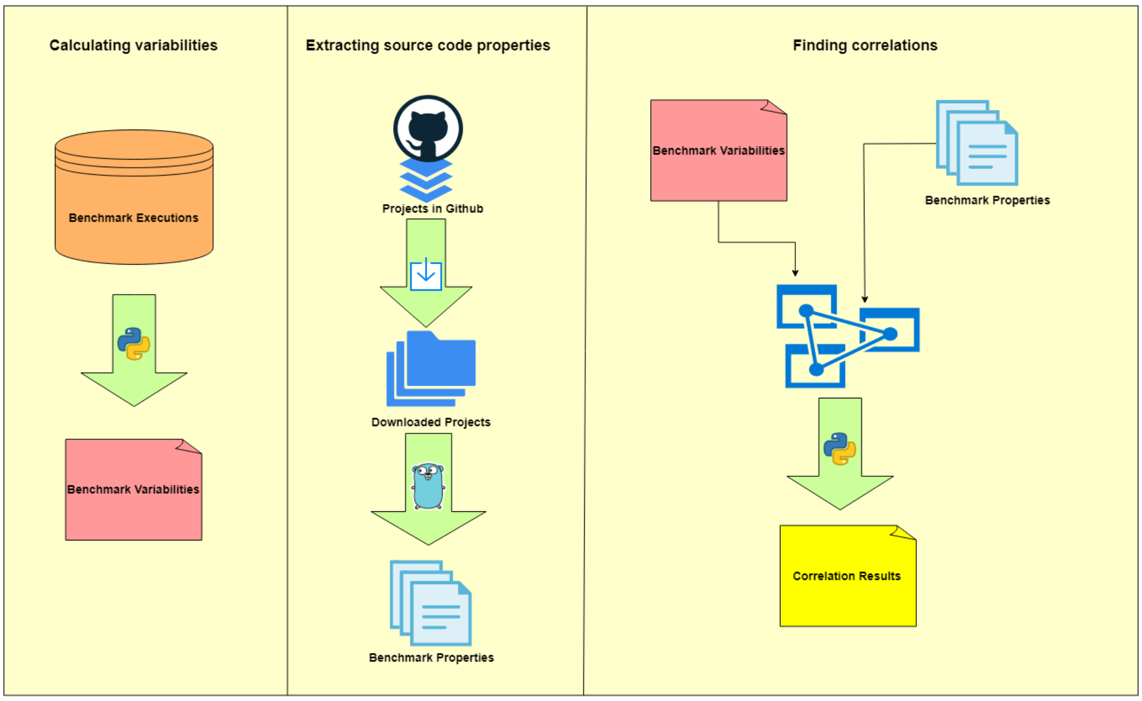


Figure 3.1: 3 main steps of the methodology.

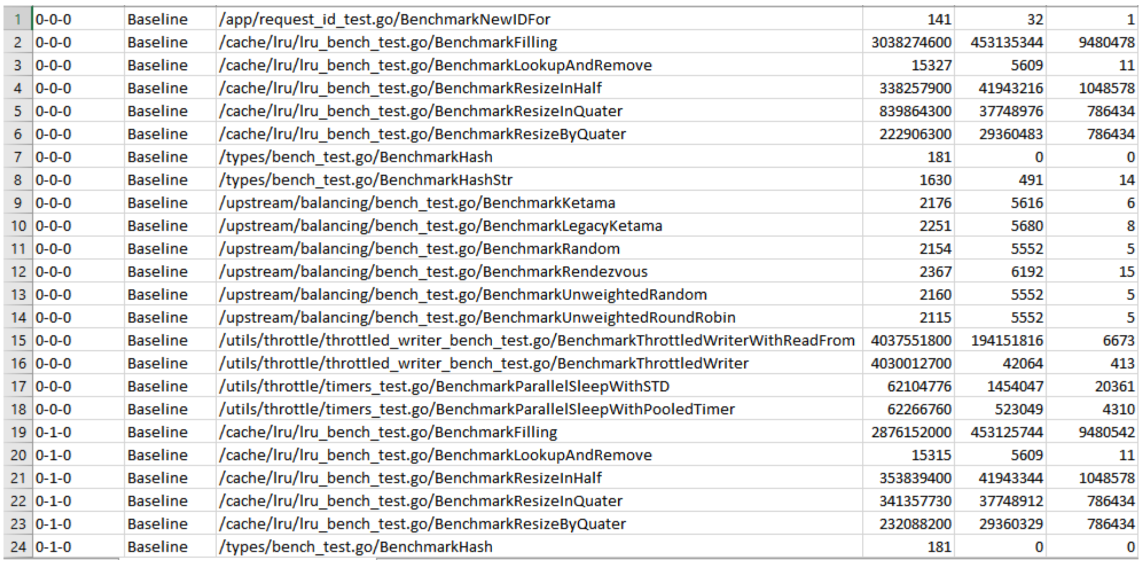
3.1 Calculating variabilities

First part of this thesis involves doing a quantitive analysis by analyzing the given data set. From this data set, I firstly extract the valid projects, i. e. choose the projects that have a positive number of individual benchmark results. Secondly, I calculate CV, RCIW95 and RCIW99 for each of benchmarks found in the dataset. For RCIW95 and RCIW99, I use a helper tool, called "pa tool", which helps me use the bootstrapping technique with randomly sampling, described in Section 2.1.2. The outcome of this part is explained in detail in Section 4.1

3.1.1 Dataset

The dataset to analyze the variations from was given me from my supervisor Christoph Laaber. This dataset with a size of 43.8 MB contains **go-results.csv**, **go-results-2.csv** and 4 folders **cumulus-1 to cumulus-4**. In the **go-results-2.csv**, I find which project's result is on which relative path (which are in cumulus folders) and has how many individual results. **go-results.csv** differs from this file in having no commit of the projects. That's why, I start by analyzing **go-results-2.csv** in particular. In this file, I look for the column named "c1_results": if the value in this column is above 0 (i.e., it is not -1), it means that there are results for that specific project on the special commit, which is found in the column "c1_commit". From a total of 481 entries in this file, I get 230 projects which have individual results and filter them out. In the next step, I map the relative filepath of the project's result file to the name and commit of the project by querying the **go-projects.csv** file, which is to be found in every cumulus folder. In the end, I have all the valid projects with their results file.

Results file of a projects looks like in Figure 3.2. In this file, the middle part in the first column specifies the number of run, the second column specifies the benchmark including the relative path and file where the benchmark is located in, the forth column reports the execution time in nanoseconds, the fifth and sixth column are related to memory performance, bytes/operation and allocations/operation respectively.



1	0-0-0	Baseline	/app/request_id_test.go/BenchmarkNewIDFor	141	32	1
2	0-0-0	Baseline	/cache/lru/lru_bench_test.go/BenchmarkFilling	3038274600	453135344	9480478
3	0-0-0	Baseline	/cache/lru/lru_bench_test.go/BenchmarkLookupAndRemove	15327	5609	11
4	0-0-0	Baseline	/cache/lru/lru_bench_test.go/BenchmarkResizeInHalf	338257900	41943216	1048578
5	0-0-0	Baseline	/cache/lru/lru_bench_test.go/BenchmarkResizeInQuater	839864300	37748976	786434
6	0-0-0	Baseline	/cache/lru/lru_bench_test.go/BenchmarkResizeByQuater	222906300	29360483	786434
7	0-0-0	Baseline	/types/bench_test.go/BenchmarkHash	181	0	0
8	0-0-0	Baseline	/types/bench_test.go/BenchmarkHashStr	1630	491	14
9	0-0-0	Baseline	/upstream/balancing/bench_test.go/BenchmarkKetama	2176	5616	6
10	0-0-0	Baseline	/upstream/balancing/bench_test.go/BenchmarkLegacyKetama	2251	5680	8
11	0-0-0	Baseline	/upstream/balancing/bench_test.go/BenchmarkRandom	2154	5552	5
12	0-0-0	Baseline	/upstream/balancing/bench_test.go/BenchmarkRendezvous	2367	6192	15
13	0-0-0	Baseline	/upstream/balancing/bench_test.go/BenchmarkUnweightedRandom	2160	5552	5
14	0-0-0	Baseline	/upstream/balancing/bench_test.go/BenchmarkUnweightedRoundRobin	2115	5552	5
15	0-0-0	Baseline	/utils/throttle/throttled_writer_bench_test.go/BenchmarkThrottledWriterWithReadFrom	4037551800	194151816	6673
16	0-0-0	Baseline	/utils/throttle/throttled_writer_bench_test.go/BenchmarkThrottledWriter	4030012700	42064	413
17	0-0-0	Baseline	/utils/throttle/timers_test.go/BenchmarkParallelSleepWithSTD	62104776	1454047	20361
18	0-0-0	Baseline	/utils/throttle/timers_test.go/BenchmarkParallelSleepWithPooledTimer	62266760	523049	4310
19	0-1-0	Baseline	/cache/lru/lru_bench_test.go/BenchmarkFilling	2876152000	453125744	9480542
20	0-1-0	Baseline	/cache/lru/lru_bench_test.go/BenchmarkLookupAndRemove	15315	5609	11
21	0-1-0	Baseline	/cache/lru/lru_bench_test.go/BenchmarkResizeInHalf	353839400	41943344	1048578
22	0-1-0	Baseline	/cache/lru/lru_bench_test.go/BenchmarkResizeInQuater	341357730	37748912	786434
23	0-1-0	Baseline	/cache/lru/lru_bench_test.go/BenchmarkResizeByQuater	232088200	29360329	786434
24	0-1-0	Baseline	/types/bench_test.go/BenchmarkHash	181	0	0

Figure 3.2: Example results file from [2].

3.1.2 Metrics as the variability indicators

As described in Section 4.1, I orient myself at 3 main metrics to calculate the variabilities of the benchmarks. These are CV, RCIW95 and RCIW99 respectively. In this thesis, I'm only interested in the execution times, hence, I only take the forth column of the results file for each benchmark into consideration. To calculate the CV, I firstly find the standard derivation and mean of the execution times of each benchmark. Dividing the standard derivation by the mean gives me the CV.

For the RCIW part, I use a tool by Christoph Laaber, called "pa-tool" [22]. This tool works in the following way: For a given benchmark with all its execution times, it randomly samples a subset of the execution times and saves the mean of this new subset. This process is repeated for a considerable amount of time, e.g., 1000 times (number of bootstrap simulations), which guarantees that the new collection of means of the initial benchmark's execution times is now normalized. Finally, the tool gives a confidence interval of the new collection of means with a default significance level of 0.05. Listing 3.1 illustrates an example output of pa-tool, showing the confidence interval of each benchmark in project ironsmile/nedomi after bootstrapping 10.000 times with the 0.01 significance level [2].

```
C:\Users\Mikael\pa>pa −bs 10000 −sig 0.01
"C:\Users\Mikael\go−calculation\pa_input_projects\1&ironsmile&nedomi_benchmarks.csv"
#Execute CIs:
# cmd = CI
# number of cores = 8
# bootstrap simulations = 10000
# significance level = 0.01
# statistic = Mean
# invocation sampling = Mean
# files 1 = [C:\Users\Mikael\go−calculation\pa_input_projects\1&ironsmile&nedomi_benchmarks.csv]
# files 2 = []

/app/request_id_test.go/BenchmarkNewIDFor;;;1.372879e+02;1.396515e+02;0.99
/cache/lru/lru_bench_test.go/BenchmarkFilling;;;2.950686e+09;2.982955e+09;0.99
/cache/lru/lru_bench_test.go/BenchmarkLookupAndRemove;;;1.527660e+04;1.532752e+04;0.99
/cache/lru/lru_bench_test.go/BenchmarkResizeByQuater;;;2.239956e+08;2.292164e+08;0.99
/cache/lru/lru_bench_test.go/BenchmarkResizeInHalf;;;3.551978e+08;4.518111e+08;0.99
/cache/lru/lru_bench_test.go/BenchmarkResizeInQuater;;;3.811996e+08;5.008146e+08;0.99
/types/bench_test.go/BenchmarkHash;;;1.813582e+02;1.834179e+02;0.99
/types/bench_test.go/BenchmarkHashStr;;;1.634015e+03;1.637060e+03;0.99
/upstream/balancing/bench_test.go/BenchmarkKetama;;;2.185776e+03;2.194403e+03;0.99
/upstream/balancing/bench_test.go/BenchmarkLegacyKetama;;;2.233333e+03;2.242545e+03;0.99
/upstream/balancing/bench_test.go/BenchmarkRandom;;;2.160121e+03;2.165439e+03;0.99
/upstream/balancing/bench_test.go/BenchmarkRendezvous;;;2.340515e+03;2.347833e+03;0.99
/upstream/balancing/bench_test.go/BenchmarkUnweightedRandom;;;2.157939e+03;2.164152e+03;0.99
/upstream/balancing/bench_test.go/BenchmarkUnweightedRoundRobin;;;2.123652e+03;2.130727e
    +03;0.99
/utils/throttle/throttled_writer_bench_test.go/BenchmarkThrottledWriter;;;4.030817e+09;4.036552e
    +09;0.99
/utils/throttle/throttled_writer_bench_test.go/BenchmarkThrottledWriterWithReadFrom;;;4.038926e
    +09;4.040731e+09;0.99
/utils/throttle/timers_test.go/BenchmarkParallelSleepWithPooledTimer;;;6.253588e+07;6.261951e+07;0.99
```

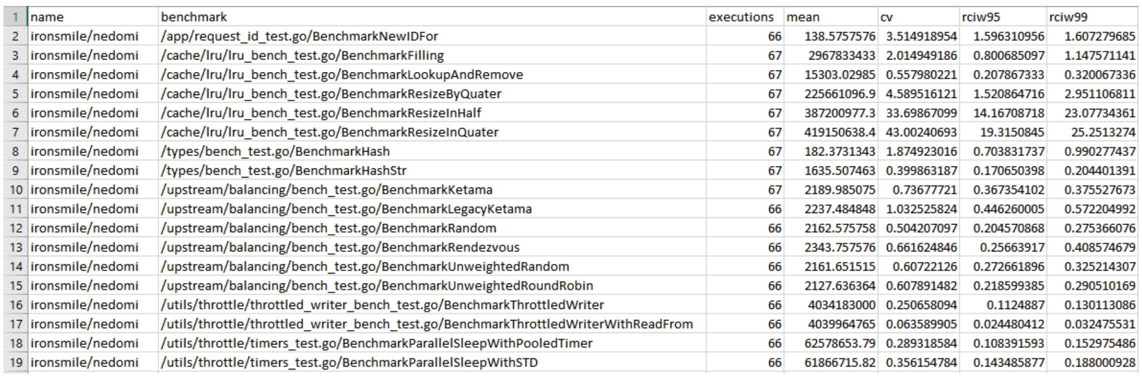
```
/utils/throttle/timers_test.go/BenchmarkParallelSleepWithSTD;;;6.182706e+07;6.191619e+07;0.99
#Total execution took 1.1923061s
```

Listing 3.1: Example pa-tool results of [2].

3.1.3 Python coding

Pa-tool requires the benchmarks of a project to be sorted in alphabetical order, along with its number of run and the execution time. After I have all the valid projects with their results file, I firstly create input csv files for the pa-tool to function accordingly. In the next step, I iterate through input files for pa-tool and run the pa-tool for each project 2 times, once with the significance level of 0.05 and once with 0.01, both having 10.000 bootstrap simulations. From the boundaries of the confidence interval acquired from the pa-tool, I calculate RCIW values by substracting the left boundary from the right boundary. As a result, I get RCIW95 for the results with 0.05 significance leve, and RCIW99 for the results with 0.01 significance level. For the CV part of each benchmark, I use *mean()* and *stddev()* from Python's *statistics* module [23].

Within the end of calculating CV, RCIW95 and RCIW99 of each benchmark, I write all the benchmarks into the **final.csv** file, which shows the name of the project, the specific benchmark, number of executions, and mean, CV, RCIW95 and RCIW99 of the benchmark respectively. Listing 3.3 shows the first lines of **final.csv**.



1	name	benchmark	executions	mean	cv	rciw95	rciw99
2	ironsmile/nedomi	/app/request_id_test.go/BenchmarkNewIDFor	66	138.5757576	3.514918954	1.596310956	1.607279685
3	ironsmile/nedomi	/cache/lru/lru_bench_test.go/BenchmarkFilling	67	2967833433	2.014949186	0.800685097	1.147571141
4	ironsmile/nedomi	/cache/lru/lru_bench_test.go/BenchmarkLookupAndRemove	67	15303.02985	0.557980221	0.207867333	0.320067336
5	ironsmile/nedomi	/cache/lru/lru_bench_test.go/BenchmarkResizeByQuater	67	225661096.9	4.589516121	1.520864716	2.951106811
6	ironsmile/nedomi	/cache/lru/lru_bench_test.go/BenchmarkResizeInHalf	67	387200977.3	33.69867099	14.16708718	23.07734361
7	ironsmile/nedomi	/cache/lru/lru_bench_test.go/BenchmarkResizeInQuater	67	419150638.4	43.00240693	19.3150845	25.2513274
8	ironsmile/nedomi	/types/bench_test.go/BenchmarkHash	67	182.3731343	1.874923016	0.703831737	0.990277437
9	ironsmile/nedomi	/types/bench_test.go/BenchmarkHashStr	67	1635.507463	0.399863187	0.170650398	0.204401391
10	ironsmile/nedomi	/upstream/balancing/bench_test.go/BenchmarkKetama	67	2189.985075	0.73677721	0.367354102	0.375527673
11	ironsmile/nedomi	/upstream/balancing/bench_test.go/BenchmarkLegacyKetama	66	2237.484848	1.032525824	0.446260005	0.572204992
12	ironsmile/nedomi	/upstream/balancing/bench_test.go/BenchmarkRandom	66	2162.575758	0.504207097	0.204570868	0.275366076
13	ironsmile/nedomi	/upstream/balancing/bench_test.go/BenchmarkRendezvous	66	2343.757576	0.661624846	0.25663917	0.408574679
14	ironsmile/nedomi	/upstream/balancing/bench_test.go/BenchmarkUnweightedRandom	66	2161.651515	0.60722126	0.272661896	0.325214307
15	ironsmile/nedomi	/upstream/balancing/bench_test.go/BenchmarkUnweightedRoundRobin	66	2127.636364	0.607891482	0.218599385	0.290510169
16	ironsmile/nedomi	/utils/throttle/throttled_writer_bench_test.go/BenchmarkThrottledWriter	66	4034183000	0.250658094	0.1124887	0.130113086
17	ironsmile/nedomi	/utils/throttle/throttled_writer_bench_test.go/BenchmarkThrottledWriterWithReadFrom	66	4039964765	0.063589905	0.024480412	0.032475531
18	ironsmile/nedomi	/utils/throttle/timers_test.go/BenchmarkParallelSleepWithPooledTimer	66	62578653.79	0.289318584	0.108391593	0.152975486
19	ironsmile/nedomi	/utils/throttle/timers_test.go/BenchmarkParallelSleepWithSTD	66	61866715.82	0.356154784	0.143485877	0.188000928

Figure 3.3: First lines from the final.csv file involving variability values for [2].

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

In this section, I present the results from the 3 parts of this study. First results correspond to the variability of benchmarks and in the first section I present some statistics regarding the distribution of benchmarks' variabilities. [will be continued]

4.1 Variabilities of benchmarks

In Section 3.1, I show which steps I go through to get variabilities of benchmarks, and in Section 3.1.3 I show the structure of **final.csv**, which lists all the benchmarks from 230 projects. In total, there are 4802 benchmarks resulting from 230 projects. A quick investigation of this data shows that there are 204 benchmarks with only 1 execution, i.e. having only the average execution time of 1 benchmark iteration. Under these benchmarks lay all the benchmarks of **mesos/mesos-go** [24] and **go-gl/mathgl** [25], as well as benchmarks *BenchmarkProtocolV2Sub128k* and *BenchmarkCallsConcurrentServer* of projects **nsqio/nsq** [26] and **uber/tchannel-go** [27], respectively. Furthermore, there are in total 9 benchmarks, which, although having more than 1 executions, have a mean of 0, which in further investigation shows that all their execution times are 0 ns. Since having only 1 execution of a benchmark and having 0 ns as a mean execution time lead to not being able to calculate the standard derivation and RCIW values, I drop these benchmarks out of the **final.csv** file, having left with 4589 benchmarks in total.

For a general view of data, I create a histograms of benchmarks using Matplotlib library [28], which is a library for Python used often for visualizing data. For looking at the results in project basis, I create tables for projects and report how many of projects fall into which bucket in terms of distribution of variabilities.

4.1.1 Variabilities in benchmark level

Figure 4.1 shows the histogram across all benchmarks and distributions of their variabilities in 10 percent buckets, taking all 3 metrics into consideration. According to the distribution, (CV) 97.50% (4474/4589), (RCIW95) 98.48% (4519/4589) and (RCIW99) 98.17% (4505/4589) of all benchmarks have a variation between 0 and 10. These are quite high percentages to tell that most of the benchmarks are very stable. Note that last bucket is for all the benchmarks that have a variation equal or higher than 100.

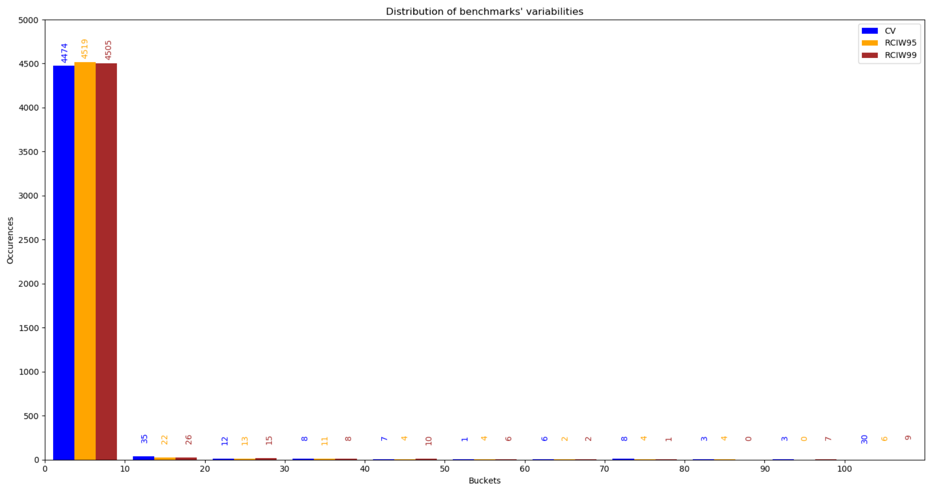


Figure 4.1: Distribution of benchmarks' variabilities 0-100 in 10% buckets.

This result pushes me to further analyze the benchmarks in the 0-10% bucket, and I create a second histogram showing the distribution of benchmarks' variabilities in the 0-10% bucket. Figure 4.2 exposes this variation. Most of the benchmarks fall into the bucket 0-1%, building (CV) 81.74% (3657/4474), (RCIW95) 89.20% (4035/4519) and (RCIW99) 87.70% (3951/4505) of the projects that are in the 1-10% bucket. Although not as high in percentage as in 0-10% bucket, most of the benchmarks can still be described as very stable, having a variation between 0 and 1. Note that last bucket in this figure represents the bucket 9-10%.

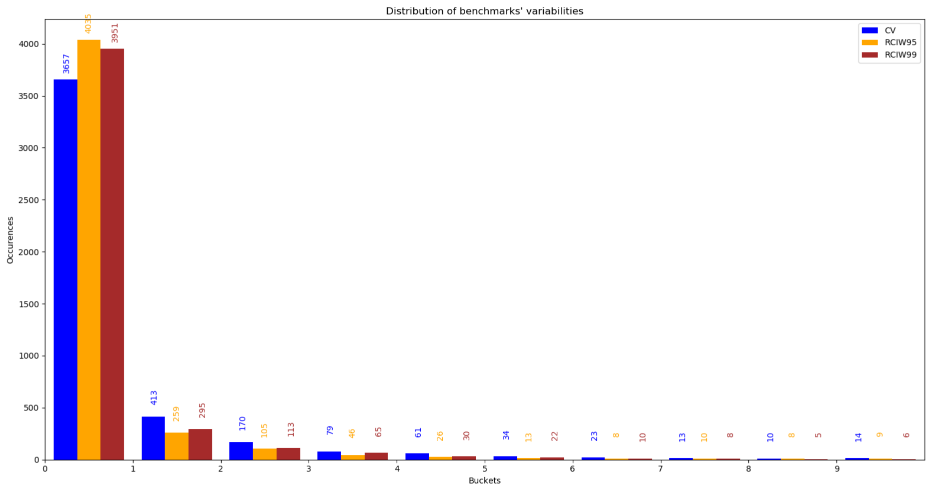


Figure 4.2: Distribution of benchmarks' variabilities 0-10 in 1% buckets.

4.1.2 Variabilities in project level

Since I eliminated 2 projects because they have 1 execution for each of their benchmarks, following statistical data is based on a total of 228 projects. Table 4.1's header shows **Percentages** as buckets for the benchmarks' RCIW99 values. Following two rows show how many distinct projects fall into this group (i.e., having at least one benchmark in this bucket), and how many of the total projects this makes in percentage. As one can see, 225 out of 228 projects have benchmarks that have at least one benchmark that has a RCIW99 score. Analyzing the 0-1% bucket further shows that there are in total 283 benchmarks from 68 projects which have a RCIW99 score of 0.0. This makes 6.16% of total benchmarks.

Table 4.1: Number of distinct projects, whose benchmarks' RCIW99 lay between 1-10% in 1% buckets and between 10-100% in 10% buckets

Percentages	0-1%	1-2%	2-3%	3-4%	4-5%	5-6%	6-7%	7-8%	8-9%	9-1
#projects	225	75	48	30	17	15	8	6	4	5
#projects %	99%	33%	21%	13%	7%	7%	4%	3%	2%	2%

Percentages	10-20%	20-30%	30-40%	40-50%	50-60%	60-70%	70-80%	80-90%	90-100%	>1
#projects	18	9	6	4	4	2	1	0	3	6
#projects %	8%	4%	3%	2%	2%	1%	0%	0%	1%	3%

Table 4.2 similarly shows the number of projects and their percentages across all projects, whose benchmarks have an RCIW99 score of more than the one provided in the **Percentage** header. One thing that attracts attention is that in the first row, there are 227 projects which have at least one benchmark that has a RCIW99 score bigger than 0. This is because one of the projects, qjpcu/sesh [29], only has 2 benchmarks, of which both have 0.0 as RCIW99 score.

Table 4.2: Number of distinct projects, whose benchmarks' RCIW99 lay more than the percent value

Percentage	0%	1%	2%	3%	4%	5%	6%	7%	8%	9%
#projects	227	98	75	58	47	53	39	35	32	31
#projects %	100%	43%	33%	25%	21%	23%	17%	15%	14%	14%

Percentage	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
#projects	28	18	13	10	8	6	6	6	6	6
#projects %	12%	8%	6%	4%	4%	3%	3%	3%	3%	3%

As I present the results for the first step of my methodology, I want to remind and answer the following research question:

- **RQ1: How variable are microbenchmark results of Go projects?**

Unlike my hypothesis, there is no normalized distribution of RCIW99 values of benchmarks. Most of the benchmarks fall to the bucket 1-10% (98.17%), from which again most them fall to 0-1% (87.70%). Based on the dataset that I analyze with 230 projects, it is clear that most of them are very stable, and 6.16% (283 in total) of benchmarks don't even vary, i.e., for each execution they have the same amount of nanoseconds as execution time. The possible reasons for the stability of benchmarks, as well as why there is such a distribution is furthermore discussed in the 5. Section.

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

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?

5.4 Future Work

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

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.

Appendix A

First Append

Bibliography

[1] D. Costa, C.-P. Bezemer, P. Leitner, and A. Andrzejak, "What's wrong with my benchmark results? studying bad practices in jmh benchmarks," *IEEE Transactions on Software Engineering*, 06 2019.

[2] "ironsmile/nedomi." https://github.com/ironsmile/nedomi. [Online; accessed 2019-08-30].

[3] "tidwall/buntdb." https://github.com/tidwall/buntdb. [Online; accessed 2019-08-30].

[4] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *2007 Future of Software Engineering*, FOSE '07, (Washington, DC, USA), pp. 171–187, IEEE Computer Society, 2007.

[5] T. H. D. Nguyen, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora, "An industrial case study of automatically identifying performance regression-causes," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, (New York, NY, USA), pp. 232–241, ACM, 2014.

[6] E. J. Weyuker and F. I. Vokolos, "Experience with performance testing of software systems: issues, an approach, and case study," *IEEE transactions on software engineering*, vol. 26, no. 12, pp. 1147–1156, 2000.

[7] C. Laaber and P. 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, (New York, NY, USA), pp. 119–130, ACM, 2018.

[8] C. Laaber, J. Scheuner, and P. Leitner, "Software microbenchmarking in the cloud. how bad is it really?," *Empirical Software Engineering*, pp. 1–40, 2019.

[9] P. Stefan, V. Horky, L. Bulej, and P. 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, (New York, NY, USA), pp. 401–412, ACM, 2017.

[10] V. Horký, P. Libič, L. Marek, A. Steinhauser, and P. Tůma, "Utilizing performance unit tests to increase performance awareness," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, (New York, NY, USA), pp. 289–300, ACM, 2015.

[11] M. Rodriguez-Cancio, B. Combemale, and B. Baudry, "Automatic microbenchmark generation to prevent dead code elimination and constant folding," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 132–143, IEEE, 2016.

[12] "OpenJDK: Java microbenchmark harness." https://openjdk.java.net/projects/code-tools/jmh/. [Online; accessed 2019-08-30].

[13] P. Leitner and J. Cito, "Patterns in the chaos&mdash;a study of performance variation and predictability in public iaas clouds," *ACM Trans. Internet Technol.*, vol. 16, pp. 15:1–15:23, Apr. 2016.

[14] A. C. Davison and D. V. Hinkley, *Bootstrap Methods and their Application*. Cambridge Series in Statistical and Probabilistic Mathematics, Cambridge University Press, 1997.

[15] S. Ren, H. Lai, W. Tong, M. Aminzadeh, X. Hou, and S. Lai, "Nonparametric bootstrapping for hierarchical data," *Journal of Applied Statistics*, vol. 37, no. 9, pp. 1487–1498, 2010.

[16] "deckarep/golang-set." https://github.com/deckarep/golang-set. [Online; accessed 2019-08-30].

[17] "segmentio/objconv." https://github.com/segmentio/objconv. [Online; accessed 2019-08-30].

[18] C. Laaber, J. Scheuner, and P. Leitner, "Performance testing in the cloud. how bad is it really?," *PeerJ PrePrints*, vol. 6, p. e3507v1, 2018.

[19] "Go programming language." https://golang.org/. [Online; accessed 2019-08-30].

[20] "Stackoverflow developer survey 2019." https://insights.stackoverflow.com/survey/2019. [Online; accessed 2019-08-30].

[21] "Package testing in go." https://golang.org/pkg/testing/. [Online; accessed 2019-08-30].

[22] "pa-tool." https://bitbucket.org/sealuzh/pa/src/master/. [Online; accessed 2019-08-30].

[23] "statistics — mathematical statistics functions." https://docs.python.org/3/library/statistics.html. [Online; accessed 2019-08-30].

[24] "mesos/mesos-go." https://github.com/mesos/mesos-go. [Online; accessed 2019-08-30].

[25] "go-gl/mathgl." https://github.com/go-gl/mathgl. [Online; accessed 2019-08-30].

[26] "nsqio/nsq." https://github.com/nsqio/nsq. [Online; accessed 2019-08-30].

[27] "uber/tchannel-go." https://github.com/uber/tchannel-go. [Online; accessed 2019-08-30].

[28] "Matplotlib." https://matplotlib.org/. [Online; accessed 2019-08-30].

[29] "qjpcu/sesh." https://github.com/qjpcu/sesh. [Online; accessed 2019-08-30].