Bachelor Thesis

September 23, 2019

A Correlation Study Between Source Code Features and Benchmark Stability

Mikael Basmaci

of Istanbul, Turkey (15-721-244)

supervised by

Prof. Dr. Harald C. Gall

Christoph Laaber

Bachelor Thesis

A Correlation Study Between Source Code Features and Benchmark Stability

Mikael Basmaci

University of Zurich^UZH

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

Bachelor Thesis
Author: Mikael Basmaci, mikael.basmaci@uzh.ch
Project period: 25.03.2019 - 25.09.2019

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

Acknowledgements

I hereby would like to thank my supervisor, Christoph Laaber, for assisting me throughout the project time, providing me with the dataset and tools that make the analysis possible, and for the whole supervision in general.

Abstract

Software microbenchmarks are used in Software Performance Engineering (SPE) to assess the performance of code fragments of a software system. Because they give non-deterministic results upon execution, there exists a certain variability in the results of microbenchmarks. Since the variability is an indicator of stability, the more variable the results, the less stable a microbenchmark is said to be. Studies show that the variability of microbenchmarks depend on different factors such as the hardware or platform the benchmarks are run on, the source code under the benchmark, or even the programming language itself. One of the factors that may affect the variability, hence the stability of a microbenchmark can be the source code features of the benchmark and the code it calls. Therefore, studying the correlation between stability and source code features of benchmarks may provide an insight into this aspect. In this thesis, I analyze variabilities of 4589 microbenchmarks from 223 open-source projects written in Go to find out about their stability, collect source code features of benchmarks by Abstract Syntax Tree (AST) parsing and callgraph analysis, and finally perform a correlation analysis based on the stability and source code features of benchmarks. Results show that 98.17% of the benchmarks have a variation below 10% as calculated by the 99th percent relative confidence interval width. Moreover, 87.70% of these benchmarks have a variability below 1%, meaning that most benchmarks are very stable. The results of correlation analysis show that 14 out of 59 collected source code features have a correlation coefficient value higher than 0.25, where the usage of sync library is the most correlating feature with a value of 0.36. This is followed by the usage of go keyword, which has the value 0.29. Generally, concurrency and control flow related features correlate with stability with a higher value than 0.22, while sync, sync/atomic and math/rand libraries correlate even more with a value above 0.25. An interesting finding reveals that pointers and defer statements are also relevant to stability. The findings from this study can be used by developers or tool designers to assess the stability of benchmarks as well as serve as a basis for predicting variability causes of microbenchmarks.

Zusammenfassung

Software-Mikrobenchmarks werden im Software Performance Engineering (SPE) verwendet, um die Leistung von Codefragmenten eines Softwaresystems zu bewerten. Da sie bei der Ausführung nicht-deterministische Ergebnisse liefern, gibt es eine gewisse Variabilität in den Ergebnissen von Mikrobenchmarks. Die Variabilität ist ein Indikator für die Stabilität, je variabler die Ergebnisse, desto instabiler soll ein Mikrobenchmark sein. Studien zeigen, dass die Variabilität von Mikrobenchmarks von verschiedenen Faktoren wie zum Beispiel der Hardware oder Plattform abhängt, auf der die Benchmarks ausgeführt werden, dem Quellcode unter dem Benchmark oder sogar der Programmiersprache selbst. Einer der Faktoren, die die Variabilität und darum auch die Stabilität beeinflussen können, kann daher der Quellcode des Benchmarks und des vom Benchmark aufgerufenen Codes sein. Aus diesem Grund kann die Untersuchung des Zusammenhangs zwischen Stabilität und Quellcode-Eigenschaften von Benchmarks einen Einblick in diesen Aspekt geben. In dieser Arbeit analysiere ich die Variabilität von 4589 Mikrobenchmarks aus 223 Open-Source-Projekten, die in Go geschrieben wurden, um ihre Stabilität herauszufinden. Ausserdem sammle ich Quellcode-Features von Benchmarks durch Abstract Syntax Tree (AST) Parsing und Callgraph-Analyse und führe schließlich eine Korrelationsanalyse basierend auf der Stabilität und den Quellcode-Features von Benchmarks durch. Die Ergebnisse zeigen, dass 98,17% der Benchmarks eine Variation unter 10% aufweisen, berechnet aus der relativen Konfidenzintervallbreite von 99%. Darüber hinaus haben 87,70% dieser Benchmarks eine Variabilität unter 1%, was bedeutet, dass die meisten Benchmarks sehr stabil sind. Die Resultate der Korrelationsanalyse zeigen, dass 14 von 59 gesammelten Quellcode-Merkmalen einen Korrelationskoeffizientswert von mehr als 0,25 aufweisen, wobei die Verwendung der Sync-Bibliothek das am meisten korrelierende Merkmal mit einem Wert von 0,36 ist. Es folgt die Verwendung des go-Schlüsselwortes, das den Wert 0,29 hat. Im Allgemeinen korrelieren die Gleichzeitigkeits- und Kontrollfluss-Features mit der Stabilität mit einem höheren Wert als 0,22, während die Bibliotheken sync, sync/atomic und math/rand noch stärker mit einem Wert über 0,25 korrelieren. Ein interessanter Befund zeigt, dass Pointers und Defer-Anweisungen auch für die Stabilität relevant sind. Die Ergebnisse dieser Studie können von Entwicklern oder Tooldesignern genutzt werden, um die Stabilität von Benchmarks zu bewerten und als Grundlage für die Vorhersage von Variabilitätsursachen von Mikrobenchmarks zu dienen.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Performance is one of the crucial qualities of a software system [5], 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 [5]. Better performance is important for many reasons. 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 the 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 in the profitableness of a company [6] [7].

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 the late 2000s, there has not been a lot of research on the field of software performance testing [8]. 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 performance regressions [8].

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 [6]. Solutions to these kinds of problems include supply of more hardware, which comes costly and is not applicable for large software systems [6] 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) activities are needed [5]. There are two general approaches found in the literature: the first one is called measurement-based SPE, which stands for experimental performance calculations made on the software to report about the performance, the second one is called model-based, which refers to creating performance models while developing to meet the performance requirements [5].

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 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 [6]. This complexity makes it hard to find performance regressions as fast as possible, causing a delay in Continuous Deployment (CD) architectures [9].

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 [10] and can test modular functions of a software system and measure their performance [7]. 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 exists for the results of an executed benchmark [9]. A benchmark is said to be stable or not stable depending on the variability of its results. Having a stable benchmark is important because being stable means having less varying results across multiple executions, and the less variable the benchmark's results, the more accurate the benchmark is able to report about performance counters. This also indicates that developers can rely on the results of a stable benchmark with less questioning about their validity. There are multiple factors that affect the variability of a benchmark such as the platform where the benchmarks are executed, the hardware that executes them, or the programming language of the benchmark itself [11]. One of the factors that might be affecting the variability of benchmarks could be the source code related features of a benchmark and it is therefore needed to study the correlation between source code features and the stability of a benchmark.

In this thesis, I am studying the stability of benchmarks from 233 open-source software projects written in Go. For this, I first analyze the variability of 4589 benchmarks across all projects from a given dataset which has benchmark results of the projects. Secondly, I extract source code features of these benchmarks by parsing their source code and doing a callgraph analysis. The source code features extracted in this thesis fall into two categories: (i) Language related source code features, which include syntactic aspects of the programming language such as loops, collections and similar, and (ii) standard library related features, with which indications about the usage of hardware and network related aspects such as I/O, HTTP calls, and similar are approximated. Thirdly, I perform an analysis using the variability results of benchmarks and their source code features to assess the correlation between the stability of benchmarks and their source code features.

To this end, I answer the following research questions:

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

My hypothesis for the first research question is: I expect many of the benchmarks to have small percentages of variability values since Go benchmarks are reported to be largely very reliable [9]. 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 valid projects, having in total 4589 valid benchmarks. Unlike my hypothesis, most of the benchmarks are very stable, in fact, 98.17% of all benchmarks have a RCIW99 value below 10%. From the rest of benchmarks, only 0.4% (19/4589) are rather unstable, having a RCIW99 value above 60%.

- **RQ2: Which source code features contribute to the stability of benchmark results and how?**

It is reported that the source of variability of a benchmark sometimes originates from the benchmark itself [10]. To that end, 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 stability of benchmark results. In particular, I suppose that cyclomatic complexity, file IO, HTTP calls and loops have a significant impact on the variability. Cyclomatic complexity and loops are directly related to the execution time of code, hence, depending on the control flow graph of the benchmark and visited nodes within the graph, a benchmark's execution time may alter. HTTP calls' performance relies on the performance of the network, that's why the usage of them may also be a cause of variability, depending on how fast the network responds at each execution of benchmarks. Similarly, file IO's performance relies on the performance of underlying hardware, which can cause a benchmark to give different results for the same code under test. For the second research question, I download all the projects from Github. Then I run my parser program Prophunt, which collects the source code features from all functions in a project. Consequently, I make a callgraph analysis for the project, by which I find all the reachable functions from a benchmark to sum up feature values of the reachable functions, resulting into the features of the benchmark itself. Finally, I use Spearman's rank-order correlation to find out about the correlation between extracted source code features and the stability of a benchmark. Results show that across all features, 11 language related and 3 library related features correlate with the stability of a benchmark having a correlation coefficient higher than 0.25 and a strong significance. Among all features, "sync" as a library feature has the highest correlation with a coefficient value of 0.35690, followed by a language feature "gos", which has the value 0,28593. All in all, concurrency and control flow related features have a correlation with a value above 0.22, while the usage of libraries sync, sync/atomic and math/rand correlate slightly more with values higher than 0.25. Also, pointers and defer statements depict a correlation with the stability.

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

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 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 Frames Per Second (FPS) score of the game run on a hardware.

Software developers can analyze the performance of parts of their software with so called software microbenchmarks, which are found either as microbenchmarks or performance unit tests in the literature [7]. 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 [7], as in performance assertions. Some studies such as [12] [13] investigate the usage of performance unit tests, while other studies such as [9] [14] work with microbenchmarks. In this thesis, I work with microbenchmarks and in the rest of this thesis I use the term performance test interchangeable with testing with microbenchmarks.

Microbenchmarks usually test for only a small fraction of the software, such as one or multiple 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 [7]. 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 [15].

Go comes with a 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 [16]. 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 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** [1].

```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 ironsmile/nedomi [1].

A benchmark in Go can be executed by using the built-in command "go test" followed by a flag "-bench", and the benchmark invokes the target code `b.N` times [16], trying to execute the underlying code as many times as possible. The default execution time of a benchmark is 1 second and the result of the benchmark is the average execution time of the target code across all runs. In this thesis, a full run of a benchmark is defined as iteration and the number of runs for a benchmark can be configured with a command-line flag "-count". The results of a benchmark refer to the collection of results from all iterations.

Go 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 configurations are especially useful for when determining whether a newly 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.

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 number of executions and the third column is the execution time per execution [16].

```
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
```

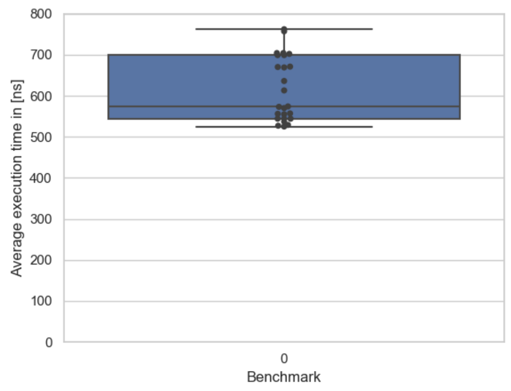
Listing 2.2: Example microbenchmark suite execution from tidwall/buntdb [3].

2.1.2 Microbenchmark variability

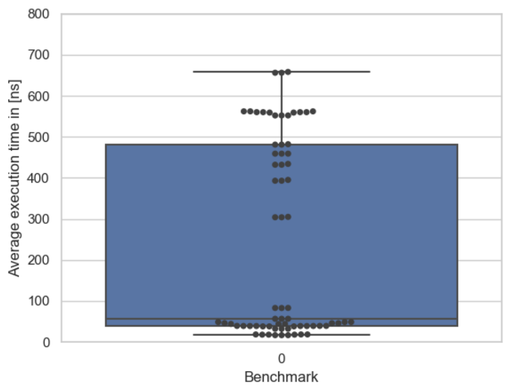
Unless otherwise configured, when a microbenchmark is run, it executes the underlying code for a certain time period (e.g., 1s) over and over for each benchmark iteration and returns a collection of average execution times for each benchmark iteration [10]. Execution times of a benchmark in Go are expressed in nanoseconds [10] [7]. 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 execution 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 deviation [10]. "CV is a statistical measure for dispersion among a population of values" [10], 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 previous studies as well [10] [17]. Since CV expects the values to be on the same relative scale (ratio), it is not a statistically sound metric for performance data, because the execution times of iterations do not form a normal distribution. 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" [10].

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 with replacement and is also used in Laaber et al.'s paper [10] [18] [19]. The reason to follow this methodology resides in the non-normalized performance measurements [10]. Bootstrapping with randomly resampling refers to randomly sampling data points from a dataset, and in this thesis, I use this technique with 10000 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 10.06%.



(b) Execution results of a microbenchmark with a RCIW99 of 53.41%.

Figure 2.1: Example for a low and high variance microbenchmark.

To illustrate a low and high variable microbenchmark: Figure 2.1a shows the execution times of the benchmark *BenchmarkSerializeStruct* from **hprose/hprose-golang** project [20]. As the box plot shows, the points are all within the whiskers and close to the mean. Also, the distribution of the points vary from 525 to 762 nanoseconds and it has a 99 percent relative confidence interval width (RCIW99) of 10.06%. In Figure 2.1b, the benchmark *BenchmarkEncoder* from **segmentio/objconv** project [21] is shown with its execution times. This benchmark's average execution times vary from 16.8 nanoseconds to 658 nanoseconds. This time, RCIW99 is 53.41%. 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 many factors such as the execution platform, the hardware the benchmarks are executed on, or, even the programming language of the microbenchmark itself [11]. 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 [10]. Some other factors include the concurrency, I/O latencies, virtualization etc [10].

Variability is a crucial factor of a benchmark because it has an impact on the stability and reliability of its results [9]. 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 higher trust. Therefore, it is essential to predict the stability of microbenchmarks. There have 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 [17], or by comparing the outcomes of forks, trials and iterations in terms of variability [10]. One way to predict the variability might be through analyzing source code features 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 feature affects the results in which way.

Executing system-wide 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 [9]. If a developer can rely on the microbenchmarks of the software, this could help to find regression causing functions, but, it is still unknown whether microbenchmarks could replace system-wide performance tests in terms of finding performance regressions. When testing the software for performance via microbenchmarks, the aim of the developer is to have the highest possible coverage with minimal effort. For this, the minimal benchmark suite is needed, which should cover all or 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 good coverage, developers might not need testing all the functionalities. That's why it is 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 prioritization 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 features 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 on November 2009 [22]. 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 fourth most active programming language in Github, and third-most highly paid language globally, according to Stack Overflow Developer Survey 2019 [23]. Furthermore, it is effectively used in the industry and has a growing community.

Functions Go offers a very flexible way of defining functions, namely, one can decide to set or omit the parameters and return values of functions. A function in Go is structured as:

```
func (receiver) Name(parameter1, parameter2) returnval1
```

Functions in Go can be called by using the name of the imported package, followed by the name of the function, as in `io.WriteString`. This way, Go evaluates the function in the resolving package and calls it in the current environment. Another type of functions that are found in Go language are the methods, which take receivers. Receivers are by definition types that are declared in packages. Methods enable object-oriented style of programming in Go by calling the specifically declared function for the type.

Concurrency Go has built-in support for concurrent programming, which is meant for both CPU parallelism and asynchrony of processes. With the keyword **"go"**, developers can create Goroutines, which are lightweight processes that can do some tasks while the main processes is running. While Go is great for separated code execution, it's also important to have access to inter-process communication. For this manner, Go implements the **channel** data structure, to which one can send and from which one can retrieve data values. The usage of channels is by default synchronized, which means that the developers do not need to program the mutual exclusion of processes additionally when they rely on channels. However, there might be a need to use asynchronous programming. For this, Go also provides great support with the `Mutex.Lock` and `Mutex.Unlock` methods from the **Sync** package. Developers can use these methods at the start and end of the functions to ensure that only one process can reach certain values at a time.

The usage of these concepts brings complexity to programs written in Go while keeping the code simple, however, with complexity, the need for testing for performance increases as well, because a complex structure may cause performance regressions under some circumstances. At the same time, these concepts can be a part of the source code features that come up in a benchmark or in the functions a benchmark calls.

2.2 Related Work

Performance testing, microbenchmarking and identifying performance regression causes are some of the prevalent topics in the field of Software Performance Engineering [6, 7, 9, 10, 12, 13, 24–27]. Another extensive study field is analyzing performance variability of and within the production clouds [17, 28].

To the best of my knowledge, there exist no study analyzing the stability of software microbechmarks written in Go by trying to find a correlation with the underlying source code features in these benchmarks. Furthermore, at the time of writing this thesis, most of the existing work is on the research of performance unit testing or microbenchmarking in the Java ecosystem [7,12,13,27], as also similarly stated in Stefan et al.'s paper [12]. However, although being a fairly new language (introduction goes back to 2009), Go has become more popular in recent years and has taken its place in the research of software performance testing [9, 10].

Laaber et al. report on the variability of microbenchmark results in various cloud environments [10]. For their study, they use 19 microbenchmarks from 4 open-source projects, 2 of them being written with Java and the other 2 with Go. To analyze the stability of these benchmarks, they pick 3 well known Infrastructure as a Service (IaaS) providers and 1 bare-metal instance from IBM. From the 4.5 million unique microbenchmarking data points that they obtain from the executions in these environments, they show that the variation of benchmarks' CVs range from 0.03% to over 100 %. It is stated that the bare-metal instance's results are very stable, nearly followed by Amazon's Amazon Web Services (AWS) cloud. Google's Google Compute Engine (GCE) and Microsoft's Azure, on the other hand, do not retrieve reliable results. As a second research topic, they cast about the slowdown detectability of benchmarks and report that while low number of instances and trials cause high false-positive rates in detecting slowdowns, an increase of instances and trials affects slowdown detection rates positively. In a second paper by Laaber and Leitner [9], the research topic is the quality of the microbenchmark suite, which is investigated by studying 10 different OSS projects. In this study they analyze 5 Go and 5 Java projects' benchmarks suites and describe the size of benchmark suites, having 16 to 983 individual benchmarks and total execution times ranging from 11 minutes to 8.75 hours. To compare the stability of microbenchmarks, they use GCE and a self-managed bare-metal server. Their *maxSpread* analysis shows that Go's benchmarks are very stable, mostly having a *maxSpread* below 0.05 in bare-metal. Moreover, Java's benchmarks have higher variations. These conclusions indicate that not all benchmarks can be trusted in terms of discovering slowdowns. Laaber and Leitner also introduce an API benchmarking score (ABS), with which they measure the slowdown detectability of benchmark suites. For the 20 often-used methods in study subjects, they show that the benchmark suites have ABS scores ranging from 10 to 100 percent.

In my study, I follow a very similar approach to what these studies do in terms of analyzing the variabilities of benchmarks. In particular, I use CV and RCIW as variability metrics as they come up in Laaber et al.'s paper [10]. Unlike the sources of variability that are investigated separately in Laaber et al.'s study [10], my study focuses on only the inherent variability of the benchmarks and does not consider iterations or trials. Furthermore, I do a large scale study including 4589 benchmarks from 223 Go projects. Nevertheless, like in both studies, in my study the benchmarks also mostly have a low variability.

Alcocer and Bergel [26] study the performance evolution of 19 projects from Pharo ecosystem by investigating the variation of benchmarks across 1439 versions. They selectively choose 49 benchmarks and run these across different versions of projects to find performance differences.

After finding these, they dig into the source code that underlies the benchmarks to find patterns that are the causes of these performance variations. Their results show that every third project exhibits performance variations across different versions. Causes of performance variations can be grouped into 9 patterns, and, the biggest factors that play a role in the variation are loops and collections.

My study is comparable to Alcocer and Bergel's study [26] as I also extract source code features of benchmarks, however, I do not consider different versions of a project but analyze benchmarks from one single commit version. A noticeable common point of our studies is that loops affect the variability, although in my study it is not the most correlating feature.

Costa et al. present 5 bad practices existing in Java Microbenchmarking Harness and show how these affect the benchmark results [7]. They base their study on a tool developed by them, *SpotJMHBugs*, which does static analysis to find out about previously defined bad practices in JMH. Out of 123 OSS Java projects, 35 projects show at least an example of a bad practice in their benchmark suites. After fixing 105 benchmarks from 6 projects, they show the significant change in results, indicating that bad practices in JMH can affect the results of benchmarks cardinally. Therefore, they provide suggestions to developers for better design of benchmark suite frameworks.

Chen and Shang [24] investigate root causes of performance regression by analyzing different commits of 2 Java projects, Hadoop and RxJava. After executing benchmarks and tests for each commit of different versions, they look up for the commits which are reported to have performance regressions and compare their execution metrics such as CPU usage, memory usage, I/O read and I/O write with the commits that have no performance regressions. Based on the comparison, they identify the causes of performance regressions and report that most of the performance regressions occur after fixing a bug and that 12.5% of the regressions found in these versions can be circumvented if precautions are taken early enough. Nguyen et al. [6] perform 2 case studies analyzing performance regression causes of an OSS and a commercial software. For this, they mine a regression-causes repository and collect performance counter data of previously run performance tests. Using machine learning, they compare the performance counters of newer test runs with the data from the mined repository. They further show that their approach can reach up to 80% accuracy in automatically detecting regression causes and that even a very small training dataset is enough to get accurate results. Luo et al. [25] present their tool called *Perfimpact*, which is a tool to extract code changes that may be playing a role in a newly introduced performance regressions. The workwise of this tool depends on search-based input profiling, with which the tool identifies inputs for a program that may cause performance regressions. Later, they use change impact analysis to track the execution trace of the program in order to evaluate changes that might have caused the regression. They test this tool with 2 open-source web applications and report that it effectively detects regression causes.

Iosup et al. [28] collect performance traces from different services of AWS and Google App Engine (GAE) and assess the performance variability of these services. They show that there exist yearly and daily patterns, however, most of the services show periods of stable performance. Furthermore, they analyze the effects of performance variability on extensive cloud applications such as scientific computation jobs, trading of virtual goods in social networks and managing the status of social games. They state that the impact of performance variability depends on the type of application. Leitner and Cito [17] do a similar study involving the performance variability and predictability of IaaS instances. For this, they run 5 micro and application level benchmarks more times in a day for a month on different instances of 4 well known IaaS providers (Amazon Elastic Compute Cloud (EC2), Google Compute Engine (GCE), Microsoft Azure, UBM Softlayer (SL)) and collect 53918 performance measurements in total. Their further analysis shows that hardware heterogeneity is nowadays lesser important unlike other studies find. Multi-tenancy, on the other hand, is a more important factor, although its effect is not applicable to all providers. Unlike Iosup et al. [28], Leitner and Cito [17] report not to find any pattern of time on the variability and predictability of the cloud performance.

Stefan et al. [12] do a research about the adoption of performance testing in 99019 Github projects written in Java. In particular, they are interested in the adoption of performance unit tests and Java's JMH performance testing framework. Based on the statistical analysis, only 370 of all projects (0.37%) have any performance testing framework. A survey about the adoption of performance unit testing conducted on 111 open-source software developers shows that only 57% of all effectively use performance tests for their design decisions. Leitner and Bezemer [27] conduct a study about the usage of performance testing in open-source software written in Java. They analyze 111 projects and report that only a small subset of projects' test suite consists of performance tests. Moreover, they show that only low number of developers in projects implement performance tests and they do not have a standard way of implementing these tests, indicating that there is a lack of standardization in creating performance tests. The authors suggest that performance testing frameworks should focus on supporting developers better in terms of writing performance tests.

Horký et al. [13] propose an approach to make developers more aware of the performance aspect of their code. For this, they create a performance unit test framework for Java which provides developers with information about the performance of the code they are working on. Although it is hard to measure the effects of such a framework on the resulting performance of the software, they claim that developers can benefit from seeing the performance measurements of the code whilst developing, which can help them make better decisions even about small artifacts, which they normally would not take into consideration.

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 shows the steps along with the programming/scripting languages involved in this thesis. In the first step (1), 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 (2), I download all the projects that have an entry in the previous CSV file and run a parser tool written in Go (Prophunt) that collects source code features for each of the functions found in the projects. This is followed by a callgraph analysis using another tool of the same language (Callgraph Analyer), which gives CSV files containing source code features of all analyzed benchmarks for each project. In the final step (3), I do a correlation analysis using the variabilities of individual benchmarks as independent variables and their features as dependent variables. The outcomes of each step are presented in the 4. Section.

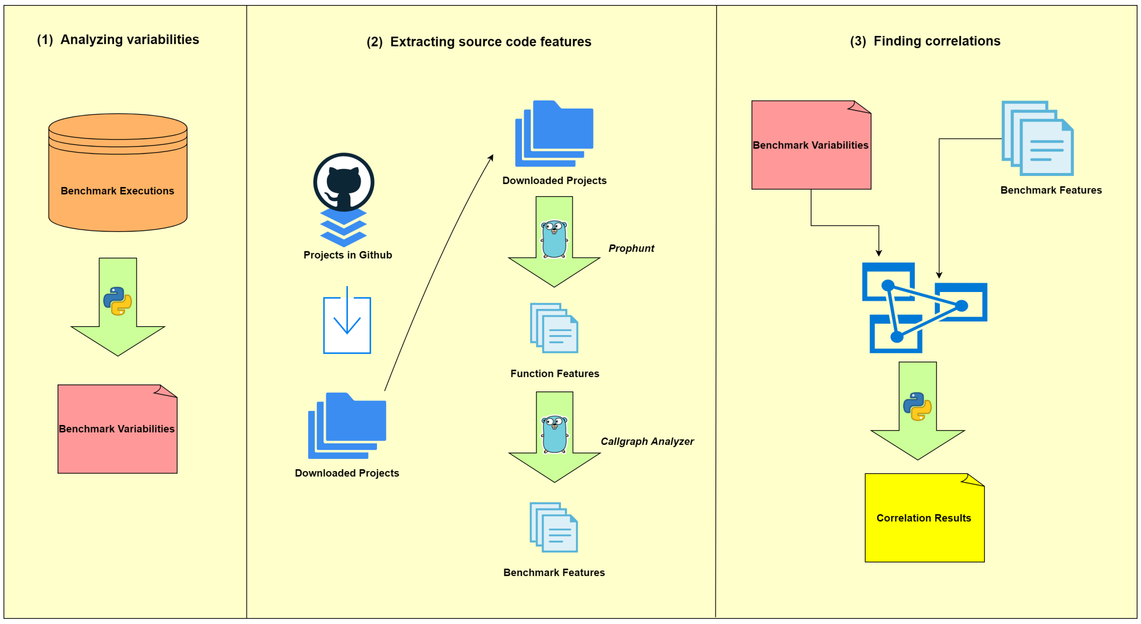


Figure 3.1: 3 main steps of the methodology.

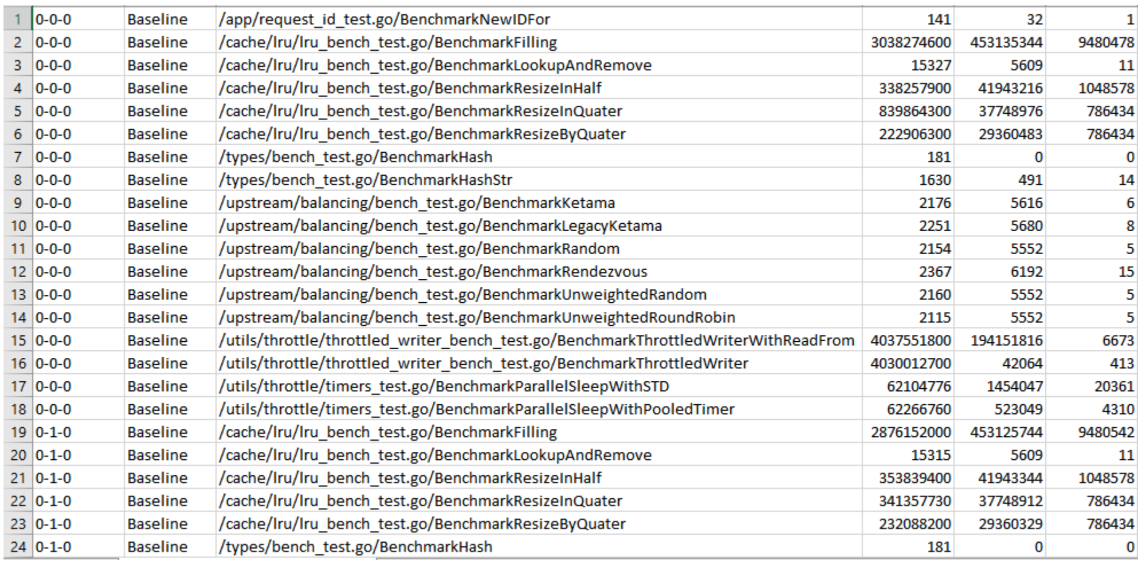
3.1 Analyzing variabilities

The first part of this thesis involves doing a quantitative analysis by analyzing the given data set (This part refers to the 1. box in Figure 3.1). From this data set, I first 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" [29], 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 to 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 project looks like in Figure 3.2. In this file, the middle part in the first column specifies the number of runs, the second column specifies the benchmark including the relative path and file where the benchmark is located in, the fourth 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 [1].

3.1.2 Metrics as the variability indicators

As described in Section 2.1.2, 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 fourth column of the results file for each benchmark into consideration. To calculate the CV, I first find the standard deviation and mean of the execution times of each benchmark. Dividing the standard deviation by the mean gives me the CV. For the mathematical representation of CV, if the collection of benchmark results is called M, then the CV of M can be shown as:

$$cv(M) = \sigma_M / \mu_M$$

where σ_M is the standard deviation and μ_M is the mean of the collection.

For the RCIW part, I use a tool by Christoph Laaber, called "pa-tool" [29]. 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., 10000 times (number of bootstrap simulations), and returns the normally distributed set of mean execution times. This is because the distribution of the bootstrap means is normal due to the central limit theorem. Having the normal distribution is a requirement to take the confidence interval of the set. The tool finally 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 10000 times with the 0.01 significance level [1].

```
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 ironsmile/nedomi [1].

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, divided by the mean of the benchmark results. For the mathematical representation, if we call the benchmark B, collection of benchmark results M, the normal distributed output of pa-tool M_n, and the confidence interval of M_n *CI*, then the RCIW value of a benchmark can be shown as:

$$RCIW(B) = (CI_R - CI_L)/\mu_M$$

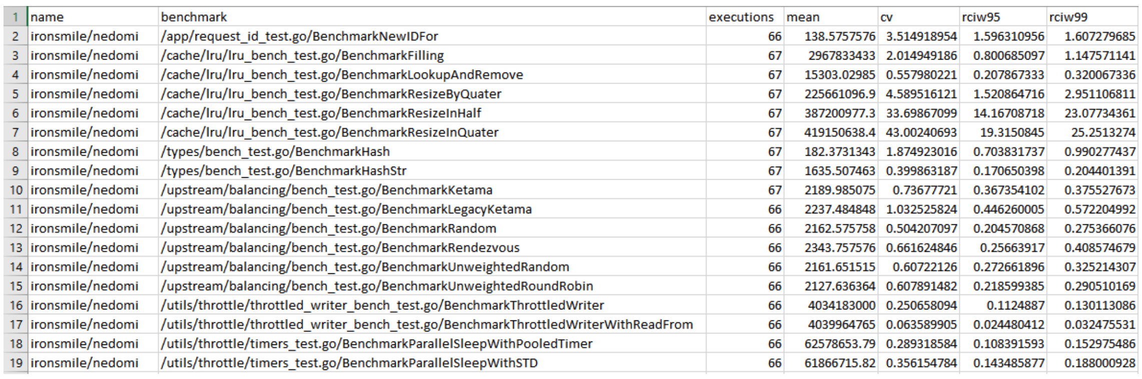
where CI_R is the right, CI_L is the left boundary of the confidence interval and μ_M is the mean of benchmark's execution results.

3.1.3 Benchmark Variabilities

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 first 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 10000 bootstrap simulations.

By subtracting boundaries from pa-tool output, I get RCIW95 for the results with 0.05 significance level of confidence interval, and RCIW99 for the results with 0.01 significance level of confidence interval. For the CV part of each benchmark, I use *mean()* and *stddev()* from Python's *statistics* module [30].

Within the end of calculating CV, RCIW95 and RCIW99 of each benchmark, I write all the benchmarks into the **Benchmark_Variabilities.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 **Benchmark_Variabilities.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 Benchmark_Variabilities.csv file involving variability values for [1].

3.2 Extracting source code features

This is the second part of the study, which involves extracting the source code features of benchmarks, for which I calculated the variabilities in Section 3.1. This part consists of a downloading projects from Github, extracting information about all kinds of functions (normal, test and benchmarks) in these projects and finally creating CSV files for each project, having the benchmarks and their features, which can visually be seen in the second box of Figure 3.1. Results of this part are to be found in Section 4.2.

3.2.1 Decision on source code features

Prior to programming for each of the parts, I do a qualitative analysis to decide on the features that I want to extract for each benchmark. For this, I start by analyzing some of the benchmarks with particularly high variabilities and look for the features that come up in these. While this gives me some basic ideas about what to analyze, I also scroll through the standard library of Go [31] and look for the libraries, which in my opinion might make an impact to the variability of the benchmarks. At the time of doing that, my particular inspectation goes to the libraries which implement functionalities about handling file IO, HTTP calls and other functionalities that might otherwise make an impact on the variability.

As a second source for the source code features, I have a look at the Go Programming Language [22] and its language features. Near features such as data structures, control flow elements and error handling mechanisms, Go is also fully equipped with concurrency related features, which enable developers to code parallel programs without having to code for underlying data structures in the first place. With this, I put language related features on top of usage of standard libraries whilst deciding for the source code features. Additionally, I look up for source code metrics that are often used/extracted in the literature and find cyclomatic complexity as a metric to measure the depth of control flow graph of a function [32].

Finally, I create the list of features in Table 3.1 to analyze when extracting information out of functions. In total, there are 4 different types of features that I analyze. First one is the usage of the specified standard library. In this type, there are 31 libraries that can play a role in the variability of the benchmarks. In particular, I inspect **io, io/ioutil** which represents file I/O operations; **net/http, net/http/httptest, net/http/httptrace, net/http/httputil** which represent HTTP calls and other HTTP related functions. All the remaining features are for me of interest as well: For instance, I wonder if the usage of random functions has an impact on the variability and look for the occurrences of **math/rand**; similarly, I'm interested in the usage of synchronization primitives and seek for **sync, sync/atomic** to see whether count of usage of this library within a benchmark affects the benchmark in a significant way. Second feature type in the list is signature of the function, which stands for the features that are directly related to the function. In this type, there are 6 features, which can be extracted by looking at the place of function's definition and the identifiers of the function. Third feature type is called body of the function, which contains all the features that can be extracted from the body of the benchmarks. For the variability of the benchmark, these can also be a predictor of stability. Last feature type is other, which has cyclomatic complexity.

For the extraction of these features, I use the methodology of parsing the Abstract Syntax Tree of a source code file. While signature features require only the inspection of function declarations within the AST, the other 3 types require analysis of the body of the function within the declaration, going into a deeper level.

Feature Type	**Feature Name**	**Explanation**
Standard library usage	bufio	Library to handle buffer actions.
	bytes	Library to manipulate byte slices.
	crypto	Library that has common cryptographic constants.
	database/sql	Library for database interactions.
	encoding	Libraries for handling different type of encodings.
	encoding/binary	
	encoding/csv	
	encoding/json	
	encoding/xml	
	io	Libraries that hold io primitives and io functionalities.
	io/ioutil	
	math	Libraries for math functions and random implementations.
	math/rand	
	mime	Library implementing parts of MIME spec.
	net	Library for handling network I/O.
	net/http	Libraries implementing HTTP client and server, utilities for HTTP testing, mechanisms to trace events and other HTTP utility functions.
	net/http/httptest	
	net/http/httptrace	
	net/http/httputil	
	net/rpc	Libraries that implement remote procedure calls for objects.
	net/rpc/jsonrpc	
	net/smtp	Library to handle Simple Mail Transfer Protocol.
	net/textproto	Library that implements generic support for text based request/response protocols.
	os	Library to handle OS functionality.
	os/exec	Library to run external commands.
	os/signal	Library to access incoming signals.
	sort	Library to sort slices and user defined collections.
	strconv	Library that implements conversion to and from string representations.
	sync	Libraries that implement basic synchronization primitives and low-level atomic memory primitives.
	sync/atomic	
	syscall	Library that implements an interface to low-level operating system primitives.
Signature of the function	pkgfiles	Files in the package where the function belongs to.
	fileloc	Lines of code of the file where the function belongs to.
	namelength	Length of the name of the function.
	parameters	Parameters of the function.
	returns	Return values of the function.
	loc	Lines of code of the function.
Body of the function	funccalls	Function calls within the function.
	loops	For and while loops within the function.
	nestedloops	Nested for/while loops within the function.
	channels	Channel creations within the function.
	sends	Sending data to the channel within the function.
	receives	Receiving data from the channel within the function.
	closes	Channel terminations within the function.
	gos	Go keywords (new threads) within the function.
	concrranges	Channel loops within the function.
	selects	Select statements within the function.
	selectcases	Cases in select statements within the function.
	variables	Variable, pointer, slice and map declarations within the function.
	pointers	
	slices	
	maps	
	ifelses	If-else statements within the function.
	switches	Switch statements within the function.
	switchcases	Cases in switch statements within the function.
	panics	Panic statements within the function.
	recovers	Recover statements within the function.
	defers	Defer statements within the function.
Other	cyclomaticcomplexity	Cyclomatic complexity of the function.

Table 3.1: List of source code features.

3.2.2 Downloading projects

To download all the valid projects that resulted with 4589 benchmarks in the first analysis, I first create a little CSV file which stores the name of the projects and its commit, where the execution results are from. I then write a Python script, which reads this file and downloads the projects sequentially from Github. For downloading, I use Go's *get CLI flag*, as this is intended to download and install a Go project along with its dependencies. However, downloading these projects with:

```
go get github.com/{owner_name}/{project_name}
```

does not suffice, because the commits of these projects are from a time when Go Modules did not use to exist. This means, at the time of these commits projects had their own package management tools, which ensured getting the right dependencies for the project [33]. Because projects had their own GOPATH environment back then, I create a GOPATH for each of the projects prior to downloading them. On one hand, I ensure that their dependencies from their commits can be stored in their own GOPATH, on the other hand, I avoid clashing dependencies from other projects, which would then share the same GOPATH for all the dependencies. Post download, I then use a locally installed Git to checkout the project folder from master to the given commit in the CSV file. As an output, I get a CSV file containing the name, commit and installed path of each project. This output file is useful for the parser tool that can iterate through different projects, which is explained in the next section. Figure 3.4 illustrates the process of downloading projects.

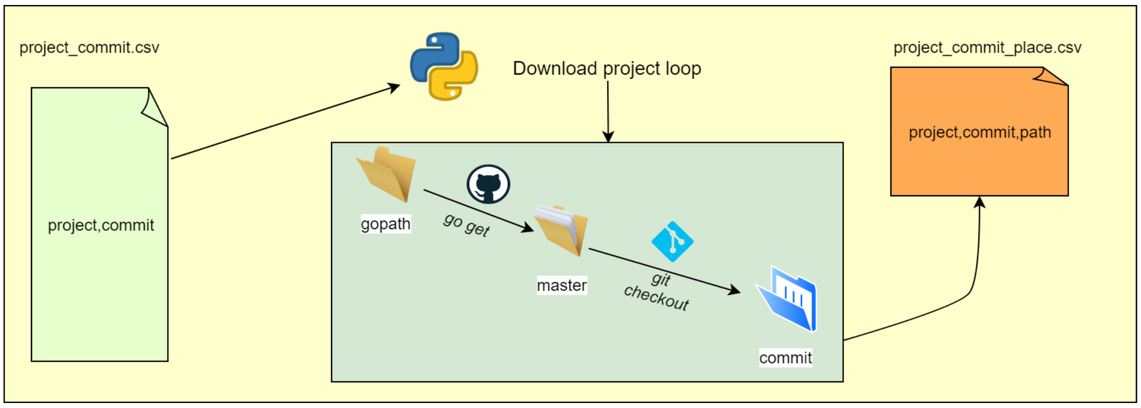


Figure 3.4: Process of downloading projects from Github.

Extracting source code features in this thesis consists of two main parts. First part has to do with AST parsing of source code files to extract all the functions within a project. The second part is about using a callgraph CLI tool which resides in Go's *golang.org/x/tools/cmd/* repository [34]. In particular, I use the parser tool in the first part to create a CSV file from a project which has all the functions with their source code features. Following that, I use the callgraph tool to iterate through the callgraph of each benchmark found in the resulting CSV file from the parser tool in order to find all the called functions from a benchmark. This way, I'm able to match all the called functions from the benchmark in the first CSV file and calculate the sum of each feature value for the benchmark. The output of the callgraph tool is a CSV file for the project, which contains only the benchmark functions with their source code features.

For both of the parts, I use Go programming language as it offers great functionabilities when it comes to parsing Go's source code files and collect source code information. I present the resulting tools "Prophunt" and "Callgraph Analyzer" in the next sections.

3.2.3 Prophunt

After having all the projects residing in their own GOPATHs, next step is to start with parsing the source code of each file found in a project. But before that, there is still a step that needs to be taken, which is getting the dependencies. Downloading the projects with "go get" without any further flag only causes downloading and installing them in the specified GOPATH, fetching only the dependencies that can be fetched via Go Modules and only for the master commit. However, to be able to parse the source code correctly and have no compilation errors, all the dependencies that are required for that special commit need to be fetched. Because of that reason, I first implement a mechanism to iterate through all the project paths, set the GOPATH accordingly, call *"go get -t ./..."* inside the project's root folder and use *deps/fetch.go/Fetch* method from Christoph Laaber's GoABS implementation [2].

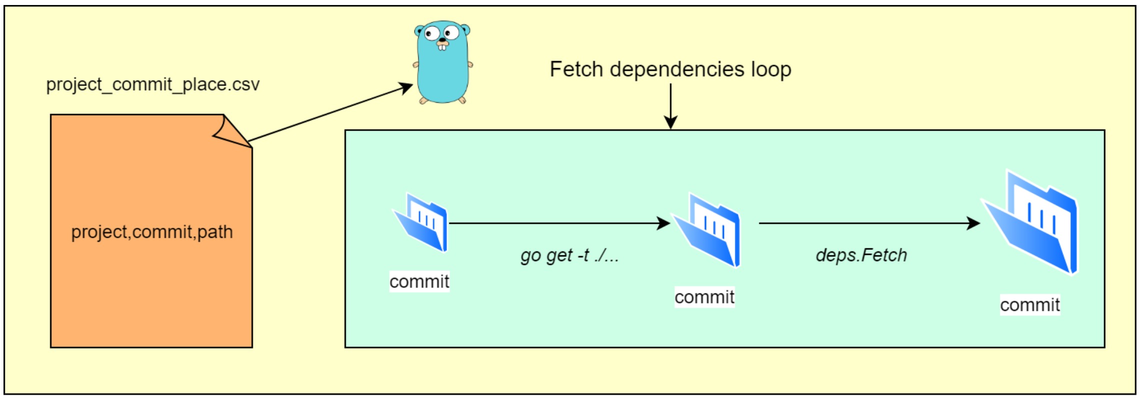


Figure 3.5: Process of fetching dependencies for each project.

Figure 3.5 shows how this process runs in Go. The optional "-t" flag of *go get* ensures the installation of all required test packages and the latter *./...* tells *go get* to fetch all dependencies for this project. This command is crucial since without getting dependencies for test packages some of the source code files cannot be compiled correctly, which leads to not being able to parse them correctly, using Prophunt. For all the projects with their specific commits, which are compatible with Go Modules, this step suffices in terms of being ready to be parsed. However, there are projects which use other package management tools [33]. For these kinds of projects, I use *deps.Fetch* method and give it the path of the project, from which it automatically extracts GOPATH of the projects and uses one of the preinstalled dependency management tools to get the dependencies. As of writing this thesis, this method has support for the tools (which need to be installed before using the method) listed in Table 3.2.

Table 3.2: List of dependency management tools supported in GoABS [2].

Tool	Repo
Get	built-in
dep	https://github.com/golang/dep
Glide	https://github.com/Masterminds/glide
Godep	https://github.com/tools/godep
Govendor	https://github.com/kardianos/govendor
gvt	https://github.com/FiloSottile/gvt
govend	https://github.com/govend/govend
trash	https://github.com/rancher/trash
gom	https://github.com/mattn/gom
gopm	https://github.com/gpmgo/gopm
Gogradle	https://github.com/blindpirate/gogradle
gpm	https://github.com/pote/gpm
glock	https://github.com/robfig/glock

I install these tools prior to fetching dependencies in my Ubuntu 18.04 Virtual Machine (VM) environment. After fetching dependencies for all the projects, I am ready for the next step in Prophunt, which is parsing.

Parsing I first start my parsing processes by importing Go's **parser** library and giving its *ParseFile* method some source code files written in Go and the *parser.ParseComments* mode as parameters to parse everything including comments in a file. This method, unless errors occur, returns an ***ast.File** instance, which represents a file in Go environment. This instance, as shown in Listing 3.2, has all the information about a .go file. This return value can now be investigated by using its attributes, for example, **ast.File.Name* gives name of the package this .go file belongs to.

```
type File struct {
Doc *CommentGroup // associated documentation; or nil
Package token.Pos // position of "package" keyword
Name *Ident // package name
Decls []Decl // top-level declarations; or nil
Scope *Scope // package scope (this file only)
Imports []*ImportSpec // imports in this file
Unresolved []*Ident // unresolved identifiers in this file
Comments []*CommentGroup // list of all comments in the source file
}
```

Listing 3.2: *ast.File declaration in Go.

Since I am interested in the function declarations in the file, I investigate the **Decls** slice and search for the function declarations in this slice. A function declaration is defined as ***ast.FuncDecl** within this library, and has attributes such as **Doc**, **Recv**, **Name**, **Type** and **Body**, as in Listing 3.3. From the **Type** attribute, one can read information about parameters and return values of the function. From the **Recv** attribute, the receiver of the method can be read. Rest of the features that I look for are located in the **Body** of the function.

```
FuncDecl struct {
   Doc *CommentGroup // associated documentation; or nil
   Recv *FieldList // receiver (methods); or nil (functions)
   Name *Ident // function/method name
   Type *FuncType // function signature: parameters, results,
                  // and position of "func" keyword
   Body *BlockStmt // function body; or nil for external (non-Go) function
}
```

Listing 3.3: *ast.FuncDecl declaration in Go.

Signature features are trivially extracted from the ***ast.FuncDecl**, however, for all the body, standard library usage and other types of features, one needs to know what to look for inside the **Body** of the ***ast.FuncDecl**. To this end, I first look for Go's "A Tour of Go" documentation [35] and find code examples about features of the language. While one can use these code examples in the programming environment, I aim for a GUI approach to better understand how to look for features in the bodies of functions. Fortunately, I come across a web application from yuroyoro [4], which offers a GUI to visualize a Go AST, given Go source code. Listing 3.4 shows an example output from the AST Viewer of a main function with a "Hello Golang" print on the stdout.

```
23 . . 1: *ast.FuncDecl {
24 . . . Name: *ast.Ident {
25 . . . . NamePos: 7:6
26 . . . . Name: "main"
27 . . . . Obj: *ast.Object {
28 . . . . . Kind: func
29 . . . . . Name: "main"
30 . . . . . Decl: *(obj @ 23)
31 . . . . }
32 . . . }
33 . . . Type: *ast.FuncType {
34 . . . . Func: 7:1
35 . . . . Params: *ast.FieldList {
36 . . . . . Opening: 7:10
37 . . . . . Closing: 7:11
38 . . . . }
39 . . . }
40 . . . Body: *ast.BlockStmt {
41 . . . . Lbrace: 7:13
42 . . . . List: []ast.Stmt (len = 1) {
43 . . . . . 0: *ast.ExprStmt {
44 . . . . . . X: *ast.CallExpr {
45 . . . . . . . Fun: *ast.SelectorExpr {
46 . . . . . . . . X: *ast.Ident {
47 . . . . . . . . . NamePos: 8:2
48 . . . . . . . . . Name: "fmt"
49 . . . . . . . . }
50 . . . . . . . . Sel: *ast.Ident {
51 . . . . . . . . . NamePos: 8:6
52 . . . . . . . . . Name: "Printf"
53 . . . . . . . . }
54 . . . . . . . }
55 . . . . . . . Lparen: 8:12
56 . . . . . . . Args: []ast.Expr (len = 1) {
57 . . . . . . . . 0: *ast.BasicLit {
58 . . . . . . . . . ValuePos: 8:13
59 . . . . . . . . . Kind: STRING
60 . . . . . . . . . Value: "\"Hello, Golang\\n\""
61 . . . . . . . . }
62 . . . . . . . }
63 . . . . . . . Ellipsis: -
64 . . . . . . . Rparen: 8:30
65 . . . . . . }
66 . . . . . }
67 . . . . }
68 . . . . Rbrace: 9:1
69 . . . }
70 . . }
```

Listing 3.4: Sample output from yuroyoro's Ast Viewer [4].

Generally, the features to collect are reachable from the **Body** by querying for the correct parser instance. Table 3.4 presents how I extract these features or which data structure I aim for when visiting the nodes in a function declaration. For the cyclomatic complexity, I adapt the calculation to an existing calculation approach by fzipp/gocyclo [36].

Table 3.4: Extraction of features based on parser library.

Feature Name	According Parser Instance / Extraction Method
funccalls	*ast.CallExpr
loops	*ast.ForStmt / *ast.RangeStmt
nestedloops	none, counting loops inside loops
channels	*ast.ChanType
sends	*ast.SendStmt
receives	*ast.UnaryExpr.Op being "<-"
closes	*ast.CallExpr having name "close"
gos	*ast.GoStmt
concrranges	none, counting loops with channel ranges
selects	*ast.SelectStmt
selectcases	*ast.SelectStmt.Body.List length
variables	*ast.DeclStmt –>*ast.GenDecl or *ast.AssignStmt right side being *ast.BasicLit
pointers	*ast.AssignStmt right side having ast.UnaryExpr.Op being "&"
slices	*ast.ArrayType
maps	*ast.MapType
ifelses	*ast.IfStmt
switches	*ast.SwitchStmt
switchcases	*ast.SwitchStmt.Body.List length
panics	*ast.CallExpr having name "panic"
recovers	*ast.CallExpr having name "recover"
defers	*ast.DeferStmt
cyclomaticcomplexity	calculated by counting each instance of: ast.FuncDecl ast.IfStmt ast.ForStmt ast.RangeStmt ast.CaseClause ast.CommClause ast.BinaryExpr within the function
standard library usages	checking resolved funccalls within a function

To be able to parse each function correctly and later match them with the output of the callgraph tool, I need to resolve the package of the function or the package of the receiver of the method if a receiver exists. Unfortunately, without having types info for the file to be parsed, this is impossible unless all the information is found in the same file. In practice, this is rarely the case.

Package packages I use a library called **packages** from Go's *golang.org/x/tools/go/* repository [37], which offers the functionality to load a package's related info to the programming environment. This comes as a perfect solution to the problem of resolving types, since it gives all the found .go files in the folder with their according package and a slice of ***ast.File** for the files that compile within the folder. Furthermore, since it automatically parses all the files, there is no need to parse all the files one by one as I practiced in the previous section. The crucial feature of this functionality is, that it automatically generates a ***types.Info** instance for each package, which can later be used to query for the resolving type of an existing ***ast.Ident** node in the AST tree. Best practice to use this package is by giving the path of a folder containing Go source files as a parameter to the configuration of *packages.Load*, while also telling the configuration to include all test files.

Go's standard practice tells to put the test files in the "_test" version of the package found in the folder. If that is the case for a folder, *packages.Load* returns both the normal and "_test" version of the package with their compiled .go files. However, in the dataset that I analyze there are projects which don't follow this practice, hence, they use only one package per folder and include the tests in the same package. In that case, *packages.Load* returns all the .go files in the "package [package.test]" package.

Having **packages** library as the skeleton to my parsing technique, I create **Prophunt**, which is a tool that takes a project folder as a parameter and starts walking all the folders of the project, starting from the root folder. For each folder, resulting packages are acquired via *packages.Load* and function declarations are extracted for each of the compiling .go files. Then, the features of each function are collected from the functions, using algorithms that aim for the according parser instance or extraction method of the feature to be extracted. At the end of visiting all possible functions, the functions are written into a CSV file which includes all 3 kinds of functions from a project, namely: **normal**, **test** and **benchmark** functions. Figure 3.6 illustrates the workflow of Prophunt. Simply explained, it takes the CSV from Python downloader script to iterate through projects, creates a map containing each function and their features, and fills it by loading each package, filtering the functions and extracting their features. At the end of visiting all folders of a project, the map is transformed into a CSV for the project, which is saved in *Prophunt_Output* folder. For convenience, also an *index.csv* file is written in this folder for later usage with callgraph analyzer, which contains the names and Prophunt_Output paths of the projects.

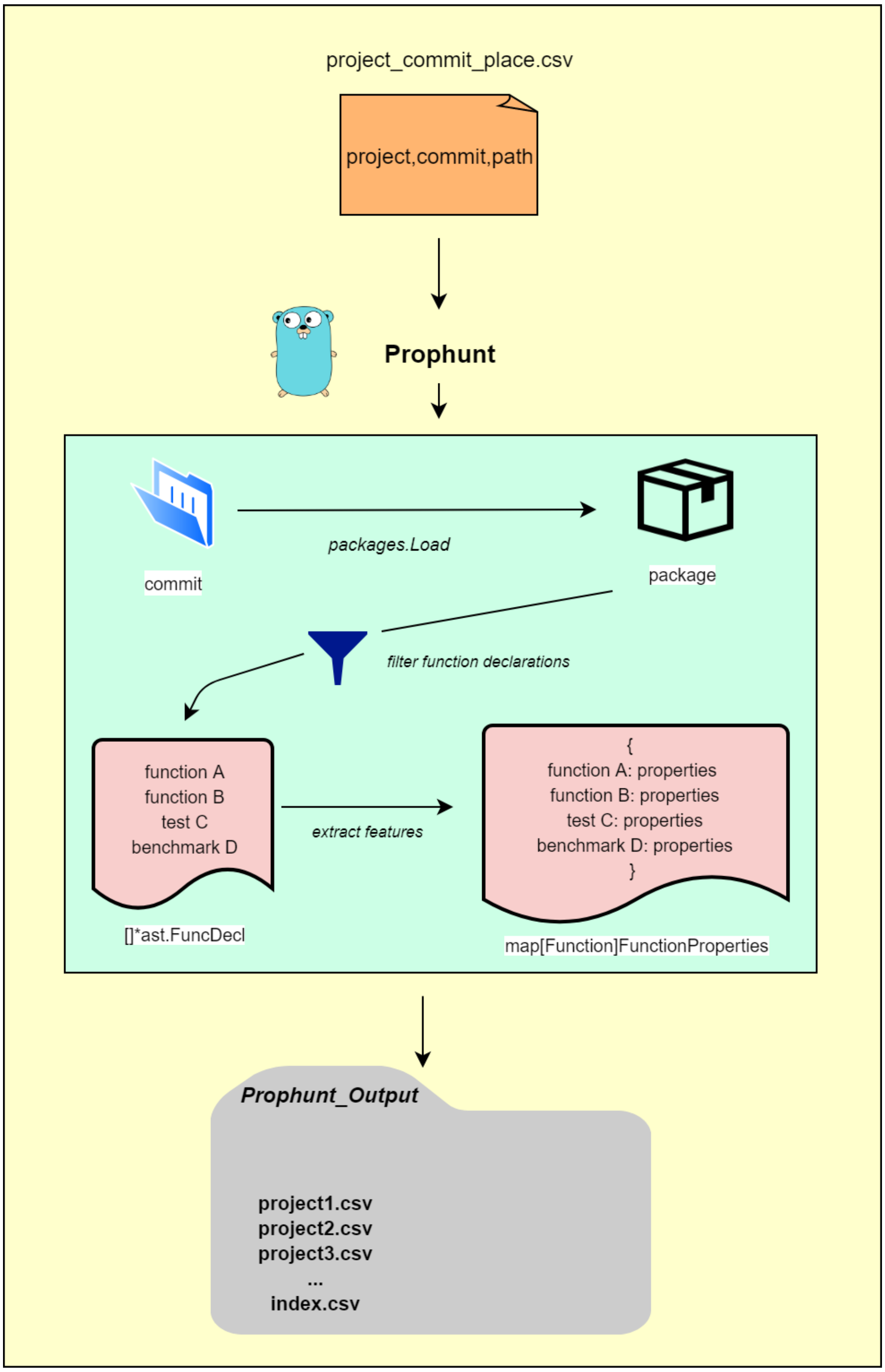


Figure 3.6: Workwise of Prophunt iterating through downloaded projects.

3.2.4 Callgraph Analyzer

Creating Prophunt outputs for all the projects is only half of the story, because the information in these CSV files are only relevant to the functions themselves, i.e., having only Prophunt outputs I am limited to features of an individual function. However, to make a proper correlation analysis, it is interesting and important to look at the callgraphs of the benchmarks, because these reveal information about all the functions that are called by a benchmark. The idea behind the callgraph analyzer is to build a directed graph, which has function nodes and edges between function nodes representing a function call. In such a graph, the outgoing node of an edge is the caller, and the incoming node of an edge is the callee. Having such a datastructure can be used to return all the visited functions from a benchmark. To this end, I search for a tool which can in some way return me the edges between functions, which in whole builds the callgraph of the project. Fortunately, I come across the callgraph CLI tool, which resides in Go's *golang.org/x/tools/cmd/* repository [34].

callgraph CLI tool The tool expects several flags to give the callgraph of a project as output: the first and most important flag is the algorithm ("-algo") that is used to create the callgraph. Under 4 different algorithm options (**Static**, Class Hierarchy Analysis (**cha**), Rapid Type Analysis (**rta**) and inclusion-based Points-to-Analysis (**pta**)), I use inclusion-based Points-to-Analysis. **pta** is often referred as Points-to-Analysis and is used in computer science as a static code analysis technique. In its pure form, the algorithm tries to find about which pointer or heap reference can be pointing to which variable or storage location at the runtime. This algorithm, as **rta**, requires a whole program to give correct output and includes only functions that are reachable from main [34]. This means, that if a benchmark has function calls that are not included in the main or tests of the whole project, this benchmark will likely not be included in the callgraph tool's output, which is the only analyzed drawback by me. The other flags of the tool include "-test", which is to enable creating callgraphs of tests as well and "-format", which is to format the output of the callgraph tool. Using "-format" with "digraph" gives all the edges in the form:

```
"package.functionA" "package.functionB"
"(package.receiver).functionC" "(*package.receiver).functionD"
"(package.receiver).functionE" "(package.receiver).functionC"
```

Listing 3.5: Output format of callgraph CLI tool.

where each **functionX** is defined by the signature of the function in a specific way. For instance, if a function is a method, it has the resolved package of the receiver concatenated with the receiver in parentheses before the function name. If the receiver is a pointer type, it has an asterisk at the beginning of the receiver's resolved package. In any other case, the function has the resolved package followed by the function name. Listing 3.5 shows all kinds of examples explained here.

Next step in order to have a sound directed graph of a project is to find a datastructure, which can store all the nodes with their edges. For this, I import the graph library of **gonum/gonum** [38], which is a performant library that has implementations of different graph types. In particular, I use the ***simple.DirectedGraph** data structure to feed all the edges that can be acquired from the project. Once this graph is fully fed with every found edge from the callgraph tool, the graph can be queried for all the callees of a benchmark.

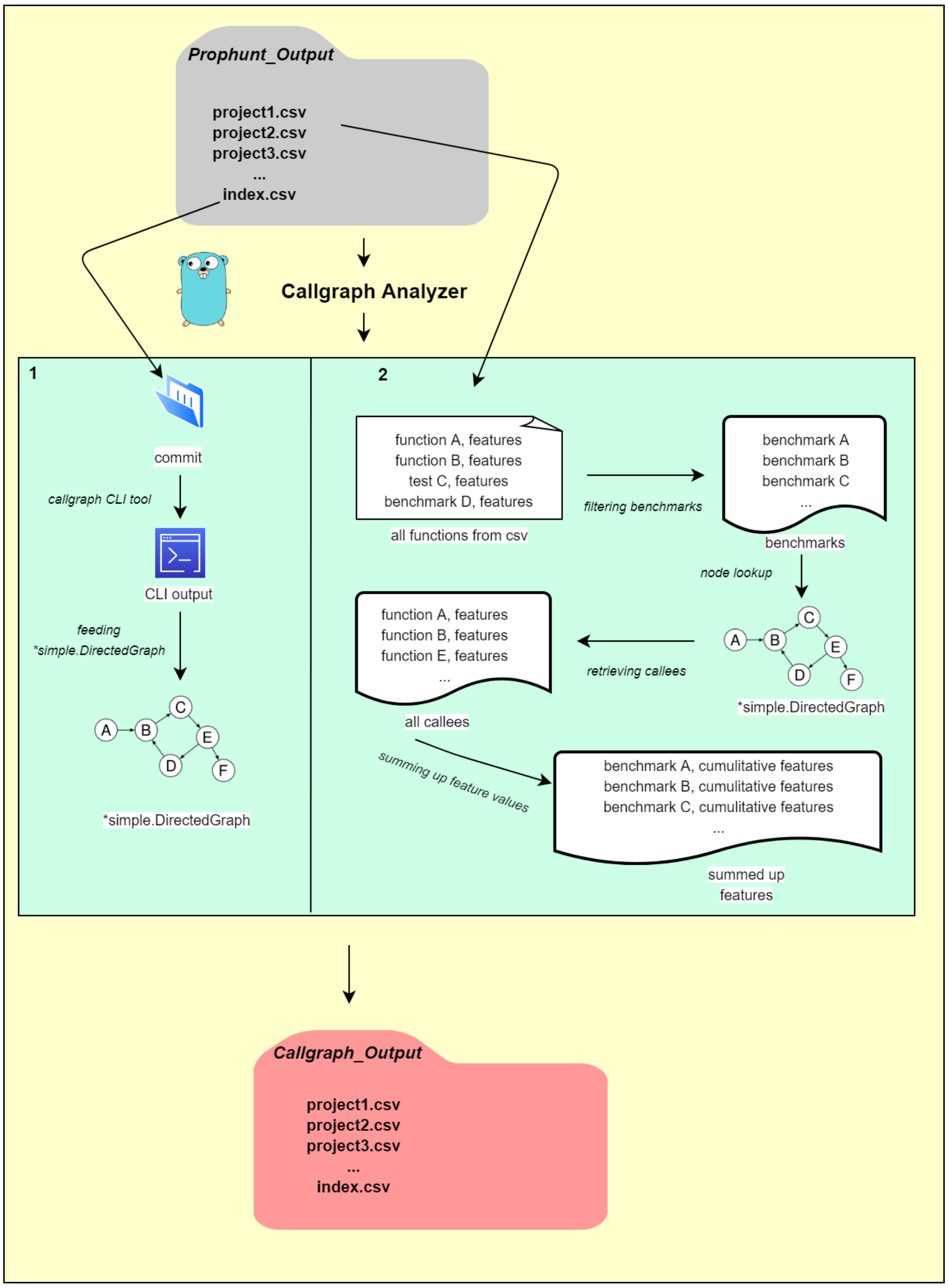


Figure 3.7: Workwise of Callgraph Analyzer iteration through all parsed projects.

Figure 3.7 shows how Callgraph Analyzer works. Similar to Prophunt, Callgraph Analyzer takes the full path of the project to be analyzed from Prophunt_Output's index.csv. A single loop execution consists of 2 phases. In the first phase, the callgraph tool is called for each of the projects folder because of 2 reasons: (1) a project might not have a main in the root folder, hence, the callgraph fails to create the edges and (2) there might exist more than 1 main in the whole project structure and they may have different scopes for the functions across the project. After each call of the CLI tool, the output is read and the nodes and their edges are collected. These are directly fed to the ***simple.DirectedGraph** instance of the project. Due to the precautions in the implementation of ***simple.DirectedGraph**, a node cannot be added to the graph twice, hence, I eliminate edges that occurred in a previous output from the process and only feed the nodes if they were not in the graph before. A full walk through the project path results as a full callgraph of a project.

In the second phase of the same loop execution, according CSV file for the project is read from Prophunt_Output folder and its benchmarks are filtered. Having all benchmarks on one place, a lookup for all direct callees of one benchmark node can be made by using *simple.From()* method, when the node of the benchmark is given as a parameter. Since I am interested in not only the direct callees but also all reachable nodes from a benchmark node, I implement a recursive algorithm which calls *simple.From()* method for all the nodes that can be visited from the initial benchmark node. This ensures collecting all the reachable nodes in the callgraph, and as a next step, I search for the callees in the according projectX.csv file. If reachable nodes are present in the CSV file of the project, I collect their CSV entries to later sum up all the feature values, resulting in the benchmark's cumulative features. Note that "namelength" and "parameters" are not summed up, as their cumulative value is not useful for the correlation analysis. Using this methodology, there are 3 possible outcomes for every benchmark analyzed by Prophunt: (1) the benchmark cannot be found in the callgraph, resulting into a nil node in the programming environment. In this case, I skip this benchmark and do not record it in the output; (2) the benchmark is found in the callgraph, however, there is no reachable node from this benchmark, either because there was a compilation error in the callgraph CLI tool, or the callees of the benchmark were not reachable from main of the project; (3) the benchmark is found in the callgraph and there is at least 1 reachable node from the benchmark in the callgraph. For the completeness of the results, I anticipate that there are not many occurrences of the first and second outcomes, nevertheless, the numbers are reported in the 4.2. Section. Finally, Callgraph Analyzer gives an output folder called Callgraph_Output with a CSV for each project containing only the benchmarks with their cumulative feature values, as well as an index.csv file which is used later at the correlation analysis.

3.3 Finding correlations

Third and last part of the study is to find correlations between source code features of benchmarks and their variability, which is shown in the 3rd box of Figure 3.1. In this last part, I use Spearman's rank-order correlation on the two variables I acquired in the 2 previous sections (variability values and benchmark features). Results of this part are to be found in Section 4.3.

3.3.1 Correlation analysis

For the analysis, I have 2 outputs from the previous sections, namely the variability values of each benchmark in Benchmark_Variabilities.csv file and features of these benchmarks in Callgraph_Output folder. Before I start with the analysis, I bring the feature values of each project into one CSV file which I name Benchmark_Features.csv to be able to match the benchmarks in Benchmark_Variabilities.csv file easily. Next, I load these CSV files in Python environment using Pandas library [39] and merge the files using a SQL style left join to match benchmarks.

For the calculation of Spearman's rank-order correlation coefficient, I use the implementation from SciPy library [40]. *Spearmanr* method of this library takes as parameters two series of variables and returns the coefficient value and the p-value representing the significance of the correlation value. Spearman's rank-order correlation depicts monotonic relations, which means it looks for the behaviour of change in one variable while observing the change on other variable. In this manner, the value of coefficient lies between -1 and 1, where -1 means a perfect negative correlation and 1 a perfect positive correlation. The p-value for the coefficient is interpreted as follows: If the value is between 0.1-0.05, the significance of the coefficient is weak, if it is between 0.05 and 0.01, the significance is strong, and if it is below 0.01, the significance is very strong.

From the merged pandas dataframe I create a feature series for all of the 59 different feature types and use these as the first variable in the *Spearmanr* parameters. For the second variable, I create CV, RCIW95 and RCIW99 series. Finally, I let SciPY's *Spearmanr* method calculate the coefficients and p-values for all feature types and variability values. For instance, *Spearmanr* of **pkgfiles** and RCIW99 returns 0.28000358145089915 as coefficient value and 8.003648264207533e-79 as p-value, meaning that pkgfiles is somewhat correlated with the variability of the benchmark and this correlation is very significant. Detailed analysis follows in Section 4.3.

3.4 Threats to validity

The validity of this study depends on the correctness and completeness of multiple aspects and these are discussed in this section.

Number of projects For the correlation analysis, I use a dataset given by my supervisor, which has in total entries for 481 OSS projects, from which only 230 are effectively used in this study because they have valid benchmark results. From these 230 valid projects, 223 qualify to the final correlation analysis, resulting in an analysis done on 4330 individual benchmarks. Although there is not a minimal variable set size requirement for a correlation analysis done with Spearman's rank-order correlation, I believe that 4330 benchmarks from 223 projects is a large scale to reliably accept the correlation results.

Type of projects Another aspect is, that all the projects used in this study are open-source software, meaning that the results of this study can not be generalized for all kinds of software projects existing in the world. Also, the project pool has projects from different directions (libraries, databases, utilities etc.), which means that the results are not applicable for a specific kind of software project. Regarding the microbenchmarks, this study only takes benchmarks written in Go into consideration and shows the correlation of their features with their stability, which means that the results of this study cannot be generalized for all the benchmarks written in different programming languages. However, the approaches introduced in this thesis can be easily adapted for other programming languages as well to replicate the study on different projects.

Chosen features To find correlations between benchmark stability and different benchmark features, the features are chosen by means of observations, assumptions and hypotheses. This on one hand means that the features are not chosen with a standard procedure, which means there is a high chance that work of other researchers won't easily match with this work. On the other hand, it indicates that there may be potential source code features left out, which might have a strong correlation with the stability of benchmarks. However, this study does not claim that other source code features or libraries have no effect on the stability of benchmarks, and this study can be extended with new features or libraries of interest.

It should also be stated that this work is only based on the features that come up in the programming language Go and on features that can be extracted from count of the usage of Go standard libraries, which means that third party libraries are not particularly observed for the features. This poses a threat to the validity of correlations between hardware-network related aspects and stability of a benchmark because if a project strongly relies on a third party library to implement its I/O or networking capabilities, this is not tracked by Prophunt. Since Go has strict rules and has simplicity as part of its core values, my assumption is that there are not many projects among the 230 projects which rely on third parties for such capabilities.

Static Analysis The extraction of source code features of benchmarks is implemented by an AST traversal of the .go files found in the projects. This means that the extraction is purely static and there is no evidence whether the nodes that come up in the AST are visited in the runtime of the benchmarks in reality. This poses a threat to the validity in the way that the extracted features of a benchmark are not as accurate as when they would be extracted by doing a dynamic analysis and tracking the execution traces of benchmarks. However, performing the correlation analysis based on the source code features that are extracted with a dynamic callgraph analysis would be expensive in terms of time and performance. Moreover, since the analyses are performed on my laptop and not on a high performance computing cloud, there are limitations of hardware under the system to overcome such expenses.

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 Section 4.1, I present statistical information regarding the distribution of benchmarks' variabilities. From the given dataset of 4802 total benchmarks, 4589 valid benchmarks in total qualify to final correlation analysis. Only 1.83% of all benchmarks have a RCIW99 value above 10%, showing that most of the benchmarks are very stable. Second results are about the outcome of Prophunt and Callgraph Analyzer and are presented in Section 4.2. From successfully downloaded 223 projects, Prophunt parses 4926 benchmarks and their features, of which 4500 are matched with the ones from variability analysis. As a next step, Callgraph Analyzer takes Prophunt_Output as input and returns 4837 benchmarks with cumulative source code features. Matching the output of Prophunt and Callgraph Analyzer shows that there are 121 benchmarks which did not have cumulative feature values, which are therefore also removed. Finally, last results are presented in Section 4.3, which are based on 4330 matched benchmarks, which are found both in the results of variability and callgraph analysis. Of all correlation coefficients acquired from Spearman's rank-order correlation, "sync" has the highest correlation with benchmark stability with a value of 0.35690, followed by "gos" with 0.28593, both being strongly significant. Figure 4.1 sums up the removal of benchmarks and shows the number of benchmarks remaining to the correlation analysis.

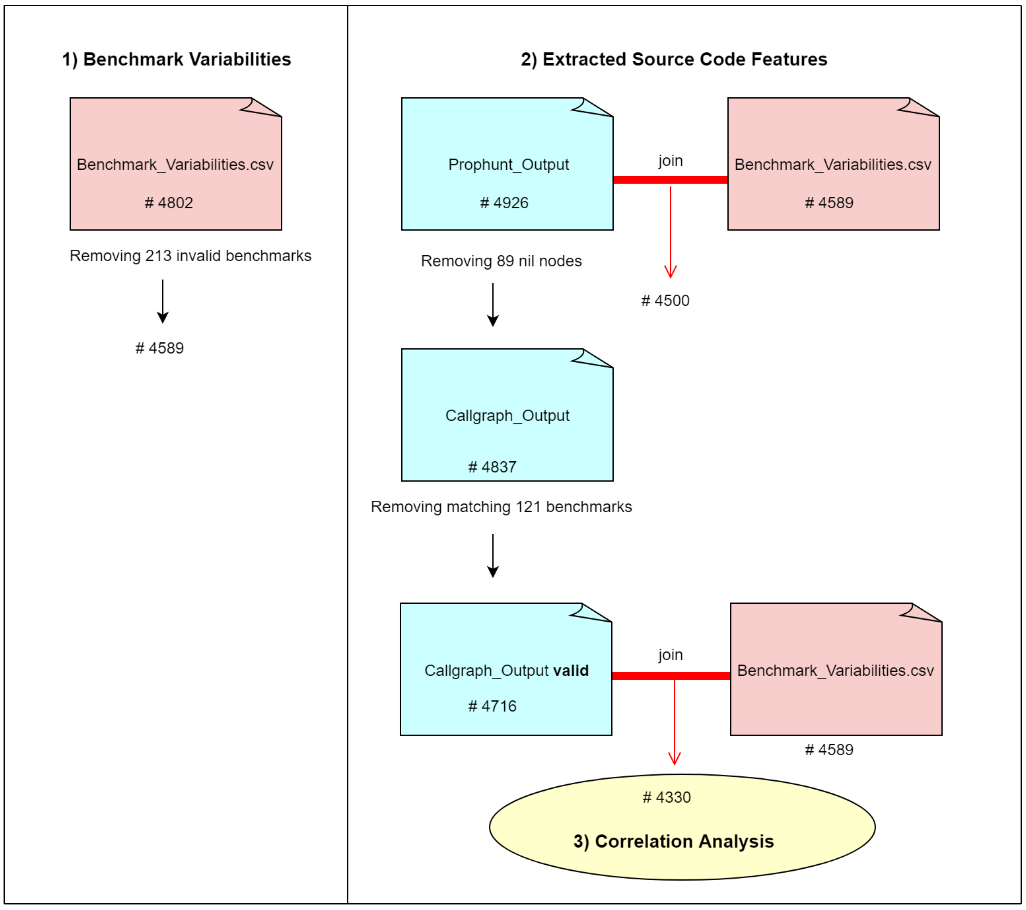

1) Benchmark Variabilities
Benchmark_Variabilities.csv
# 4802
Removing 213 invalid benchmarks
# 4589
2) Extracted Source Code Features
Prophunt_Output
# 4926
join
Benchmark_Variabilities.csv
# 4589
Removing 89 nil nodes
# 4500
Callgraph_Output
# 4837
Removing matching 121 benchmarks
Callgraph_Output valid
# 4716
join
Benchmark_Variabilities.csv
# 4589
# 4330
3) Correlation Analysis

Figure 4.1: General view of processing results.

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 **Benchmark_Variabilities.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 are all the benchmarks of **mesos/mesos-go** [41] and **go-gl/mathgl** [42], as well as benchmarks *BenchmarkProtocolV2Sub128k* and *BenchmarkCallsConcurrentServer* of projects **nsqio/nsq** [43] and **uber/tchannel-go** [44], respectively. Furthermore, there are in total 9 benchmarks, which, although having more than 1 execution, 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 **Benchmark_Variabilities.csv** file, having left with 4589 benchmarks in total. The count of removed benchmarks and their belonging projects can be seen in Table 4.1.

Table 4.1: Removal of invalid benchmarks from Benchmark_Variabilities.csv.

Project Name	Benchmark Count	Reason of Removal
mesos/mesos-go	178	Benchmarks have only 1 iteration.
go-gl/mathgl	24	Benchmarks have only 1 iteration.
nsqio/nsq	1	Benchmark has only 1 iteration.
uber/tchannel-go	1	Benchmark has only 1 iteration.
prometheus/common	1	All counters are 0 ns.
codegangsta/martini-contrib	2	All counters are 0 ns.
DNAProject/DNA	3	All counters are 0 ns.
TuftsBCB/io	1	All counters are 0 ns.
cgrates/cgrates	2	All counters are 0 ns.

For a general view of data, I create histograms of benchmarks' variabilities using Matplotlib library, which is a library for Python used often for visualizing data. For looking at the results on a 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 on benchmark level

Figure 4.2 shows the histogram across all benchmarks found in **Benchmark_Variabilities** 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 percent. 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.

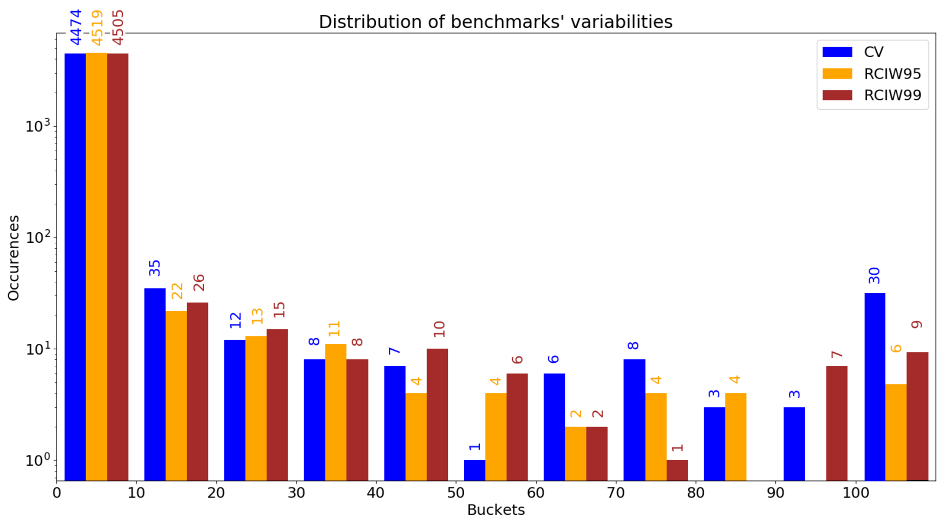


Figure 4.2: Distribution of benchmarks' variabilities 0-100 in 10% buckets in log-scaled y-axis.

This result requires further analysis of 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 4.3 exposes the variabilities in percentage. 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 %.

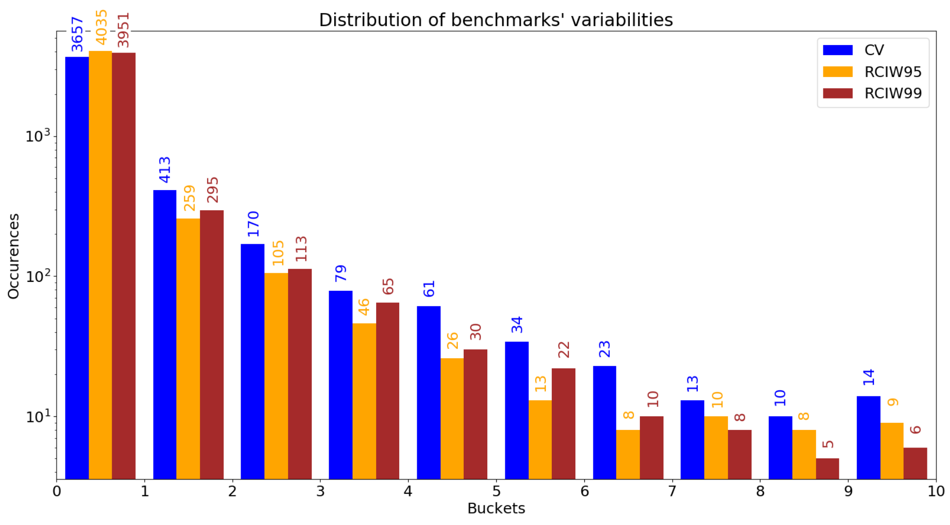


Figure 4.3: Distribution of benchmarks' variabilities 0-10 in 1% buckets in log-scaled y-axis.

4.1.2 Variabilities on project level

Since I eliminate 2 projects because they have 1 execution for each of their benchmarks, following statistical data is based on a total of 228 projects. In the rest of the analysis, I focus on RCIW99 values of variability, because CV is not a valid measure for performance data and RCIW99 is more significant than RCIW95. Table 4.3'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 value between 0-1%. Analyzing the 0-1% bucket further shows that there are in total 283 benchmarks from 68 projects which have a RCIW99 value of 0.0%. This makes 6.16% of total benchmarks. A RCIW99 value of 0.0% means that there is no variation in the results of the benchmark, which can be confirmed by looking at the iteration values of these benchmarks. The number of iterations for these benchmarks varies from 4 to 1519 with an average of 97 iterations and a standard deviation of 167, and the average execution time in ns varies from 0.48 ns to 2930.0 ns, having an average of 140 ns and a standard deviation of 332 ns. Generally, the benchmarks that have 0.0% of RCIW99 are short running benchmarks.

Table 4.3: 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%	9-10%
#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%**	**>100%**
#projects	18	9	6	4	4	2	1	0	3	6
#projects %	8%	4%	3%	2%	2%	1%	0%	0%	1%	3%

Table 4.4 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 column 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** [45], only has 2 benchmarks, of which both have 0.0 as RCIW99 score.

Table 4.4: 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%

A notable finding is that starting with 60% of RCIW99 in Table 4.4, the number of projects, whose benchmarks have variabilities more than the given percentage, do not decrease. Further inspection shows that these 6 projects are always the same 6 projects from 60% to over 100%. Table 4.5 depicts these projects and their benchmarks with their RCIW99 values.

Table 4.5: Unstable benchmarks.

Project Name	Benchmark	RCIW99
tendermint/go-merkle	/benchmarks/bench_test.go/BenchmarkLevelDBBatchSizes	101.91
tendermint/go-merkle	/benchmarks/bench_test.go/BenchmarkMedium	97.91
tendermint/go-merkle	/benchmarks/bench_test.go/BenchmarkRandomBytes	99.08
tendermint/go-merkle	/benchmarks/bench_test.go/BenchmarkSmall	96.86
tendermint/merkleeyes	/benchmarks/bench_test.go/BenchmarkLevelDBBatchSizes	98.60
tendermint/merkleeyes	/benchmarks/bench_test.go/BenchmarkMedium	93.97
tendermint/merkleeyes	/benchmarks/bench_test.go/BenchmarkMemKeySizes	63.07
tendermint/merkleeyes	/benchmarks/bench_test.go/BenchmarkRandomBytes	111.32
tendermint/merkleeyes	/benchmarks/bench_test.go/BenchmarkSmall	96.49
xtaci/kcp-go	sess_test.go/BenchmarkSinkSpeed1M	79.15
xtaci/kcp-go	sess_test.go/BenchmarkSinkSpeed256K	170.20
xtaci/kcp-go	sess_test.go/BenchmarkSinkSpeed4K	90.77
xtaci/kcp-go	sess_test.go/BenchmarkSinkSpeed64K	133.51
uber/tchannel-go	relay_benchmark_test.go/BenchmarkRelayNoLatencies	196.26
micro/go-micro	/broker/http_broker_test.go/BenchmarkPub128	140.02
micro/go-micro	/transport/http/http_test.go/BenchmarkTransport1	121.92
segmentio/objconv	/cbor/cbor_test.go/BenchmarkCodec	204.07
segmentio/objconv	/json/json_test.go/BenchmarkCodec	188.40
segmentio/objconv	/objutil/int_test.go/BenchmarkParseUintHex	64.18

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?**

Similar to my hypothesis, benchmarks generally have a low variation. 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. On the other side, 19 benchmarks from 6 projects manifest a RCIW99 above 60%, which makes only 0.4% of all the benchmarks. The possible reasons for the stability of benchmarks, as well as why there is such a distribution is furthermore discussed in Section 5.

4.2 Extracted source code features

In this section, I explain how many projects are downloaded, how many functions are parsed, as well as how many benchmarks result from the final callgraph analysis.

4.2.1 Downloaded Projects

Using all the 228 projects and their commits from the first results, my Python downloader script is able to download 223 of the projects successfully. From the remaining 5 projects, 3 (**tendermint/go-merkle**, **pp2p/paranoid**, **eleme/banshee**) are not found, either because they terminated the project on Github, or because they moved the project to another repository within or out of Github. 1 project (**stratumn/sdk** [46]) changed its repo and its redirection from Github results in a non-Go-project. 1 project (**eaburns/T**) changed the repo to a new name (**eaburns/T_old**) [47], however, it's commit from the initial dataset does not match any commits in the new repo.

As described in Section 3.2.3 and 3.2.4, parsing with Prophunt results with Prophunt_Output and using the Callgraph Analyzer results with Callgraph_Output. Prior to running both of the tools, I set up a Virtual Machine for Ubuntu 18.04 in my Windows 10 installation, because some projects have dependencies to some standard library packages, which are not included in a Windows installation of Go, such as "syscall". Not having such libraries causes compilation errors for some projects, hence, I run both of the tools in Ubuntu. This ensures a smoother experience when parsing and extracting features that rely on the standard library packages.

4.2.2 Prophunt Output

Prophunt returns in total 223 CSV files with a size of 35.2 MB. In total, Prophunt parses 163474 functions across all 223 projects. This makes an average of 733 functions per project and a standard deviation of 2945, and Figure 4.4 illustrates the number of parsed functions per project. From this many functions, there are 4926 parsed benchmarks, which on average equals to 22 benchmarks per project with a standard deviation of 28. The frequency of parsed benchmarks per project is illustrated in Figure 4.5. This is more than the total number of benchmarks found in the Benchmark_Variabilities.csv (4589) and means that the parser was able to find 337 more benchmarks than in the given dataset. However, when matching them with the benchmarks from Benchmark_Variabilities.csv, there are in total 4500 matching benchmarks. That is explainable with the missing benchmarks from unsuccessful downloads and some compilation problems in 2 projects albeit all dependency fetching efforts. In Table 4.7 is the projects of benchmarks (89) that are present in Benchmark_Variabilities.csv, yet could not be parsed with Prophunt (See in A.1):

Table 4.7: Information about projects of not matching benchmarks

Project Name	Benchmark Count	Reason of not match
tendermint/go-merkle	6	Not found
pp2p/paranoid	14	Not found
eleme/banshee	19	Not found
micro/go-micro	10	Not compiling
coredns/coredns	1	Not compiling
stratumn/sdk	2	Changed repo
eaburns/T	37	Changed repo

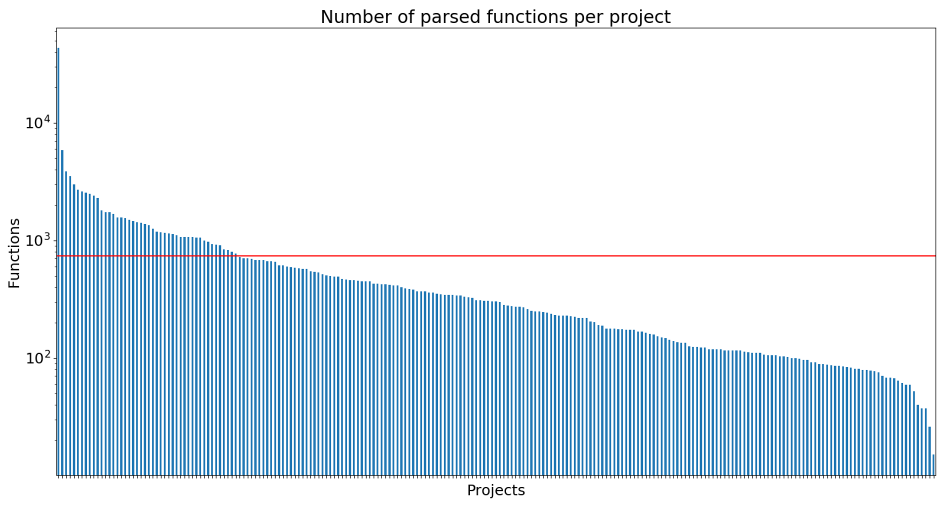


Figure 4.4: Number of parsed functions per project in log-scaled y-axis. Red line indicates the average function count.

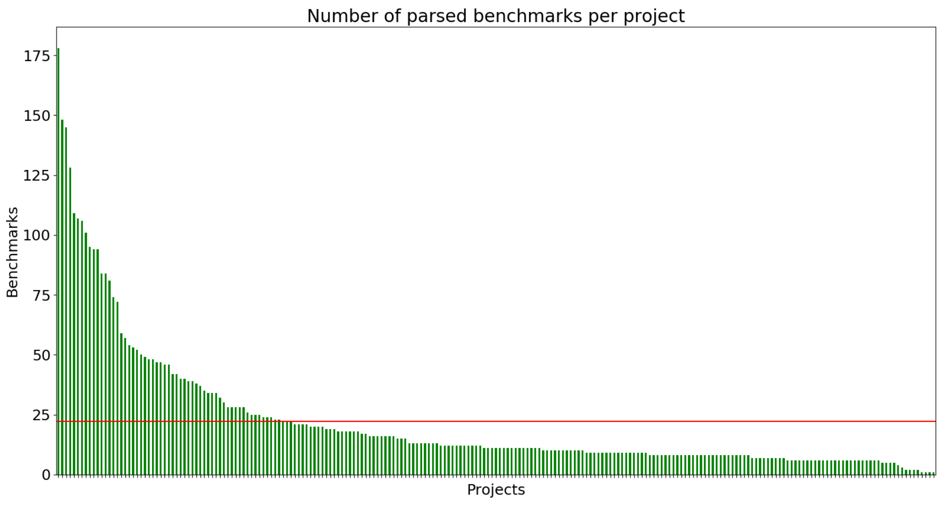


Figure 4.5: Number of parsed benchmarks per project in log-scaled y-axis. Red line indicates the average benchmark count.

4.2.3 Callgraph Analyzer Output

Callgraph Analyzer takes as input the CSV files that are found in Prophunt_Output. I run Callgraph Analyzer and collect 223 CSV files again. The valid 223 CSV files size up to 1.38 MB, with a total of 4837 benchmarks. This shows that 89 of the benchmarks in Prophunt_Output were not analyzed via Callgraph Analyzer, because they had a nil node in the ***simple.DirectedGraph** instance of the project (See in A.1). In the next step, I compare the benchmarks from Prophunt_Output with the ones from Callgraph_Output to see whether any benchmark has exactly same feature values in both of the versions, and I find 121 exact same benchmarks (See in A.1). This indicates that Callgraph Analyzer could not find any reachable node from these benchmarks, hence, it took the existing version of benchmark entry from Prophunt_Output to Callgraph_Output. For the sake of proper analysis, I eliminate these 121 benchmarks from Callgraph_Output, having left with 4716 valid benchmarks and their features.

Finally, I match the benchmarks from Benchmark_Variabilities.csv with the valid ones from Callgraph_Output and see that there are 4330 matching benchmarks. This means that in total there are 259 not matching benchmarks of which 89 were not present in Prophunt_Output in first place. Rest of 170 benchmarks do not match due to *simple.DirectedGraph not giving any reachable node, or callgraph tool cannot find the benchmark in the folders.

4.3 Correlation between variabilities and source code features

Section 4.1 and 4.2 are preliminary steps to build the final dataset of benchmarks, which includes the variability metrics RCIW99, RCIW95, CV, as well as the 59 source code feature values for each of the 4330 benchmarks. As one can see in Figure 4.2, the distribution of variabilities is not normalized, hence, for the correlation analysis, I choose Spearman's rank-order correlation coefficient, because unlike other correlation coefficients such as Pearson, Spearman's correlation does not assume that the distribution of variables is normalized. Additionally, because "parameters" has 1 and "net/http/httptrace" has 0 as value across all benchmarks, there is no correlation found between them and the stability, thus, they are removed from the analysis. For a better focus on different types of features, I present the results in Subsections 4.3.1 and 4.3.2.

4.3.1 Language related features

There are in total 27 language related features including "cyclomaticcomplexity" [32] that are the part of this analysis. Table 4.9 shows the correlation values of these features with the according variability metric (a concatenating * means a p-value lower than 0.05, and ** means a p-value lower than 0.01).

Table 4.9: Correlation between stability and language related features.

Feature Name	rciw99	rciw95	cv
pkgfiles	0.28000**	0.28310**	0.20296**
fileloc	0.25547**	0.26357**	0.17676**
namelength	0.10807**	0.11421**	0.05583**
returns	0.28535**	0.28778**	0.22448**
loc	0.24654**	0.24957**	0.19555**
funccalls	0.25208**	0.25542**	0.20048**
loops	0.23889**	0.24372**	0.18443**
nestedloops	0.18190**	0.18672**	0.13891**
channels	0.27211**	0.27868**	0.23050**
sends	0.22830**	0.23290**	0.18796**
receives	0.24083**	0.24562**	0.19546**
closes	0.26428**	0.27345**	0.21205**
gos	0.28593**	0.29192**	0.24832**
concrranges	0.04556**	0.04388**	0.03319*
selects	0.25025**	0.25777**	0.18467**
selectcases	0.25005**	0.25751**	0.18493**
variables	0.24846**	0.25292**	0.20061**
pointers	0.28430**	0.29096**	0.22130**
slices	0.17991**	0.18075**	0.13344**
maps	0.17304**	0.17262**	0.16990**
ifelses	0.25134**	0.25509**	0.20447**
switches	0.08179**	0.08671**	0.04099**
switchcases	0.07659**	0.08068**	0.04293**
panics	0.16285**	0.17181**	0.10483**
recovers	0.07039**	0.06250**	0.07748**
defers	0.27512**	0.27477**	0.24704**
cyclomaticcomplexity	0.25656**	0.26004**	0.20563**

Taking RCIW99 correlations into consideration, the most correlating 3 features are "gos", "returns" and "pointers", all with a correlation coefficient higher than 0.28 and with a very high significance. On the other side, "concrranges", "recovers" and "switchcases" are the least correlating features. 11 of these features have a correlation coefficient higher than 0.25. A positive correlation means that with the increase of this feature in the source code executed by the benchmark, the benchmark's variability also increases, hence, its stability decreases. In my hypothesis for RQ2, I expected to see a significant effect of loops and cyclomatic complexity on the benchmark stability. Evaluating these results, this expectation holds, however, "cyclomaticcomplexity" has a higher correlation coefficient (0.25656) than "loops" (0.23889). Still, among all language features, these do not make up the highest correlation values.

An interesting observation is that all concurrency related features ("channels", "sends", "receives", "closes", "gos", "concrranges", "selects", "selectcases") except for "concrranges" have a value above 0.22, which shows that these features have rather a negative effect on the stability of a benchmark. Similarly, signature related features such as "pkgfiles", "fileloc", "loc" and "returns" also reveal noteworthy correlations. The more files and lines of code found in the package of the benchmark and its reachable functions, the more variable benchmarks get. Same applies for the lines of code that are in the benchmark and its reachable functions. These correlations are explainable with the CPU having to process more instructions as the lines of code grows. Besides signature related properties, "pointers" and "defers" as body related features point to a notable correlation as well. Regarding pointers, dereferencing pointers and sharing data via pointers come with computational costs, and the more pointers come up in the source code features of a benchmark, the more data shared via pointers will be stored in the heap. Since heap data can only be freed by the garbage collector, the correlation may have to do with the performance of garbage collector, or the memory allocation/freeing mechanisms of the underlying OS. Using defer statement,s on the other hand, is also not free and a lookup in the disassembled code shows that function calls made with defer statement causes at least 2 extra function calls in assembly level. The more defer statements in the benchmark, the larger the list of defer statements gets, which I suspect might be causing the higher variability of the benchmark. Defer statement has got several performance improvements in the history of Go [48] (the overhead of deferred function calls reduced by the half), [49] (performance of most uses of defer improved by 30%) and since there are projects in the dataset, whose commits are going back to several years (Go 1.7 and earlier), it might be that the version of Go is responsible for the correlation with defers.

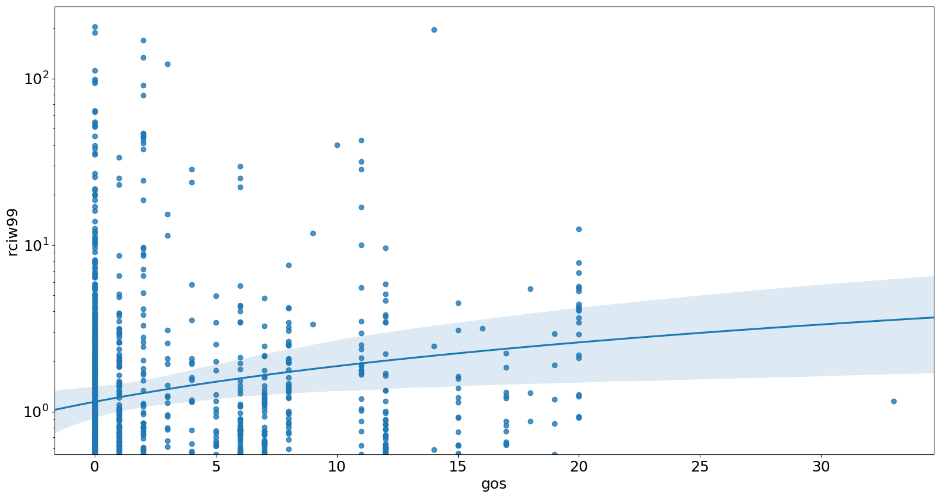


Figure 4.6: "Gos" and rciw99 values plotted with a linear regression model in log-scaled y-axis.

Figure 4.6 shows the linear relationship between "gos" and RCIW99 values across all 4330 benchmarks. The count of Go keyword in these benchmarks vary from 0 to 33, but most of the benchmarks have 0 to 5 "gos", as the figure indicates.

4.3.2 Standard library related features

The number of standard library related features used in the correlation analysis is 30 and Table 4.10 shows the correlation coefficients of these features and the variability metrics (a concatenating * means a p-value lower than 0.05, and ** means a p-value lower than 0.01).

Table 4.10: Correlation between stability and standard library related features.

Feature Name	rciw99	rciw95	cv
bufio	0.15467**	0.16217**	0.10351**
bytes	0.16678**	0.16779**	0.15004**
crypto	-0.02982*	-0.03062*	-0.02616
database/sql	0.01040	0.00881	-0.00608
encoding	0.12044**	0.11960**	0.07911**
encoding/binary	0.08718**	0.09103**	0.04150**
encoding/csv	-0.00911	-0.01151	0.01892
encoding/json	0.09332**	0.09174**	0.06628**
encoding/xml	0.05489**	0.05191**	0.05730**
io	0.20088**	0.20167**	0.15897**
io/ioutil	0.17902**	0.18179**	0.13742**
math	0.13606**	0.14590**	0.08024**
math/rand	0.26986**	0.27009**	0.22528**
mime	0.14102**	0.14973**	0.11820**
net	0.23078**	0.23471**	0.18983**
net/http	0.15577**	0.15516**	0.13530**
net/http/httptest	0.02879	0.02164	0.06075**
net/http/httputil	0.02795	0.02721	0.02797
net/rpc	0.01805	0.01221	0.02851
net/rpc/jsonrpc	0.01783	0.01700	0.01678
net/smtp	-0.01906	-0.01700	-0.00840
net/textproto	0.01988	0.01981	0.01227
os	0.16799**	0.16924**	0.16754**
os/exec	0.09034**	0.08932**	0.08138**
os/signal	0.01988	0.01981	0.01227
sort	0.11224**	0.11057**	0.09498**
strconv	0.08764**	0.08236**	0.08342**
sync	0.35690**	0.36464**	0.28974**
sync/atomic	0.25871**	0.27040**	0.18262**
syscall	0.06219**	0.06013**	0.06330**

Based on the table and RCIW99 correlations, the most correlating library usage with the stability is "sync" with 0.35690, which is also the most correlating feature across all analyzed features. It is followed by "math/rand" (0.26986) and "sync/atomic" (0.25871) respectively. Unlike with Table 4.9, in this table, there are only the 3 features that have a higher correlation than 0.25. Moreover, there are even negative correlations, although they are very close to 0. Also note that 9 of 30 features do not have any * concatenating, which means these correlation values are not significant.

Another expectation from my hypothesis for RQ2 was that file IO and https calls would have an impact on the stability of a benchmark due to the performance of the underlying hardware (file IO) or the performance of the network. To start with, file IO, by looking at "io" (0.20088) and "io/ioutil" (0.17902) as corresponding libraries, seems to have rather a small correlation with the RCIW99 variation, meaning that there is somewhat an impact to the stability. Http call, on the other hand, can be derived from the usage of "net/http" (0.15577), "net/http/httptest" (0.02879) and "net/http/httputil" (0.02795). Because the latter two are not significant, and because "net/http" has a value of 0.15577, it is rather to say that HTTP calls are weakly to not correlated with the benchmark stability, unlike expected in my hypothesis. In contrast to http related libraries, "net" has a higher correlation with significance, showing that network IO related things affect the stability more.

Regarding the correlation with sync and sync/atomic: These are the libraries that have mechanisms for syncing and scheduling processes in Go and these mechanisms operate in language level instead of OS level. Therefore, the correlation of these libraries with the instability of benchmark can arise from Go's implementations. Apart from sync libraries, the correlation of math-/rand library is also outstanding. This correlation may be related to the methods defined in this library requiring extra CPU cycles to generate pseudo-random numbers.

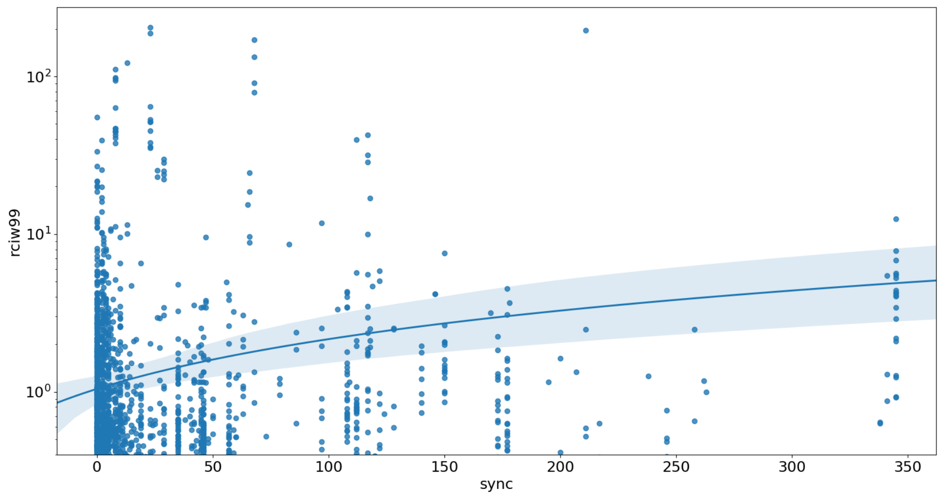


Figure 4.7: "sync" and rciw99 values plotted with a linear regression model in log-scaled y-axis.

Figure 4.7 illustrates the linear relationship between "sync" and RCIW99 values across all 4330 projects, which has the highest correlation coefficient across all features. The usage of sync library varies from 0 to 345 and most of the benchmarks have 0 to 50 usages of the sync library.

All in all, I want to repeat my second research question and answer it with the analysis done in this section:

- **RQ2: Which source code features contribute to the stability of benchmark results and how?**

Across all features that I analyze, 11 language related and 3 library related features correlate with the variability of a benchmark with a coefficient value higher than 0.25 and a strong significance. This indicates that these features contribute to the stability of a benchmark negatively since an increase in their numbers also increases the variability of the benchmark. Among all features, "sync" has the highest correlation with a coefficient value of 0.35690, followed by a language feature "gos", whose value is 0.28593.

A noticeable fact in these results is that the correlations are fairly low in general. Note that the extracted source code features of a benchmark are based on a static callgraph analysis, which means that the execution traces of benchmarks are not taken into account. A dynamic callgraph analysis would allow for tracking the execution traces of a benchmark and as a result, one would better inspect which nodes in the control flow graph of a benchmark are visited. Consequently, this would allow for more accurate extraction of source code features, which would be interesting to look at for future directions.

The results so far show the correlations based on the cumulative source code features of a benchmark, which means feature values consist of the sum of benchmark's and its reachable functions' feature values. For a comparison, I repeat the same analysis using only the features of the benchmarks without considering the reachable functions from the benchmarks. The results are depicted in Table 4.11 (a concatenating * means a p-value lower than 0.05, and ** means a p-value lower than 0.01).

Table 4.11: Correlations analyzed using source code features from Prophunt Output.

Feature Name	rciw99	rciw95	cv
pkgfiles	0.11012**	0.11197**	0.05742**
fileloc	0.04035**	0.05191**	-0.04041**
namelength	0.10297**	0.10871**	0.05374**
returns	0.00000	0.00000	0.00000
loc	-0.08238**	-0.08828**	-0.02195
funccalls	-0.02776	-0.03134*	0.00863
loops	-0.12540**	-0.12974**	-0.07682**
nestedloops	-0.01958	-0.01780	-0.00504
channels	-0.00625	-0.00983	0.02506
sends	-0.03424*	-0.03679*	-0.00908
receives	-0.03067*	-0.03367*	0.00146
closes	0.00816	0.00464	0.04448**
gos	0.03725*	0.03384*	0.05859**
concrranges	0.00000	0.00000	0.00000
selects	-0.00971	-0.01043	0.01115
selectcases	-0.00974	-0.01046	0.01112
variables	-0.16944**	-0.17341**	-0.11699**
pointers	-0.00616	-0.00997	0.01252
slices	-0.09739**	-0.09746**	-0.08867**
maps	-0.04350**	-0.04723**	-0.01780
ifelses	-0.07513**	-0.07924**	-0.01399
switches	0.00126	0.00103	-0.00010
switchcases	0.00126	0.00103	-0.00010
panics	0.01972	0.01730	0.02056
recovers	0.00000	0.00000	0.00000
defers	0.06999**	0.06274**	0.11278**
cyclomaticcomplexity	-0.13129**	-0.13613**	-0.06775**
bufio	-0.00821	-0.00811	0.00693
bytes	-0.01819	-0.01675	0.00143
crypto	-0.08598**	-0.08755**	-0.09682**
database/sql	-0.00142	-0.00163	-0.00613
encoding	-0.04291**	-0.04359**	-0.03434*
encoding/binary	-0.00500	-0.00703	0.00065
encoding/csv	0.00000	0.00000	0.00000
encoding/json	-0.04770**	-0.04717**	-0.04095**
encoding/xml	0.00000	0.00000	0.00000
io	-0.05632**	-0.05885**	-0.02208
io/ioutil	-0.04248**	-0.04603**	-0.01104
math	-0.01591	-0.01448	-0.02878
math/rand	0.00324	0.00145	-0.00236
mime	0.00000	0.00000	0.00000
net	0.02582	0.02055	0.06135**
net/http	0.03032*	0.02613	0.06388**
net/http/httptest	0.00654	0.00285	0.03144*
net/http/httputil	0.00000	0.00000	0.00000
net/rpc	0.02525	0.02145	0.02969*
net/rpc/jsonrpc	0.01810	0.01735	0.01682
net/smtp	0.00000	0.00000	0.00000
net/textproto	0.00000	0.00000	0.00000
os	0.00509	0.00043	0.04499**
os/exec	-0.01068	-0.01123	-0.00894
os/signal	0.00000	0.00000	0.00000
sort	0.01083	0.00743	0.02556
strconv	0.03328*	0.03448*	0.03859**
sync	0.02661	0.02726	0.02867
sync/atomic	0.04551**	0.04709**	0.04148**
syscall	0.00000	0.00000	0.00000

As one can see, the coefficients of the correlation analysis based on only source code features of the benchmark itself are very low in contrast to the previous correlation analysis. Moreover, 11 of the features do not even correlate, having 0.00 as the coefficient. These results show that even the correlations with cumulative source code features are not very high, the callgraph analysis of benchmarks is still needed, since there is a significant difference in the results of the analyses.

Chapter 5

Discussion

5.1 Benchmark Stability

The results in Section 4.1 shows that 98.17% of the benchmarks' RCIW99 values fall into the 0-10% bucket, and further investigation shows that 87.70% of these fall into 0-1% bucket. While these percentages indicate that most of the benchmarks are stable, it is not clear why there is such a distribution. Just by analyzing given dataset and without looking at outer factors, we can say that benchmarks written in Go are likely to be stable according to the variability analysis made in this thesis. Although there are slight differences between distributions of benchmarks' variabilities across different metrics, the distributions of benchmarks' variabilities are close to each other.

The environment where the benchmarks are executed is one of the factors that play a role in the stability of benchmarks. Considering that the benchmarks are executed on a different, but less stable environment, it is expected that the stability of benchmarks would also be different. For instance, the performance of clouds is extensively discussed in the literature and I assume that the stability of benchmarks would generally be lower in public clouds. Particularly, IO related operations are shown to be negatively affected by noisy neighbors [17]. Consequently, IO related features may correlate stronger with the stability when the benchmarks are executed on such environments. Similarly, in an environment where the network performance is not stable, I would expect HTTP related features to correlate stronger.

5.2 Chosen features

To find the correlation between benchmark stability and source code features, the chosen features cover the fields which are of interest to answer the research questions. For the "which source code features" part of the second research question, a more detailed look at language related features could result in other interesting correlations. For instance, I extract "nestedloops" feature by counting loops (either for or range loops in Go) which have at least one another loop inside its body. While this does not show a very high correlation with the benchmark variability (0.18190), maybe if I collected the information about how many levels every nested loop has (for example a triple nested for loop has 3 levels), this information might show a stronger correlation with the benchmark stability.

As for the library related features, there are over 150 standard libraries listed in Go's package documentation [31] and in this thesis, I look at 31 individual libraries and their usages across 223 projects. Note that I choose these 31 libraries because I assume that their usage may have a correlation with the variability of the benchmarks, however, there is no proof that the rest of the standard libraries do not have a correlation with the stability of benchmarks. Prior to performing the correlation analysis, I detect that "net/http/httptrace" has a value of 0 across all benchmarks, which is interesting because this means that from all 223 projects, no project has a benchmark that tests tracing of a http event for performance with this library. In general, 16 of 30 libraries that are part of this analysis have a correlation coefficient below 0.10, which shows that less than the half of libraries analyzed in this thesis are correlated with the stability at all. It is therefore important to extend this study by increasing the number of libraries to find out whether there are other library usages which are affecting the stability and if yes, to what extent.

In this thesis, I focused on features that can be directly parsed from the source code of the benchmark and its reachable functions. Yet, these features are not the only kinds of features that one can analyze. Features which are not directly parsable from the source code could be analyzed using specific tools on top of the source code, or other tools could be employed in the process of extracting source code which do dynamic analysis. This way, features could be extracted which cannot be extracted statically.

5.3 Correlation results

From the signature based features, most correlating feature with the benchmark stability is "pkg-files" with 0.28000, which represents the number of files in the same package as the benchmark. Note that this value is cumulated with the usage of Callgraph Analyzer, which means that for a benchmark, count of all files is taken which is found in the same package of all called functions. If we were to take only package files of the benchmark, the same analysis gives a value of 0.11012, which shows that cumulative value has a higher correlation. Similarly, "fileloc" not only considers lines of code of the file where benchmark is located, but the cumulative value of all functions called from benchmark as well. The same analysis only with the benchmark's own value results with 0.04035, which is a much lower value than 0.25547. Unfortunately, the analysis is not made with the cumulative version of "parameters", which is why that feature is out of the results. However, cumulative version of "parameters" could be interesting to look at, since counted parameters across all called functions of a benchmark might impact the stability of a benchmark.

From the body related features, concurrency related features correlate with the stability, "gos" being the feature with the highest correlation value. This means that concurrent programming affects the variability of a benchmark and this could be due to the performance of the CPU and memory, on which the benchmarks are run, or maybe due to the uncertain nature of thread scheduling. Since this dataset has benchmark iterations from only one source, it might be interesting to repeat this study with different benchmark results from different hardware. This way, one could compare the correlations of different results to see whether CPU and memory also change the way concurrency related features affect the benchmark stability. Also interesting to look at are "pointers" and "defers" as they have the second and third highest correlation value in body related features. Go does not offer pointer arithmetic, however, the usage of pointers still seems to affect the stability of benchmarks. This might have to do with the performance of OS's memory management, or the performance of the garbage collector which is responsible for reducing the heap overhead caused by the pointers. Defers are statements in Go which are executed after the surrounding function returns. This statement has seen a performance improvement in Go versions 1.8 [48] and 1.13 [49], reducing the time per operation. Since the benchmarks might be executed with a Go version going back to 1.7, the responsible for defer statements correlation could be the handling of Go's compiler on defer statements. For a comparison, the benchmarks could be executed with different installations of Go and their respective variabilities could be used as a base for the correlation analysis.

It is noticeable that "cyclomaticcomplexity" has a score similar to those of "funccalls", "loops" and "ifelses", which are the features that play a role on the calculation of "cyclomaticcomplexity". While this is an expected result, it is very exceptional that neither "switches" nor "switchcases" have a correlation value over 0.10. Switches and switch cases are also features that play a role on the cyclomatic complexity, however, they don't seem to affect the stability when their cumulative value is analyzed.

It should be noted that there is an impact of the callgraph analysis on the results of the correlation analysis. Static callgraph analysis, unlike a dynamic callgraph analysis, suffers from finding the actual visited nodes in the control flow graph of a benchmark in the runtime. As a result of that, number of function calls and reachable functions within a benchmark are overly approximated. Because of this, the cumulative feature values may not be as accurate as when the features would be extracted with a dynamic callgraph analysis. Performing the correlation analysis based on such a dynamic callgraph analysis would enable the researchers to find more accurate correlation results.

5.4 Future Work

This study reveals correlation results between different features and benchmark stability, however, the results should not be accepted as to directly affect the stability of benchmarks because correlation does not imply causation. It is for some features expected that with the increase in their values the benchmark variability will also be affected, but this assumption is based on the study with 223 projects and 4330 benchmarks.

To answer the hypotheses, I take several libraries as correspondence to source code features, such as "io" and "io/ioutil" as a representer of file IO. Although Prophunt can extract the usage of these libraries, we don't particularly know which functions of these libraries are in fact used in the benchmarks or in the called functions from these benchmarks. Extending this study by looking up for usages of libraries with functional level would show more detailed results in this correlation analysis, for example, whether *Write* method of **io** package is more correlating to benchmark stability than *Read* method of the same package.

The results of this study can also be used to predict the stability of a benchmark by using machine learning models. Examples for that include adopting a linear regression model to predict the stability using selected source code features, or reformulation of the variability to a classification problem. A first step into that field would be by applying a feature reduction to the Callgraph_Output. This way, features which have similar behaviors on the change of RCIW99 value could be grouped together, and then some of the features could be chosen to see whether a learning effect from these features can be generated.

This study shows how statically analyzing code blocks could be used for assessing source code features that play a role in microbenchmark variability. In the current state of our industry, static analysis of source code is mostly used in improving code readability (such as in the case of linters) or helping in restructuring code (such as in the case of automatic refactoring and renaming). I believe that software developers and software engineering tool designers may also utilize static analysis for improving program performance. For example, they can include the findings from this study to create extensions for Integrated Development Environments (IDEs), which show which parts of the code may require revising for performance enhancement by searching for the existence of source code features which are reported to affect the stability the most. Furthermore, employing a predictive methodology might help them identify the cause of slowdowns in a newer version of the software more easily.

Chapter 6

Conclusion

In this thesis, I analyzed the correlation between stability and source code features of software microbenchmarks written in Go.

To achieve this, I first analyzed a given dataset of benchmarks' execution results to find out about their variability, which is an indicator for the stability. As metrics of variation, I used coefficient of variation (CV) and relative confidence interval width (RCIW) with 99 and 95 percent confidence levels. Results show that 98.17% of 4589 benchmarks have a RCIW99 value between 1-10%. Furthermore, 87.70% of the benchmarks in this range have a RCIW99 value between 0-1%. This shows that most of the benchmarks are very stable, and only 1.83% of all benchmarks have a higher RCIW99 value than 10%. The variabilities based on CV and RCIW95 fall similar to those of RCIW99.

Secondly, I wrote a parser tool called "Prophunt" to extract language and library related source code features from Go projects and ran this tool on 223 projects, of which the benchmarks in variability analysis are from. Prophunt parsed 163474 functions and 4926 benchmarks. Next, I wrote another tool called "Callgraph Analyzer" to analyze the callgraphs of benchmarks and ran this on the output of Prophunt. This tool found all called functions from a benchmark and summed the feature values of all called functions, resulting into cumulative values of benchmark. As a result, Callgraph Analyzer successfully collected cumulative source code features of 4716 benchmarks, of which 4330 matched with the benchmarks from variability analysis.

Finally, I performed a correlation analysis based on the 59 source code features and variability of 4330 benchmarks to see which source code features have a correlation with the stability of the benchmark. For this, I used Spearman's rank-order correlation and looked at the coefficients of correlations. Results show that, among all features, the usage of sync package is the most correlating feature with a value of 0.35690, followed by a language feature "gos", which represents the count of go keyword, with a value of 0.28593. In total, 14 features have a coefficient value higher than 0.25 and 20 features have a coefficient value below 0.10. An interesting finding is that concurrency related features generally correlate with the stability and as expected, control flow elements are also correlated to the instability of benchmarks.

Researchers and practitioners can use the approaches presented in this thesis to analyze the stability of their benchmarks. Furthermore, the results of this study can be used as a basis for future directions on the topic of microbenchmark variability.

Appendix A

First Append

A.1 Not matching benchmarks

Following benchmarks were in final.csv, yet not in Prophunt_Output:

```
tendermint/go—merkle /benchmarks/bench_test.go/BenchmarkLevelDBBatchSizes
tendermint/go—merkle /benchmarks/bench_test.go/BenchmarkMedium
tendermint/go—merkle /benchmarks/bench_test.go/BenchmarkMemKeySizes
tendermint/go—merkle /benchmarks/bench_test.go/BenchmarkRandomBytes
tendermint/go—merkle /benchmarks/bench_test.go/BenchmarkSmall
tendermint/go—merkle iavl_test.go/BenchmarkImmutableAvlTreeMemDB
pp2p/paranoid /libpfs/commands/benchmark/commands_benchmark_test.go/BenchmarkAccess
pp2p/paranoid /libpfs/commands/benchmark/commands_benchmark_test.go/BenchmarkCreat
pp2p/paranoid /libpfs/commands/benchmark/commands_benchmark_test.go/BenchmarkLink
pp2p/paranoid /libpfs/commands/benchmark/commands_benchmark_test.go/BenchmarkMkDir
pp2p/paranoid /libpfs/commands/benchmark/commands_benchmark_test.go/BenchmarkRead
pp2p/paranoid /libpfs/commands/benchmark/commands_benchmark_test.go/BenchmarkReadDir
pp2p/paranoid /libpfs/commands/benchmark/commands_benchmark_test.go/BenchmarkReadLink
pp2p/paranoid /libpfs/commands/benchmark/commands_benchmark_test.go/BenchmarkRename
pp2p/paranoid /libpfs/commands/benchmark/commands_benchmark_test.go/BenchmarkRmDir
pp2p/paranoid /libpfs/commands/benchmark/commands_benchmark_test.go/BenchmarkStat
pp2p/paranoid /libpfs/commands/benchmark/commands_benchmark_test.go/BenchmarkSymLink
pp2p/paranoid /libpfs/commands/benchmark/commands_benchmark_test.go/BenchmarkTruncate
pp2p/paranoid /libpfs/commands/benchmark/commands_benchmark_test.go/BenchmarkUtimes
pp2p/paranoid /libpfs/commands/benchmark/commands_benchmark_test.go/BenchmarkWrite
micro/go—micro /broker/http_broker_test.go/BenchmarkPub1
micro/go—micro /broker/http_broker_test.go/BenchmarkPub128
micro/go—micro /broker/http_broker_test.go/BenchmarkPub32
micro/go—micro /broker/http_broker_test.go/BenchmarkPub64
micro/go—micro /broker/http_broker_test.go/BenchmarkPub8
micro/go—micro /broker/http_broker_test.go/BenchmarkSub1
micro/go—micro /broker/http_broker_test.go/BenchmarkSub128
micro/go—micro /broker/http_broker_test.go/BenchmarkSub32
micro/go—micro /broker/http_broker_test.go/BenchmarkSub64
micro/go—micro /broker/http_broker_test.go/BenchmarkSub8
eleme/banshee /filter/filter_test.go/BenchmarkRules1KNativeBest
eleme/banshee /filter/filter_test.go/BenchmarkRules1kBest
eleme/banshee /filter/filter_test.go/BenchmarkRules1kWorst
eleme/banshee /filter/filter_test.go/BenchmarkRules2kWorst
eleme/banshee /models/rule_test.go/BenchmarkRuleTest
eleme/banshee /models/rule_test.go/BenchmarkRuleTestWithDefaultThresholdMaxsNum4
eleme/banshee /models/rule_test.go/BenchmarkRuleTestWithDefaultThresholdMaxsNum8
eleme/banshee /storage/indexdb/db_test.go/BenchmarkGet10K
eleme/banshee /storage/indexdb/db_test.go/BenchmarkPut
eleme/banshee /storage/metricdb/db_test.go/BenchmarkGet100K
eleme/banshee /storage/metricdb/db_test.go/BenchmarkPut
eleme/banshee /storage/metricdb/db_test.go/BenchmarkPutX10
eleme/banshee /util/idpool/pool_test.go/BenchmarkAllocate
eleme/banshee /util/mathutil/mathutil_test.go/BenchmarkAverageNum605
eleme/banshee /util/mathutil/mathutil_test.go/BenchmarkStdDevNum605
eleme/banshee /util/trie/trie_test.go/BenchmarkPutAndGetPrefixedKeys
eleme/banshee /util/trie/trie_test.go/BenchmarkPutAndGetRandKeys
eleme/banshee /util/trie/trie_test.go/BenchmarkPutPrefixedKeys
eleme/banshee /util/trie/trie_test.go/BenchmarkPutRandKeys
stratumn/sdk /dummystore/benchmark_test.go/BenchmarkDummystore
stratumn/sdk /filestore/benchmark_test.go/BenchmarkFilestore
eaburns/T /edit/addr_bench_test.go/BenchmarkLinex1K
```

eaburns/T /edit/addr_bench_test.go/BenchmarkLinex1M
eaburns/T /edit/addr_bench_test.go/BenchmarkLinex32
eaburns/T /edit/addr_bench_test.go/BenchmarkLinex32M
eaburns/T /edit/addr_bench_test.go/BenchmarkRegexpEasy0x1K
eaburns/T /edit/addr_bench_test.go/BenchmarkRegexpEasy0x1M
eaburns/T /edit/addr_bench_test.go/BenchmarkRegexpEasy0x32
eaburns/T /edit/addr_bench_test.go/BenchmarkRegexpEasy0x32M
eaburns/T /edit/addr_bench_test.go/BenchmarkRegexpEasy1x1K
eaburns/T /edit/addr_bench_test.go/BenchmarkRegexpEasy1x1M
eaburns/T /edit/addr_bench_test.go/BenchmarkRegexpEasy1x32
eaburns/T /edit/addr_bench_test.go/BenchmarkRegexpEasy1x32M
eaburns/T /edit/addr_bench_test.go/BenchmarkRegexpHardx1K
eaburns/T /edit/addr_bench_test.go/BenchmarkRegexpHardx1M
eaburns/T /edit/addr_bench_test.go/BenchmarkRegexpHardx32
eaburns/T /edit/addr_bench_test.go/BenchmarkRegexpHardx32M
eaburns/T /edit/addr_bench_test.go/BenchmarkRegexpMediumx1K
eaburns/T /edit/addr_bench_test.go/BenchmarkRegexpMediumx1M
eaburns/T /edit/addr_bench_test.go/BenchmarkRegexpMediumx32
eaburns/T /edit/addr_bench_test.go/BenchmarkRegexpMediumx32M
eaburns/T /edit/addr_bench_test.go/BenchmarkRunex1K
eaburns/T /edit/addr_bench_test.go/BenchmarkRunex1M
eaburns/T /edit/addr_bench_test.go/BenchmarkRunex32
eaburns/T /edit/addr_bench_test.go/BenchmarkRunex32M
eaburns/T /edit/runes/bench_test.go/BenchmarkRead1
eaburns/T /edit/runes/bench_test.go/BenchmarkRead10k
eaburns/T /edit/runes/bench_test.go/BenchmarkRead1k
eaburns/T /edit/runes/bench_test.go/BenchmarkRead4k
eaburns/T /edit/runes/bench_test.go/BenchmarkRune10kRand
eaburns/T /edit/runes/bench_test.go/BenchmarkRune10kScan
eaburns/T /edit/runes/bench_test.go/BenchmarkRuneCacheRand
eaburns/T /edit/runes/bench_test.go/BenchmarkRuneCacheScan
eaburns/T /edit/runes/bench_test.go/BenchmarkWrite1
eaburns/T /edit/runes/bench_test.go/BenchmarkWrite10k
eaburns/T /edit/runes/bench_test.go/BenchmarkWrite1k
eaburns/T /edit/runes/bench_test.go/BenchmarkWrite4k
eaburns/T /editor/benchmark_test.go/BenchmarkDo
coredns/coredns /test/proxy_test.go/BenchmarkProxyLookup

Following benchmarks resulted as nil nodes in the **`*simple.DirectedGraph`**:

pilosa/pilosa /test/attr.go/BenchmarkAttrStore_Duplicate
pgpst/pgpst /internal/github.com/gin−gonic/gin/benchmarks_test.go/Benchmark404Many
pgpst/pgpst /internal/github.com/gin−gonic/gin/githubapi_test.go/BenchmarkGithub
pgpst/pgpst /internal/github.com/gin−gonic/gin/benchmarks_test.go/BenchmarkManyHandlers
pgpst/pgpst /internal/github.com/gin−gonic/gin/benchmarks_test.go/BenchmarkManyRoutesLast
pgpst/pgpst /internal/github.com/gin−gonic/gin/githubapi_test.go/BenchmarkParallelGithub
pgpst/pgpst /internal/github.com/gin−gonic/gin/benchmarks_test.go/BenchmarkOneRoute
pgpst/pgpst /internal/github.com/gin−gonic/gin/benchmarks_test.go/Benchmark5Params
pgpst/pgpst /internal/github.com/gin−gonic/gin/benchmarks_test.go/BenchmarkOneRouteSet
pgpst/pgpst /internal/github.com/gin−gonic/gin/benchmarks_test.go/Benchmark404
pgpst/pgpst /internal/github.com/gin−gonic/gin/githubapi_test.go/BenchmarkParallelGithubDefault
pgpst/pgpst /internal/github.com/gin−gonic/gin/benchmarks_test.go/BenchmarkOneRouteJSON
pgpst/pgpst /internal/github.com/gin−gonic/gin/benchmarks_test.go/BenchmarkRecoveryMiddleware
pgpst/pgpst /internal/github.com/gin−gonic/gin/benchmarks_test.go/BenchmarkOneRouteString
pgpst/pgpst /internal/github.com/gin−gonic/gin/benchmarks_test.go/BenchmarkOneRouteHTML
pgpst/pgpst /internal/github.com/gin−gonic/gin/benchmarks_test.go/BenchmarkManyRoutesFist
pgpst/pgpst /internal/github.com/gin−gonic/gin/benchmarks_test.go/BenchmarkLoggerMiddleware
CodisLabs/codis /pkg/proxy/request_test.go/BenchmarkRequestChan512
CodisLabs/codis /pkg/proxy/request_test.go/BenchmarkRequestGoChannel
CodisLabs/codis /pkg/proxy/request_test.go/BenchmarkRequestChan128
CodisLabs/codis /pkg/proxy/request_test.go/BenchmarkRequestChan256
CodisLabs/codis /pkg/proxy/request_test.go/BenchmarkRequestChan2048
CodisLabs/codis /pkg/proxy/request_test.go/BenchmarkRequestChan1024
nats−io/go−nats /test/bench_test.go/BenchmarkAsyncSubscriptionCreationSpeed
nats−io/go−nats /encoders/protobuf/protobuf_test.go/BenchmarkPublishProtobufStruct
nats−io/go−nats /test/netchan_test.go/BenchmarkPublishSpeedViaChan
nats−io/go−nats /test/bench_test.go/BenchmarkSyncSubscriptionCreationSpeed
nats−io/go−nats /encoders/builtin/json_test.go/BenchmarkPublishJsonStruct
nats−io/go−nats /test/bench_test.go/BenchmarkRequest
nats−io/go−nats /test/bench_test.go/BenchmarkInboxCreation
nats−io/go−nats /test/bench_test.go/BenchmarkPublishSpeed
nats−io/go−nats /test/bench_test.go/BenchmarkOldRequest
nats−io/go−nats /encoders/builtin/json_test.go/BenchmarkJsonMarshalStruct
nats−io/go−nats /test/bench_test.go/BenchmarkPubSubSpeed
nats−io/go−nats /encoders/builtin/gob_test.go/BenchmarkPublishGobStruct
nats−io/go−nats /encoders/protobuf/protobuf_test.go/BenchmarkProtobufMarshalStruct
Redundancy/go−sync /comparer/comparer_bench_test.go/BenchmarkWeakComparison
Redundancy/go−sync /comparer/comparer_bench_test.go/BenchmarkStrongComparison
Redundancy/go−sync gosync_test.go/BenchmarkIndexComparisons
Everlag/poeitemstore /dbTest/IndexQueryBench_test.go/BenchmarkMultiLeagueIndexQuerySlow
Everlag/poeitemstore /dbTest/IndexQueryBench_test.go/BenchmarkFiveIndexQuerySlow

Everlag/poeitemstore /dbTest/StashMetaBench_test.go/BenchmarkCompactFast
Everlag/poeitemstore /dbTest/IndexQueryBench_test.go/BenchmarkSingleIndexQueryFast
Everlag/poeitemstore /dbTest/StashMetaBench_test.go/BenchmarkAddStashesFast
Everlag/poeitemstore /dbTest/StashMetaBench_test.go/BenchmarkCompactAddStashesFast
Everlag/poeitemstore /dbTest/IndexQueryBench_test.go/BenchmarkSingleIndexQuerySlow
Everlag/poeitemstore /dbTest/IndexQueryBench_test.go/BenchmarkFiveIndexQueryFast
Everlag/poeitemstore /dbTest/IndexQueryBench_test.go/BenchmarkMultiLeagueIndexQueryFast
dustin/gomemcached /server/server_test.go/BenchmarkTransmitRes
dustin/gomemcached /server/server_test.go/BenchmarkTransmitResNull
dustin/gomemcached /server/server_test.go/BenchmarkTransmitResLarge
dustin/gomemcached /server/server_test.go/BenchmarkReceive
fabiolb/fabio /proxy/http_integration_test.go/BenchmarkProxyLogger
fabiolb/fabio /proxy/http_headers_test.go/BenchmarkUint16Base16
sni/lmd /lmd/benchmark_test.go/BenchmarkSingleFilter_1k_svc_10Peer
sni/lmd /lmd/benchmark_test.go/BenchmarkTacStats
sni/lmd /lmd/benchmark_test.go/BenchmarkQuery
sni/lmd /lmd/benchmark_test.go/BenchmarkServicelistLimit_1k_svc_10Peer
sni/lmd /lmd/benchmark_test.go/BenchmarkSimpleStats
sni/lmd /lmd/benchmark_test.go/BenchmarkSingleFilter_1k_svc__1Peer
sni/lmd /lmd/benchmark_test.go/BenchmarkTacStats_1k_svc_100Peer
sni/lmd /lmd/benchmark_test.go/BenchmarkTacStats_1k_svc_10Peer
sni/lmd /lmd/benchmark_test.go/BenchmarkMultiFilter
sni/lmd /lmd/benchmark_test.go/BenchmarkTacStats_5k_svc_500Peer
sni/lmd /lmd/benchmark_test.go/BenchmarkSingleFilter
sni/lmd /lmd/benchmark_test.go/BenchmarkTacStats_1k_svc__1Peer
sni/lmd /lmd/benchmark_test.go/BenchmarkServicelistLimit_1k_svc__1Peer
getlantern/zenodb math_bench_test.go/BenchmarkMathFloatBigEndian
getlantern/zenodb math_bench_test.go/BenchmarkMathUintLittleEndian
getlantern/zenodb math_bench_test.go/BenchmarkMathIntLittleEndian
getlantern/zenodb math_bench_test.go/BenchmarkMathFloatLittleEndian
getlantern/zenodb math_bench_test.go/BenchmarkMathIntBigEndian
getlantern/zenodb math_bench_test.go/BenchmarkMathUintBigEndian
tsuru/planb /reverseproxy/reverseproxy_test.go/BenchmarkServeHTTP_Fast
tsuru/planb /reverseproxy/reverseproxy_test.go/BenchmarkServeHTTPInvalidFrontends_Fast
tsuru/planb /reverseproxy/reverseproxy_test.go/BenchmarkServeHTTPInvalidFrontends_Native
tsuru/planb /reverseproxy/reverseproxy_test.go/BenchmarkServeHTTP_Native
rivine/rivine /types/block_bench_test.go/BenchmarkEncodeBlock
rivine/rivine /sync/threadgroup_test.go/BenchmarkThreadGroup
rivine/rivine /types/block_bench_test.go/BenchmarkDecodeEmptyBlock
rivine/rivine /types/validtransaction_bench_test.go/BenchmarkStandaloneValid
rivine/rivine /modules/consensus/consensusset_bench_test.go/BenchmarkCreateServerTester
rivine/rivine /sync/threadgroup_test.go/BenchmarkWaitGroup
Workiva/go−datastructures /btree/immutable/rt_test.go/BenchmarkBulkAdd
Workiva/go−datastructures /btree/immutable/rt_test.go/BenchmarkGetitems
coredns/coredns /middleware/file/lookup_test.go/BenchmarkFileLookup
coredns/coredns /middleware/file/dnssec_test.go/BenchmarkFileLookupDNSSEC
coredns/coredns /middleware/cache/cache_test.go/BenchmarkCacheResponse
coredns/coredns /middleware/file/file_test.go/BenchmarkFileParseInsert

Following benchmarks had no reachable callees in the **`*simple.DirectedGraph`**:

gobwas/glob glob_test.go/BenchmarkAlternativesCombineHardRegexpMatch
gobwas/glob glob_test.go/BenchmarkAlternativesSuffixFirstRegexpMismatch
gobwas/glob /match/match_test.go/BenchmarkRuneLenFromTable
gobwas/glob /util/runes/runes_test.go/BenchmarkLastIndexStrings
gobwas/glob /util/runes/runes_test.go/BenchmarkIndexStrings
gobwas/glob /util/runes/runes_test.go/BenchmarkNotEqualStrings
gobwas/glob glob_test.go/BenchmarkSuffixRegexpMatch
gobwas/glob glob_test.go/BenchmarkMultipleRegexpMatch
gobwas/glob glob_test.go/BenchmarkSuffixRegexpMismatch
gobwas/glob /match/match_test.go/BenchmarkRuneLenFromUTF8
gobwas/glob glob_test.go/BenchmarkPlainRegexpMismatch
gobwas/glob /util/runes/runes_test.go/BenchmarkIndexAnyStrings
gobwas/glob glob_test.go/BenchmarkPrefixRegexpMismatch
gobwas/glob glob_test.go/BenchmarkAllRegexpMatch
gobwas/glob glob_test.go/BenchmarkAllRegexpMismatch
gobwas/glob glob_test.go/BenchmarkPlainRegexpMatch
gobwas/glob /util/runes/runes_test.go/BenchmarkIndexRuneStrings
gobwas/glob /util/runes/runes_test.go/BenchmarkEqualStrings
gobwas/glob glob_test.go/BenchmarkMultipleRegexpMismatch
gobwas/glob glob_test.go/BenchmarkAlternativesCombineLiteRegexpMatch
gobwas/glob glob_test.go/BenchmarkPrefixSuffixRegexpMismatch
gobwas/glob glob_test.go/BenchmarkAlternativesSuffixSecondRegexpMatch
gobwas/glob glob_test.go/BenchmarkAlternativesSuffixFirstRegexpMatch
gobwas/glob glob_test.go/BenchmarkPrefixSuffixRegexpMatch
gobwas/glob glob_test.go/BenchmarkAlternativesRegexpMismatch
gobwas/glob glob_test.go/BenchmarkAlternativesRegexpMatch
gobwas/glob glob_test.go/BenchmarkPrefixRegexpMatch
gobwas/glob glob_test.go/BenchmarkParseRegexp
dustin/go−humanize ftoa_test.go/BenchmarkStrconvF
dustin/go−humanize ftoa_test.go/BenchmarkFmtF

dustin/go—humanize ftoa_test.go/BenchmarkFtoaRegexTrailing
siddontang/go /list2/list_bench_test.go/BenchmarkGoList
nsqio/nsq /nsqd/guid_test.go/BenchmarkGUIDCopy
nsqio/nsq /nsqd/guid_test.go/BenchmarkGUIDUnsafe
prataprc/goparsec /json/json_test.go/BenchmarkEncJSONNull
prataprc/goparsec /json/json_test.go/BenchmarkEncJSONBool
prataprc/goparsec /json/json_test.go/BenchmarkEncJSONInt
prataprc/goparsec /json/json_test.go/BenchmarkEncJSONMap
prataprc/goparsec /json/json_test.go/BenchmarkEncJSONMedium
prataprc/goparsec /json/json_test.go/BenchmarkEncJSONLarge
prataprc/goparsec /json/json_test.go/BenchmarkEncJSONArray
prataprc/goparsec /json/json_test.go/BenchmarkEncJSONFloat
prataprc/goparsec /json/json_test.go/BenchmarkEncJSONString
tinylib/synapse map_test.go/BenchmarkStdMapInsertDelete
Comcast/rulio /core/util_test.go/BenchmarkMutex
Comcast/rulio /core/util_test.go/BenchmarkLoop
Comcast/rulio /core/util_test.go/BenchmarkDeferWith
Comcast/rulio /core/util_test.go/BenchmarkDeferWithout
Comcast/rulio /core/util_test.go/BenchmarkRWMutex
wgliang/goreporter /linters/spellcheck/misspell/stringreplacer/replace_test.go/BenchmarkByteByteReplaces
dustin/go—jsonpointer bytes_test.go/BenchmarkReplacerTilde
dustin/go—jsonpointer bytes_test.go/BenchmarkReplacerSlash
json—iterator/go jsoniter_int_test.go/Benchmark_itoa
cosmos72/gomacro benchmark_test.go/BenchmarkArithCompiler1
cosmos72/gomacro /experiments/stmt_old_test.go/BenchmarkThreadedStmtFunc0
cosmos72/gomacro /experiments/stmt_new_test.go/BenchmarkThreadedFuncX6
cosmos72/gomacro /experiments/stmt_old_test.go/BenchmarkThreadedStmtFunc3
cosmos72/gomacro /experiments/stmt_new_test.go/BenchmarkThreadedStmtFunc6
cosmos72/gomacro /experiments/stmt_old_test.go/BenchmarkThreadedStmtFunc4Adaptive
cosmos72/gomacro /experiments/stmt_new_test.go/BenchmarkThreadedStmtStruct6Unroll
cosmos72/gomacro /experiments/stmt_new_test.go/BenchmarkThreadedStmtStruct6
cosmos72/gomacro /experiments/stmt_old_test.go/BenchmarkThreadedStmtFunc5
cosmos72/gomacro /experiments/stmt_new_test.go/BenchmarkThreadedStmtFunc6Adaptive
cosmos72/gomacro /experiments/stmt_old_test.go/BenchmarkThreadedStmtFunc4Terminate
cosmos72/gomacro /experiments/stmt_old_test.go/BenchmarkThreadedStmtFunc4Unroll
cosmos72/gomacro /experiments/stmt_old_test.go/BenchmarkThreadedStmtFunc4
cosmos72/gomacro /experiments/stmt_new_test.go/BenchmarkThreadedStmtFunc6Unroll
cosmos72/gomacro /experiments/stmt_new_test.go/BenchmarkThreadedStmtFuncX6
cosmos72/gomacro /experiments/stmt_old_test.go/BenchmarkThreadedStmtFunc2
cosmos72/gomacro /experiments/stmt_new_test.go/BenchmarkThreadedStmtFunc6Terminate
cosmos72/gomacro /experiments/stmt_old_test.go/BenchmarkThreadedStmtFunc1
cosmos72/gomacro /experiments/stmt_new_test.go/BenchmarkThreadedStmtStruct6Terminate
cosmos72/gomacro /experiments/stmt_new_test.go/BenchmarkThreadedStmtStruct6Adaptive
jmoiron/sqlx /reflectx/reflect_test.go/BenchmarkFieldNameL1
jmoiron/sqlx /reflectx/reflect_test.go/BenchmarkFieldPosL4
jmoiron/sqlx /reflectx/reflect_test.go/BenchmarkFieldPosL1
jmoiron/sqlx /reflectx/reflect_test.go/BenchmarkFieldNameL4
DataDog/datadog—go /statsd/statsd_benchmark_test.go/BenchmarkStatBuildGauge_Sprintf
DataDog/datadog—go /statsd/statsd_benchmark_test.go/BenchmarkStatBuildGauge_BytesAppend
DataDog/datadog—go /statsd/statsd_benchmark_test.go/BenchmarkStatBuildGauge_Concat
DataDog/datadog—go /statsd/statsd_benchmark_test.go/BenchmarkStatBuildCount_BytesAppend
DataDog/datadog—go /statsd/statsd_benchmark_test.go/BenchmarkStatBuildCount_Concat
DataDog/datadog—go /statsd/statsd_benchmark_test.go/BenchmarkStatBuildCount_Sprintf
hprose/hprose—golang /util/util_test.go/BenchmarkFormatInt
hprose/hprose—golang /util/util_test.go/BenchmarkFormatUint
hprose/hprose—golang /util/util_test.go/BenchmarkStrconvItoa
Redundancy/go—sync /index/index_bench_test.go/Benchmark_256Split_Map
tendermint/tendermint /benchmarks/map_test.go/BenchmarkSomething
digitalocean/captainslog parser_test.go/BenchmarkJSONCheckFirstChar
dearplain/fast—shadowsocks /shadowsocks/encrypt_test.go/BenchmarkRC4Init
google/netstack /sleep/sleep_test.go/BenchmarkGoWaitOnSingleSelect
google/netstack /sleep/sleep_test.go/BenchmarkGoAssertNonWaiting
google/netstack /sleep/sleep_test.go/BenchmarkGoWaitOnMultiSelect
google/netstack /sleep/sleep_test.go/BenchmarkGoSingleSelect
google/netstack /sleep/sleep_test.go/BenchmarkGoMultiSelect
DataDog/datadog—trace—agent /watchdog/info_test.go/BenchmarkReadMemStats
henrylee2cn/faygo /freecache/cache_test.go/BenchmarkMapGet
henrylee2cn/faygo /freecache/cache_test.go/BenchmarkMapSet
orcaman/concurrent—map concurrent_map_bench_test.go/BenchmarkStrconv
Workiva/go—datastructures /hashmap/fastinteger/hashmap_test.go/BenchmarkGoInsertWithExpand
Workiva/go—datastructures /queue/queue_test.go/BenchmarkChannel
Workiva/go—datastructures /hashmap/fastinteger/hashmap_test.go/BenchmarkGoDelete
rackspace/rack /internal/github.com/dustin/go—humanize/ftoa_test.go/BenchmarkStrconvF
OneOfOne/xxhash xxhash_test.go/BenchmarkCRC64ISOString
OneOfOne/xxhash xxhash_test.go/BenchmarkFnv64MultiWrites
OneOfOne/xxhash xxhash_test.go/BenchmarkCRC32IEEEShort
OneOfOne/xxhash xxhash_test.go/BenchmarkFnv64Short
OneOfOne/xxhash xxhash_test.go/BenchmarkFnv64
OneOfOne/xxhash xxhash_test.go/BenchmarkCRC32IEEEString
OneOfOne/xxhash xxhash_test.go/BenchmarkCRC64ISO
OneOfOne/xxhash xxhash_test.go/BenchmarkFnv32

OneOfOne/xxhash xxhash_test.go/BenchmarkAdler32
OneOfOne/xxhash xxhash_test.go/BenchmarkCRC32IEEE
OneOfOne/xxhash xxhash_test.go/BenchmarkCRC64ISOShort
patrickmn/go−cache cache_test.go/BenchmarkRWMutexInterfaceMapGetStruct
patrickmn/go−cache cache_test.go/BenchmarkRWMutexInterfaceMapGetString
patrickmn/go−cache cache_test.go/BenchmarkRWMutexMapGetConcurrent
patrickmn/go−cache cache_test.go/BenchmarkRWMutexMapSetDeleteSingleLock
patrickmn/go−cache cache_test.go/BenchmarkRWMutexMapGet
patrickmn/go−cache cache_test.go/BenchmarkRWMutexMapSet
patrickmn/go−cache cache_test.go/BenchmarkRWMutexMapSetDelete

Bibliography

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

[2] sealuzh/goabs. https://github.com/sealuzh/GoABS, 2019. [Online; accessed 2019-08-30].

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

[4] Goast viewer. https://github.com/yuroyoro/goast-viewer, 2019. [Online; accessed 2019-08-30].

[5] Murray Woodside, Greg Franks, and Dorina C. Petriu. The future of software performance engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 171–187, Washington, DC, USA, 2007. IEEE Computer Society.

[6] Thanh H. D. Nguyen, Meiyappan Nagappan, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. An industrial case study of automatically identifying performance regression-causes. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 232–241, New York, NY, USA, 2014. ACM.

[7] Diego Costa, Cor-Paul Bezemer, Philipp Leitner, and Artur Andrzejak. What's wrong with my benchmark results? studying bad practices in jmh benchmarks. *IEEE Transactions on Software Engineering*, 06 2019.

[8] Elaine J Weyuker and Filippos I Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *IEEE transactions on software engineering*, 26(12):1147–1156, 2000.

[9] 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.

[10] Christoph Laaber, Joel Scheuner, and Philipp Leitner. Software microbenchmarking in the cloud. how bad is it really? *Empirical Software Engineering*, pages 1–40, 2019.

[11] Christoph Laaber, Joel Scheuner, and Philipp Leitner. Performance testing in the cloud. how bad is it really? *PeerJ PrePrints*, 6:e3507v1, 2018.

[12] Petr Stefan, Vojtech Horky, Lubomir Bulej, and Petr 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, pages 401–412, New York, NY, USA, 2017. ACM.

[13] Vojtěch Horký, Peter Libič, Lukáš Marek, Antonin Steinhauser, and Petr Tůma. Utilizing performance unit tests to increase performance awareness. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 289–300, New York, NY, USA, 2015. ACM.

[14] Marcelino Rodriguez-Cancio, Benoit Combemale, and Benoit Baudry. Automatic microbenchmark generation to prevent dead code elimination and constant folding. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 132–143. IEEE, 2016.

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

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

[17] Philipp Leitner and Jürgen Cito. Patterns in the chaos&mdash;a study of performance variation and predictability in public iaas clouds. *ACM Trans. Internet Technol.*, 16(3):15:1–15:23, April 2016.

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

[19] Shiquan Ren, Hong Lai, Wenjing Tong, Mostafa Aminzadeh, Xuezhang Hou, and Shenghan Lai. Nonparametric bootstrapping for hierarchical data. *Journal of Applied Statistics*, 37(9):1487–1498, 2010.

[20] hprose/hprose-golang. https://github.com/hprose/hprose-golang, 2019. [Online; accessed 2019-08-30].

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

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

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

[24] J. Chen and W. Shang. An exploratory study of performance regression introducing code changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 341–352, Sep. 2017.

[25] Q. Luo, D. Poshyvanyk, and M. Grechanik. Mining performance regression inducing code changes in evolving software. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 25–36, May 2016.

[26] Juan Pablo Sandoval Alcocer and Alexandre Bergel. Tracking down performance variation against source code evolution. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015, pages 129–139, New York, NY, USA, 2015. ACM.

[27] 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 on International Conference on Performance Engineering*, ICPE '17, pages 373–384, New York, NY, USA, 2017. ACM.

[28] A. Iosup, N. Yigitbasi, and D. Epema. On the performance variability of production cloud services. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 104–113, May 2011.

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

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

[31] Packages. https://golang.org/pkg/, 2019. [Online; accessed 2019-08-30].

[32] Martin Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, 1988.

[33] Package management tools. https://github.com/golang/go/wiki/PackageManagementTools, 2019. [Online; accessed 2019-08-30].

[34] Callgraph tool. golang.org/x/tools/cmd/callgraph, 2019. [Online; accessed 2019-08-30].

[35] A tour of go. https://tour.golang.org/list, 2019. [Online; accessed 2019-08-30].

[36] fzipp/goycylo. https://github.com/fzipp/gocyclo, 2019. [Online; accessed 2019-08-30].

[37] Package packages. https://godoc.org/golang.org/x/tools/go/packages, 2019. [Online; accessed 2019-08-30].

[38] gonum/gonum. https://github.com/gonum/gonum, 2019. [Online; accessed 2019-08-30].

[39] Pandas. https://pandas.pydata.org, 2019. [Online; accessed 2019-08-30].

[40] Scipy spearmanr. https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.stats.spearmanr.html, 2019. [Online; accessed 2019-08-30].

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

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

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

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

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

[46] stratumn/sdk. https://github.com/stratumn/sdk-js, 2019. [Online; accessed 2019-08-30].

[47] eaburns/t_old. https://github.com/eaburns/T_old, 2019. [Online; accessed 2019-08-30].

[48] Go 1.8 defer improvement. https://golang.org/doc/go1.8, 2019. [Online; accessed 2019-08-30].

[49] Go 1.13 version notes. https://golang.org/doc/go1.13, 2019. [Online; accessed 2019-08-30].