Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Using Software-Performance-Antipatterns and Profiling Traces to Perform Code-Refactorings

Niko Stadelmaier

Course of Study:	Softwaretechnik
Examiner:	Dr.-Ing. André van Hoorn
Supervisor:	Dr. Dušan Okanović, Dr. Catia Trubiani, GSSI, Italy
Commenced:	July 30, 2019
Completed:	January 30, 2020
CR-Classification:	I.7.2

Abstract

Today, usability, user satisfaction, as well as enterprise adoption of a software application, are highly influenced by the performance of the software application. Therefore, it is required to resolve performance issues as early as possible during the development of the software. Many issues can be resolved during the planning and design phase by integrating a model-based antipattern detection. Such approaches can be easily integrated with continuous development and integration pipelines, which are often used in modern software development following an agile development methodology.

The focus of this thesis is to develop an approach that can automatically detect performance antipatterns and suggest refactorings for the found problems. In contrast to model-based approaches, the intention is to detect the problems on the code-level.
To tackle the problem, we make use of profiling traces that record the execution of an application. After the initial research on antipatterns in Go, we introduce the identified code-based antipatterns. We then present the benchmark application, where we implemented the problems. This benchmark is then used to generate the profile traces. Now, we analyze how the problems can be detected in the profiles. We then extract our novel *code-* and *profile-patterns* from the profiling information. These patterns are then used by our detection tool to identify the problems in the profiles and suggest the respective refactorings.

Our results show that our approach can automatically detect performance antipatterns in the profiling data. However, more tests need to be conducted to conclude if the approach can detect antipatterns in the data of other systems.

Kurzfassung

Heutzutage spielen die Gebrauchstauglichkeit, Kundenzufriedenheit und Adaption der Software in der Industrie eine wichtige Rolle. Diese werden stark von der Leitung, in Bezug auf die Geschwindigkeit der Software beeinflusst. Daher ist es notwendig Probleme mit Leistung der Software bereits in der Entwicklung zu finden und zu lösen. Dazu gibt es Werkzeuge, wie model-basierte Antipattern-Detection Tools, welche sich in den Entwicklungsprozess einbinden lassen.

Das Hauptaugenmerk der Arbeit ist die Entwicklung eines Prozesses, welcher automatisiert „performance antipatterns“ auffinden und Vorschläge zur deren Lösung machen kann. Im Gegensatz zu modellbasierten Methoden, entwickeln wir einen Prozess, der auf der Quelltextebene arbeitet. Dazu nutzen wir sogenannte Profiler Daten. Nach der initialen Untersuchung der Antipatterns in Go, stellen wir diese vor. Danach zeigen wir die entwickelte Benchmark Anwendung, welche die Anti Pattern implementiert. Diesen Benchmark nutzen wir, um die Profiler Daten zu erzeugen. Anschließend extrahieren wir unsere so genannten *code-* und *profile* patterns aus den Daten und stellen diese vor. Die Patterns werden nun in der automatisierten Antipattern-Erkennung verwendet, um diese zu identifizieren und Lösungsvorschläge anzubieten.

Unsere Ergebnisse zeigen, dass der automatisierte Erkennungsprozess funktioniert und Antipatterns aus den Profiler Daten filtern kann. Es ist jedoch notwendig weitere Tests durchzuführen, um festzustellen, ob der entwickelte Prozess auch für andere Systeme und Daten funktioniert.

Contents

List of Figures

List of Tables

List of Acronyms

ADDL Architecture Analysis & Design Language

API application programming interface

APM Application Performance Monitoring

AST Abstract Syntax Tree

CD Continuous Deployment

CI Continuous Integration

CPU Central Processing Unit

EDA Excessive Dynamic Allocation

EP Extensive Processing

I/O Input/Output

LOC Lines of Code

OLB One-Lane Bridge

protobuf Protocol Buffers

Regex Regular Expression

SPA Software Performance Antipattern

List of Listings

List of Algorithms

Chapter 1

Introduction

This thesis introduces an approach for automatic detection of performance antipatterns in Go programs using profiler data as well as source code and suggesting possible refactorings to solve the found issues.

This chapter provides an overview of the thesis and motivation as to why such an approach is necessary. Firstly, the motivation behind this research is explained, followed by the goals of the thesis, a short overview of the approach, and finally, the structure of this document.

1.1. Motivation

In today's world where cloud computing, microservices, and web services are omnipresent and play an increasingly important role, application performance becomes more and more important as it is one of the major factors for customer satisfaction.
Most major companies that provide web services like Amazon, Facebook, or Netflix, employ some form of monitoring strategy for their services to monitor the performance and correctness of their applications. Unusual events in performance or other issues can then be visualized using various tools to get a better overview of the state of the application. Many unusual events, like a spike in Central Processing Unit (CPU) usage or an increase of the response time of a service, can then be observed as a peak in the different graphs of a performance monitoring tool.

After noticing that something went wrong with the application or system, the developer starts looking for the causes of the problem. This can be a very difficult and tedious task since there is a lot of data to analyze to find the root of the problem. To tackle this, application performance monitoring tools, for example, Kieker [HWH12], are used to automatically screen the data and help point out eventual points of failure. This

process is called application performance management. Its main concerns are to monitor and manage the performance of software applications, as well as their availability. Besides application performance monitoring tools, there are profilers, to help monitor the application in an often earlier stage of their lifecycle than Application Performance Monitoring (APM) tools. APM tools are mainly used to monitor deployed software, whereas profilers are often used during the development of an application. With the help of profilers, it is possible to instrument the code to record low-level performance data like CPU or memory utilization per function or even line of code during the development stage. Although profilers record a lot of data, there are many language-specific tools to help visualize and analyze the recorded profile data to ease the process of finding issues in the code. This way, performance issues can be detected and fixed before they make it into the later deployed application.

The problem with this method of analysis is that it is still a largely manual task that can be very time-consuming. Further, it requires expert knowledge about the limitations and capabilities of the used profiler and tools.
This thesis introduces an approach to automate the detection of software performance antipatterns using profiler data and provide possible refactorings for the found issues to the user. The goal of the approach is to simplify and speed up the process of analyzing the given data and finding bottlenecks in the given software.

1.2. Goals

The goal of the thesis is to develop an approach that automates the Software Performance Antipattern (SPA) detection for Go applications on the code level by utilizing the provided profiler data and finally provide possible refactorings to solve the detected problems.

Investigate SPAs in Go As a first step, it is important to research and understand the basics of the language for which we are developing the approach. Here, the goal is to find possible code structures or bad practices that can be indicative of introducing performance issues to an application.

Evaluate pprof The next goal is to evaluate the capabilities of the Go profiler pprof. We have to get familiar with its usage and how to utilize it from within an application instead of using the interactive command-line. It is essential to know what profiles can be recorded and how they can be analyzed and viewed using pprof.

Build a benchmark application Additionally, we want to develop a benchmark application that implements some of the found bad practices. We need this to recognize how the problems manifest themselves in the profiler data to be able to build the

detection program. Further, it helps to analyze whether the applied refactoring results in a performance increase or not.

SPA research and categorization of problems Further, it is essential to research what SPAs exist and how the bad practices from above can be categorized into the existing antipatterns. This is not as clear cut as it first seems since the code level issues can manifest in multiple antipatterns at the same time. Meaning, it is possible for a code problem to cause excessive processing and bad memory usage at the same time. Such a problem could either fit into Extensive Processing or Wrong Cache Strategy. Therefore, it is crucial to have a precise categorization to differentiate the issues better and better align with existing research.

A concept for the detection To be able to detect the SPAs, we need to develop a concept of how to find them in the available profiling data. For example, are there certain patterns or code structures that we can look for, or are there other indicators that we have to consider as well. The concept process should be applicable to many antipatterns to that it can be easily extended to detect other antipatterns with minor changes.

Resolving SPAs The next goal of the thesis is the implementation of the detection software. The software should use the found patterns from above and utilize the pprof profiler to identify the SPAs. After the detection, the software should output the results and possible refactorings to help solve the found issues.

Evaluation Finally, our implemented approach has to be tested. We need to confirm that the software can detect SPAs and if the suggested refactorings provide a speed-up of the application. For the evaluation, we will use our benchmark application since we know which SPAs are present. This helps to confirm a successful proof of concept for our approach.

1.3. Approach Overview

For this research, which aims to detect SPAs at the code level, we first require a baseline benchmark application. The goal of this benchmark is to help identify how certain bad coding practices manifest in the recorded profile. First, to develop such an application, it was necessary to research the Go language and study its documentation as well as other sources that are concerned with performance-related code problems. Following this research, we developed a benchmark application that implements some of the found bad practices and their suggested solutions. The benchmark further generates a CPU and memory profile and additionally records the run-times of the functions which represent a problem. Each function is run 100 times to counteract some fluctuations in the run-time

and reduce the impact of other random influences on the machine. We have to do this since we do not have full control over the scheduler, CPU clock, or core assignment on the CPU.

After running the benchmark, we analyzed how the SPAs could be detected in the given profile data. Additionally, we confirmed that the refactorings do improve the run-time of the affected functions, so that we can suggest these refactorings to the user.
In order to automate the antipattern detection we came up with the notion of *code-patterns* and *profile-patterns*. Code-patterns can be found in the source code of the application, whereas profile-patterns only concern the profiler output. These patterns are extracted for each of the antipatterns by manually evaluating the profiler output and source code of the benchmark app. Now, to detect SPAs we search for matching patterns in the source code and the profiler output; if both are detected, we have found an antipattern.

An overview of the approach is shown in Figure 1.1.

The detection mechanism can be summarized as a five-step approach. To run the detection the program requires three inputs 1. the source code of the application, 2. the CPU profile, and 3. the memory profile. For now the *profile-* and *code-patterns* are hardcoded into the application. Once all three inputs are given, the detection works as follows:

1. Get the top 100 slowest functions in the profile.
2. Filter the above by functions which appear in the given source code.
3. Search for *profile-patterns* for each function.
4. Search for *code-patterns* for each function.
5. Generate output with metrics and report findings and refactorings.

A more detailed explanation of the implemented detection process can be found in Chapter 5 on page 41.

1.4. Thesis Structure

The thesis is structured as follows:

Chapter 2 – Foundations: The foundations for the thesis are described in this chapter.

Chapter 3 – Related Work: This chapter provides an overview of the related work regarding the topic of performance antipatterns.

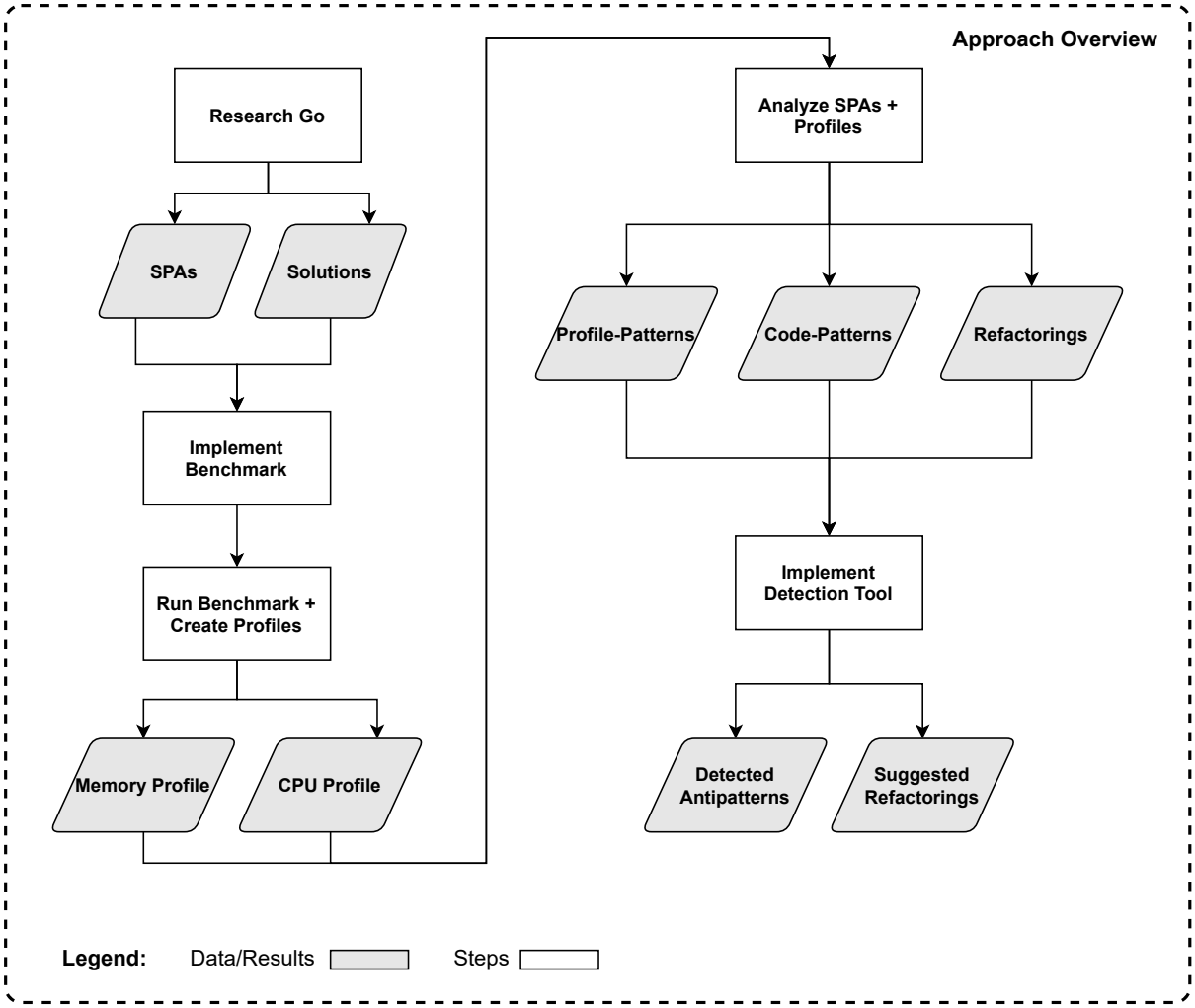


Figure 1.1.: Approach overview

Chapter 4 – Performance Antipatterns in Go: Here we show the found, source code level, performance antipatterns for Go as well as our benchmark application.

Chapter 5 – Antipattern Detection: This chapter explains the automated detection process in detail and how the results from chapter 4 are used to detect SPAs. The process is explained for each antipattern and what the issues and limitations are.

Chapter 6 – Antipattern Solutions: Solutions for the software-performance antipatterns are discussed in this chapter.

Chapter 7 – Evaluation: An evaluation of the developed approach is conducted in this chapter. Further, the research question are presented and lastly the discussion of the results.

Chapter 8 – Conclusion: The final chapter summarizes the results of the thesis, provides the conclusion and future work.

Chapter 2

Foundations

This chapter outlines the important foundations of the thesis. These include a short introduction to the Go language, an explanation of patterns, antipatterns, and performance antipatterns as well as an overview of profiling computer programs and, lastly, code-refactoring.

2.1. Go Language

The Go language [Goo20e] is an open-source project run by Google. It is statically typed and often referred to as a very clean and efficient language. Go has a mature and sophisticated concurrency model that allows the programmer to get the best performance out of multicore systems with ease. Further notable features are garbage collection and runtime reflection.

The language has been developed with regard to today's modern computer systems, which often focus on scaleable cluster- and cloud computing oriented applications. The language's syntax is similar to the syntax of the C language, which should help system programmers to adapt to Go more easily. While it is oriented at the C syntax and also offers pointers, it does abstain from the error-prone pointer arithmetic that is possible in C. Furthermore, it features closures, type-security, and automatic memory management, which is a mostly manual task in other languages like C or C++. Object-oriented programming is also supported using Go's interfaces, mixins, and the possibility to declare methods on types. Lastly, it is possible to organize the source code in modules by using Go's package feature.

The following example Listing 2.1 demonstrates Go's syntax and basic output to the console using the standard hello-world example from the Go documentation. Explanation of the example:

Listing 2.1 Go Hello World Example [Goo20d]

```
package main

import "fmt"

func main() {
      fmt.Println("Hello, Wolrd!")
}
```

On the first line is the declaration of the package. This specifies to which package the file and it's code belong to; here, it is the main package – a special package meant for executable programs. We then import the package `fmt`, which implements formatted Input/Output (I/O) and is later used to print "Hello World!" to the console.

Next, we find the `main` function. It serves as the entry point of an executable program similar to Java or C. Inside the `main` function we print the string "Hello, World!" to the console using the `Println` function of the previously imported `fmt` package.

2.2. Performance Antipatterns

In general, *patterns* or *design-patterns* are common solutions to recurring problems. They provide general solutions which can be adapted to the given context of an application or architecture. In short: patterns represent best practices for software development. A pattern or design-pattern as introduced in [Gam95] by Gamma generally consists of the following four elements:

- The **Pattern name** should be a meaningful and descriptive name to capture the problem, solution, and possible consequences in only a couple of words.
- The **Problem** description describes the problem and context and when to apply the pattern.
- The **Solution** expresses the elements, their relationships, responsibilities, and collaborations. It does not describe a concrete implementation; instead, the focus is on an abstract description of a design problem and its general solution.
- **Consequences** describe possible outcomes and trade-offs which come with the application of the pattern. This information is vital for evaluating alternatives.

Antipatterns follow the same concepts and principles as design patterns. They document recurring solutions to common problems. In contrast to a design-pattern, the roles are "reversed", in that the use of an antipattern produces a negative outcome, and the use

of a design pattern generally produces a positive outcome. That means, antipatterns document common mistakes in software development. Antipatterns are very useful since they can help the developers to avoid mistakes during development or to easier identify and correct them later.

Performance antipatterns are specialized in that they mainly focus on performance-related problems and solutions to those problems. Most of the performance antipatterns we will use for the thesis stem from the work of Smith et al. in their works [SW00] [SG02] and [SW03] where they propose a number antipatterns in software development which typically cause performance problems. They show that these antipatterns cause performance problems and also negatively impact other quality attributes and that refactoring those antipatterns leads to better performance. Analog to design-patterns, each performance antipattern is defined by (as defined in [SW00]):

- Name: descriptive name for the antipattern
- Problem: a description of the situation that causes the problem
- Solution: how to avoid the antipattern or correct it

2.2.1. An overview of relevant performance antipatterns for the thesis

In this chapter, we introduce the relevant antipatterns for the thesis. First, each SPA is described in detail, following the detailed description is a table Table 2.1 to provide a better overview of the antipatterns.

Extensive Processing (EP)

[SG02] Extensive Processing occurs when a long-running process monopolizes the available processor. The processor is then removed from the available pool, and no other workload can be assigned to it while the process is still running. This is particularly problematic if the extensive processing is on the processing path of the most frequent workload. To solve the problem one has to identify the processing steps that may cause slowdowns. These can then be delegated to processes that will not disrupt the fast path.

One-Lane Bridge (OLB)

A One-Lane Bridge [SW01] is a point in the execution of a program where only one or few processes may run at the same time, and all other processes must wait. It frequently occurs when accessing a database or when multiple processes make synchronous calls to another non-multi-threaded process or another resource. These processes must then take turns in order to "cross the bridge". To solve the problem, one can use shared resources that allow concurrent access. For example: When accessing a database, use multiple connections to reduce the round-trip time of the requests or utilize database features like bulk-inserts or transactions.

Excessive Dynamic Allocation (EDA)

This antipattern generally occurs when an application unnecessarily creates and destroys a large number of objects. The overhead of the object creation and destruction negatively influences the performance. In garbage-collected languages, this can cause the garbage collector to run too frequently, which creates an unnecessary overhead. One solution is to recycle objects. This can be achieved by pre-allocating a pool of objects. New objects are then requested from the pool, and unused objects are returned to the pool [SW00].

2.3. Profiling

Generally, profiling can be summarized as the process of extrapolating information and creating a useful general view of a subject of interest.

In this case, the subject of interest is a computer program. In computer science and software engineering, profiling generally refers to the analysis of a computer program to, for example, measure its time complexity, memory allocations, or the execution time of functions. Therefore, profiling and the obtained profiling information serve as an excellent aid to optimize software code.

2.3.1. Instrumentation

To enable a program to be profiled, the source code or the binary has to be instrumented to support profiling. Instrumentation refers to the addition of certain instructions to

Antipattern	Problem	Solution
Extensive Processing	Occurs when processing is not needed or not needed at that time.	Delete the extra processing steps, reorder steps to detect unnecessary steps earlier, or restructure to delegate those steps to a background task.
Excessive Dynamic Allocation	Occurs when an application unnecessarily creates and destroys large numbers of objects during its execution. The overhead required to create and destroy these objects has a negative impact on performance.	1) "Recycle" objects (via an object "pool") rather than creating new ones each time they are needed. 2) Use the Flyweight pattern to eliminate the need to create new objects.
One-Lane Bridge	Occurs at a point in the execution where only one, or a few, processes may continue to execute concurrently (e.g., when accessing a database). Other processes are delayed while they wait for their turn.	To alleviate the congestion, use the Shared Resources Principle to minimize conflicts.

Table 2.1.: Excerpt of Performance Antipatterns used in this thesis [SW03]

the code, which allow the measurement of the execution time of functions or memory allocations made by the program.

Instrumentation and, therefore, profiling is limited by the execution path of the program. Meaning, that instrumented code that does not get executed cannot produce any measurement results. This means one has to make sure that the point of interest – a function or other any other code of the program – is actually executed.

2.3.2. Profiling Go Programs

The Go language provides the means to profile an application out of the box. There are multiple ways to enable profiling of a Go program. We can enable profiling by using the `runtime/pprof` package [Goo20b] or the `net/http/pprof` package [Goo20a]. These packages write runtime data in the protobuf format expected by the pprof visualization tool [Goo20b]. More information about the Protocol Buffers (protobuf) format can be found here, [Goo20f]. We do not explain the format here because we do not use the profile-files directly. Instead, we use the pprof visualization tool [Goo19] to analyze the profiles, see Section 4.1 on page 23. These different options are discussed in the following.

These options are:

1. Manually instrumenting the code and configuring of the profiler with the `runtime/pprof` package.
2. Using Go's `net/http/pprof` package, which automatically registers http endpoints to retrieve the profiles from.

Option one is better suited for short-running applications like command line or standalone applications or instances where one needs more control over the profiler. For example, if one wants to add a stack-trace of a certain value to the profile or create a custom profile. It is also a great option if the application to profile runs on the local machine, and one has access to the local filesystem to save and load the profiles into the pprof tool. The Listing 2.2 from the go documentation [Goo20d] shows how to enable profiling for an application using the first option.

Listing 2.2 Enable profiling for a standalone application [Goo20d] [Goo20b]

```
var cpuprofile = flag.String("cpuprofile", "", "write cpu profile to ‘file‘")
var memprofile = flag.String("memprofile", "", "write memory profile to ‘file‘")

func main() {
   flag.Parse()
   if *cpuprofile != "" {
      f, err := os.Create(*cpuprofile)
      if err != nil {
         log.Fatal("could not create CPU profile: ", err)
      }
      defer f.Close()
      if err := pprof.StartCPUProfile(f); err != nil {
         log.Fatal("could not start CPU profile: ", err)
      }
      defer pprof.StopCPUProfile()
   }

   // ... rest of the program ...

   if *memprofile != "" {
      f, err := os.Create(*memprofile)
      if err != nil {
         log.Fatal("could not create memory profile: ", err)
      }
      defer f.Close()
      runtime.GC() // get up-to-date statistics
      if err := pprof.WriteHeapProfile(f); err != nil {
         log.Fatal("could not write memory profile: ", err)
      }
   }
}
```

Listing 2.3 Enable profiling for long-running services [Goo20d] [Goo20a]

```
// import the net/http/pprof package, this will create debub/pprof http handlers

import _ "net/http/pprof"

//if there is no http server already running start one by importing "net/http" and "log"
// and add the following lines to the main function

go func() {
   log.Println(http.ListenAndServe("localhost:6060", nil))
}()
```

The second option is better suited for long-running processes like web applications or other web services, where one does not necessarily have access to the local file system to save and retrieve the profile files. As above motioned, this option automatically creates http endpoints to retrieve the different profiles from. These endpoints can be adjusted to fit the desired API of the application so that it integrates better with existing web services or remote applications.
To enable profiling with the second option one has to follow the instructions of Listing 2.3 Different profiles can now be retrieved under the `http://localhost:6060/debug/pprof` endpoint. For example, to create a 30 second CPU profile and open it in pprof, type the following line in the terminal.

```
go tool pprof http://localhost:6060/debug/pprof/profile?seconds=30
```

We chose option one to profile our benchmark application since it provides more control, and is better suited for stand-alone applications like our benchmark. This way, we can ensure that the profiler captures the entire execution of the program. The advantage of this option is that we can start and stop the profiling as we require.

Available Profiles

The profiler provides seven different predefined profiles, as found on [Goo20c].

cpu CPU profile determines where a program spends its time while actively consuming CPU cycles (as opposed to while sleeping or waiting for I/O).

heap Heap profile reports memory allocation samples; used to monitor current and historical memory usage and to check for memory leaks.

threadcreate Thread creation profile reports the sections of the program that lead to the creation of new OS threads.

goroutine Goroutine profile reports the stack traces of all current goroutines.

block Block profile shows where goroutines block waiting on synchronization primitives (including timer channels). Block profile is not enabled by default; use `runtime.SetBlockProfileRate` to enable it.

mutex Mutex profile reports the lock contentions. When you think your CPU is not fully utilized due to a mutex contention, use this profile. Mutex profile is not enabled by default, see `runtime.SetMutexProfileFraction` to enable it.

[Goo20c] Note, that the CPU profile is not available as a profile directly, because it streams its output to a writer. It has its own application programming interface (API), the `StartCPUProfile` and `StopCPUProfile` functions as shown in Listing 2.2. All other profiles can be retrieved by calling `pprof.Lookup(name string) *Profile`, where `name` is the name of the profile in lowercase letters.

2.4. Refactoring

[Fow19] In software engineering, *refactoring* refers to the process of either manual or automated restructuring of the code without changing its original behavior. Generally, the goal is to improve the structure of the code and its non-functional attributes like maintainability, readability, or complexity.

Refactoring is often done after finding a code smell. A code smell is a surface level indication in the code that there is a deeper problem in the system [Fow06]. Code smells are not necessarily bugs, or technically incorrect; they rather represent weaknesses in design and development. For example, one finds a very lengthy method in the source code, which can lead to bad readability and negatively influence the maintainability of the program. To combat this, one can then extract parts of the method into a subroutine to split the lengthy code into smaller pieces with clear-cut concerns and improved naming, if required. This practice would also allow the use of the new subroutine in other functions or methods which may require its functionality. There are many other code smells or antipatterns that can exist in software. Duplicate code, where the same implementation appears multiple times in the code or the shotgun surgery where when one makes a little change to the code, many other changes have to follow in a lot of different places [Fow19].

The refactoring process is an ongoing process during the lifecycle of a software system. It aims to improve the structure, readability, complexity, and other qualitative aspects of the software.

The refactoring process generally works as follows [Sea16] [Fow19].

1. develop new code and tests
2. test the new code to ensure that there are no changes to the functionality
3. apply the refactorings i.e., the developed and tested code
4. test the refactoring and software system to ensure the functionality
5. repeat the process

To summarize, refactoring is the ongoing process of improving software code by applying internal changes regarding its structure and other qualitative features such as readability or maintainability, without changing its external behavior.

Chapter 3

Related Work

In the following sections, we provide a brief introduction to the topic "Performance Antipatterns" what they are and why they are helpful for the detection of performance problems in architectural models. Further, we present some of the existing approaches in academia with the focus on antipattern based approaches to detect and possibly solve performance antipatterns in software architectural models. The latter implies that some form of a model is required, which holds true for many of the current approaches, but there is some recent development towards using trace and profiling based approaches [TBH+18] for detecting performance antipatterns. The most relevant papers for this thesis are covered in the next sections.

3.1. Performance Antipatterns

As described in [CMT10], we are mostly interested in antipatterns, which can be described without domain-specific notions and are therefore independent of their representation. This is important as it can help to automate the detection and solution of performance antipatterns. Further, the suggested approach would be independent of any architectural modeling or performance modeling language, require fewer domain experts and therefore be more applicable in the real world. A number of notion-independent antipatterns can be found in the work of Smith and Williams in [SW03]. This thesis will mostly build on these findings and definitions as well as those mentioned in [CMT10]. The following table 3.1 presents a subset of these antipatterns, which are also found and presented in later sections, see 4.

Antipattern	Problem	Solution
Unnecessary Processing	Occurs when processing is not needed or not needed at that time.	Delete the extra processing steps, reorder steps to detect unnecessary steps earlier, or restructure to delegate those steps to a background task.
Circuitous Treasure Hunt	Occurs when an object must look in several places to find the information that it needs. If a large amount of processing is required for each "look," performance will suffer.	Refactor the design to provide alternative access paths that do not require a Circuitous Treasure Hunt (or to reduce the cost of each "look").
Empty Semi Trucks	Occurs when an excessive number of requests is required to perform a task. It may be due to the inefficient use of available bandwidth, an inefficient interface, or both.	The Batching performance pattern combines items into messages to make better use of available bandwidth. The Coupling performance pattern, Session Facade design pattern, and Aggregate Entity design pattern provide more efficient interfaces.

Table 3.1.: An excerpt of Performance Antipatterns as in [SW03]

3.2. Antipattern Detection

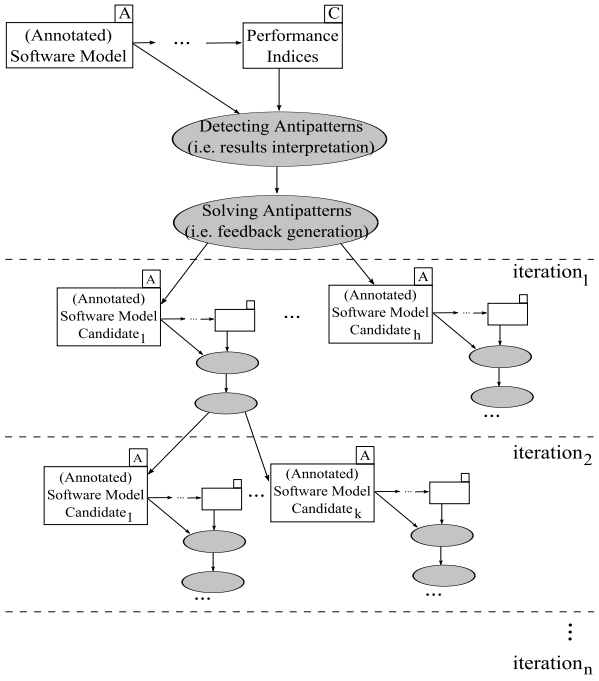
This section focuses on the most relevant approaches for this thesis, the so-called antipattern-based approaches. Firstly, we introduce some of the existing model-based approaches. They utilize the capabilities of well-known modeling languages such as UML [OMG17] and PCM [Pal19] and extensions such as MARTE [OMG08] or OCL [OMG06] to define, detect, and solve performance antipatterns.

In the following, we will review the state of the art with respect to the topic goals and specifically focus on the currently available performance antipattern detection approaches and their applications to locate and remove performance problems.
Currently, there are many approaches to detecting performance antipatterns. Many of them are model-based solutions like [ABGM09] or [PM08], which require some form of an architectural model of the software application. There are many different modeling languages and language extensions available to apply to UML and other modeling languages. Further, there are tools and applications that make use of these modeling languages to perform antipattern detection and refactoring.

Palladio Component Model (PCM)

The Palladio Component Model provides means to model any software architecture and provide substantial QoS (Quality of Service) analysis, such as performance problem detection based on the defined model [BKR09]. Palladio is not just a modeling language; instead, it provides many tools to help to create and work with the model in the so-called Palladio-Bench [Pal19]. PCM is a widespread tool in the domain of software performance modeling and analysis. The following approach shows how to use the PCM to solve performance antipatterns [TK11]. Their general approach is to 1. Start with an annotated Software Model (PCM), 2. transform it into a Performance Model, 3. solve the Performance Model, 4. interpret performance metrics acquired from previous step, 5. refactor the base model of step 1 according to feedback from the performance metrics, 6. iterare previous steps until problems are solved. The figure 3.1 shows a visual representation of the process as found in [TK11].

Figure 3.1.: Antipattern based process for PCM [TK11]



Unified Modeling Language (UML)

UML [OMG17] is arguably one of the most well-known and widespread modeling languages in modern software development. It is used to describe and model the

architecture of software applications. There are extensions to UML like MARTE [OMG08] or OCL [OMG06] that aim to extend the language by performance-related attributes. Using those extensions, it is possible to simulate and predict the performance of an architectural model defined in UML. Both have been successfully utilized to detect and resolve performance antipatterns in software architectural models. In [CDE+10] the authors show an approach to define and detect performance antipatterns in UML+OCL, but it lacks the automation of resolving the antipatterns. Another approach [CDT12] utilizes UML+MARTE to detect performance antipatterns though it is still a challenge to automate their solution.

Other Model-based Approaches

In addition to the aforementioned detection methods, there are further approaches like [Xu12] and [PM08] to automate refactoring of software architectures based on detected performance antipatterns. Also, there are multiple frameworks and applications available to perform antipattern based software or architecture refactoring. One of those frameworks–called "Performance Antipattern Detection" (PAD)–was created by Parsons et. al. and is introduced in [PM08]. It focuses on Enterprise Java Beans applications and therefore is domain-specific to the EJB world. However, its principles of a rule-based detection using runtime metrics can be applied to other languages as well. In parts, it is very similar to the approach we are proposing.

Another tool for architecture optimization is *ArcheOpterix* [ABGM09]. It is based on the Eclipse framework and aimed at embedded systems described with AADL models. The Architecture Analysis & Design Language (ADDL) is used to model the software- and hardware architecture of embedded programs. It is a so-called architecture description language. Although it is not specifically based on performance antipatterns, their use of evolutionary algorithms and Pareto-Optimization is very promising and might help drive the automated solution of detected performance antipatterns in our approach.

Code and Profiling Based Solutions

Code-based solutions mostly work on the code directly with additional input from monitoring or profiling tools. That means the code is being analyzed without transforming it into a model first. This approach is often used for static analysis to find bugs or patterns that can have a negative influence on the performance. This method can produce false positives if the code is only analyzed using performance antipatterns. In some cases, non-guilty antipatterns are detected, which have no real influence on the execution time

of a program. With the input of additional profiling or monitoring information as in [PM08], we can reduce or even eliminate such false positives.

Trubiani et al. successfully used such an approach [TBH+18] to improve the performance of an application. The authors demonstrate how to use profiling information and source code in conjunction with performance antipatterns to detect the latter in the given application. The profiling information was acquired by driving load tests that represent realistic workloads for the software. The output is then matched with the specification of the antipatterns to detect their location in the source code. More details can be found in the thesis by Bran [Bra17].

Another profile-based approach is called "Profile-based Detection of Layered Bottlenecks" by Inagaki et al. Since this approach also uses the Go language and the pprof profiler tools, it is specifically relevant for this thesis. In their novel approach, the authors extract a performance model; in this case, a thread-dependency graph from the execution profiles of the target application to detect layered bottlenecks. The authors use a customized version of the Go compiler and profiler to generate the thread and novel *wake-up profiles*. Both profiles are then merged to generate their so-called *thread-dependency graph*. By top-down graph traversal along the largest thread counts, the authors were able to successfully detect bottlenecks in the application [IUNO19].

3.3. Open Challenges Addressed by this Thesis

In this thesis, we will develop a method that does not require extensive architectural models and domain experts to refactor the software and fix performance problems. Just as the aforementioned method [TBH+18], the developed method can be easily applied to existing applications that do not have a complete model that represents their architecture. Another advantage is that we do not need the input of multiple domain experts to find and correct critical performance issues. Other approaches often require specifically annotated architectural models like the Palladio Component Model [TK11] [BKR09] or MARTE [OMG08] [CDE+10] [CDT12] which are often used in model-based software development but require specific knowledge and experience in those technologies. Our method eliminates those problems since we focus on the code level, a developer's comfort zone, therefore none or fewer domain experts are required.

The goal is to extend the approach of Trubiani et al. [TBH+18] and automate the feedback generation so that less or even no manual refactoring is required. The tool to develop should automatically detect performance antipatterns and automatically suggest possible refactorings to the user and inform about their trade-offs. Further, we will use the language Go, which is not as widespread as Java and has less tooling

support. By applying the approach above to another language and extending it, we hope to further prove its effectiveness and use in successfully refactoring real-world software applications.

The authors of [IUNO19] have successfully shown that it is possible to detect bottlenecks in Go application using the available profiling data. While their approach can show where the bottlenecks are located in an application, it can not point to the exact locations in code that are responsible for the performance issues.
With our approach, we aim to bring the detection down to the code level, so that we can identify the Lines of Code (LOC) accountable for the performance problem.

Developing such an approach brings up the following questions and challenges:

1. Which performance antipatterns exist on the code level in Go?
2. How well do the existing profiling tools work for Go?
    - Do they produce meaningful output?
      For example: Is the stack trace deep enough to capture an entire run of a program?
      Are the function names captured with their real names, or are they reported using their references i.e., pointer addresses?
    - How can the output be used or transformed into other common formats?
3. How can we automate the antipattern detection?
4. How can the refactoring be automated?

We will research and answer those questions in the thesis. More details on the research questions can be found in later sections of the thesis.

Chapter 4

Performance Antipatterns in Go

To be able to detect SPAs on the code-level, it is essential to know which antipatterns exist in Go and how they manifest in the source code and profile data. In this chapter, we describe how the antipatterns are identified. Firstly, the identification process and methodology are explained, followed by an explanation of the benchmark program that was created for this purpose. Further, we introduce the notion of *code patterns* and *profile patterns*, which are extracted from the code and profiling information of the benchmark and aid in the detection of Chapter 5. Lastly, we list the identified performance-antipatterns for the Go programming language. Each antipattern will be defined and explained in the following sections. Further, an overview is provided in table 4.1 in Section 4.2.

4.1. Identifying Performance Antipatterns

The first step of detecting performance antipatterns is to identify which code constructs can have a negative impact on performance. Further, it is required to identify how those code constructs are represented in code and caught by the profiler. We then have to specify characteristics/metrics and patterns which are specific to each antipattern for both of these cases. Our methodology to find SPAs in Go is described in Section 4.1.1, afterwards Section 4.1.2 explains the developed benchmark program. We then introduce our novel *profile-* and *code-patterns* in Section 4.1.3 and Section 4.1.4. Finally, an overview of the manual analysis process of the profiles is provided in Section 4.1.5. Going forward, we can then build a tool that looks for these antipattern-specific characteristics in the code and in the profile of an application, see Chapter 5.

4.1.1. Methodology

The Go documentation [Goo20d] provides general information on code constructs, which are simple solutions to recurring problems but can cause poor performance for some use-cases. They also list suggestions on how to achieve better performance on these problems. Additionally, there are several online resources like [Sta19a], [Gry16], or [MG18], which analyze the performance of Go in certain scenarios and list the results of their tests as well as corrections and refactorings to speed up the found problems. This thesis shows an excerpt of these problems from the sources above and implements a small benchmark [Sta19b] to confirm the suggested performance antipatterns. Since the benchmark incorporates the findings of the research, it serves as a baseline and case study for further investigations on how to identify these problems in the recorded profile.

4.1.2. Benchmark Program

The testing program runs one hundred iterations of each antipattern or problem and calculates the average runtime for each of the problems. Additionally, it runs the same number of iterations for each refactored antipattern to calculate their average runtime as well to later calculate the potential speedup. It is important to note that these tests are synthetic, and the antipatterns may not appear exactly comparable in real-life applications. But they provide a valuable baseline for our research. The aim of these tests is to confirm the existence of an antipattern and serve as an example to extract the above-mentioned “patterns” for the following detection of the problem in Chapter 5.

The benchmark implements three initial problems, one variation per problem, and two solutions for each problem. In total, we have six problems and twelve solutions implemented in the benchmark. As mentioned above, these problems are executed in a loop of 100 iterations to calculate the average runtime per problem and solution. The number of iterations can be changed via a parameter `numRuns` on the command line. Two additional parameters are required to run the benchmark. First, the `cpuprofile` flag which specifies the filename of the CPU profile, second the `memprofile` which specifies the filename for the memory profile.
The benchmark can be found at [Sta19b].
To build the benchmark clone the repository, change into the `benchmarks` folder and type:

```
go build .
```

into the console.

To run the benchmark, run the now created executable `benchmark`:

```
./benchmark -cpuprofile="profile.prof" -memprofile="memory.prof"
```

4.1.3. Profile Patterns

Profile Patterns are patterns that are responsible for causing performance problems. These patterns are extracted from the profiler output, in this case, the *web* command of pprof. The command prints a call graph of the executed program and highlights the hot paths of the code, meaning; it highlights the cumulative slowest functions captured in the samples. Further, the graph shows the runtimes of the called functions with their absolute time and a percentage of the total execution time available in the profile.
When identifying profile patterns, we are interested in those functions that are called by other slow, user-defined functions, and also appear in the hot path of the graph. To find these functions, we analyze the called function of every problematic function in our benchmark program. Using this process, we hope to identify functions that are indicative of a performance problem. The found functions are our so-called *profile-pattern* for which we scan the provided profile.

Running the above-mentioned test program for each antipattern and profiling the runs allows us to identify the slowest function calls that generally appear in the profile for each of the antipatterns. In short: Profile Patterns are function calls that generally appear in the call graph of a program and are indicative of causing performance problems.

4.1.4. Code Patterns

Code patterns are patterns that appear in the source code of an application and are identified to cause performance issues.

The code pattern extraction process is comparable to the extraction of the profile patterns. We run the benchmark application to create a CPU profile. To identify the code patterns, we then look at the annotated output of pprof's *list* functionality. This outputs the source code of a function, where each line of code is annotated with their respective timings. Long-running LOC that are indicative of a performance problem are extracted from the profile and added to our pool of code patterns. With the manual review of the problematic source code, we hope to isolate certain code constructs that are responsible for the bad performance of a function. Our intention is to have a baseline of code examples that we can use to identify antipatterns in the automated detection.

4.1.5. Manual Analysis Process

In this section, we describe the manual process used to identify problems in the profile produced by the benchmark program. Since the benchmark produces both a CPU-profile and memory profile, we can identify bottlenecks in the execution or response time of a function as well as memory-related issues like memory leaks. The process generally looks like follows:

1. Run and profile the benchmark application.
2. Analyze pprof's *web* output to get an overview of the slowest functions.
3. Compare results to pprof's *topN* output.
4. Use pprof's `peek` to get callers and callees per function.

   } *Profile-Patterns*

5. Identify "guilty" LOC using the `list` feature to get annotated code listing of the slowest functions from the previous steps.

   } *Code-Patterns*

6. Fix the found issues.

An example of the output of the *web* command from step 1 of the list above can be found at Appendix B.1 on page 95. The callgraph can get very big; we advise to use pprof's filtering features, like *show_from* or *show*, to inspect the areas of interest. In Appendix B.2 on page 96, we provide an example out of pprof's *top* command, which is used in step 2 of the process.

The process remains the same for CPU and memory profiles. In the latter, we look for most memory allocations and created objects instead of the runtime of the functions or lines of code. This process is used for each antipattern or problem. The following section, Section 4.2, shows the identified issues and explains the analysis process and presents the results of the process.

4.2. Found Antipatterns

This section provides an overview of the found antipatterns in Go. The problems are listed with an id that will be used throughout the document and problem description. The id can be used to find the implementations of the problem in the benchmark application. The naming scheme with the problem ids and function names is explained in Section 4.2.1. Afterward, the identified antipatterns and the application of the analysis process in practice are shown in Section 4.2.2 for the first problem, Section 4.2.2 for the

second problem and finally Section 4.2.4 for the third problem. We list the categorization for each problem i.e., which antipattern does the problem conform to, e.g., OLB, EP, or another antipattern. Further, the found *profile-* and *code-patterns* are presented in the respective section of the problems. We only show the analysis process for the first variants of the problems because the process remains the same for all the other variants, and the *code-* and *profile-patterns* are the same as well.

4.2.1. Function and Problem Naming Scheme

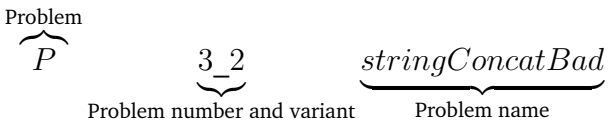
The function names of the benchmark follow a specific scheme, which is also used in the thesis. Each function name starts with the id of the problem, followed by a descriptive name of and possible settings used in the problem, for example, the length of a string. Each id starts with a P for Problem, or S for solution followed by a number (number of the problem). If the problem has a variant, the initial number after P is followed by an underscore and the number of the variant.

Example:

`P3_1stringConcatBad`

This translates to Problem 3 variant 1, string concatenation. “Bad” further indicates that this is a problematic implementation.

`P3_2stringConcatBad`

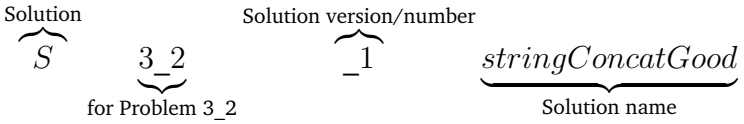


The second example is read as follows: Problem 1, variant 2, string concatenation.

A similar naming scheme applies to the functions that present a solution to the problem. As mentioned above, a solution always starts with the letter S, for example, `S1stringConcatGood`. Further, it contains the word *Good* at the end to additionally clarify that it is a good version of the problem, hence a solution. Each problem has two solutions. That means the solutions are numbered similarly to the problems.

Example:

`S3_2_1stringConcatGood`



The example reads as follows: Solution for problem 3_2, solution version 1, string concatenation. In contrast to the word “bad” from the problem above, “good” further implicates a better implementation to solve the problem.

Note: For the solutions, it is helpful to read the ids backward. Here, version one of the solution for the problem three variant two.

4.2.2. P1 Frequent Object Creation - EDA

Frequently creating new objects requires a lot of CPU time and is expensive on memory and garbage collection. The frequent creation and deletion of objects cause many allocations and de-allocations on the heap, which causes a heavy workload for the garbage collector. In the following example P1_1 Listing 4.1, many objects are created and pre-populated with fixed properties.

Variant 1 (P1_1)

In the first variant of the problem, we simply create many objects to analyze the impact of the frequent object creation. Objects can be created differently, depending on the required type of the object. Here, we chose the struct literal notation with the &, which creates the book object and returns a pointer to the book. More on structs and struct literals can be found in the Go documentation [Goo20d]. The code of Listing 4.1 shows the essential parts of the implementation.

Variant 2 (P1_2)

In the second variant P1_2 of the problem, we additionally pass the created objects to a function `changeObj` that changes some properties of the received object. With the second version, we aim to implement a more realistic scenario, where the created objects are

Listing 4.1 Example one for frequent object creation (P1_1)

```
var book *Book
   for n := 0; n < 1000000; n++ {
      book = &Book{
         Title: "some longer book title",
         Author: "amazing author",
         Pages: 672,
      }
   }
```

Listing 4.2 Example two for frequent object creation (P1_2)

```
for n := 0; n < 1000000; n++ {
   book = &Book{
      Title: "The Art of Computer Programming, Vol. 1",
      Author: "Donald E. Knuth",
      Pages: 672,
   }
   numObj++
   changeObj(book, n)
}
```

actually used as they could be in a real application. The implementation can be found in Listing 4.2.

After running the examples, we analyze the profiler output to find the profile and code patterns which we plan to use later in the thesis when building the automated detection. To get the profile patterns, we analyze the callgraph of the run. A callgraph can be generated with pprof's `web` command, which outputs an annotated graph and highlights the hot path of the execution. Assuming the profile is named `p1.profile`, we can then get the graph as follows:

```
go tool pprof p1.profile
```

This starts pprof with the profile of P1 in interactive mode. Now we can use the `web` command to get the callgraph as an SVG file and analyze it. The graph in Figure 4.1 shows an excerpt of the complete graph. We only show the direct callers of the `P1_1` function as well as the most relevant functions, which are causing a performance issue.

Profile Patterns

With the callgraph, we now note the functions that are called by `P1_1createObjectsBad` and other subsequent functions, which we identified as problematic. From the graph, we can see that the most time spent in the garbage collection (`runtime.mallocgc`) which in

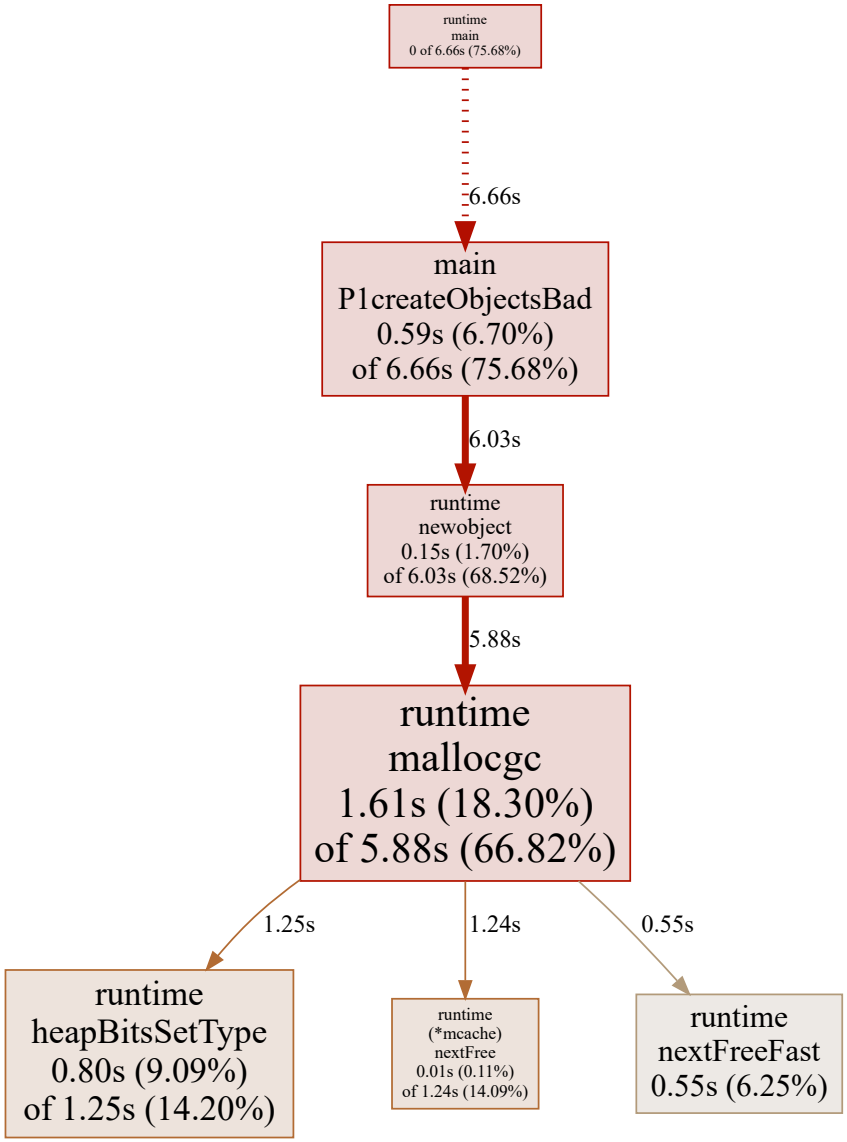


Figure 4.1.: Excerpt of Profiler callgraph for P1

return is triggered by the `runtome.newobject` function. This way, we can identify which functions to look for in the profile. The following information is extracted from the annotated profile callgraph. Patterns to match:

- `runtime.newobject`
- `runtime.mallocgc`
- `runtime.heapBitsSetType`
- `runtime.nextfree`

Listing 4.3 Output of pprof's `list` command

```
 60ms     60ms   19: for n := 0; n < 1000000; n++ {
   .        .    20:       book = &Book{
230ms    330ms   21:             Title: "some longer book title",
100ms    150ms   22:             Author: "amazing author",
   .     4.90s   23:             Pages: 672,
   .        .    24:       }
 20ms     20ms   25:       numObj++
   .        .    26: }
```

Code Patterns

After we specified the profile patterns, we have to go down to the code level to be able to tell which lines of code are responsible for the performance issue. Pprof provides the functionality to get the annotated source code of a function from the given profile. The command in question is called `list`. Again, we open the profile in pprof's interactive mode by calling:

```
go tool pprof p1.profile
```

on the command line. We can now continue to analyze the function of interest by typing `list P1_1createObjectsBad`. The output of the list command can be found in Listing 4.3.

From the output of Listing 4.3, we can see that line 19, 21, 22 and 23 are the top slowest lines of the function. We can ignore line 19 since it is our testing loop, and we are not interested in the loops runtime. In our case, the loop only acts as a multiplier to create many objects. Line 20 is the line we are looking for since this code is actually responsible for the actual object creation. The times of lines 21, 22, and 23 are part of the object creation as well, but these are not the ones responsible for the object creation. Instead of using the object literal notion `&obj{}` to create an object, Go also has a `new` operation that can be used as well.

After analyzing the output and with our knowledge about the Go language, we define the following code patterns:

LOC responsible for causing the problem:

- `var a = &b{}`
- `var a = new*`
- `var a = b{}`

Categorization

To better align with other research on SPAs we categorize our problems into antipatterns like EDA, EP, or other fitting categories. Since P1 frequently creates and deletes objects, which causes many memory operations and excessive garbage collection, we chose to categorize it as *Excessive Dynamic Allocation*.

4.2.3. P2 Non Compiled Regular Expressions - EP

A regular expression can be assigned to a string variable, since a Regular Expression (Regex) consists of a set of characters. This way, the same expression can be reused throughout a program. This is a so-called uncompiled regular expression. In the `regexp` package [Goo20d], Go provides two ways to use such a Regex; The first option is to use functions that take the input to match and the regular expression as an argument for example `regexp.MatchString(pattern string, s string)`. The second option is to pre-compile the regular expression and then use it for further matching, more on that is explained in Chapter 6 on page 53.
The following example Listing 4.4 shows the problematic use of option one.

Variant one (P2_1)

The first variant of the second problem shown in Listing 4.4 implements the use of a non-compiled regular expression. The expression is stored in the variable `testRegexp` and used in the function `MatchString`, to match the given string with the Regex. The purpose of this test program is to show the inefficiency of compiling the regular expression with every iteration. In the test, we match an example string to verify if it follows the correct form of an email address.

Listing 4.4 Example one for non-compiled regular expressions (P2_1)

```
var testRegexp = '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]+$'
   for n := 0; n < 100000; n++ {
      _, err := regexp.MatchString(testRegexp, "test@example.com")
      if err != nil {
         panic(err)
      }
   }
```

Variant two (P2_2)

In the second version of the problem (Listing 4.5) , we implemented a typical scenario where one want to test of a given string contains certain words. This example was chosen, because it shows the unnecessary use of a regular expression. It is possible to solve the problem differently by using the functionality provided by Go's `strings` package. More on that in Section 6.3.2 on page 60.

Listing 4.5 Example two for non-compiled regular expressions (P2_2)

```
var regEx = ‘first|second|third‘
examples := []string{"first ex", "second exp", "test"}
defer timeTrack(time.Now(), "P2.2")
for i := 0; i < 100000; i++ {
   for _, i2 := range examples {
      _, err := regexp.MatchString(regEx, i2)
      if err != nil {
            panic(err)
      }
   }
}
```

Running these examples with the CPU profiler allows us to then analyze the problem in more detail. Using the web option of pprof extracts a callgraph with the highlighted "hot path" – the slowest path in the graph – which shows the slowest functions to look for in the output of the profiler. Figure 4.2 shows the callgraph of the example P2.

Profile Patterns

We can see that most of the time is spent in `syntax.Parse` and `regexp,compileOnePass`. Both of the functions are called by `regexp.compile` which itself is triggered by the `MatchString` function of the regexp package. This concludes that we have to look for uses of the latter in the profiles callgraph. After the inspection of the callgraph, we extracted the following profile patterns.

1. `Regexp.MatchString` – if `MatchString` function is used
2. `regexp.compile`
3. `regexp.Compile`
4. `syntax.parse`
5. `regexp.compileOnePass`

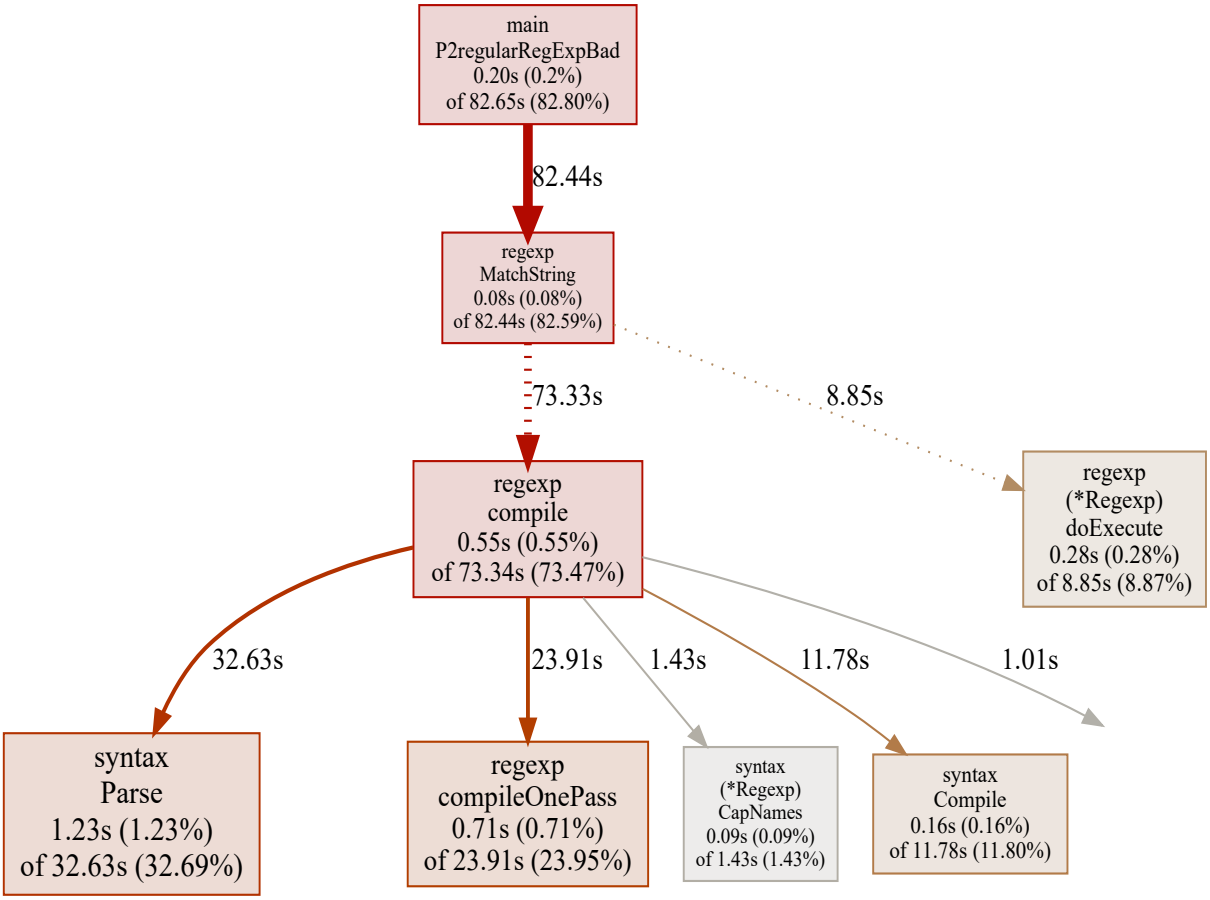


Figure 4.2.: Excerpt of Profiler callgraph for P2

Code Patterns

As for pattern in the code, we have to look for calls to functions like `MatchString`, which can take non-compiled regular expressions as arguments either as an inline definition or as a variable. To confirm this, we have to evaluate the output of pprof's `list` feature, see Listing 4.6, to get detailed information about the source code.

Listing 4.6 Annotated Source Code of Problem 2 (P2_1)

```
    30ms     30ms   14:  for n := 0; n < 100000; n++ {
   140ms 1.18mins   15:        _, err := regexp.MatchString(testRegexp,"test@example.com")
      .        .    16:        if err != nil {
      .        .    17:              panic(err)
      .        .    18:        }
      .        .    20:  }
```

With the code listing, we can confirm that line 15 that contains the call to `MatchString` is indeed taking up most of the runtime of our function. Now, we consult the Go documentation [Goo20d] to find other functions of the regexp package that also use un-compiled regular expressions. Once we combine our findings with the results from the documentation, we get the following code patterns:

1. `regexp.MatchString`
2. `regexp.Match`
3. `regexp.MatchReader`

Categorization

Repeatedly parsing and compiling the regular expression consumes a lot of unnecessary processing power when done extensively, as it is in our example. In this case, the required processing exceeds the overall response time of the matching itself. Further, the solution for the problem is to move the compilation to either the very beginning of the function or to the package definition as a global variable, see Chapter 6 on page 53. For the above-stated reasons, we categorize problem 2 as *Extensive Processing*.

4.2.4. P3 String Concatenation - OLB

Concatenating strings and creating new strings can be memory expensive and slow. This can be true for concatenating large strings, as well as many small strings to create a single string that holds all the data. For example, one could read a large file line by line to do some verification and then concatenate these lines to form a big string representing the initial file contents.

Variant one (P3_1)

In the following example Listing 4.7, we concatenate the letter “a” to a string until it reaches the final length of 100000 characters. Since we only concatenate a single character *a*, the `strLength` is equal to the number of concatenations that is performed in the example. As stated in the code comment of Listing 4.7 it also possible to use the $+=$ operator to join strings.

Listing 4.7 Example one for string concatenation (P3_1)

```
var concString string
   for i := 0; i < strLength; i++ {
      concString = concString + "a"
      //another variant is to use the += operator
      //concString += "a"
   }
```

Variant two

Comparable to the other variants, the second version implements a more realistic use case of the original problem. Here we read a file *AutGen1.txt* from the filesystem. The file size is 5MB and it contains around 70000 lines with 78 characters per line. In the benchmark, the function reads 10000 lines, given in variable `readLines`, from the file and concatenates them to a string.

Listing 4.8 Example tow for string concatenation (P3_2)

```
var concString string
file, err := os.Open("./AutoGen1.txt")
if err != nil {
   log.Fatal(err)
}
scanner := bufio.NewScanner(file)
defer timeTrack(time.Now(), "P3.2 "+strconv.Itoa(readLines))
for i := 0; i < readLines; i++ {
   scanner.Scan()
   concString += scanner.Text()
}
```

As with the other examples, we collect the CPU profile of the run and analyze it for potential profile and code patterns. Again, we use pprof's `web` command to get a callgraph with a highlighted hot-path. By further inspecting the hot path, we can see where the program spends most of its execution time. An excerpt from the graph with the most relevant functions can be found in Figure 4.3.

Profile patterns

In the callgraph we can see that a lot of time is spent in the `memmove` function which copies a number `n` of bytes from one location to another. Memmove gets called by `runtime.concatstring` and that is called by `runtime.concatstring2`. These three function form the main profile patterns of this antipattern. Additioanlly, it is important to

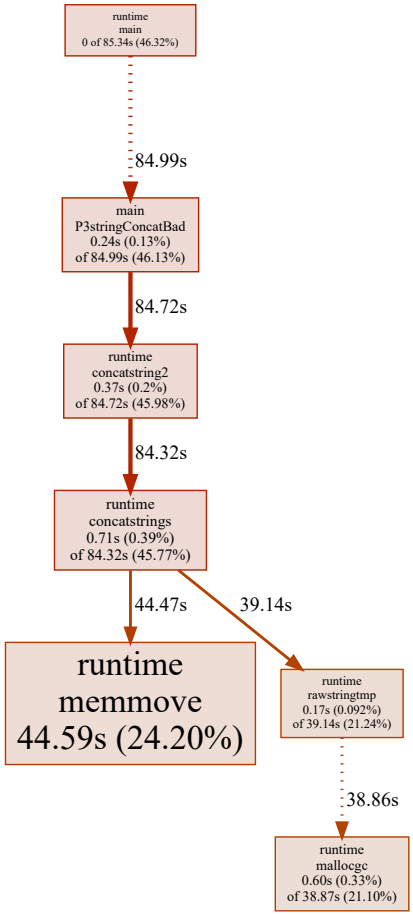


Figure 4.3.: Callgraph of string concatenation (P3)

note that the += operator and `concstring = concstring + "a"` cause different function calls from `P3stringConcatBad`. The first calls `runtime.concatstring` while the latter calls `runtime.concatstring2`. This results in four profile patterns to match. See the following summary:

- `runtime.concatstring`
- `runtime.concatstring2`
- `runtime.concatstrings`
- `runtime.memmove`

Listing 4.9 Annotated Source Code of Problem 3 (P3)

```
50ms     50ms   72:  for i := 0; i < strLength; i++ {
  .        .    73:
70ms  1.07mins  74:        concString += "a"
  .        .    75:        numStrConcats++
  .        .    76:  }
```

Code patterns

To get the relevant code patterns for this problem, we analyze the output of pprof's `list` function as we did for the other issues. See Listing 4.9 for the output of the profiler.

From the Listing 4.9, we can see that most of the execution time is spent in line 74. Therefore we can identify the code in line 74 as problematic. As a result, we get the pattern `x += y` for which we can match in the source code. Another pattern to match is `x = x + "y"` because it can be used to concatenate strings as well. Therefore we get the following code patterns:

- `x += y`
- `x = x + y`

Categorization

For this problem, it is not easy to put it into a specific category, because it causes multiple SPAs at once. It is heavy on memory consumption as well as CPU usage. Further, it can block other processes from executing due to its heavy CPU usage, which could also fit the OLB antipattern. We chose to categorize it as OLB since it often occurs when reading or parsing files, which is a blocking operation.

Id	Antipattern	Category	Problem	Solution
P1	Frequent Object Creation	EDA	Frequently creating new objects causes a processing overhead due to unnecessary memory allocation.	Re-use already created objects if possible.
P2	Non compiled Regular Expressions	EP	Using non compiled regular Expressions in Go in expensive an requires recompilation for each use of the RegExp.	Use compiled Regular Expressions in Go or functionality of the `strings` package.
P3	String Concatenation	OLB	Concatenating strings in expensive as it causes frequent memory allocation/deallocation and copying the the string.	Use Buffers or the strings.Builder available in Go.

Table 4.1.: Overview of Performance Antipatterns in Go

Chapter 5

Antipattern Detection

This chapter discusses the detection process for different performance antipatterns. With the data and results of the previous chapter (Chapter 4), we can now build the automated detection tool. The general idea is, to automate the manual analysis process used in Chapter 4, so that one does not have to use the profiler pprof by hand to search the profile for performance problems. In Section 5.1, we explain the envisioned detection process in detail as well as the challenges and shortcomings of the idea.

5.1. An outline of the detection process

In this section, we describe the detection process in detail. We present the initial idea of the process in Section 5.1.1 and how we can reuse the patterns from the previous chapter. Further, we discuss how we envision integrating our detection tool into a software development process.

5.1.1. Initial Idea

The initial idea for the process was to reuse the previously defined *profile-* and *code-patterns* from Section 4.1 on page 23 and automate the process explained in Section 4.1.5. With the patterns, we can easily search the given profiles for potential matches, collect metrics, and output our findings. With the help of the *code-patterns*, we are able to trace a problem to the exact lines of code that are responsible for the issue. Further, the goal is to have one detection algorithm that can detect different SPAs depending on the input. Here, the inputs are the patterns.
The idea is to get an extensible application, where one can easily add new antipatterns

to the detection by implementing an interface and passing new *code-* and *profile patterns* to it.

To avoid possible "overfixing", some form of filtering is necessary. We define over fixing as follows: Solving problems that have little to no influence on the performance of an application and fixing parts of an application that are not in the direct control of the developer, for example, third party libraries.
The idea of avoiding over fixing is simple. We first filter the given source code with Go's Abstract Syntax Tree (AST) parser to get all user-defined functions. This way, we can consider only functions that are implemented by the developers of an application. Further, to avoid functions with little influence on the performance, we make use of pprof *topN* command. This command only outputs the top `N` (default: $N = 100$) time consuming or memory heavy functions. Now we can filter the *top* output of pprof by our user-defined functions and get their metrics. Those results serve as the base of the detection process.

With the filtered result, that contains the top slow functions, we can now move on to the actual detection. As mentioned above, we then search for the patterns in the profiler output. Here, we aim to automate the manual detection process of Section 4.1.5. To do this, we can use pprof's non-interactive versions of the *peek* command, which outputs callers and callees of a given function and *list* command that outputs the annotated source code of a function. Idea: For each slow function, get the output of pprof's *peek* and filter it by the *profile-patterns*. If a pattern matches, get the output of the *list* command and filter it by the code patterns. If we get another match, we have found an antipattern.

Generally, the idea for the detection of antipatterns can be summarized as follows:

1. extract functions from source files
2. filter TopN output of pprof by the above functions
3. get metrics of the functions (runtime, allocated memory etc.)
4. scan profile output of the functions for "Profile-Patterns"
5. scan code of the functions and their called runtime methods for "Code-Patterns"
6. report results

5.1.2. Integration

In software development, applications are often developed using agile methods and Continuous Integration (CI) as well as Continuous Deployment (CD). Our tool can be integrated into the CI process. Since the tool is a command-line application, it can be called from and integrated with virtually any testing process.
For example: On the build server, the instrumented source code is checked out from the repository, and the application is then built. After the build step, the application is deployed to a production-like environment, and load tests are run on the deployed software to produce the required profiles for our detection tool. Now, the profiles and source code are passed to our tool, which then produces a report. The report contains all information about the detected antipatterns, as well as the affected source code. With this information, the developers can then fix the problems. This process can be continued until all issues are cleared from the code. The schematic workflow of the integration is shown in Figure 5.1.

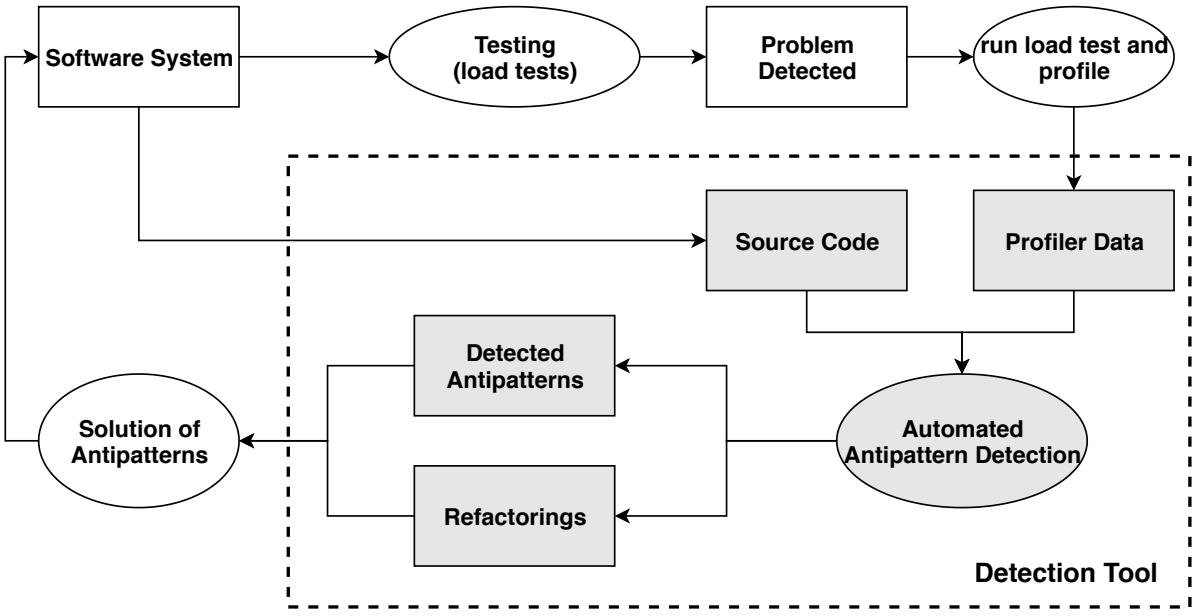


Figure 5.1.: Development process integration of our detection tool

5.2. Available Data

In this section, we discuss the data that is available for the detection. We show how the data can be used and the limitations of the available profiles.

		Antipattern		
Profile	Data	EDA	EP	OLB
CPU-Profile	function names	X	X	X
	function times	X	X	X
	call graph	X	X	X
Memory-Profile	allocated space			X
	allocated objects	X		

Table 5.1.: Overview of the used data

5.2.1. Profiles

As mentioned in the previous section, we primarily use the CPU profile and the memory profile produced by the `pprof` profiling library of Go. Both profiles contain information about the resource usage of the profiled application. While the CPU profile provides detailed information on the runtime of functions, call-trees, or the runtimes for each profiles line of code, the memory profile contains information about the memory usage of functions. This includes the number of allocated objects, as well as allocated memory in bytes. Chapter 4 shows how the profiler can be used to extract that information from the provided profiles. In addition to the interactive mode of pprof, which is used in Chapter 4, it also has a non-interactive or command-line mode, which prints the results to standard out. In the implementation, see Section 5.3, we make use of the command-line mode, because this way, we can call the pprof tool from within another application. Table 5.1 shows what data is used for which antipattern in the detection.

5.2.2. Limitations

Due to the availability of only two profile types, we are limited in our detection strategy. That means we are not able to detect blocking operations since we do not have the *blocking* profile available. Further, we can not detect problems regarding multi-threaded applications. We would require the *goroutine* profile to analyze problems that can occur in threaded applications. Another issue is that we do not have metrics regarding memory or CPU usage in percentage. For example, with our information, it is not possible to tell, that function x consumed $y\%$ of the CPU's time. The same applies to percentage-based memory usage. Another problem could be that some function might not get profiled at all, or only a limited amount of samples is collected. This can happen if one uses the profiling method two from Section 2.3.2 on page 11. With this option, one would know the possible causes of the problem to enable profiling before triggering the suspected action.

5.3. Implementation

In the implementation, the procedure from Section 5.1.1 from above is split into two parts or steps. These steps are explained in this section, and the respective pseudo-algorithms are shown in Algorithm 5.1 for the first step, and Algorithm 5.2 for the second step of the implementation.
To build a detection application, we first have to define the inputs that are required to implement the idea from Section 5.1.1. Our idea uses both, the CPU profile and the memory profile. Additionally, we require the path to the directory of the source files for our analysis. Hence, we have three parameters:

1. `cpuProfile` - the CPU profile
2. `memProfile` - the memory profile
3. `srcdir` - the path to source files

First (Section 5.3.1), items one, two, and three of Section 5.1.1 are done in the *main* function of the application. This happens before the actual antipattern detection. Here, the slow user functions are extracted and saved in an internal `files` array that holds the file's information. Each file has its filename, package name, and path as well as all slow functions that are found in this file. Further, each slow function contains its name, metrics (runtimes and memory allocations), and affected LOC. The second step of Section 5.3.2 checks for profile-patterns by checking the callers and callees of the slow function. Once a match is found, the code of the function is scanned for code-patterns. More details on the steps are explained in the following sections.

5.3.1. Extract and filter - Step One

First, we create an array (slice in Go) that holds all source files that can be found in the given source directory. The file's information is added to the respective file in the array. For each source file, the functions are extracted using the AST parser of the Go language and then saved to the matching file in the array.

Now, we filter the given CPU-profile by the functions of the files. For this task, we make use of pprof's `topN` command, which outputs the top slowest functions. Matching functions are then saved to a `SlowFunctions` array of each file. With this, we have an array of files, where each file stores information about its file name, the path to the file, package name, all functions defined in the file, and slow functions that were found while matching the profile.

Metrics like runtime and memory usage are now extracted from the given profiles and stored with each slow function of the `SlowFunctions` array. Runtimes are read from `topN` output of pprof and the memory statistics are read from the memory profile using the `-alloc_objects` and `-alloc_space` options. The antipattern detection is performed in the second step and works as follows.

5.3.2. Detect and report - Step Two

Each antipattern detection is implemented in its own package. The packages have two files, one for the detection and one for the intended refactoring. In the detection file, we implement the actual detection of the SPA. Each detection file has a detection function that performs that detection. That function has to be called for each antipattern-package to start the detection. For example, the detection function for the frequent object creation problem is called `DetectObejctCreation` and the containing package is called `objectcreation`.

Now, the detection is started by calling `objectcreation.DetectObejctCreation` from the `main` function of the application. The parameters for the detection function are the CPU profile, the internal files array, the filtered profile, and a boolean parameter to enable the use of an AST parser. The last parameter is not implemented yet and intended for future use. Since every detection package has a detection function, we have to call that function for each package to start the detection.

Each detection function will now loop over all of the `SlowFunction` of the files in the `files` array. For each of the functions, we get the callers and callees using the `peek` command of the pprof tool. The output is matched with the previous defined *profile-patterns* of Section 4.2. If there is a match, the detection then goes on to look for *code-patterns* in the source code of the function. For this, we utilize pprof's `list` command, which returns the annotated source code of the given function. If there is a match as well, the function is reported as an antipattern, with the previously gathered metrics.

The source code of the application can be found at [Sta19c] in the detection folder.

5.3.3. Report Generation

A report is generated for each detected antipattern. In this section, we describe how the reports are created. The report generation is called `PrintResults` in the pseudo-code in Algorithm 5.2 and will be explained in more detail in this section. Further, the generation of the refactoring feedback works analogously to the report generation.

Algorithm 5.1 Detection - Step One

```
procedure FilterProfileAndGetMetrics(cpuProfile, memProfile, srcFilesPath)
    files ← getGoFilesFromDir(srcFilesPath)
    for all f ∈ files do
        f.Functions, f.Package ← getFunctionsAndPackage(f.Path)
    end for
    pprofOut ← pprofTop(cpuProfile)
                                                    // filter profile and get cpu metrics
    filteredProfile, files ← getSlowUserFunctions(cpuProfile, pprofOut, files)
    getMemoryStats(memProfile, files)
end procedure
```

Algorithm 5.2 Detection - Step Two

```
procedure DetectAntipattern(cpuProfile, files, filteredProfile)
    for all file ∈ files do
        for all sFunc ∈ file.SlowFunctions do
            peekOut ← pprofPeek(sFunc, cpuProfile)
            pDetected ← detectProfilePatterns(peekOut)
            if pDetected then
                listOut ← pprofList(sFunc, cpuProfile)
                cDetected ← detectCodePatterns(listOut)
                if cDetected then
                    PrintResults(sFunc)
                    SuggestRefactoring(sFunc)
                end if
            end if
        end for
    end for
end procedure
```

To generate the reports, we use the templating feature of Go. A template can be understood as a predefined text with placeholders. The placeholders are then replaced with the given data, and a complete text can be reported to the user. An example can be found in Listing 5.1.

Explanation:
The constant `outputTempl` defines the template string. The placeholders of the template can be found between the curly brackets, for example, {{.FunctionName}}, which will be replaced by the respective property of the template data. In the last line, we create an "instance" of the template by parsing it. This way, the template can be reused with

Listing 5.1 Example of a report template

```
const outputTempl = ‘One-Lane Bridge (OLB) performance antipattern is detected
File {{.Filename}} on {{.FunctionName}} function, since the following bad practices are
found: total memory used {{.MemAlloc.Cum}} runtime
execution is {{.Timings.Cum}} seconds {{.Timings.CumPercent}} % of the captured execution
    time.‘

var outputTemplate = template.Must(template.New("example").Parse(outputTempl))
```

Listing 5.2 Example for template usage

```
buf := &bytes.Buffer{}
   if err := outputTemplate.Execute(buf, data); err != nil {
      fmt.Printf("%s", err)
      panic(err)
   }

fmt.Println(buf.String())
```

different data to generate multiple reports. An example of using a template is shown in Listing 5.2.

Explanation:
In line one, we declare a buffer that will hold the string after the template is filled with data. The second line shows how a template can be filled with data. The first argument of `Execute` is our buffer; it will receive the completed template. The second argument is the data to be used when filling in the placeholder from Listing 5.1. After calling `Execute` on the template, we can print the result to the console with `fmt.Println(buf.String)`. This is how we generate the reports for the detected SPAs.

5.4. Limitations and Problems

The detection has some limitations and problems. These issues are discussed in this section. First, we will look at general limitations. Afterward, we show the limitations and problems per antipattern.

5.4.1. General limitations

Firstly, the limitations from Section 5.2.2 apply globally to the whole application and the process in general. Since we only have two profile types available, we are very

limited in the detection. Certain SPAs can not be detected due to missing data. For example, bottlenecks due to race conditions or blocking operations can not be detected because the required profile type is not available to us. Another issue is that we do not have a percentage based usage of the CPU and memory. Due to this, we can not work with thresholds, as the authors of [Bra17] did. Instead, we would have to use some timing-based threshold. The problem with this option is that every program runs differently on other machines. Even on the same machine, the outcomes can vary drastically. Therefore, we do not use timing-based thresholds.

5.4.2. Process Limitations

Other limitations come from the detection process itself. Our filtering of the profile to only consider user-defined functions causes issues when the problem comes from third-party libraries. Those issues are simply undetected since they fly under the radar with our process. While they might appear in the CPU and memory profile, we filter them out in our process. That causes us to miss potential SPAs in the given application. Further, with the current process, we are only able to detect problems that match our pre-defined *profile-* and *code-patterns* from Section 4.2 on page 26. If we want to detect more antipatterns, the manual inspection of Section 4.1.5 has to be done for every additional antipattern we want to find.

Additionally, since we use the same detection process for every antipattern, we cannot address all the possible differences and characteristics for each antipattern. Using the same process helps the extensibility of the detection tool but also limits its capabilities to detect more complex SPAs like architecture-related patterns that involve many parts of the application.

5.4.3. One-Lane Bridge

In the previous Chapter 4, we categorized the string concatenation as OLB. Our reasoning is that it represents the extensive us of a single blocking resource. One could make strong arguments against this categorization. For example, the OLB generally refers to problems that can be solved by using a shared resource-principle. This applies to database access but not access to a single variable where order matters.

While we know how much memory is used by the function that contains the antipattern, we can not report how many strings are concatenated or how long the resulting string is. That information is not available from the profiles, although it would be helpful for further investigation.

5.4.4. Extensive Processing

Currently, we can detect one code-based form of this problem, only. Further, automatically detecting not needed processing is almost impossible. To be able to detect other EP instances, we would have to know the intentions of the developer. Meaning, we would need some semantics extraction tool to detect other forms of these problems. To be able to detect further instances, one has to define the *profile-* and *code-patterns* for every new instance.

Further, it is not clear which metrics are helpful for the report. In our case, we report the number of function calls of the `compile` function for regular expressions since this is the cause of the EP. But with other problems, it is not as clear what exactly is the cause of the problem. This has to be manually inspected beforehand, and the report generation has to be adjusted for other problems.

5.4.5. Excessive Dynamic Allocation

Sometimes it is impossible to avoid dynamic allocations due to the design of the software or the used language. For example, in Go everything is “pass by value” unless one passes a pointer to a function. While this helps with functional programming and immutability, it leads to more allocations of new objects on the heap. Even with pointers as a parameter, Go creates a new variable with the object’s address as its value. Defining a threshold for the number of created objects is too arbitrary since the performance varies between systems. For example, a certain amount of object creations might cause a problem on one machine while it can be a lot less impacting on another, faster machine.

5.5. Tool implementation details

In this section, we show some screenshots of the detection tool and explain how to use it. The tool can be found at [Sta19c]. To use the tool, it has to build first. At the moment, we do not provide binaries of the software.

Please take note that we only provide the necessary commands for Linux based systems with bash as the standard shell.

5.5.1. Building the Tool

To build the tool, one has first to check out the source code from the repository. This can be done by pasting the following line into your terminal and pressing enter.
Note: This will clone the repository into the current working directory and create a go-antipatterns folder.

```
git clone https://github.com/nstadelmaier-dev/go-antipatterns.git
```

Now, change into the `go-antipatterns` folder and then change into the `detection` folder. Use the following commands:

`cd go-antipatterns/detection` or
`cd go-antipatterns` followed by `cd detection`

The tool can now be built by typing:

```
go build .
```

This will create a `detection` executable in the current directory.

5.5.2. Using the Tool

Before one can use the tool, make sure the CPU profile, memory profile, and source code of the application are available. We assume that the `detection` folder as the current working directory. Now, one can call the previously created binary and pass the required arguments to the detection tool. As mentioned in Chapter 5 the application has three parameters. The `cpuProfile`, `memProfile` and `srcdir`. To redirect the output of the application from standard out to a file use the > operator, i.e. `command > file`.

To start the detection use the following command:

```
./detection -cpuProfile=<cpuProfile> -memProfile=<memProfile> -srcdir="srcDir"
```

Please note the quotation marks around the path of the `srcdir` parameter.

The program will now automatically detect the antipatterns using the provided profiling data and output the results on the console. To save the report in a file, use the following command:

```
./detection -cpuProfile=<cpuProfile> -memProfile=<memProfile> -srcdir="srcDir" > report.txt
```

An example output of the tool is shown in Figure 5.2. In the output you can see that the tool outputs which detection is currently running, which files and functions are

```
1
2   Detecting string concat started...
3   Scan file p1.go
4   Analyze Function: P1_2createObjectsBad
5   Analyze Function: changeObj
6   Analyze Function: P1createObjectsBad
7   Analyze Function: S1_3reuseObjectsGoodCPool
8   Analyze Function: S1reuseObjectsGood
9   Analyze Function: S1_2ReuseObjectGod
10  Scan file p2.go
11  Analyze Function: P2regularRegExpBad
12  Analyze Function: S2compliedRegExpGood2
13  Analyze Function: S2compliedRegExpGood
14  Scan file p3.go
15  Analyze Function: P3stringConcatBad[      60ms   1.03mins     74:       concString += "a" concString]
16
17  detected String Concatenation
18  File: p3.go Function: {P3stringConcatBad {0.10s 0.031% 16.97% 61.81s 18.97%} {{493.70GB 81.99% 81.99% 493.72GB 81.99%}
19  Affected Code:
20  [      60ms   1.03mins     74:      concString += "a"]
21
22  [74]
23  One-Lane Bridge (OLB) performance antipattern is detected
24  File p3.go on P3stringConcatBad function, since the following bad practices are
25  found: total memory used 493.72GB runtime
26  execution is 61.81s seconds 18.97% % of the captured execution time.
27
```

Figure 5.2.: Output of the detection tool

analyzed in the process. If it is used on the command line, this provides decent progress information, as it informs the user of the current status. By redirecting the output to a file, the progress information is lost, because it is written to the file. This behavior is subject to change. The report shows which antipattern is detected, see line 17 of Figure 5.2, the file that contains the antipattern in line 18 and the affected LOC in line 20 of the output. The last part of the report is the categorized antipattern; in this case, OLB is detected. For the OLB the report also shows the total memory usage and runtime of the function that is causing the problem.

Chapter 6

Antipattern Solutions

Since we want to suggest refactorings to the user, we have to research which refactorings provide a performance increase for the detected problems before suggesting it to the user. This chapter is complementary to Chapter 4 on page 23, which deals with the initial problems and defines how they can be detected. In contrast to Chapter 4, this chapter provides solutions to these problems. In the following sections, we show the found solutions for each antipattern and explain how we measure the execution time of functions. The implemented solutions are based on the findings of [MG18], Stackimpact and [Gry16]. Firstly, our measurement methodology is explained in Section 6.1, followed by the solutions for problem one (EDA) in Section 6.2, problem two (EP) in Section 6.3 and problem three (OLB) in Section 6.4. Lastly, we then discuss refactoring of Go, and it's problems in Section 6.5. Finally, an overview of the solutions is provided in Section 6.6.

6.1. Measuring the execution time of functions

To measure the execution time of a function, we get the time at the beginning of the function and a measurement after the function returns. The first measurement is then subtracted from the second measure. This gives us the runtime of the function in question. In order to help with this process, we implemented a `timeTrack` function. This function does the calculation and extracts the name of the function to measure from the callers provided by `runtime.Caller()`. Further, it adds the measurements to an internal map to keep track of all measurements.

The `timeTrack` functions logs the runtime of its calling function, here the `S1reuseObjectsGood` function. We take advantage of the `defer` keyword that causes a function to be executed after its containing function returns. This way, we can leave

Listing 6.1 Implementation of the `timeTrack` function [Moh17]

```
func timeTrack(start time.Time, prefix string) {
   // nameLength := 30
   elapsed := time.Since(start).Seconds()

   // Skip this function, and fetch the PC and file for its parent.
   pc, _, _, _ := runtime.Caller(1)

   // Retrieve a function object this functions parent.
   funcObj := runtime.FuncForPC(pc)

   // Regex to extract just the function name (and not the module path).
   runtimeFunc := regexp.MustCompile(`^.*\.(.*)$`)
   name := runtimeFunc.ReplaceAllString(funcObj.Name(), "$1")
   // for i := len(name); i < nameLength; i++ {
   //   name += " "
   // }
   var fName = ""
   if prefix != "" {
      fName = prefix + " " + name
   } else {
      fName = name
   }

   measaures[fName] = append(measaures[fName], elapsed)
}
```

the runtime calculation to one single function, and the calculation will not impede the runtime of the function we want to measure. The implementation shown in Listing 6.1 follows the solution provided by user Mohsin of [Moh17].

6.2. Excessive Dynamic Allocation (P1)

In the following sections, we show the solutions for the EDA performance antipattern. We present two solutions for each variant of the initial problem. All solutions focus on minimizing the creation of new objects by reusing already existing objects. The solutions for the first version of the problem are presented in Section 6.2.1, and the solutions for the second version are found in Section 6.2.2.

6.2.1. Solutions for P1 variant one (P1_1)

For the first variant of the problem – P1 –, we implemented two solutions to solve the problem. The solutions are explained in the following sections.
In order to solve the problem, we followed the suggested solution from [MG18] and the generally suggested solution for the antipattern shown in Table 2.1. Those solutions aim to "recycle" objects and make use of a pool of already created objects to alleviate the creation of new objects.

Solution One (S1_1_1)

The function `S1reuseObjectsGood()` implements the first solution for the problem P1. In this implementation, we make use of Go's **sync.Pool**. The pool is a set of temporary objects that can be saved and retrieved from the pool [Goo20d]. The purpose of the pool is to manage a group of temporary items that are shared across concurrent clients. Further, the cost of allocations is amortized by the implementation of the `snyc.Pool`. Listing 6.2 shoes the implementation of the solution.

Explanation of `sync.Pool`:

In our implementation, we modify a rather large amount of objects, 1 million in our case, to test the effectiveness of the `Pool`. According to the Go documentation, the `Get` functions works as follows:

Get selects an arbitrary item from the pool, removes it from the pool, and returns it to the caller. Get may choose to ignore the pool and treat it as empty. Callers should not assume any relation between values passed to Put and the values returned by Get [Goo20d].

That means we will most likely not have one million objects creations since the function either returns an already created object or create a new one at random. Therefore, we expect an improvement in the performance of the function and a reduced amount of created objects.

Solution Two (S1_1_2)

The second solution implemented in `S1_2ReuseObjectGod` of Listing 6.3 does not make use of a pool a all. Instead, we create a single instance of a `book` that is then passed to a function `changeObj`, which changes some properties of the object. The idea is to reduce the initial object creation by reusing a single instance of the same object. While this solution prevents the initial creation of many objects, the Go language will create copies

Listing 6.2 Solution one (S1_1_1) for problem one variant one (P1_1)

```
func S1reuseObjectsGood() {
   defer timeTrack(time.Now(), "S1.1.1")

   for n := 0; n < 1000000; n++ {
      book := pool.Get().(*Book)
      book.Title = "The Art of Computer Programming, Vol. 1"
      book.Author = "Donald E. Knuth"
      book.Pages = 672

      pool.Put(book)
   }
}
```

Listing 6.3 Solution two (S1_1_2) for problem one variant one (P1_1)

```
func S1_2ReuseObjectGod() {
   var book Book
   defer timeTrack(time.Now(), "S1.1.2")
   book = Book{
      Title: "The Art of Computer Programming, Vol. 1",
      Author: "Donald E. Knuth",
      Pages: 672,
   }
   for n := 0; n < 500000; n++ {

      _ = changeObj2(book, n)

   }
}
```

of the object that is passed to `changeObj`. That is true for every variable that is passed to a function. See the Go documentation [Goo20d] for more details on the language's design. Overall we expect a reduction in runtime and less created object.

6.2.2. Solutions for P1 variant two (P1_2)

In the following section, we show the solution to the second variant of problem one (P1_2). Again, we intend to "recycle" already existing objects, to reduce the number of object allocations. In the first example, we use a custom pool, and in the second, we implemented a flyweight pattern to reduce the object creation.

Listing 6.4 Solution one (S1_2_1) for problem one variant two (P1_2)

```
var customPool = make([]*Book, 1000, 1000)

//prepare books pool

for i := 0; i < 1000; i++ {
   customPool[i] = &Book{
      Title: "The Art of Computer Programming, Vol. 1",
      Author: "Donald E. Knuth",
      Pages: 672,
   }
}

func S1_2_1reuseObjectsGoodCPool() {
   defer timeTrack(time.Now(), "S1.2.1")

   for n := 0; n < 1000000; n++ {
      book := customPool[n%1000]
      changeObj(book, n)
   }
}
```

Solution one (S1_2_1)

In this solution, we implemented a custom pool that holds the initial objects intended for reuse. The pool has no logic; it is a simple slice that holds the objects we want to reuse in the `S1_2_1reuseObjectsGoodCPool` function. We initially pre-populate the pool with 1000 books.

The idea is that these 1000 books can be used later at any time in the application, and we only have one cycle of object creations at the initialization of the pool. In this case, we have 1000 objects, instead of the 1000000 created by the P1_2 function. Listing 6.4 shows the implementation of the pool and the solution function `S1_2_1reuseObjectsGoodCPool`.

Solution two (S1_2_2)

For this solution, we chose to implement the flyweight design pattern, as suggested in Table 2.1. The flyweight pattern aims to reduce the number of created objects in an application in order to increase the performance and reduce the memory consumption. In the function `S1_2_2reuseObjectsGood` from Listing 6.5 we demonstrate the use of the flyweight pattern.

Listing 6.5 Solution one (S1_2_2) for problem one variant two (P1_2)

```
func S1_2_2reuseObjectsGood() {
   defer timeTrack(time.Now(), "S1.2.2")
   factory := NewFlyweightFactory()

   for n := 0; n < 1000000; n++ {
      book := factory.GetFlyweight("The Art of Computer Programming")
      _ = book
   }
}
```

The flyweight pattern minimizes memory usage by sharing as much data as possible with other objects. This is achieved by reusing similar, already existing objects and only creating new objects if no matching object is found [Gam95].

Our implementation is a simplified version of the flyweight pattern and follows the implementation of [Wel19]. We do not use the intrinsic and extrinsic properties exactly as they are defined in the pattern of [Gam95]. Our intention is to show the potential benefits of applying the flyweight pattern. We don't intend to provide a textbook flyweight implementation. The complete implementation of the pattern can be found in Listing A.1 on page 93.

6.3. Extensive Processing (P2)

In this section, we discuss the solutions for the second problem (P2), which represents an EP antipattern. P2 focuses on the use of regular expressions and how to use them more efficiently in the code. The approaches focus on reusing regular expressions to avoid repeated compilation or avoid them completely in certain situations. The solutions for variant one are shown in Section 6.3.1, solutions for the second variant in Section 6.3.2.

6.3.1. Solutions for P2 variant one (P2_1)

As with the previous problem, we implemented two solutions to evaluate whether they are effective in improving the performance of the problem. Our solutions are based on the findings of [MG18] and adopted to fit our benchmark program.

Listing 6.6 Solution one (S2_1_1) for problem two variant one (P2_1)

```
func S2_1_1compliedRegExpGood() {
   r, err := regexp.Compile(testRegexp)
   if err != nil {
      panic(err)
   }
   email := []string{"jsmith@example.com", "jsmithexample.com", "jsmith@example.com",
       "jsmith@example.com", "jsmithexample.com", "jsmith@example.com"}

   defer timeTrack(time.Now(), "S2.1.1")
   for n := 0; n < 100000; n++ {
      r.MatchString(email[n%6])
   }
}
```

Solution One (S2_1_1)

To solve the extensive processing that stems from repeatedly compiling the regular expression, we use a modified version of the initial problem (P2_1). In the implementation of `S2\_1\_1compliedRegExpGood` in Listing 6.3, the regular expression is compiled once at the start of the function. We then reuse the compiled Regex in the loop to do the matching of the email addresses. By applying this solution, we intend to improve the performance of the function and reduce the number of compilations.

Solution Two (S2_1_2)

In the original problem, we want to validate email addresses. We solved this by repeatedly matching the input, the elements of the `email` slice, with a regular expression. The Regex is compiled in every iteration of the original problem, which causes a performance problem.

To solve the problem, we implemented the following idea:

We only match the input with the regular expression, if it contains an @ symbol. The intention here is to filter out some of the non-email inputs and not match them at all, in order to reduce the number of compilation runs. Generally, email addresses contain an @ symbol. Therefore, we assume that if the input also contains the @ symbol, we can continue to validate it. By applying the conditional matching, we hope to increase the performance of the function. The Listing 6.7 displays the implementation of the solution.

Listing 6.7 Solution two (S2_1_2) for problem two variant one (P2_1)

```
func S2_1_2compliedRegExpGood2() {

   email := []string{"jsmith@example.com", "jsmithexample.com", "jsmith@example.com",
       "jsmith@example.com", "jsmithexample.com", "jsmith@example.com"}

   defer timeTrack(time.Now(), "S2.1.2")

   for i := 0; i < 100000; i++ {
      if strings.Contains(email[i%6], "@") {
         r, err := regexp.Compile(testRegexp)
         if err != nil {
            panic(err)
         }
         r.MatchString(email[i%6])
      }
   }
}
```

6.3.2. Solutions for P2 variant two (P2_2)

The second variant of the problem uses a Regex to determine whether a given input contains certain words. This version also uses the problematic `regexp.MatchString` method, which causes the compilation of the regular expression with each use of the function.

Solution One (S2_2_1)

Solution one for this variant intents to solves the problem by pre-compiling the regular expression and reusing it whenever it is required. This approach corresponds to the approach used in the first solution for the first variant of the problem, see Section 6.3.1. This should substantially reduce the number of compiles and improve the runtime of the function `S2_2_1ContainsRegExp` of our implementation in Listing 6.8.

With this solution at hand, we ask the following question: Is a regular expression required to do this task? Or can we solve it differently to avoid regular expression where we possibly don't need them. More on this thought can be found in the next solution.

Listing 6.8 Solution one (S2_2_1) for problem two variant two (P2_2)

```
func S2_2_1ContainsRegExp() {
   re := regexp.MustCompile('first|second|third')
   examples := []string{"first ex", "second exp", "test"}
   defer timeTrack(time.Now(), "S2.2.1")

   for i := 0; i < 100000; i++ {
      for _, i2 := range examples {
         re.MatchString(i2)
      }
   }
}
```

Listing 6.9 Solution two (S2_2_2) for problem two variant two (P2_2)

```
func S2_2_2ContainsStrContains() {
   examples := []string{"first ex", "second exp", "test"}
   defer timeTrack(time.Now(), "S2.2")
   for i := 0; i < 100000; i++ {
      for _, i2 := range examples {
         ContainsSubstring(i2, "first", "second", "third")
      }
   }
}
```

Solution Two (S2_2_2)

To pick up on the question of the previous section, we further investigated the problem and came up with the following answer: For this particular use case, one does not need a regular expression to perform the matching. Therefore, we implemented a solution that abstains from the use of regular expressions.

Go's `strings` package provides a function called `Contains(s, substr string)` that returns if a string `s` contains a substring `substr`. Since this is exactly the functionality that the initial problem implements by making use of regular expressions, we can use the contains function to solve the problem differently.

In our second variant of the solution shown in Listing 6.9, we implemented a helper function called `ContainsSubstring(str string, subs ...string)`, which uses the contains function from the strings package to match the given inputs. It returns, whether the string str contains any of the substrings subs.
The implementation of `ContainsSubstring` can be found in Listing A.2.

Listing 6.10 Solution one (S3_1_1) for problem three variant one (P3_1)

```
func S3_1_1stringConcatGood(strLength int) {
   var concString strings.Builder
   defer timeTrack(time.Now(), "S3.1 "+strconv.Itoa(strLength))
   for i := 0; i < strLength; i++ {
      concString.WriteString("a")
   }
   concString.String()
}
```

6.4. One-Lane Bridge (P3)

This section explains the solutions for the third problem, the OLB antipattern. The solutions try to reduce the memory overhead and improve the performance by introducing buffers to the original problems. Section 6.4.1 shows the solutions for the first variant, and Section 6.4.2 shows the solutions for the second version of the problem P3.

6.4.1. Solutions for P3 variant one (P3_1)

In the problem, we concatenated many characters to build a big string. We found that the function was running slow, as in, it took the most time in our test system to complete. As with the other problems, we implemented two solutions for each variant of the problem. The solutions for the first version are discussed in the following sections.

Solution one (S3_1_1)

The blocking part in this problem in the actual string variable and how the Go language handles the concatenation of strings internally. To solve the problem, we make use of a “buffer”, which can accept input at a much faster rate. Instead of allocating new memory for the new string created with each iteration, the buffer only stores the actual values. The concatenation to a single string is done only once after we added all of the characters to the buffer.

Golang provides a buffer specifically for building strings. It is called `strings.Builder`. We use the builder to build the string efficiently and reduce the memory overhead. Instead of using the builder, one could also use the `bytes.Buffer`, which accepts any byte input like a character. The solution can be found in Listing 6.10. We chose the `strings.Builder` because it was intended for this exact case.

Listing 6.11 Solution two (S3_1_2) for problem three variant one (P3_1)

```
func S3_1_2stringConcatGoodCBuffer(strLength int) {
   var concStringBuf = make([]string, strLength, strLength)
   defer timeTrack(time.Now(), "S3.2 "+strconv.Itoa(strLength))

   for i := 0; i < strLength; i++ {
      concStringBuf = append(concStringBuf, "a")
   }
   _ = strings.Join(concStringBuf, "")
}
```

Solution two (S3_1_2)

In the second solution to problem P3_1, we use a custom buffer in an attempt to speed up the functions runtime. In this case, the "custom-buffer" is a simple slice (dynamic array in Go) of type `string`. Since the implementation of slices is very efficient and appending to a slice amortizes well, we think one can take advantage of this and re-purpose a slice to buffer of strings.

Instead of using any special buffer or builder, we now use a slice and append single characters to it, to build a big string at the end of the function. We hope to see an increase in performance since, in theory, we should have less memory usage using a slice. The implementation of the solution is shown in Listing 6.11. As we can see from the code, we create a slice with the length and capacity of our desired string length. Then we append `strLength` "a" characters to the slice and finally build the entire string with a call to `strings.Join()`.

6.4.2. Solutions for P3 variant two (P3_2)

We chose a more realistic implementation for the section variant of the problem three because the original problem of this variant is a realistic scenario as well. In the problem three variant two, we read many lines from a file on the file-system. Those lines are then concatenated to rebuild the file in memory.

Solution one (S3_2_1)

We decided to test the effectiveness of the `strings.Builder` with a more realistic scenario. That is why we implemented the first solution with the same builder from the first solution to problem P3_1. Just as with solution one to the first version of problem three,

Listing 6.12 Solution one (S3_2_1) for problem three variant two (P3_2)

```
func S3_2_1stringConcatGood2(strLength int) {
   file, err := os.Open("./AutoGen1.txt")
   if err != nil {
      log.Fatal(err)
   }
   var buf strings.Builder
   scanner := bufio.NewScanner(file)
   defer timeTrack(time.Now(), "S3.2.1 "+strconv.Itoa(strLength))
   for i := 0; i < strLength; i++ {
      scanner.Scan()
      buf.WriteString(scanner.Text())
   }
   _ = buf.String()
}
```

we expect an increase in performance for the function `S3_2_1stringConcatGood2` of Listing 6.12. Instead of building a string of the size `strLength`, we read `strLength` lines from the file *AutoGen1.txt*. The file contains a little over 70000 lines with 78 characters.

Solution two (S3_2_2)

Again, we chose to try the second solution to the previous problem in a more realistic scenario. In the implementation (Listing 6.13) of the solution, the `concStringBuf` variable is our buffer that we use to build the file contents. The buffer is a slice of the type string. With this implementation, we want to test how the slice behaves when larger strings are appended instead of single characters. As another alternative, we could have chosen a `bytes.Buffer` as a buffer for the file's content. We decided against this alternative to be able to compare the effectiveness of the second solutions for the variants of P3. If this version shows an increase in performance, we can safely suggest it as a refactoring.

6.5. Refactoring Go

Automatically refactoring source code can be very challenging. In this section, we show some of the available refactoring tools for Go (Section 6.5.1) and discuss the challenges we faced while trying to automate the refactoring in Section 6.5.2.

Listing 6.13 Solution two (S3_2_2) for problem three variant two (P3_2)

```
func S3_2_2stringConcatGood2(strLength int) {
   file, err := os.Open("./AutoGen1.txt")
   var concStringBuf = []string{}
   if err != nil {
      log.Fatal(err)
   }
   scanner := bufio.NewScanner(file)
   defer timeTrack(time.Now(), "S3.2.2 "+strconv.Itoa(strLength))
   for i := 0; i < strLength; i++ {
      scanner.Scan()
      concStringBuf = append(concStringBuf, scanner.Text())
   }
   _ = strings.Join(concStringBuf, "")
}
```

6.5.1. Refactoring tools for Go

To language provides some refactoring tools out of the box. Further, there are third-party tools that can be used to refactor Go code. Most of the available tools focus on simpler tasks like renaming a variable, function, or field of a struct, while others provide more sophisticated possibilities. These tools are explained in the following sections.

gofmt

[Goo20d] Generally, *gofmt* formats Go programs. Additionally it provides the functionality to refactor a codebase when the `-r` flag is provided. The refactoring works as follows:

```
pattern -> replacement
```

One has to provide a pattern and a replacement for said pattern, where both are valid Go expressions. An example from the go documentation shows how to use gofmt to convert explicit slice upper bounds to implicit ones. See the following example:

```
gofmt -r 'a[b:len(a)] -> a[b:]'  -w $GOROOT/src [Goo20d]
```

This enables us to rename variables or fields of structs or replace certain code with the provided replacement. The problem with gofmt is that if one has to do more sophisticated refactoring like introducing new variables and types to a function and then use those newly created types in other parts of the application, there is no option to perform this task with the simple pattern and replacement approach. One would have to

change the source file first and introduce new types or variables and then use multiple runs of *gofmt* to achieve the wanted results.

gorename

Another refactoring tool is the *gorename* application. As the name suggests, its main purpose is to rename parts of the code, for example, function or fields of a go source file [GoD15]. While it can perform type-safe renaming, it cannot introduce new code-structures into the source file.

Example usage: `gorename -from '"bytes".Buffer.Len'-to Size` [GoD15]

This example from the rename documentation renames the `Len` function of the `Buffer` to Size. The problem with the tool is that it only supports renaming. For our problems, we require more source code transformations that a simple rename of a field. We have to introduce new variables and additional code to solve the problems. Therefore, the gorename tool does not work for us.

eg

[GoD17] The *eg* tool is an example-based refactoring tool for go. It uses template files, which are go source files, to perform refactoring tasks. A template file implements a `before` and `after` function, where the before function defined the examples to match and the `after` function defines the transformations of the code. An example for a template form the tools help command is shown in Listing 6.14 [GoD17].

Listing 6.14 Example for the eg tool

```
package P
    import ( "errors"; "fmt" )
    func before(s string) error { return fmt.Errorf("%s", s) }
    func after(s string) error { return errors.New(s) }
```

The tool can only change one expression with another. Arbitrary statements and expressions cannot be refactored with, e.g., Further, it is not possible to replace an expression of one type with an expression of another type [GoD17]. This is a problem for us since we have to introduce new types and code structures into the source code to apply our refactorings.

6.5.2. Challenges

Renaming variables or functions is a rather trivial problem with refactoring. The available tools from the previous sections support such tasks very well. Changing the structure of the code is very challenging and not fully supported by the tools that we revived earlier. One could try a multi-step approach for changing the source code with *eg* or *gofmt*. Another problem is that some tools are not well documented, which leads to a lot of trial and error when using the tools. These problems make it difficult to develop an automated refactoring tool, to apply the suggested refactorings after the detection. In our case, we did not succeed in developing an automated refactoring approach.

Some of the refactorings are too substantial and require big structural changes to the code. They may even change the architecture of the application, like the application of the flyweight pattern in the second solution to problem P1_2, see Section 6.3.1. For such cases it may be better, to leave the refactoring to the developer and architect of the given application to avoid unwanted side-effects.
In summary, while there are multiple refactoring tools for Go that support renaming and example-based refactoring, introducing new types and structural changes to the code remains very challenging.

6.6. Antipattern Solutions - Overview

The table 6.1 shows an overview of the identified problems and their solutions.

Problem	Antipattern	Pattern(Profile)	Pattern(Code)	Refactoring
P1	EDA	runtime.newobject runtime.mallocgc runtime.heapBitsSetType runtime.nextFree(Fast)	var a = b{} var a = &b{} var a = new*	use `sync.Pool`, a custom pool implementation, or the flyweight design-pattern
P2	EP	regexp.MatchString regexp.Compile regexp.compileOnePass regexp/syntax.Parse	.MatchString() .Match() .MatchReader()	use regex.MustCompile(), regex.Compile(), or avoid regex if not needed
P3	OLB	runtime.concatstring runtime.concatstring2 runtime.mallocgc	s = s + "string" | s = s + a s += "a" | s += a	use string builder var a strings.Builder() a.writeString("a") | a.writeString(a) b = a.String() or a slice as buffer e.g. a = []string append(a, "string")

Table 6.1.: Overview of Performance Antipatterns and Solutions

Chapter 7

Evaluation

In this chapter, we present the evaluation of our developed approach. This includes the antipattern solutions and automated detection tool. We analyze whether the suggested refactorings provide an increase in performance and the effectiveness of our implemented detection tool. Section 7.1 describes our testing methodology, and Section 7.2 shows the research questions we want to answer in the evaluation. Further, the experiment settings are described in Section 7.3, and finally, the results are discussed in Section 7.4.

7.1. Methodology

Analog to the methodology of Section 4.1.1, we implemented some of the solutions suggested by [MG18], [Sta19a] and [Gry16]. Further, we implemented our own variants of the solutions to provide more potential refactorings. The solutions are implemented in our benchmark application from Section 4.1.2, which can be found at [Sta19b] in the benchmarks folder. Each benchmark is run 100 times to get an average runtime of the implemented solutions. We run the solutions several times because the runtime of an application or function can vary drastically with each run. By repeating it several times, we get a much more meaningful average runtime and reduce the influence of outliers in the collected runtime samples. We then compare the average runtimes of the problems and solutions to conclude whether the solution provides a performance improvement or not. Additionally, we use the gathered data, function runtimes, and profiles, as well as manual inspection of the results, to determine if the detection tool works as intended.

7.2. Experiment Goals

First, we want to evaluate if the implemented solutions provide a performance improvement for the problems in Chapter 4. Secondly, we want to evaluate the effectiveness of our detection tool. Lastly, we are interested in possible improvements for the detection. To evaluate these goals, we want to answer the following questions:

RQ1: Do the provided refactorings show a performance increase?

This question is essential to our approach since we want to suggest possible refactorings for the antipatterns of Chapter 4 on page 23. The research question one will be evaluated for each implemented solution per antipattern and can be found in Section 7.4.

RQ2: Does the developed tool detect the antipatterns correctly?

With this question, we intend to determine if our developed approach works correctly. Meaning, we want to evaluate if the detection tool is effective in finding the antipatterns of the test application.

RQ3: How can the detection be improved?

Lastly, we are interested in finding how the detection can be improved. To answer these questions, we evaluate the findings of the previous questions and suggest possible improvements for the approach.

7.3. Experiment Settings

For our tests, we use the most recent version of the source code in [Sta19b] at the time of writing. The compiled and built source code produces the benchmark application binary. The benchmark is run on a desktop machine with an i5 3750k CPU @ 4.5 Ghz and 8 Gb of ram. The following data is used to analyze the results:

1. cpu profile - generated from the benchmark
2. memory profile generated - from the benchmark
3. runtimes of the functions in question - recorded from the benchmark
4. output of the detection tool

The baseline for the comparison is the problems introduced in Chapter 4. We use the recorded runtimes and metrics from the recorded profiles to analyze, if the suggested solutions show an increase in performance and a decrease in resource utilization.

The conducted test scenarios to answer the research questions are the following:

1. In the first test scenario, every function gets run 100 times, and all runtimes are recorded and logged to a file. This is done for the following reason. The runtime of a function can vary a lot between runs, depending on the scheduler or the assigned CPU core, amongst other influences. By running multiple repetitions, we aim to decrease the influence of outliers and obtain a better average of the runtime. We then compare the mean and median runtimes of the problem and the implemented solution to conclude whether we can observe an improvement in performance and resource use. This scenario is used to evaluate if we can observe a performance increase in our solutions.
2. In a second scenario, we artificially introduce false positives into our benchmark to test if our detection will identify these problems as antipatterns when they should not be detected. The false positives are created by inserting *code-patterns* of one problem into the code of another problem that has a different set of antipatterns. When running the benchmark with the new functions, they create different, or rather additional *profile-patterns* of other problems. With this scenario, we want to evaluate if the detection works correctly and what some possible improvements are. For this purpose we inserted an EDA *code-pattern* into an EP instance, secondly we inserted an EP pattern into an OLB instance and lastly inserted an OLB patterns into an EDA instance.

Running each scenario creates profiling traces. Those traces serve as the base for the evaluation. Additionally, we use the recorded function runtimes as described in Section 7.1.

7.4. Experiment Results

This section shows and discusses the results of the experiment and answers the research questions. We describe the results for each antipattern and scenario in the Sections 7.4.2 to 7.4.4. Before that, we show how we evaluated the recorded runtimes of the function in the benchmark in Section 7.4.1. An overview of the detection results and metrics is provided in Table 7.1.

7.4.1. Evaluating the function runtimes

In order to conclude whether the refactorings provide an improvement, we have to perform statistical testing on the recorded runtimes. From our records, we could see that the data is not always normally distributed. Meaning, the distributions of the samples are different. We used QQ-plots to test for normality of the samples and the KS-tests to test for normality. Additionally, the KS-test was used to test the samples for a difference in the distribution. The solution can be equally distributed, while the problem can be normally distributed. Those distributions can change with each run of the benchmark. So we cannot assume that the distribution for each function stays the same.

Since we have to deal with different distributions, we decided to use the Mann–Whitney or U-test, to test our samples for differences in the populations. With the Mann-Whitney test, we can evaluate whether one of two random variables is stochastically larger than the other [MW47]. We use a one-sided, two-sample test, to test for a stochastical difference in the samples.

Definitions

The samples of the problems are referred to as X.
The samples of the solutions are referred to as Y. The chosen significance level is $\alpha = 0.05$.

We keep the numbering scheme for the problems and solutions, as introduced in Section 4.2.1 on page 27.

For example:

X_{1_1} refers to the samples of P1_1.
$Y_{1\_1\_1}$ refers to the samples of S1_1_1.

We use the R statistics language for the computations. R implements this test in its *statistics* package as a Wilcoxon Rank-Sum Test. This test is equivalent to the U-test [Bau72].

To analyze our solutions, we formulate the following general hypothesis:

(7.1) $H_0 : A = Y$

There is no location shift in the distribution of A and B. Meaning the distributions are equal, and we do not have an increase in performance.

(7.2) $H_1 : A > Y$

Distribution A is shifted to the right of distribution B. Therefore, we have an increase in performance.

7.4.2. Excessive Dynamic Allocation (P1)

Here, we present the results of the scenarios from above. First, the evaluation scenario one is shown, followed by the evaluation of scenario two.

Scenario one

This test is run to decide if the solutions provide an increase in performance to the problem. In the following, we show the results from the first scenario.

Solutions for P1_1:

Solution one (S1_1_1)

Hypothesis:

$H_1 : X_{1\_1} = Y_{1\_1\_1}$

$H_0 : X_{1\_1} > Y_{1\_1\_1}$

$p - value < 2.2e - 16(Computer); \quad 2.2e - 16 < \alpha$

The p-value is smaller than the significance level $\alpha = 0.05$. Therefore, we reject the null hypothesis H_0 and accept the alternative hypothesis H_1.

It follows that we can observe an increase in performance for *S1_1_1*. Consequently, *S1_1_1* provides a valid solution to the problem P1_1. The average runtime of P1_1 is recorded at 0.0607 seconds, where the solution S1_1_1 took only 0.0186 seconds, which is a reduction of 69,36 %.

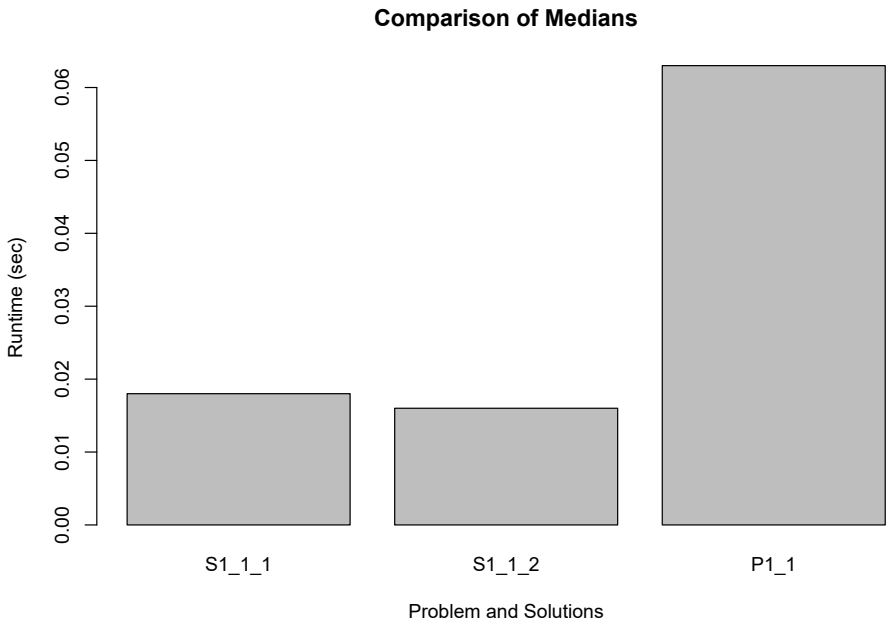


Figure 7.1.: Comparison of medians for P1_1

Solution two (S1_1_2)

Hypothesis:

$H_1 : X_{1\_1} = Y_{1\_1\_2}$

$H_0 : X_{1\_1} > Y_{1\_1\_2}$

$p - value < 2.2e - 16(Computer); \quad 2.2e - 16 < \alpha$

The p-value is smaller than the significance level $\alpha = 0.05$. Therefore, we reject the null hypothesis H_0 and accept the alternative hypothesis H_1.

It follows that we can observe an increase in performance for *S1_1_2*. Consequently, *S1_1_2* provides a valid solution to the problem P1_1. The runtime of the second solution S1_2_2 is recorded with an average runtime of 0.0329 seconds, which is a reduction of 45,80 %. A comparison of the median values in Figure 7.1 of the problem and solutions, further indicates, that the suggested solutions provide an increase in the performance.

Solutions for P1_2:

Solution one (S1_2_1)

Hypothesis:

$H_1 : X_{1\_2} = Y_{1\_2\_1}$

$H_0 : X_{1\_2} > Y_{1\_2\_1}$

$p - value < 2.2e - 16(Computer); \quad 2.2e - 16 < \alpha$

The p-value is smaller than the significance level $\alpha = 0.05$. Therefore, we reject the null hypothesis H_0 and accept the alternative hypothesis H_1.

It follows that we can observe an increase in performance for *S1_2_1*. Consequently, *S1_2_1* provides a valid solution to the problem P1_2. The runtime of problem 1_2 is recorded at 0.0925 seconds. Here, the runtime of the solution is recorded at an average of 0.0158 seconds, which corresponds to a reduction of 82,92 % in runtime.

Solution two (S1_2_2)

Hypothesis:

$H_1 : X_{1\_2} = Y_{1\_2\_2}$

$H_0 : X_{1\_2} > Y_{1\_2\_2}$

$p - value < 2.2e - 16(Computer); \quad 2.2e - 16 < \alpha$

The p-value is smaller than the significance level $\alpha = 0.05$. Therefore, we reject the null hypothesis H_0 and accept the alternative hypothesis H_1.

It follows that we can observe an increase in performance for *S1_2_2*. Consequently, *S1_2_2* provides a valid solution to the problem P1_2. The second solution is recorded with an average runtime of just 0.0046 seconds, which is a reduction of 95.03 % compared to the runtime of P1_2 at 0.0925 seconds.

As with the previous solutions, we provide a comparison of the means of problem and solutions. In Figure 7.2, one can see a strong indication, that the provided solutions show an actual increase in performance for the given problem P1_2.

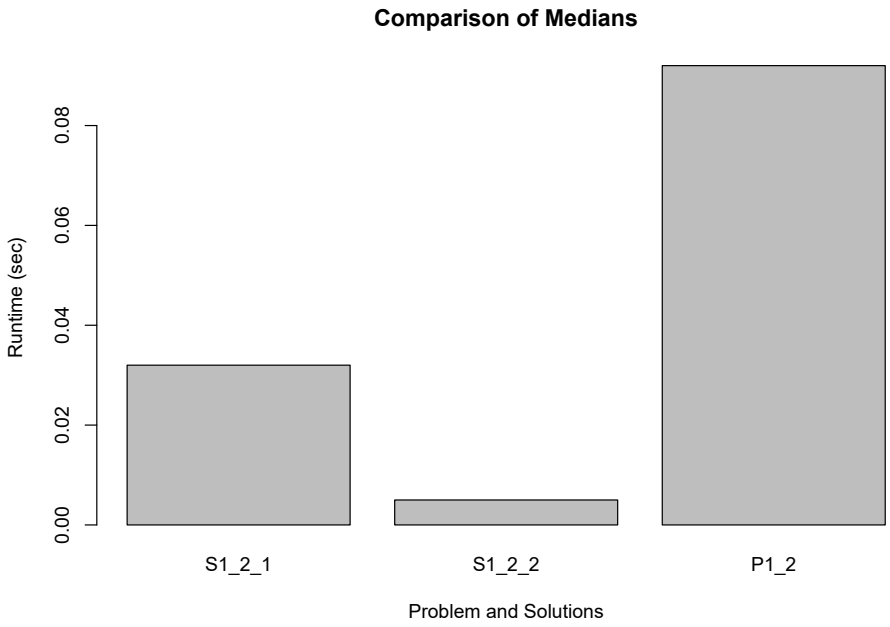


Figure 7.2.: Comparison of medians for P1_2

Scenario two

In scenario two, we research how many antipatterns were correctly detected by the software.

Both of the problems are successfully detected in this scenario. The tool reported both P1_1 and P1_2 as EDA antipatterns. Problem one is reported with an average runtime of 0.0607 seconds and 146572060 objects created. The second variant is reported with a runtime of 0.0925 seconds and 2108442600 created objects. We could confirm the results with a manual inspection of the profiles.

As we mentioned in the description of the second scenario, we implemented false positives for the fist variants of the problems. The detection tool wrongfully reports one of the false positives as an antipattern. To clarify, the function is still a slow function that contains an antipattern, but the categorization is wrong. We inserted an object creation *code-pattern* into an EP problem. Our detection tool then reports the EP as both, an EP and an EDA antipattern, even though the inserted code-patterns are not responsible for the problem.

7.4.3. Extensive Processing (P2)

Analogous to the previous section, we first discuss the results from the first scenario and then the results from the second scenario.

Scenario one

In the first scenario, we run the benchmark as described in Section 7.1. Then, we gather the results generated by the benchmark and test the solution for a possible performance increase. The results are shown in the following sections.

Solutions for P2_1:

Again, we provide a comparison of the median values of the problem P2_1 and its solutions. In Figure 7.3, one can see that the median of the solutions is lower than the median of the problem. This is an indication that the solutions provide a performance increase. The statistical tests are shown in the next sections.

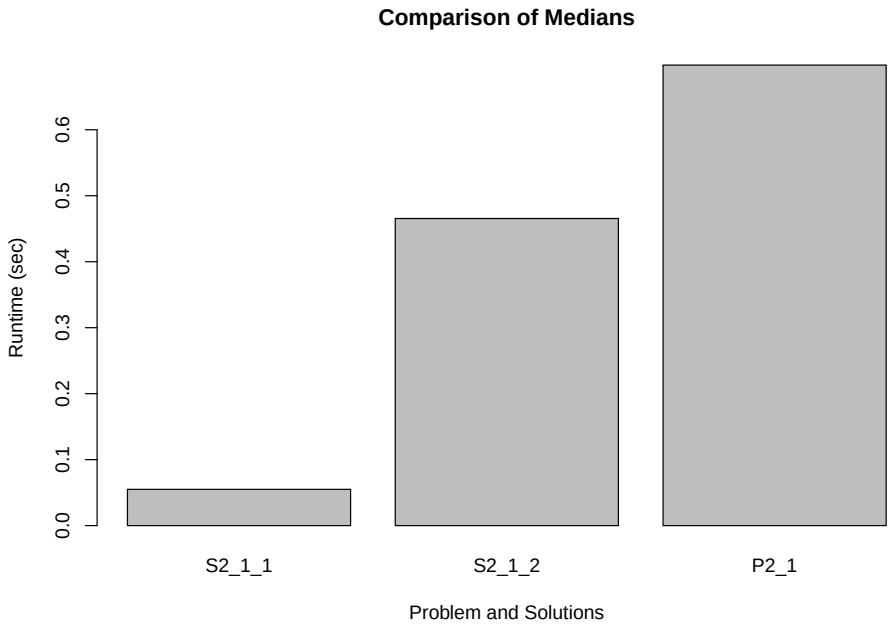


Figure 7.3.: Comparison of medians for P2_1

Solution one (S2_1_1)

Hypothesis:

$H_1 : X_{2\_1} = Y_{2\_1\_1}$

$H_0 : X_{2\_1} > Y_{2\_1\_1}$

$p - value < 2.2e - 16(Computer); \quad 2.2e - 16 < \alpha$

The p-value is smaller than the significance level $\alpha = 0.05$. Therefore, we reject the null hypothesis H_0 and accept the alternative hypothesis H_1.

It follows that we can observe an increase in performance for *S2_1_1*. Consequently, *S2_1_1* provides a valid solution to the problem P2_1. This solution ran for 0.0566 seconds, and the problem ran for 0.7222 seconds. Therefore, S1_2 provides a massive runtime reduction of 92,16 %.

Solution two (S2_1_2)

Hypothesis:

$H_1 : X_{2\_1} = Y_{2\_1\_2}$

$H_0 : X_{2\_1} > Y_{2\_1\_2}$

$p - value < 2.2e - 16(Computer); \quad 2.2e - 16 < \alpha$

The p-value is smaller than the significance level $\alpha = 0.05$. Therefore, we reject the null hypothesis H_0 and accept the alternative hypothesis H_1.

It follows that we can observe an increase in performance for *S2_1_2*. Consequently, *S2_1_2* provides a valid solution to the problem P2_1. The second solution for this problem took 0.4840 seconds to complete which is an improvement of 32,98 %.

With both tests indicating that the distributions of the solutions are significantly lower than the distributions of the problems, we can conclude that the solutions provide a significant performance improvement.

Solutions for P2_2

First, we show a comparison chart of the mean values of the problem P2_2 and the solutions to the problem. The chart, Figure 7.4 strongly indicates that the solutions indeed provide a huge performance increase. We are supporting that claim in the next sections.

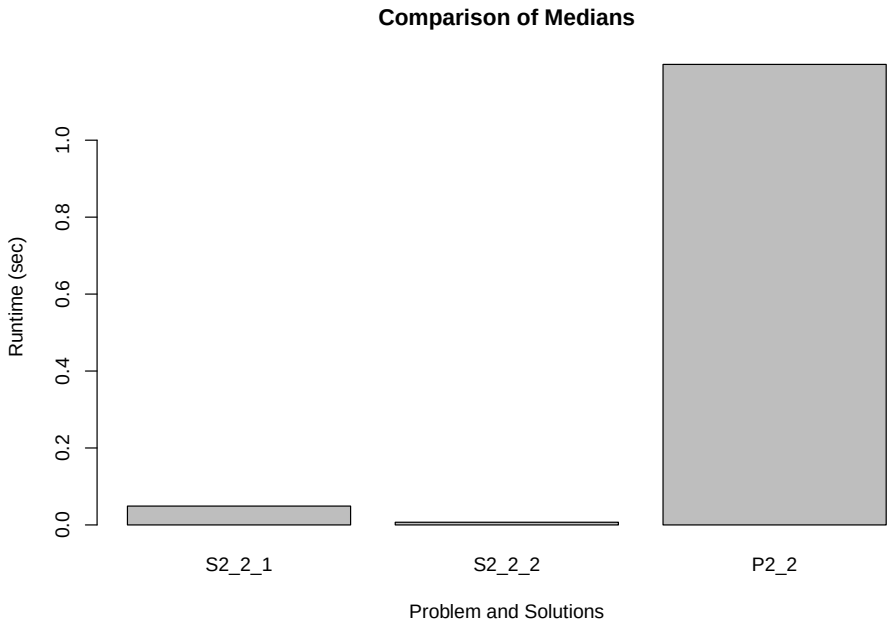


Figure 7.4.: Comparison of medians for P2_2

Solution one (S2_2_1)

Hypothesis:

$H_1 : X_{2\_2} = Y_{2\_2\_1}$

$H_0 : X_{2\_2} > Y_{2\_2\_1}$

$p - value < 2.2e - 16(Computer); \quad 2.2e - 16 < \alpha$

The p-value is smaller than the significance level $\alpha = 0.05$. Therefore, we reject the null hypothesis H_0 and accept the alternative hypothesis H_1.

It follows that we can observe an increase in performance for *S2_2_1*. Consequently, *S2_2_1* provides a valid solution to the problem P2_2. This solution took 0.0079 seconds to complete, and the problem ran for 1.2002 seconds. Here we can see another massive runtime reduction of 99.34

Solution two (S2_2_2)

Hypothesis:

$H_1 : X_{2\_2} = Y_{2\_2\_2}$

$H_0 : X_{2\_2} > Y_{2\_2\_2}$

$p - value < 2.2e - 16(Computer); \quad 2.2e - 16 < \alpha$

The p-value is smaller than the significance level $\alpha = 0.05$. Therefore, we reject the null hypothesis H_0 and accept the alternative hypothesis H_1.

We can again observe an increase in performance for *S2_2_2*. Consequently, *S2_2_2* provides a valid solution to the problem P2_2. Here, the response time of the solution is measured at 0.0503 seconds, which is a reduction of 98.81 %. With the provided results, we can reason that all of the solutions for P2 are valid and provide a significant improvement in the performance.

Scenario two

Then the second scenario targets the correctness of the approach. Here, we analyze if the detection is effective in finding the implemented performance antipatterns.

Both of the EP instances are detected in this scenario. The tool reports both P2_1 and P2_2 as EP antipatterns. The first problem is reported with an average runtime of 0.7222 seconds and 7649 sampled calls to `regexp.Compile`. The second variant is reported with a runtime of 1.2002 seconds and 11657 calls to `regexp.Compile`. We could confirm the results again with a manual inspection of the profiles.

Just like in the previous section about scenario two for the EDA antipattern, the detection tool wrongfully reports one of the false positives. The categorization is wrong again. We inserted an EP *code-pattern* into an OLB problem. Our detection tool then reports the OLB as both, an EP and an OLB antipattern. We suspect an error in the implementation of the *code-pattern* matching.

7.4.4. One-Lane Bridge (P3)

The results for the OLB antipattern of the experiment scenario one and scenario two, are presented in the following section.

Scenario One

The first scenario describes the test-run that we use to get the runtimes of the functions, here problems and solutions. In the following sections, we discuss the results of the first scenario for the OLB antipattern.

Solutions for P3_1:

First, we show an overview of the mean runtimes. This comparison allows us to draw the first conclusion about the question if the solutions provide a performance improvement. In Figure 7.5, one can already see that the implemented solutions show big performance improvement compared to the initial problem. This hypothesis is tested in the following sections.

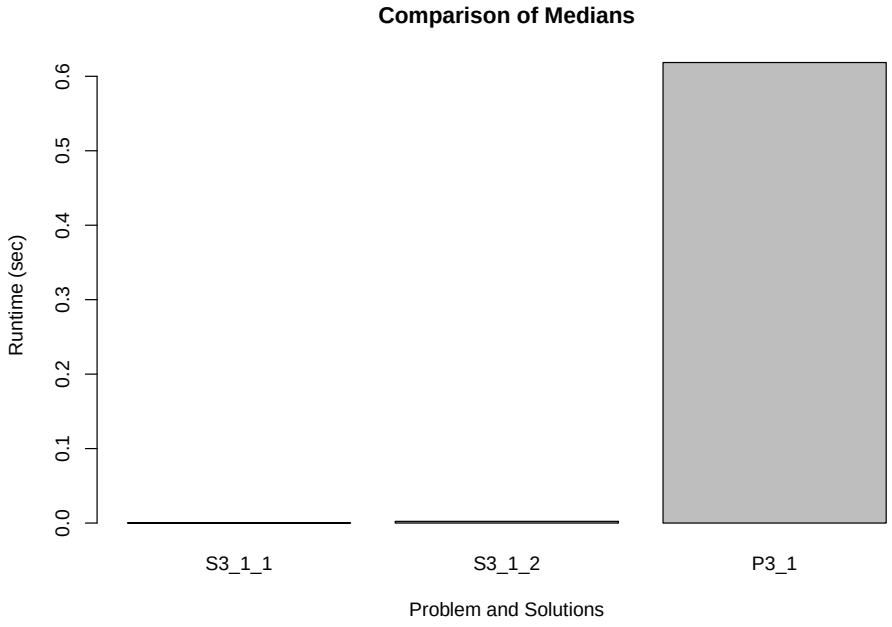


Figure 7.5.: Comparison of medians for P3_1

Solution one (S3_1_1)

Hypothesis:

$H_1 : X_{3\_1} = Y_{3\_1\_1}$

$H_0 : X_{3\_1} > Y_{3\_1\_1}$

$p - value < 2.2e - 16(Computer); \quad 2.2e - 16 < \alpha$

The p-value is smaller than the significance level $\alpha = 0.05$. Therefore, we reject the null hypothesis H_0 and accept the alternative hypothesis H_1. It follows that we can observe an increase in performance for *S3_1_1*. Consequently, *S3_1_1* provides a valid solution to the problem P3_1.

For the first solution, the recorded runtime is 0.0003 seconds. The runtime of the is 0.6794 seconds. Therefore, the improvement in runtime is a stunning 99.96 %.

Solution two (S3_1_2)

Hypothesis:

$H_1 : X_{3\_1} = Y_{3\_1\_2}$

$H_0 : X_{3\_1} > Y_{3\_1\_2}$

$p - value < 2.2e - 16(Computer); \quad 2.2e - 16 < \alpha$

The p-value is smaller than the significance level $\alpha = 0.05$. Therefore, we reject the null hypothesis H_0 and accept the alternative hypothesis H_1. It follows that the implemented solution provides an improvement in performance for the given problem. The runtime of the solution S3_1_2 is recorded at 0.0090 seconds, which is an improvement of 98,68 % compared to the runtime of P3_1.

Solutions for P3_2:

As in the previous sections, we first analyze the median runtimes of the problem and its solutions. For this purpose, we show a comparison chart in Figure 7.6. From the chart, we can make the observation that the implemented solutions seem to improve the performance of the problem P3_2. This observation is tested in the following sections.

Solution one (S3_2_1)

Hypothesis:

$H_1 : X_{3\_2} = Y_{3\_2\_1}$

$H_0 : X_{3\_2} > Y_{3\_2\_1}$

$p - value < 2.2e - 16(Computer); \quad 2.2e - 16 < \alpha$

The p-value is smaller than the significance level $\alpha = 0.05$. Therefore, we reject the null hypothesis H_0 and accept the alternative hypothesis H_1.

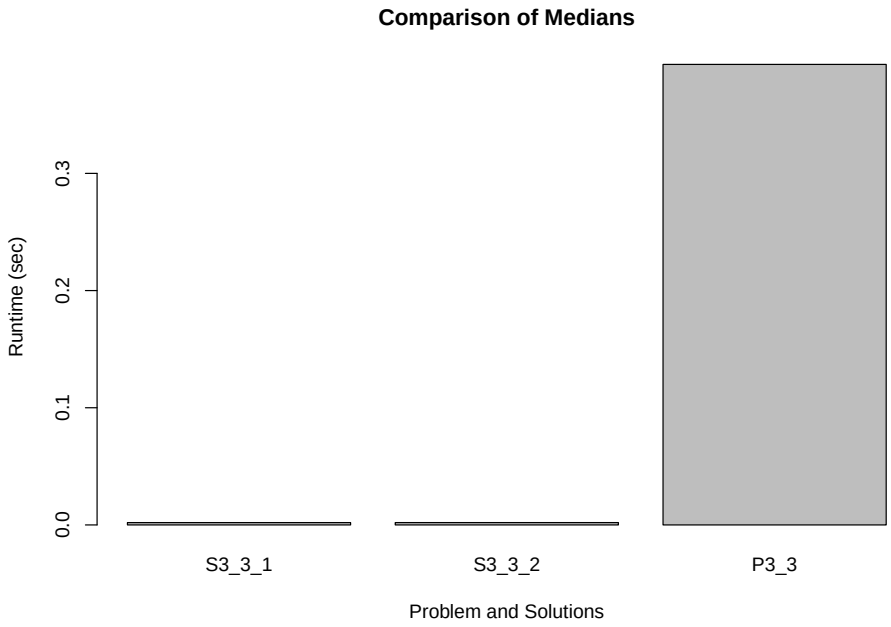


Figure 7.6.: Comparison of medians for P3_2

It follows that we can observe an increase in performance for *S3_2_1*. Consequently, *S3_2_1* provides a valid solution to the problem P3_2. This solution took 0.0018 seconds to complete, and the problem ran for 0.4064 seconds. Here we can see a runtime reduction of 99,56 %.

Solution one (S3_2_2)

Hypothesis:

$H_1 : X_{3\_2} = Y_{3\_2\_2}$

$H_0 : X_{3\_2} > Y_{3\_2\_2}$

$p - value < 2.2e - 16(Computer); \quad 2.2e - 16 < \alpha$

The p-value is smaller than the significance level $\alpha = 0.05$. Therefore, we reject the null hypothesis H_0 and accept the alternative hypothesis H_1.

It follows that we can observe an increase in performance for *S3_2_2*. Consequently, *S3_2_2* provides a valid solution to the problem P3_2. This solution took 0.0016 seconds, which is a reduction of 99.61 %. From the results for problem 2, we reason that the provided solutions improve the performance of the given problem.

Scenario two

In this section, we show the results from the second scenario and discuss the correctness of our approach.

Both of the OLB instances are detected in this scenario. The tool reports both P3_1 and P3_2 as OLB antipatterns. The first problem is detected with an average runtime of 0.6794 seconds and 494.64GB of allocated memory during the test run. The second variant is reported with a runtime of 0.4064 seconds and 341.86GB of memory allocated during the benchmark. We were able to confirm the reported results by manually inspecting the recorded profiles.

As for the false positives, the detection tool wrongfully reports one of the false positives. The falsely reported antipattern is put into the wrong category. We inserted an OLB *code-pattern* into an EDA problem. Our detection tool then reports the EDA as both, an EDA and an OLB antipattern. This is the wrong behavior since the inserted *code-pattern* is not responsible for the bad performance of the EDA problem. We can observe the same behavior in the previous reports on the second scenario for the other antipatterns. All our prepared false positives were detected and reported by the detection tool. Therefore, we suspect an error in the implementation of the *code-pattern* matching. It seems, that the matching process does not properly compare the runtimes of the found *code-patterns* to the runtimes of the function they are detected in.

7.5. Discussion of Results

7.5.1. Scenario One

The first scenario is the most important one. That is because the focus of the thesis is on "performing refactorings" with the help of profiling traces and the found antipatterns. To be able to refactor the code, we first have to know which parts to refactor and what the possible refactorings are.

The results for scenario one of the previous section show that our solutions are valid and provide a performance increase, as well as a reduction in resource use. Meaning, they improve the runtime of an application and reduce the CPU- and memory utilization. While the first variants of the implemented problems are for testing purposes only and don't have any real-life application, the second variants of the problems provide examples that could be used in production software systems. The results of scenario one are important for RQ1, since these results answer the question if the implemented refactorings provide a performance increase.

7.5.2. Scenario Two

The second scenario was conducted to find if the implemented detection tool works correctly. We found that the implemented antipatterns are detected correctly. Meaning 4 out of four antipatterns were reported by the detection tool. Further, we could confirm that the reported metrics for the found antipatterns are indeed correct, as they appear in the recorded profiles.

The second part of this scenario was to see if the tool can handle possible false positives and if it can categorize the antipatterns correctly. For this purpose, we introduced some code patterns of one antipattern into another one. This leads to our tool reporting the detected antipattern in multiple categories, while only one of the *code-patterns* is responsible for the problem. That means all three of the false positives are reported as antipatterns of additional different categories, which is wrong in this case. Therefore, we conclude that the detection and error handling has to be improved.

The Table 7.1 provides an overview of the results. It shows the recorded runtimes and other metrics that were extracted from the detection and used in the evaluation.

7.6. Conclusion

RQ1: The evaluation shows that the provided refactorings improve the performance of the respective problems. That means, we can suggest these refactorings as valid solutions to the implemented problems.

RQ2: Form the evaluations follows that the tool correctly detects the implemented antipatterns. The problem that remains is that antipatterns that contain multiple code patterns, where only one code-patterns is responsible for the problem, are reported in multiple categories.

RQ3: The evaluation shows that the detection can be improved by correcting the *code-pattern* detection and matching algorithm. This algorithm currently produces false positives, that can be eliminated by further comparing the runtimes of the LOC with the runtime of the containing function. Another possible improvement is to traverse the graph directly instead of parsing the textualized output of pprof, as the authors of [IUNO19] show in their approach.

		EDA		EP		OLB	
		#created Objects	runtimeExec (s)	compile function calls in samples	runtimeExec (s)	allocated Memory	runtimeExec (s)
Initial System	Problem-instance1	146572060	0.0607	7649	0.7222	494.64GB	0.6794
	Problem-instance2	2108442600	0.0925	11657	1.2002	341.86GB	0.4064
EDA-instance1	EDA-S1	5705	0.0186				
	EDA-S2	23036531	0.0329				
EDA-instance2	EDA-S1	48432343	0.0158				
	EDA-S2	1820	0.0046				
EP-instance1	EP-S1			100	0.0566		
	EP-S2			4046	0.4840		
EP-instance2	EP-S1			0(uses different solution)	0.0079		
	EP-S2			100	0.0503		
OLB-instance1	OLB-S1					43.79MB	0.0003
	OLB-S2					1.25GB	0.009
OLB-instance2	OLB-S1					429.58MB	0.0018
	OLB-S2					226.46MB	0.0016

Table 7.1.: Overview of the results with detected metrics

7.7. Threads to Validity

7.7.1. Conclusion validity

We only have one benchmark application that we implemented ourselves. That means we have no comparison to real-life applications. Further, the measurement implementation for the function runtimes can be incorrect or produce unreliable results. Therefore, it may be possible that the function runtimes are generated by chance. Additionally, our results would have to be confirmed by additional, independent results.

7.7.2. Internal Validity

We are in full control, how the profiles and function times are recorded. This should reduce the risk of introducing variation in the input. The problem here is that we have no control over the application's execution on the CPU or how exactly the profiler collects the profiles. Since pprof is a sampling profiler, there can be samples where a function is missing because it ran in between the sampling period. Further, our detection process is only based on the results of the profiler. Therefore, we could miss other important factors in our approach.

7.7.3. Construct Validity

Since we only have the one benchmark application and use its results for our statistical tests and implementation of the detection tool, we can not be certain that the collected measurements are correct. There could be an error in the benchmark that produces wrong measurements, which would, in return, invalidate our test results. Additionally, a wrongly assumed causality between the profiling information and SPAs can lead to an erroneous detection process.

7.7.4. External Validity

It may be problematic to apply our results to other applications. The reason for this is that we are dependant on the results produced by the pprof profiler. Other profilers may produce different traces and are therefore not applicable to our approach. This makes it hard to generalize our results for other available profilers.

Similar reasoning applies to the approach itself. We only use the results from our benchmark application and can only detect the antipatterns implemented in the benchmark app. Therefore it is difficult to generalize the approach and detect other antipatterns or use other profilers for the detection.

Chapter 8

Conclusion

This chapter provides an overview of the thesis. Afterward, future work is introduced.

8.1. Summary

We implemented an approach that can successfully detect performance antipatterns of an application and suggest refactorings for the problems. Our approach uses profiling traces and the source code of the applications to scan for pre-defined *profile-* and *code-patterns*. If both are detected, we report the found antipattern and its metrics. We then suggest a refactoring, which is confirmed to improve the performance of the application.

In the beginning, we needed to research possible performance antipatterns for the Go language. Here, we gathered information from several sources, see Chapter 4 on page 23, to implement a benchmark application. This application contains our identified antipatterns and serves as a baseline for the thesis. Additionally, we had to research how the pprof profiler works and what information it provides to us. Now, we had to find out how the antipatterns can be detected in the profiling information. For this task, we introduced our novel *profile-* and *code-patterns* and categorized the antipatterns.

After the initial research, we then designed our detection approach in Chapter 5 on page 41. Here, we re-used our previous findings, to then implement the automated antipattern detection tool, as to the defined approach. The tool uses the *profile-* and *code-patterns* to search for problems in the code and then reports the detected antipatterns with their metrics and solutions. For the detection, we use a five-step approach, where we first extract user-defined functions from the code. As the second step, we then filter the profiles by the user-defined functions. Then, the metrics for the functions are extracted. In the fourth step, the patterns are matched with the profiler output. The last step then reports the results to the user.

After the detection approach, we introduce our solutions to the problems. We created two variations for each problem and two solutions per variation. This resulted in a total of four problems and 12 solutions. We then had to confirm that our implemented solutions provide a performance increase. This was done in the evaluation of the thesis, see Chapter 7.

Finally, we evaluated our developed approach. Here, we evaluated how well the detection tool works, and if the solutions provide a statistically significant performance increase.

8.2. Retrospective

In this section we answer the question's from Chapter 1 in section 1.2.

The first goal was to research the Go language and find antipatterns on the code level. We fulfilled this goal and provide the found antipatters in Chapter 4 on page 23

With the first question, we also answered the second question on how pprof can be used for our approach. With the help of pprof we were able to identify the performance problems and extract the code and profile patterns.

Next, we successfully built a benchmark application as a baseline for our approach. Therefore, we achieve this goal as well.

Then we researched SPAs and show an overview of our research in the foundations chapter. See Chapter 2 on page 7. With those results we were able to complete this goal.

The concept for the detection can be found in Chapter 5. Here, we describe our approach and the implementation of the detection tool. Thus follows that we were able to fulfill this goal.

We finally provide the solutions for the antipatterns in Chapter 6 and the evaluation in Chapter 7. With the results of these chapters, we conclude that we were able to achieve the last two goals of Section 1.2.

Future Work

In the following, we present some future work that could be done to improve the developed approach.

Firstly, we recommend testing our approach with more realistic data. Since we only used our benchmark application, we are limited to those results. For this purpose, one could use any Go application that has real-life use, instrument it, and produce the required profiles. Then, use our tool to detect possible antipatterns in the code. That way, one could verify that the approach works for other applications as well.

Another improvement could be achieved by using additional profile types. We only use the CPU and memory profile. This limits the approach to timing- and memory-based problems. With the help of goroutine or blocking profiles, it would be possible to detect more antipatterns. For example, we could find possible congestions with the blocking profile. Those can be indicative of a OLB antipattern.

Further, Go provides an AST-parser. This parser could be used to parse the file into an abstract syntax tree. Then one could possibly perform type-safe refactorings and write the AST back to the source file. This would be a good idea to automate the refactoring process, which we could not do in our approach.

Finally, it may be better to use the graphs produced by pprof instead of using the text output. Graph traversal can be done very easy with Go since it provides many libraries for this purpose. With the help of the graph, it is possible to detect structural antipatterns. The authors of [IUNO19] showed that it is possible to detect layered-bottlenecks with such an approach.

Appendix A

Supplementary Listings

A.1. Flyweight Implementation

Listing A.1 Implementation of the flyweight design-pattern [Wel19]

```
// NewFlyweight creates a new Flyweight object.
func NewBook(title string) *Book {
      return &Book{Title: title}
}

// FlyweightFactory is a factory for creating and storing flyweights.
type FlyweightFactory struct {
      pool map[string]*Book
}

// NewFlyweightFactory creates a new FlyweightFactory.
func NewFlyweightFactory() *FlyweightFactory {
      return &FlyweightFactory{pool: make(map[string]*Book)}
}

// GetFlyweight gets or creates a flyweight depending on whether a
// flyweight of the same title already exists.
func (f *FlyweightFactory) GetFlyweight(title string) *Book {
      flyweight, okay := f.pool[title]
      if !okay {
            flyweight = NewBook(title)
            f.pool[title] = flyweight
      }
      return flyweight
}
```

A.2. Implementation of ContainsSubstring

Listing A.2 Implementation of the ContainsSubstring function

```
func ContainsSubstring(str string, subs ...string) bool {
	matches := 0
	isCompleteMatch := true

	for _, sub := range subs {
		if strings.Contains(str, sub) {
			matches += 1
		} else {
			isCompleteMatch = false
		}
	}

	return isCompleteMatch
}
```

Appendix B

Example Output

B.1. Complete Callgraph - pprof web output

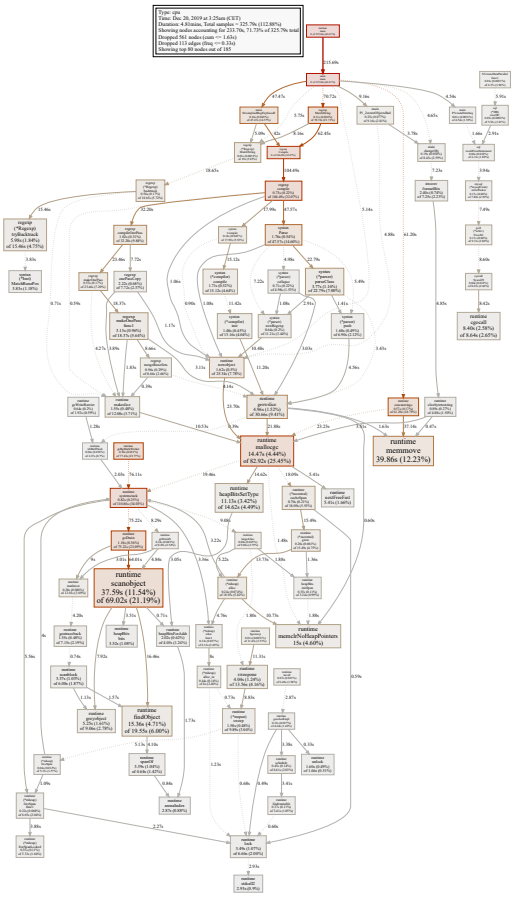


Figure B.1.: Callgraph of the benchmark

B.2. Example output for pprof -top

```
Type: cpu
Time: Dec 20, 2019 at 3:25am (CET)
Duration: 4.81mins, Total samples = 325.79s (112.88%)
Showing nodes accounting for 269.75s, 82.80% of 325.79s total
Dropped 561 nodes (cum <= 1.63s)
      flat  flat%   sum%        cum   cum%
         0     0%     0%    215.69s 66.21%  main.main
         0     0%     0%    215.69s 66.21%  runtime.main
     0.82s  0.25%  0.25%    110.86s 34.03%  runtime.systemstack
         0     0%  0.25%    104.49s 32.07%  regexp.Compile
     0.73s  0.22%  0.48%    104.49s 32.07%  regexp.compile
    14.47s  4.44%  4.92%     82.92s 25.45%  runtime.mallocgc
     0.10s 0.031%  4.95%     77.43s 23.77%  runtime.gcBgMarkWorker
     0.02s 0.0061%  4.95%     75.27s 23.10%  runtime.gcBgMarkWorker.func2
     1.18s  0.36%  5.32%     75.22s 23.09%  runtime.gcDrain
     0.18s 0.055%  5.37%     70.90s 21.76%  main.P2regularRegExpBad
     0.11s 0.034%  5.41%     70.72s 21.71%  regexp.MatchString
    37.59s 11.54% 16.94%     69.02s 21.19%  runtime.scanobject
     0.10s 0.031% 16.97%     61.81s 18.97%  main.P3stringConcatBad
     0.40s  0.12% 17.10%     61.73s 18.95%  runtime.concatstring2
     0.57s  0.17% 17.27%     61.20s 18.79%  runtime.concatstrings
     1.76s  0.54% 17.81%     47.57s 14.60%  regexp/syntax.Parse
     0.14s 0.043% 17.86%     47.47s 14.57%  main.S2compliedRegExpGood2
    39.86s 12.23% 30.09%     39.86s 12.23%  runtime.memmove
     1.02s  0.31% 30.40%     32.20s  9.88%  regexp.compileOnePass
     4.96s  1.52% 31.93%     30.66s  9.41%  runtime.growslice
     1.62s   0.5% 32.42%     25.34s  7.78%  runtime.newobject
     0.23s 0.071% 32.49%     23.49s  7.21%  runtime.rawstringtmp
     0.55s  0.17% 32.66%     23.46s  7.20%  regexp.makeOnePass
     0.03s 0.0092% 32.67%     23.26s  7.14%  runtime.rawstring
     3.77s  1.16% 33.83%     22.79s  7.00%  regexp/syntax.(*parser).parseClass
    15.36s  4.71% 38.54%     19.55s  6.00%  runtime.findObject
     0.01s 0.0031% 38.55%        19s  5.83%  regexp.(*Regexp).MatchString
     0.10s 0.031% 38.58%     18.99s  5.83%  regexp.(*Regexp).doMatch
     0.29s 0.089% 38.67%     18.96s  5.82%  runtime.(*mcache).nextFree
     0.24s 0.074% 38.74%     18.95s  5.82%  runtime.(*mheap).alloc
     0.24s 0.074% 38.81%     18.89s  5.80%  regexp.(*Regexp).doExecute
     0.56s  0.17% 38.99%     18.65s  5.72%  regexp.(*Regexp).backtrack
     3.13s  0.96% 39.95%     18.37s  5.64%  regexp.makeOnePass.func1
     0.18s 0.055% 40.00%     18.27s  5.61%  runtime.(*mcache).refill
     0.70s  0.21% 40.22%     18.09s  5.55%  runtime.(*mcentral).cacheSpan
     0.14s 0.043% 40.26%     17.99s  5.52%  regexp/syntax.Compile
     0.20s 0.061% 40.32%     15.49s  4.75%  runtime.(*mcentral).grow
     5.98s  1.84% 42.16%     15.46s  4.75%  regexp.(*Regexp).tryBacktrack
     1.71s  0.52% 42.68%     15.12s  4.64%  regexp/syntax.(*compiler).compile
       15s  4.60% 47.29%        15s  4.60%  runtime.memclrNoHeapPointers
```

```
11.13s  3.42% 50.70%     14.62s  4.49%  runtime.heapBitsSetType
 4.04s  1.24% 51.94%     13.56s  4.16%  runtime.sweepone
 1.48s  0.45% 52.40%     13.16s  4.04%  regexp/syntax.(*compiler).inst
 1.55s  0.48% 52.87%     12.08s  3.71%  runtime.makeslice
 0.28s 0.086% 52.96%     12.01s  3.69%  runtime.markroot
 0.01s 0.0031% 52.96%     11.43s  3.51%  runtime.bgsweep
 0.64s   0.2% 53.16%     11.21s  3.44%  regexp/syntax.(*parser).newRegexp
 0.61s  0.19% 53.34%      9.92s  3.04%  regexp/syntax.(*compiler).rune
 1.58s  0.48% 53.83%      9.89s  3.04%  runtime.(*mspan).sweep
 0.25s 0.077% 53.91%      9.16s  2.81%  main.P1_2createObjectsBad
 0.08s 0.025% 53.93%      9.16s  2.81%  runtime.mallocgc.func1
 0.13s  0.04% 53.97%      9.11s  2.80%  internal/poll.(*ioSrv).ExecIO
 0.08s 0.025% 53.99%      9.08s  2.79%  runtime.largeAlloc
 5.25s  1.61% 55.61%      9.06s  2.78%  runtime.greyobject
 0.96s  0.29% 55.90%      8.66s  2.66%  regexp.mergeRuneSets
 0.04s 0.012% 55.91%      8.65s  2.66%  syscall.Syscall9
 8.40s  2.58% 58.49%      8.64s  2.65%  runtime.cgocall
 0.11s 0.034% 58.53%      8.64s  2.65%  runtime.gcAssistAlloc
 0.02s 0.0061% 58.53%      8.53s  2.62%  runtime.gcAssistAlloc.func1
    0     0% 58.53%      8.51s  2.61%  runtime.gcAssistAlloc1
 0.19s 0.058% 58.59%      8.43s  2.59%  main.changeObj
 0.14s 0.043% 58.63%      8.29s  2.54%  runtime.gcDrainN
 0.12s 0.037% 58.67%      8.12s  2.49%  runtime.(*mheap).alloc.func1
 0.11s 0.034% 58.70%      8.04s  2.47%  strconv.Itoa
 0.44s  0.14% 58.84%         8s  2.46%  runtime.(*mheap).alloc_m
 0.65s   0.2% 59.04%      7.93s  2.43%  strconv.FormatInt
 0.13s  0.04% 59.08%      7.80s  2.39%  github.com/go-sql-driver/mysql.(*mysqlConn).writePacket
 2.22s  0.68% 59.76%      7.72s  2.37%  regexp.onePassCopy
 0.03s 0.0092% 59.77%      7.67s  2.35%  net.(*conn).Write
 0.02s 0.0061% 59.77%      7.64s  2.35%  net.(*netFD).Write
 0.06s 0.018% 59.79%      7.63s  2.34%  internal/poll.(*FD).Write
    0     0% 59.79%      7.40s  2.27%  internal/poll.(*FD).Write.func1
 0.05s 0.015% 59.81%      7.40s  2.27%  syscall.WSASend
 2.40s  0.74% 60.55%      7.25s  2.23%  strconv.formatBits
 1.55s  0.48% 61.02%      7.15s  2.19%  runtime.gentraceback
 1.60s  0.49% 61.51%      6.90s  2.12%  regexp/syntax.(*parser).push
 3.49s  1.07% 62.58%      6.66s  2.04%  runtime.lock
 0.22s 0.068% 62.65%      6.65s  2.04%  runtime.(*mheap).freeSpan.func1
 0.45s  0.14% 62.79%      6.61s  2.03%  runtime.schedule
 0.07s 0.021% 62.81%      6.40s  1.96%  runtime.mcall
 0.01s 0.0031% 62.81%      6.37s  1.96%  main.S7createDataParallel.func1
 0.02s 0.0061% 62.82%      6.36s  1.95%  database/sql.(*DB).Exec
 0.01s 0.0031% 62.82%      6.34s  1.95%  database/sql.(*DB).ExecContext
 0.01s 0.0031% 62.83%      6.33s  1.94%  database/sql.(*DB).exec
 0.06s 0.018% 62.84%      6.11s  1.88%  database/sql.resultFromStatement
 3.37s  1.03% 63.88%      6.08s  1.87%  runtime.scanblock
 0.03s 0.0092% 63.89%      5.91s  1.81%  database/sql.(*DB).execDC
 0.04s 0.012% 63.90%      5.89s  1.81%  runtime.markroot.func1
 0.22s 0.068% 63.97%      5.85s  1.80%  runtime.scang
```

```
0.03s 0.0092% 63.98%      5.78s  1.77%  main.S2compliedRegExpGood
0.36s  0.11% 64.09%      5.67s  1.74%  main.P1createObjectsBad
    0     0% 64.09%      5.59s  1.72%  database/sql.ctxDriverStmtExec
0.01s 0.0031% 64.09%      5.59s  1.72%  github.com/go-sql-driver/mysql.(*mysqlStmt).ExecContext
0.03s 0.0092% 64.10%      5.41s  1.66%  github.com/go-sql-driver/mysql.(*mysqlStmt).Exec
5.41s  1.66% 65.76%      5.41s  1.66%  runtime.nextFreeFast
0.21s 0.064% 65.82%      5.38s  1.65%  regexp/syntax.(*parser).concat
0.35s  0.11% 65.93%      5.33s  1.64%  runtime.(*mheap).freeSpanLocked
0.16s 0.049% 65.98%      5.32s  1.63%  runtime.scanstack
    0     0% 65.98%      5.18s  1.59%  runtime.markrootBlock
0.10s 0.031% 66.01%      5.13s  1.57%  regexp/syntax.(*parser).literal
0.04s 0.012% 66.02%      5.13s  1.57%  runtime.(*mheap).freeSpan
0.95s  0.29% 66.32%      5.09s  1.56%  regexp.mergeRuneSets.func2
    0     0% 66.32%      5.02s  1.54%  runtime.mstart
0.71s  0.22% 66.53%      4.98s  1.53%  regexp/syntax.(*parser).collapse
0.89s  0.27% 66.81%      4.88s  1.50%  runtime.slicebytetostring
1.42s  0.44% 67.24%      4.85s  1.49%  regexp/syntax.appendRange
0.57s  0.17% 67.42%      4.76s  1.46%  runtime.(*mheap).allocSpanLocked
0.12s 0.037% 67.45%      4.64s  1.42%  runtime.goschedImpl
3.39s  1.04% 68.50%      4.64s  1.42%  runtime.spanOf
0.01s 0.0031% 68.50%      4.54s  1.39%  main.P7createDataSeq
0.63s  0.19% 68.69%      4.50s  1.38%  regexp/syntax.cleanClass
1.03s  0.32% 69.01%      4.20s  1.29%  regexp/syntax.(*parser).repeat
0.26s  0.08% 69.09%      4.09s  1.26%  regexp/syntax.(*Inst).MatchRune
2.02s  0.62% 69.71%      4.05s  1.24%  runtime.heapBitsForAddr
0.09s 0.028% 69.74%      4.03s  1.24%  github.com/go-sql-driver/mysql.(*mysqlStmt).writeExecutePa
0.52s  0.16% 69.89%      3.87s  1.19%  sort.Sort
3.83s  1.18% 71.07%      3.83s  1.18%  regexp/syntax.(*Inst).MatchRunePos
0.13s  0.04% 71.11%      3.56s  1.09%  regexp/syntax.(*parser).op
3.52s  1.08% 72.19%      3.52s  1.08%  runtime.heapBits.bits
1.16s  0.36% 72.55%      3.49s  1.07%  regexp/syntax.(*parser).parseClassChar
    0     0% 72.55%      3.44s  1.06%  database/sql.(*Stmt).Exec
0.02s 0.0061% 72.55%      3.44s  1.06%  database/sql.(*Stmt).ExecContext
0.37s  0.11% 72.67%      3.41s  1.05%  runtime.findrunnable
    0     0% 72.67%      3.41s  1.05%  runtime.profilealloc
0.04s 0.012% 72.68%      3.35s  1.03%  runtime.park_m
0.01s 0.0031% 72.68%      3.32s  1.02%  runtime.mProf_Malloc
0.35s  0.11% 72.79%      3.24s  0.99%  runtime.heapBits.initSpan
0.20s 0.061% 72.85%      3.22s  0.99%  runtime.(*mcentral).freeSpan
0.12s 0.037% 72.89%      3.21s  0.99%  main.S1_3reuseObjectsGoodCPool
0.46s  0.14% 73.03%      3.21s  0.99%  sort.quickSort
0.09s 0.028% 73.06%      2.96s  0.91%  runtime.gosched_m
0.01s 0.0031% 73.06%      2.94s   0.9%  runtime.semasleep
2.93s   0.9% 73.96%      2.93s   0.9%  runtime.stdcall2
1.67s  0.51% 74.47%      2.91s  0.89%  runtime.(*mheap).coalesce
2.87s  0.88% 75.35%      2.87s  0.88%  runtime.arenaIndex
0.09s 0.028% 75.38%      2.86s  0.88%  runtime.callers
0.02s 0.0061% 75.39%      2.77s  0.85%  runtime.callers.func1
0.02s 0.0061% 75.39%      2.74s  0.84%  database/sql.withLock
```

```
0.77s  0.24% 75.63%      2.71s  0.83%  runtime.pcvalue
1.29s   0.4% 76.02%      2.67s  0.82%  regexp/syntax.nextRune
2.66s  0.82% 76.84%      2.66s  0.82%  regexp.(*bitState).shouldVisit
0.01s 0.0031% 76.84%      2.52s  0.77%  database/sql.ctxDriverPrepare
0.02s 0.0061% 76.85%      2.51s  0.77%  runtime.scanstack.func1
0.02s 0.0061% 76.86%      2.50s  0.77%  github.com/go-sql-driver/mysql.(*mysqlConn).PrepareContext
0.03s 0.0092% 76.87%      2.49s  0.76%  regexp/syntax.(*compiler).init
0.13s  0.04% 76.91%      2.49s  0.76%  runtime.scanframeworker
0.73s  0.22% 77.13%      2.49s  0.76%  runtime.wbBufFlush1
0.02s 0.0061% 77.14%      2.46s  0.76%  github.com/go-sql-driver/mysql.(*mysqlConn).Prepare
0.84s  0.26% 77.39%      2.44s  0.75%  sort.insertionSort
0.01s 0.0031% 77.40%      2.43s  0.75%  runtime.semawakeup
2.40s  0.74% 78.13%      2.42s  0.74%  runtime.stdcall1
0.04s 0.012% 78.15%      2.41s  0.74%  runtime.notewakeup
0.11s 0.034% 78.18%      2.33s  0.72%  runtime.deductSweepCredit
2.09s  0.64% 78.82%      2.32s  0.71%  runtime.heapBits.next
0.12s 0.037% 78.86%      2.29s   0.7%  regexp/syntax.(*Regexp).CapNames
0.06s 0.018% 78.88%      2.27s   0.7%  runtime.wbBufFlush
0.12s 0.037% 78.91%      2.25s  0.69%  runtime.newstack
2.24s  0.69% 79.60%      2.24s  0.69%  runtime.acquirem
0.02s 0.0061% 79.61%      2.21s  0.68%  runtime.wbBufFlush.func1
0.02s 0.0061% 79.61%      2.15s  0.66%  database/sql.(*driverStmt).Close
0.01s 0.0031% 79.62%      2.09s  0.64%  github.com/go-sql-driver/mysql.(*mysqlStmt).Close
1.13s  0.35% 79.96%      2.09s  0.64%  regexp/syntax.(*parser).parsePerlClassEscape
0.01s 0.0031% 79.97%      2.08s  0.64%  github.com/go-sql-driver/mysql.(*mysqlConn).writeCommandPacketUi
0.03s 0.0092% 79.97%      2.08s  0.64%  runtime.gcMarkDone
0.23s 0.071% 80.05%      2.04s  0.63%  runtime.newMarkBits
0.03s 0.0092% 80.05%         2s  0.61%  regexp/syntax.(*compiler).empty
1.14s  0.35% 80.40%         2s  0.61%  runtime.gcmarknewobject
0.07s 0.021% 80.43%      1.98s  0.61%  regexp.newQueue
0.64s   0.2% 80.62%      1.92s  0.59%  runtime.gcWriteBarrier
1.10s  0.34% 80.96%      1.90s  0.58%  runtime.(*mTreap).insert
0.30s 0.092% 81.05%      1.85s  0.57%  main.S1reuseObjectsGood
0.05s 0.015% 81.07%      1.82s  0.56%  runtime.gopreempt_m
0.07s 0.021% 81.09%      1.81s  0.56%  github.com/go-sql-driver/mysql.(*mysqlConn).readPacket
0.03s 0.0092% 81.10%      1.78s  0.55%  github.com/go-sql-driver/mysql.(*mysqlConn).writeCommandPacketSt
0.03s 0.0092% 81.11%      1.78s  0.55%  runtime.morestack
1.78s  0.55% 81.65%      1.78s  0.55%  runtime.releasem
1.59s  0.49% 82.14%      1.77s  0.54%  runtime.step
0.05s 0.015% 82.16%      1.74s  0.53%  github.com/go-sql-driver/mysql.(*buffer).readNext
0.37s  0.11% 82.27%      1.71s  0.52%  runtime.(*mspan).markBitsForIndex
0.04s 0.012% 82.28%      1.69s  0.52%  github.com/go-sql-driver/mysql.(*buffer).fill
1.60s  0.49% 82.77%      1.66s  0.51%  runtime.unlock
0.01s 0.0031% 82.78%      1.65s  0.51%  net.(*conn).Read
0.02s 0.0061% 82.78%      1.64s   0.5%  internal/poll.(*FD).Read
    0     0% 82.78%      1.64s   0.5%  net.(*netFD).Read
0.05s 0.015% 82.80%      1.63s   0.5%  main.S1_2ReuseObjectGod
```

Appendix B

Bibliography

[ABGM09] A. Aleti, S. Bjornander, L. Grunske, I. Meedeniya. “ArcheOpterix: An extendable tool for architecture optimization of AADL models.” In: *2009 ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software*. May 2009, pp. 61–71 (cit. on pp. 18, 20).

[Bau72] D. F. Bauer. “Constructing Confidence Sets Using Rank Statistics.” In: *Journal of the American Statistical Association* 67.339 (1972), pp. 687–690. DOI: 10.1080/01621459.1972.10481279. URL: https://www.tandfonline.com/doi/abs/10.1080/01621459.1972.10481279 (cit. on p. 72).

[BKR09] S. Becker, H. Koziolek, R. Reussner. “The Palladio component model for model-driven performance prediction.” In: *Journal of Systems and Software* 82.1 (2009). Special Issue: Software Performance - Modeling and Analysis, pp. 3–22 (cit. on pp. 19, 21).

[Bra17] A. Bran. “Detecting software performance anti-patterns from profiler data.” Englisch. 1 Online-Ressource (xv, 66 Seiten). Hochschulschrift. Stuttgart, 2017 (cit. on pp. 21, 49).

[CDE+10] V. Cortellessa, A. Di Marco, R. Eramo, A. Pierantonio, C. Trubiani. “Digging into UML Models to Remove Performance Antipatterns.” In: *Proceedings of the 2010 ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems*. QUOVADIS ’10. Cape Town, South Africa: ACM, 2010, pp. 9–16 (cit. on pp. 20, 21).

[CDT12] V. Cortellessa, A. Di Marco, C. Trubiani. “Software Performance Antipatterns: Modeling And Analysis.” In: *Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*. Ed. by M. Bernardo, V. Cortellessa, A. Pierantonio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 290–335. URL: https://doi.org/10.1007/978-3-642-30982-3_9 (cit. on pp. 20, 21).

[CMT10] V. Cortellessa, A. D. Marco, C. Trubiani. “Performance Antipatterns as Logical Predicates.” In: *2010 15th IEEE International Conference on Engineering of Complex Computer Systems*. Mar. 2010, pp. 146–156 (cit. on p. 17).

[Fow06] M. Fowler. *Code Smells*. 2006. URL: https://martinfowler.com/bliki/CodeSmell.html (cit. on p. 14).

[Fow19] M. Fowler. *Refactoring: improving the design of existing code*. Englisch. Ed. by K. Beck. Second edition. A Martin Fowler signature book. Literaturverzeichnis: Seiten 405-407. Boston ; Columbus ; New York ; San Francisco ; Amsterdam ; Cape Town ; Dubai ; London: Addison-Wesley, 2019, xix, 418 Seiten. ISBN: 0134757599 (cit. on p. 14).

[Gam95] E. [Gamma, ed. *Design patterns: elements of reusable object-oriented software*. Englisch. 5. print. Addison-Wesley professional computing series. Institut für Thermische Strömungsmaschinen und Maschinenlaboratorium. Reading, Mass. [u.a.]: Addison-Wesley, 1995, XV, 395 Seiten (cit. on pp. 8, 58).

[GoD15] GoDoc. *gorename documentation*. 2015. URL: https://godoc.org/golang.org/x/tools/cmd/gorename (cit. on p. 66).

[GoD17] GoDoc. *eg documentation*. 2017. URL: https://godoc.org/golang.org/x/tools/cmd/eg (cit. on p. 66).

[Goo19] Google. *pprof visualization tool*. 2019. URL: https://github.com/google/pprof (cit. on p. 11).

[Goo20a] Google. *Go - net/http/pprof package documentation*. 2020. URL: https://golang.org/pkg/net/http/pprof/ (cit. on pp. xv, 11, 13).

[Goo20b] Google. *Go - runtime/pprof package documentation*. 2020. URL: https://golang.org/pkg/runtime/pprof/ (cit. on pp. xv, 11, 12).

[Goo20c] Google. *Go Diagnostics*. 2020. URL: https://golang.org/doc/diagnostics.html (cit. on pp. 13, 14).

[Goo20d] Google. *Go Documentation*. 2020. URL: https://golang.org/doc/ (cit. on pp. xv, 8, 12, 13, 24, 28, 32, 35, 55, 56, 65).

[Goo20e] Google. *Go Website*. 2020. URL: https://golang.org/ (cit. on p. 7).

[Goo20f] Google. *Protocol Buffers*. 2020. URL: https://developers.google.com/protocol-buffers/ (cit. on p. 11).

[Gry16] D. Gryski. *go-perfbook*. 2016. URL: https://github.com/dgryski/go-perfbook (cit. on pp. 24, 53, 69).

[HWH12] A. van Hoorn, J. Waller, W. Hasselbring. "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis." In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ICPE '12. Boston, Massachusetts, USA: Association for Computing Machinery, 2012, pp. 247–248. ISBN: 9781450312028. DOI: 10.1145/2188286.2188326. URL: https://doi.org/10.1145/2188286.2188326 (cit. on p. 1).

[IUNO19] T. Inagaki, Y. Ueda, T. Nakaike, M. Ohara. "Profile-Based Detection of Layered Bottlenecks." In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. ICPE '19. Mumbai, India: Association for Computing Machinery, 2019, pp. 197–208. ISBN: 9781450362399. DOI: 10.1145/3297663.3310296. URL: https://doi.org/10.1145/3297663.3310296 (cit. on pp. 21, 22, 85, 91).

[MG18] D. Melykian, S. GmbH. *Practical Go Benchmarks*. 2018. URL: https://stackimpact.com/blog/practical-golang-benchmarks (cit. on pp. 24, 53, 55, 58, 69).

[Moh17] Mohsin. *Go - calculate execution time*. 2017. URL: %5Curl%7Bhttps://stackoverflow.com/questions/45766572/is-there-an-efficient-way-to-calculate-execution-time-in-golang/45766707#45766707%7D (cit. on pp. xv, 54).

[MW47] H. B. Mann, D. R. Whitney. "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other." In: *Ann. Math. Statist.* 18.1 (Mar. 1947), pp. 50–60. DOI: 10.1214/aoms/1177730491. URL: https://doi.org/10.1214/aoms/1177730491 (cit. on p. 72).

[OMG06] O. M. G. (OMG). *OCL Specification, Document – formal/06-05-01 (Object Constraint Language, v2.0)*. 2006. URL: https://www.omg.org/cgi-bin/doc?formal/06-05-01 (cit. on pp. 18, 20).

[OMG08] O. M. G. (OMG). *UML Profile for MARTE (ptc/08-06-09)*. https://www.omg.org/omgmarte/Documents/Specifications/08-06-09.pdf. June 2008. URL: https://www.omg.org/omgmarte/Documents/Specifications/08-06-09.pdf (cit. on pp. 18, 20, 21).

[OMG17] O. M. G. (OMG). *UML Specification, Document – formal/17-12-05 (Unified Modeling Language, v2.5.1)*. 2017. URL: https://www.omg.org/spec/UML/ (cit. on pp. 18, 19).

[Pal19] Palladio. *Palladio Website*. https://www.palladio-simulator.com/home/. 2019. URL: https://www.palladio-simulator.com/home/ (cit. on pp. 18, 19).

[PM08] T. Parsons, J. Murphy. "Detecting Performance Antipatterns in Com-ponent Based Enterprise Systems." In: *Journal of Object Technology* 7.3 (Mar. 2008), pp. 55–90 (cit. on pp. 18, 20, 21).

[Sea16] J. Seaman. *Refactoring: Improving the Design of Existing Code*. 2016. URL: https://userweb.cs.txstate.edu/~js236/cs4354/refactoring.pdf (cit. on p. 14).

[SG02] C. Smith, L. G. Williams. "New Software Performance AntiPatterns: More Ways to Shoot Yourself in the Foot." In: Jan. 2002, pp. 667–674 (cit. on p. 9).

[Sta19a] Stackimpact. *Go Performance Tuning*. 2019. URL: https://stackimpact.com/docs/go-performance-tuning/#go-performance-patterns (cit. on pp. 24, 53, 69).

[Sta19b] N. Stadelmaier. *Go Antipattern Benchmark Application*. 2019. URL: https://github.com/nstadelmaier-dev/go-antipatterns/benchmarks (cit. on pp. 24, 69, 70).

[Sta19c] N. Stadelmaier. *Go Antipattern Detection Application*. 2019. URL: https://github.com/nstadelmaier-dev/go-antipatterns/detection (cit. on pp. 46, 50).

[SW00] C. U. Smith, L. G. Williams. "Software Performance Antipatterns." In: *Proceedings of the 2Nd International Workshop on Software and Performance*. WOSP '00. Ottawa, Ontario, Canada: ACM, 2000, pp. 127–136 (cit. on pp. 9, 10).

[SW01] C. Smith, L. Williams. "Software Performance AntiPatterns; Common Performance Problems and their Solutions." In: Jan. 2001, pp. 797–806 (cit. on p. 10).

[SW03] C. U. Smith, L. G. Williams. "More new software performance antipatterns: Even more ways to shoot yourself in the foot." In: *Computer Measurement Group Conference*. Citeseer. 2003, pp. 717–725 (cit. on pp. 9, 11, 17, 18).

[TBH+18] C. Trubiani, A. Bran, A. van Hoorn, A. Avritzer, H. Knoche. "Exploiting load testing and profiling for Performance Antipattern Detection." In: *Information and Software Technology* 95 (März 2018), pp. 329–345 (cit. on pp. 17, 21).

[TK11] C. Trubiani, A. Koziolek. “Detection and solution of software performance antipatterns in palladio architectural models.” In: *Proceeding of the second joint WOSP/SIPEW international conference on Performance engineering - ICPE ’11*. ACM Press, 2011 (cit. on pp. 19, 21).

[Wel19] B. Wells. *Go Patterns*. 2019. URL: %5Curl%7Bhttps://github.com/bvwells/go-patterns#design-patterns%7D (cit. on pp. xvi, 58, 93).

[Xu12] J. Xu. “Rule-based automatic software performance diagnosis and improvement.” In: *Performance Evaluation* 69.11 (2012), pp. 525–550 (cit. on p. 20).

All links were last followed on January 17, 2020.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature