Hands-On High Performance with Go

Boost and optimize the performance of your Golang applications at scale with resilience

Packt

www.packt.com

Bob Strecansky

Hands-On High Performance with Go

Boost and optimize the performance of your Golang applications at scale with resilience

Bob Strecansky

Packt

BIRMINGHAM - MUMBAI

Hands-On High Performance with Go

Commissioning Editor: Richa Tripathi
Acquisition Editor: Alok Dhuri
Content Development Editor: Pathikrit Roy
Senior Editor: Storm Mann
Technical Editor: Pradeep Sahu
Copy Editor: Safis Editing
Project Coordinator: Francy Puthiry
Proofreader: Safis Editing
Indexer: Tejal Daruwale Soni
Production Designer: Joshua Misquitta

First published: March 2020

Production reference: 1230320

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78980-578-9

`www.packt.com`

To my wife Bree, who made all of this (plus many other things!) possible.

Contributors

About the author

Bob Strecansky is a senior site reliability engineer. He graduated with a computer engineering degree from Clemson University with a focus on networking. He has worked in both consulting and industry jobs since graduation. He has worked with large telecom companies and much of the Alexa top 500. He currently works at Mailchimp, working to improve web performance, security, and reliability for one of the world's largest email service providers. He has also written articles for web publications and currently maintains the OpenTelemetry PHP project. In his free time, Bob enjoys tennis, cooking, and restoring old cars. You can follow Bob on the internet to hear musings about performance analysis:
Twitter: `@bobstrecansky`
GitHub: `@bobstrecansky`

About the reviewer

Eduard Bondarenko is a long-time software developer. He prefers concise and expressive code with comments and has tried many programming languages, such as Ruby, Go, Java, and JavaScript.

Eduard has reviewed a couple of programming books and has enjoyed their broad topics and how interesting they are. Besides programming, he likes to spend time with the family, play soccer, and travel.

> *I want to thank my family for supporting me during my work on the book and also the author of this book for an interesting read.*

Table of Contents

Preface

Hands-On High Performance with Go is a complete resource with proven methodologies and techniques to help you diagnose and fix performance problems in your Go applications. The book starts with an introduction to the concepts of performance, where you will learn about the ideology behind the performance of Go. Next, you will learn how to implement Go data structures and algorithms effectively and explore data manipulation and organization in order to write programs for scalable software. Channels and goroutines for parallelism and concurrency in order to write high-performance codes for distributed systems are also a core part of this book. Moving on, you'll learn how to manage memory effectively. You'll explore the **Compute Unified Device Architecture** (**CUDA**) driver **application programming interface** (**API**), use containers to build Go code, and work with the Go build cache for faster compilation. You'll also get a clear picture of profiling and tracing Go code to detect bottlenecks in your system. At the end of the book, you'll evaluate clusters and job queues for performance optimization and monitor the application for performance regression.

Who this book is for

This Go book is a must for developers and professionals who have an intermediate-to-advanced understanding of Go programming and are interested in improving their speed of code execution.

What this book covers

`Chapter 1`, *Introduction to Performance in Go*, will discuss why performance in computer science is important. You will also learn why performance is important in the Go language.

`Chapter 2`, *Data Structures and Algorithms*, deals with data structures and algorithms, which are the basic units of building software, notably complex performance software. Understanding them will help you to think about how to most impact fully organize and manipulate data in order to write effective, performant software. Also, iterators and generators are essential to Go. This chapter will include explanations of different data structures and algorithms, as well as how their big O notation is impacted.

`Chapter 3`, *Understanding Concurrency*, will talk about utilizing channels and goroutines for parallelism and concurrency, which are idiomatic in Go and are the best ways to write high-performance code in your system. Being able to understand when and where to use each of these design patterns is essential to writing performant Go.

`Chapter 4`, *STL Algorithm Equivalents in Go*, discusses how many programmers coming from other high-performance languages, namely C++, understand the concept of the standard template library, which provides common programming data structures and functions in a generalized library in order to rapidly iterate and write performant code at scale.

`Chapter 5`, *Matrix and Vector Computation in Go*, deals with matrix and vector computations in general. Matrices are important in graphics manipulation and AI, namely image recognition. Vectors can hold a myriad of objects in dynamic arrays. They use contiguous storage and can be manipulated to accommodate growth.

`Chapter 6`, *Composing Readable Go Code*, focuses on the importance of writing readable Go code. Understanding the patterns and idioms discussed in this chapter will help you to write Go code that is more easily readable and operable between teams. Also, being able to write idiomatic Go will help raise the level of your code quality and help your project maintain velocity.

`Chapter 7`, *Template Programming in Go*, focuses on template programming in Go. Metaprogramming allows the end user to write Go programs that produce, manipulate, and run Go programs. Go has clear, static dependencies, which helps with metaprogramming. It has shortcomings that other languages don't have in metaprogramming, such as `__getattr__` in Python, but we can still generate Go code and compile the resulting code if it's deemed prudent.

`Chapter 8`, *Memory Management in Go*, discusses how memory management is paramount to system performance. Being able to utilize a computer's memory footprint to the fullest allows you to keep highly functioning programs in memory so that you don't often have to take the large performance hit of swapping to disk. Being able to manage memory effectively is a core tenet of writing performant Go code.

`Chapter 9`, *GPU Parallelization in Go*, focuses on GPU accelerated programming, which is becoming more and more important in today's high-performance computing stacks. We can use the CUDA driver API for GPU acceleration. This is commonly used in topics such as deep learning algorithms.

`Chapter 10`, *Compile Time Evaluations in Go*, discusses minimizing dependencies and each file declaring its own dependencies while writing a Go program. Regular syntax and module support also help to improve compile times, as well as interface satisfaction. These things help to make Go compilation quicker, alongside using containers for building Go code and utilizing the Go build cache.

`Chapter 11`, *Building and Deploying Go Code*, focuses on how to deploy new Go code. To elaborate further, this chapter explains how we can push this out to one or multiple places in order to test against different environments. Doing this will allow us to push the envelope of the amount of throughput that we have for our system.

`Chapter 12`, *Profiling Go Code*, focuses on profiling Go code, which is one of the best ways to determine where bottlenecks live within your Go functions. Performing this profiling will help you to deduce where you can make improvements within your function and how much time individual pieces take within your function call with respect to the overall system.

`Chapter 13`, *Tracing Go Code*, deals with a fantastic way to check interoperability between functions and services within your Go program, also known as tracing. Tracing allows you to pass context through your system and evaluate where you are being held up. Whether it's a third-party API call, a slow messaging queue, or an $O(n^2)$ function, tracing will help you to find where this bottleneck resides.

`Chapter 14`, *Clusters and Job Queues*, focuses on the importance of clustering and job queues in Go as good ways to get distributed systems to work synchronously and deliver a consistent message. Distributed computing is difficult, and it becomes very important to watch for potential performance optimizations within both clustering and job queues.

`Chapter 15`, *Comparing Code Quality Across Versions*, deals with what you should do after you have written, debugged, profiled, and monitored Go code that is monitoring your application in the long term for performance regressions. Adding new features to your code is fruitless if you can't continue to deliver a level of performance that other systems in your infrastructure depend on.

To get the most out of this book

This book is for Go professionals and developers seeking to execute their code faster, so an intermediate to advanced understanding of Go programming is necessary to make the most out of this book. The Go language has relatively minimal system requirements. A modern computer with a modern operating system should support the Go runtime and its dependencies. Go is used in many low power devices that have limited CPU, Memory, and I/O requirements.

You can see the requirements for the language listed at the following URL: `https://github.com/golang/go/wiki/MinimumRequirements`.

In this book I used Fedora Core Linux (version 29 during the time of writing this book) as the operating system. Instructions on how to install the Fedora Workstation Linux distribution can be found on the Fedora page at the following URL: `https://getfedora.org/en/workstation/download/`.

Docker is used for many of the examples in this book. You can see the requirements listed for Docker at the following URL: `https://docs.docker.com/install/`.

In `Chapter 9`, *GPU Parallelization in Go*, we discuss GPU programming. To perform the tasks of this chapter, you'll need one of two things:

- A NVIDIA enabled GPU. I used a NVIDIA GeForce GTX 670 in my testing, with a Compute Capability of 3.0.
- A GPU enabled cloud instance. Chapter 9 discusses a couple of different providers and methodologies for this. GPUs on Compute Engine work for this. More up to date information on GPUs on Compute Engine can be found at the following URL: `https://cloud.google.com/compute/docs/gpus`.

After you read this book; I hope you'll be able to write more efficient Go code. You'll hopefully be able to quantify and validate your efforts as well.

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.

2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/bobstrecansky/HighPerformanceWithGo/`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Code in Action

Code in Action videos for this book can be viewed at `http://bit.ly/2QcfEJI`.

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781789805789_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: " The following code blocks will show the `Next()` incantation"

A block of code is set as follows:

```
// Note the trailing () for this anonymous function invocation
func() {
  fmt.Println("Hello Go")
}()
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
// Note the trailing () for this anonymous function invocation
func() {
  fmt.Println("Hello Go")
}()
```

Any command-line input or output is written as follows:

```
$ go test -bench=. -benchtime 2s -count 2 -benchmem -cpu 4
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "The **reverse algorithm** takes a dataset and reverses the values of the set"

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1 Section 1: Learning about Performance in Go

In this section, you will learn why performance in computer science is important. You will also learn why performance is important in the Go language. Moving on, you will learn about data structures and algorithms, concurrency, STL algorithm equivalents, and the matrix and vector computations in Go.

The chapters in this section include the following:

1 Introduction to Performance in Go

This book is written with intermediate to advanced Go developers in mind. These developers will be looking to squeeze more performance out of their Go application. To do this, this book will help to drive the four golden signals as defined in the *Site Reliability Engineering Workbook* (`https://landing.google.com/sre/sre-book/chapters/monitoring-distributed-systems/`). If we can reduce latency and errors, as well as increase traffic whilst reducing saturation, our programs will continue to be more performant. Following the ideology of the four golden signals is beneficial for anyone developing a Go application with performance in mind.

In this chapter, you'll be introduced to some of the core concepts of performance in computer science. You'll learn some of the history of the Go computer programming language, how its creators decided that it was important to put performance at the forefront of the language, and why writing performant Go is important. Go is a programming language designed with performance in mind, and this book will take you through some of the highlights on how to use some of Go's design and tooling to your advantage. This will help you to write more efficient code.

In this chapter, we will cover the following topics:

- Understanding performance in computer science
- A brief history of Go
- The ideology behind Go performance

These topics are provided to guide you in beginning to understand the direction you need to take to write highly performant code in the Go language.

Technical requirements

For this book, you should have a moderate understanding of the Go language. Some key concepts to understand before exploring these topics include the following:

- The Go reference specification: `https://golang.org/ref/spec`
- How to write Go code: `https://golang.org/doc/code.html`
- Effective Go: `https://golang.org/doc/effective_go.html`

Throughout this book, there will be many code samples and benchmark results. These are all accessible via the GitHub repository at `https://github.com/bobstrecansky/HighPerformanceWithGo/`.

If you have a question or would like to request a change to the repository, feel free to create an issue within the repository at `https://github.com/bobstrecansky/HighPerformanceWithGo/issues/new`.

Understanding performance in computer science

Performance in computer science is a measure of work that can be accomplished by a computer system. Performant code is vital to many different groups of developers. Whether you're part of a large-scale software company that needs to quickly deliver masses of data to customers, an embedded computing device programmer who has limited computing resources available, or a hobbyist looking to squeeze more requests out of the Raspberry Pi that you are using for your pet project, performance should be at the forefront of your development mindset. Performance matters, especially when your scale continues to grow.

It is important to remember that we are sometimes limited by physical bounds. CPU, memory, disk I/O, and network connectivity all have performance ceilings based on the hardware that you either purchase or rent from a cloud provider. There are other systems that may run concurrently alongside our Go programs that can also consume resources, such as OS packages, logging utilities, monitoring tools, and other binaries—it is prudent to remember that our programs are very frequently not the only tenants on the physical machines they run on.

Optimized code generally helps in many ways, including the following:

- Decreased response time: The total amount of time it takes to respond to a request.
- Decreased latency: The time delay between a cause and effect within a system.
- Increased throughput: The rate at which data can be processed.
- Higher scalability: More work can be processed within a contained system.

There are many ways to service more requests within a computer system. Adding more individual computers (often referred to as horizontal scaling) or upgrading to more powerful computers (often referred to as vertical scaling) are common practices used to handle demand within a computer system. One of the fastest ways to service more requests without needing additional hardware is to increase code performance. Performance engineering acts as a way to help with both horizontal and vertical scaling. The more performant your code is, the more requests you can handle on a single machine. This pattern can potentially result in fewer or less expensive physical hosts to run your workload. This is a large value proposition for many businesses and hobbyists alike, as it helps to drive down the cost of operation and improves the end user experience.

A brief note on Big O notation

Big O notation (`https://en.wikipedia.org/wiki/Big_O_notation`) is commonly used to describe the limiting behavior of a function based on the size of the inputs. In computer science, Big O notation is used to explain how efficient algorithms are in comparison to one another—we'll discuss this more in detail in `Chapter 2`, *Data Structures and Algorithms*. Big O notation is important in optimizing performance because it is used as a comparison operator in explaining how well algorithms will scale. Understanding Big O notation will help you to write more performant code, as it will help to drive performance decisions in your code as the code is being composed. Knowing at what point different algorithms have relative strengths and weaknesses helps you to determine the correct choice for the implementation at hand. We can't improve what we can't measure—Big O notation helps us to give a concrete measurement to the problem statement at hand.

Methods to gauge long term performance

As we make our performance improvements, we will need to continually monitor our changes to view impact. Many methods can be used to monitor the long-term performance of computer systems. A couple of examples of these methods would be the following:

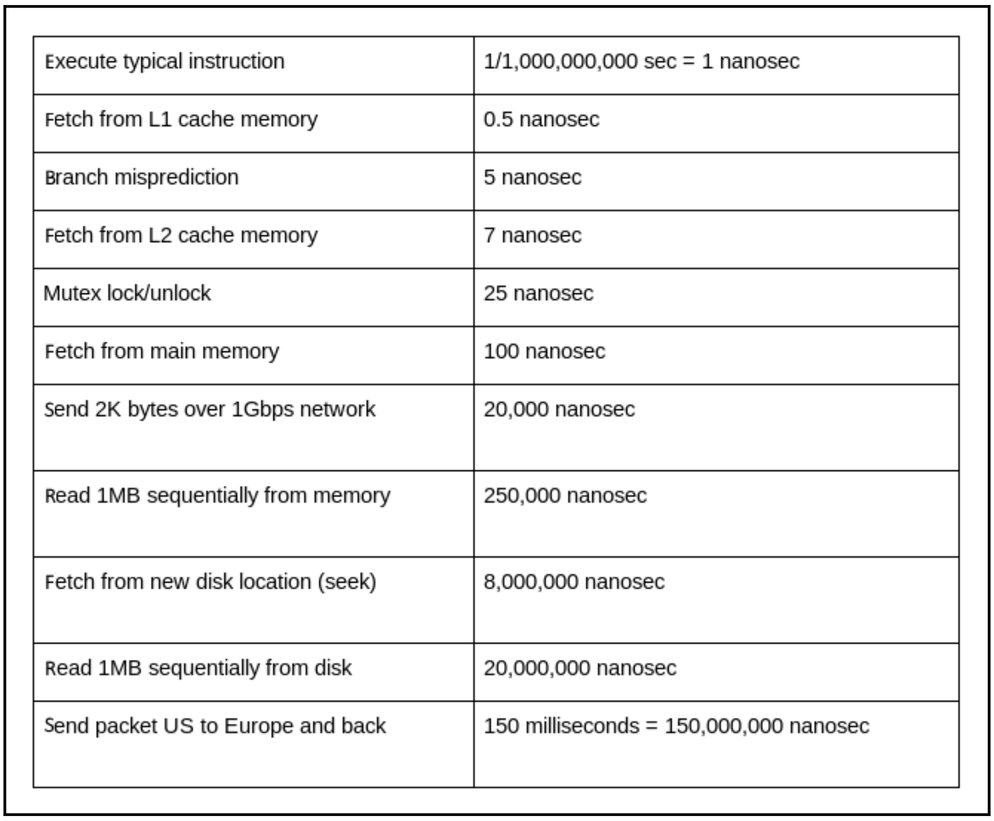
- Brendan Gregg's USE Method: Utilization, saturation, and errors (`www.brendangregg.com/usemethod.html`)
- Tom Wilkie's RED Metrics: Requests, errors, and duration (`https://www.weave.works/blog/the-red-method-key-metrics-for-microservices-architecture/`)
- Google SRE's four Golden Signals: Latency, traffic, errors, and saturation (`https://landing.google.com/sre/sre-book/chapters/monitoring-distributed-systems/`)

We will discuss these concepts further in `Chapter 15`, *Comparing Code Quality Across Versions*. These paradigms help us to make smart decisions about the performance optimizations in our code as well as avoid premature optimization. Premature optimization plays as a very crucial aspect for many a computer programmers. Very frequently, we have to determine what *fast enough* is. We can waste our time trying to optimize a small segment of code when many other code paths have an opportunity to improve from a performance perspective. Go's simplicity allows for additional optimization without cognitive load overhead or an increase in code complexity. The algorithms that we will discuss in `Chapter 2`, *Data Structures and Algorithms*, will help us to avoid premature optimization.

Optimization strategies overview

In this book, we will also attempt to understand what exactly we are optimizing for. The techniques for optimizing for CPU or memory utilization may look very different than optimizing for I/O or network latency. Being cognizant of your problem space as well as your limitations within your hardware and upstream APIs will help you to determine how to optimize for the problem statement at hand. Optimization also often shows diminishing returns. Frequently the return on development investment for a particular code hotspot isn't worthwhile based on extraneous factors, or adding optimizations will decrease readability and increase risk for the whole system. If you can determine whether an optimization is worth doing early on, you'll be able to have a more narrowly scoped focus and will likely continue to develop a more performant system.

It can be helpful to understand baseline operations within a computer system. *Peter Norvig*, the Director of Research at Google, designed a table (the image that follows) to help developers understand the various common timing operations on a typical computer (`https://norvig.com/21-days.html#answers`):



Execute typical instruction	1/1,000,000,000 sec = 1 nanosec
Fetch from L1 cache memory	0.5 nanosec
Branch misprediction	5 nanosec
Fetch from L2 cache memory	7 nanosec
Mutex lock/unlock	25 nanosec
Fetch from main memory	100 nanosec
Send 2K bytes over 1Gbps network	20,000 nanosec
Read 1MB sequentially from memory	250,000 nanosec
Fetch from new disk location (seek)	8,000,000 nanosec
Read 1MB sequentially from disk	20,000,000 nanosec
Send packet US to Europe and back	150 milliseconds = 150,000,000 nanosec

Having a clear understanding of how different parts of a computer can interoperate with one another helps us to deduce where our performance optimizations should lie. As derived from the table, it takes quite a bit longer to read 1 MB of data sequentially from disk versus sending 2 KBs over a 1 Gbps network link. Being able to have *back-of-the-napkin math* comparison operators for common computer interactions can very much help to deduce which piece of your code you should optimize next. Determining bottlenecks within your program becomes easier when you take a step back and look at a snapshot of the system as a whole.