O'REILLY®

Efficient Go

Data-Driven Performance Optimization

Early Release

RAW & UNEDITED

Bartłomiej Płotka

Efficient Go

Data-Driven Performance Optimizations

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Bartłomiej Płotka

Efficient Go

by Bartłomiej Płotka

Copyright © 2022 Alloc Limited. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Suzanne McQuade

Development Editor: Melissa Potter

Production Editor: Clare Jensen

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

December 2022: First Edition

Revision History for the Early Release

- 2021-09-16: First Release
- 2021-12-16: Second Release
- 2022-03-01: Third Release
- 2022-06-23: Fourth Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098105716 for release details.

The O’Reilly logo is a registered trademark of O’Reilly Media, Inc. *Efficient Go*, the cover image, and related trade dress are trademarks of O’Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher’s views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-10564-8

[LSI]

Chapter 1. Software Efficiency Matters

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at mpotter@oreilly.com.

> *The primary task of software engineers is the cost-effective development of maintainable and useful software.*
>
> —Jon Louis Bentley, Writing Efficient Programs (1982)

Even after 40 years, Jon's definition of development is fairly accurate. The ultimate goal for any engineer is to create a useful product that can sustain user needs for the product lifetime. Unfortunately, nowadays, not every developer realizes the significance of the software cost. The truth can be brutal; stating that the development process can be expensive might be an underestimation. For instance, it took five years and 250 engineers for Rockstar to develop a popular Grand Theft Auto 5 video game, which was estimated to cost $137.5 million. On the other hand, to create a usable, commercialized operating system, Apple had to spend way over $500 million before the first release of macOS in 2001.

Because of the high cost, when producing software, it's crucial to focus our efforts on things that matter the most. Ideally, we don't want to waste engineering time and energy on unnecessary actions, like spending weeks on subjective code refactoring that does not reduce code complexity or deep micro-optimizations of the function executed rarely. The industry is continually inventing new patterns to pursue an efficient development process. Agile, Kanban methods allowing to adapt to ever-changing requirements, specialized programming languages for mobile platforms like Kotlin, or frameworks for building websites like React are only some examples. Engineers innovate in those fields because every inefficiency increases the cost.

What makes it even more difficult is that when developing software now, we should also be aware of the future costs. Running and maintenance cost is typically higher than the initial development. Code changes to stay competitive, bug fixing, incidents, installations, or even electricity consumed are only a few examples of Total software Cost of Ownership (TOC) we, developers, tend to forget. Agile methodologies help to reveal this cost early by releasing software often and getting feedback sooner. Unfortunately, because of development complexity, it's a prevalent mistake to focus on the short-term goal of publishing the software quicker. Many dangerous shortcuts naturally come to our minds. Skipping testing parts, not investing enough in security, documentation, clean code, or code performance might indicate smart development efficiency and pragmatism. The problem is that if taken too far, they can cause an enormous cost, ranging from your product not being useful on the market to extremes like taking airplanes down[1].

At this point, based on the title "Efficient Go", you might be asking how this book will motivate you to spend more of your precious time on software execution performance characteristics like speed or efficiency. If we, software creators, should care, as Jon wrote, about development cost-effectiveness, why not focus purely on the bare minimum needed for the software to work? Waiting a few seconds more for application execution never killed anyone. On top of that, the hardware is getting cheaper and

faster every month. In 2021, it's not difficult to buy a smartphone with a dozen GBs of RAM. Finger-sized, 2TB SSD disks capable of 7 GB/s read and write throughput are available. Even home PC workstations are hitting never before seen performance scores. With0 8 CPUs or more that can perform billions of cycles per second each and with 2TB of RAM we can compute things fast. After all, improving performance on the software level alone is a complicated topic. Especially when you are new, it is common to lose time optimizing without significant program speedups. And even if we start caring about the latency introduced by our code, things like Java Virtual Machine or Go compiler will apply their optimizations anyway. Overall, spending more time on something tricky like performance that can also sacrifice our code's reliability and maintainability may sound like a bad idea. These are only a few of the numerous reasons why typically engineers put performance optimizations on the lowest position of the development priority list, far in the outskirts of the mentioned software bare minimum.

Unfortunately, as with everything extreme, there is a risk in such performance deprioritization. In essence, there is a difference between consciously postponing optimizations and making silly mistakes causing inefficiencies and slowdowns. However, don't be worried! I will not try to convince you in this book that all of this is wrong, and you should now measure the number of nanoseconds each code line introduces or how many bits it allocates in memory before putting it in your software. You should not. I am far from trying to motivate you to put a performance on the top of your development priority list. Instead, I would like to propose a subtle but essential change to how we, software engineers, should think about application performance. It will allow you to bring small but effective habits to your programming and development management cycle. Based on data and as early as possible in the development cycle, you will learn how to tell when you can safely ignore or postpone program inefficiencies. And when you can't afford to skip performance optimizations, where and how to apply them effectively, and when to stop.

> *Machines have become increasingly cheap compared to people; any discussion of computer efficiency that fails to take this into account is short-sighted. "Efficiency" involves the reduction of overall cost - not just machine time over the life of the program, but also time spent by the programmer and by the users of the program.*
>
> —Brian W. Kernighan and P. J. Plauger, The Elements of Programming Style (1978)

In "Motivation For This Book" you will learn what made me decide not to treat performance optimizations as an unnecessary evil. In "Behind Performance" we will unpack the word performance and learn how it is related to efficiency in this book's title. Through "Common Performance Misconceptions" to "Be Vigilant to Simplifications", we will challenge five serious misconceptions around efficiency and performance, often descoping such work from developer minds. You will learn that thinking about efficiency is not reserved only for "high performance" software. Finally, in "Efficiency: The Key to Pragmatic Code Performance" I will teach you why particularly efficiency will allow us to think about performance optimizations effectively without sacrificing time and other software qualities. This chapter might feel theoretical, but trust me. Those insights will train your essential programming judgment on how and if to adopt particular efficiency optimizations, algorithms, and code improvements presented in Part 3 of this book. This chapter is also fully language agnostic, so it should be practical for non-Go developers too!

Motivation For This Book

When writing this book, I was 29 years old. That might not feel like much experience, but I started full-time, professional programming when I was 19. I did full-time Computer Science studies in parallel to work at Intel around infrastructure. Initially coding in Python and C++, then jumping into Go (due to Kubernetes hype) for most of those years. I wrote or reviewed tens of thousands of code lines for various software that had to run on production, be reliable, and scale. From around 2017, in London, UK, I was

lucky enough to develop primarily open-source software in various projects written in Go. This includes a popular time-series database for monitoring purposes called Prometheus. I also had an opportunity to co-create a large distributed system project called Thanos. I would not be surprised if my code is running somewhere in your company infrastructure too!

I am grateful for those open-source opportunities. If not those, most likely, I would not decide to write this book. The reason is that nothing taught me as much about software development as doing this in the open. You interact with diverse people, from different places worldwide, with different backgrounds, goals, and needs. It's sometimes challenging. It would probably be easier to stick to working only with ex-Google, ex-Facebook, ex-Amazon like I did before. However, I was always motivated to look around more and see the bigger picture of everyday software development problems and challenges. In my opinion, that picture does not look perfect. With more people programming overall, often without a computer science background, there are plenty of mistakes and misconceptions, especially related to software performance.

Overall, with the fantastic people I had a chance to work with, I believe we achieved amazing things. I was lucky to work in environments where high-quality code was more important than decreasing code review iterations or reducing time spent addressing style issues. We thrived for good system design, code maintainability, and readability. We tried to bring those values to open source too, and I think we did a good job there. However, there is one important thing I would improve if I had a chance to write, for instance, the Thanos project again. You probably would not guess what. I would try to focus more on the pragmatic efficiency of my code and the algorithms we chose. I would focus on learning how to quickly gather data about my code module's performance, benchmark, profile, and understand many performance tools. Go, and other languages provide. I would avoid many hours I spent making mistakes and understanding the performance tooling and Go runtime behaviour. And don't get me wrong, the Thanos system nowadays is faster and uses much fewer resources than some competitors, but it took a lot of time, and there is still a massive amount of hardware

resources we could use less. If I would apply the knowledge, tips, and suggestions that you will learn in this book, I believe we could cut the development cost in half, if not more, to have Thanos in the state we have today. (I hope my ex-boss who paid for this work won't read that thesis!).

Don't you believe me? Well, hear this: Thanos was almost fully functional in February 2018, after three months of development we did with Fabian Reinartz. However, it took six more months to improve performance to the state where queries no longer crashed the process due to OOMs[2], and latencies were enough for initial production use.

What blocked me from caring about efficiency a little bit more from the start? First of all, a lack of skills. There was not much literature that would give me the practical answer to our performance or scaling questions, especially for Go. And even when I found a way to improve our Go code's efficiency, almost no one could review it and verify it properly. In both open-source and organization, not many around me had practical awareness of what to do around Go code performance. Fortunately, today we live in a better world, with consolidated literature about pragmatic efficiency. You are reading such literature right now!

The second reason is even harder to improve upon. It is about the misleading perception of performance optimizations. The impression that "premature optimization is the root of all evil" as Donald E. Knuth wrote[3], spread the world, giving everyone an excellent excuse to do less optimization work. Taken to the extreme, it demotivated people to even learn about efficiency practices and think about it, which had a strong impact on the software industry we know now. In this chapter, in "Common Performance Misconceptions", I have collected five major ones that we will unpack. Those and similar misconceptions are the main reasons why basic programs like Microsoft Excel are so slow, why the battery in your smartphone only lasts a few hours, and why your cloud provider bill is so large.

Let's start with unpacking some of those misconceptions by first exploring what performance actually means.

Behind Performance

Before discussing why software efficiency or any form of optimizations matters and when to apply them, we must first demystify the overused word performance. In engineering, this word is used in too many contexts and can mean different things, so let's unpack it to avoid confusion.

> **DID YOU KNOW?**
>
> Word performant does not exist in English vocabulary. Sadly our code cannot be performant, indicating that there is always room to improve things[4]. The question is, at what point we should say stop.

When people say "this application is performing poorly", they usually mean that this particular program is executing slowly[5]. However, if the same people would say, "Bartek is not performing well at work", they probably do not mean that Bartek is walking too slowly from the computer to the meeting room. In my experience, a significant number of people in software development consider the word performance a synonym of speed. For others, it means the overall quality of execution, which is the original definition of this word[6]. This phenomenon is sometimes called a "Semantic Diffusion". It occurs when a word starts to be used by larger groups with different meaning it originally had.].

> *The word performance in computer performance means the same thing that performance means in other contexts, that is, it means "How well is the computer doing the work it is supposed to do?*
>
> —Computer Performance Analysis with Mathematica by Arnold O. Allen, Academic Press

I think Arnold's definition describes the *performance* world as accurately as possible, so it's might be the first actionable item you can take from this book. Be specific.

CLARIFY WHEN SOMEONE USES THE WORD "PERFORMANCE".

When reading the documentation, code, bug trackers, or attending conference talks, be careful when you hear that word. Ask follow-up questions and ensure what the author means.

In practice, performance, as the quality of overall execution, might contain much more than we typically think. It might feel picky, but if we want to improve software development's cost-effectiveness, we must communicate clearly, efficiently and effectively!

I would suggest we avoid this word unless we can specify what we mean. Imagine you are reporting a bug in a bug tracker like GitHub Issues. Especially there, don't just mention "low performance", but instead specify what exactly was the unexpected behaviour of the application you were describing. Similarly, when describing improvements for a software release in the changelog[7], don't just mention "improving performance". Describe what exactly was enhanced. Maybe part of the system is now less prone to user input errors, use less RAM (if yes, how much less, in what circumstances?) or execute something faster (how many seconds faster, for what kind of workloads?). Being explicit will save time for you and your users.

When you see the word performance in my book, in the context of application computation, you can refer to it as visualized in Figure 1-1.

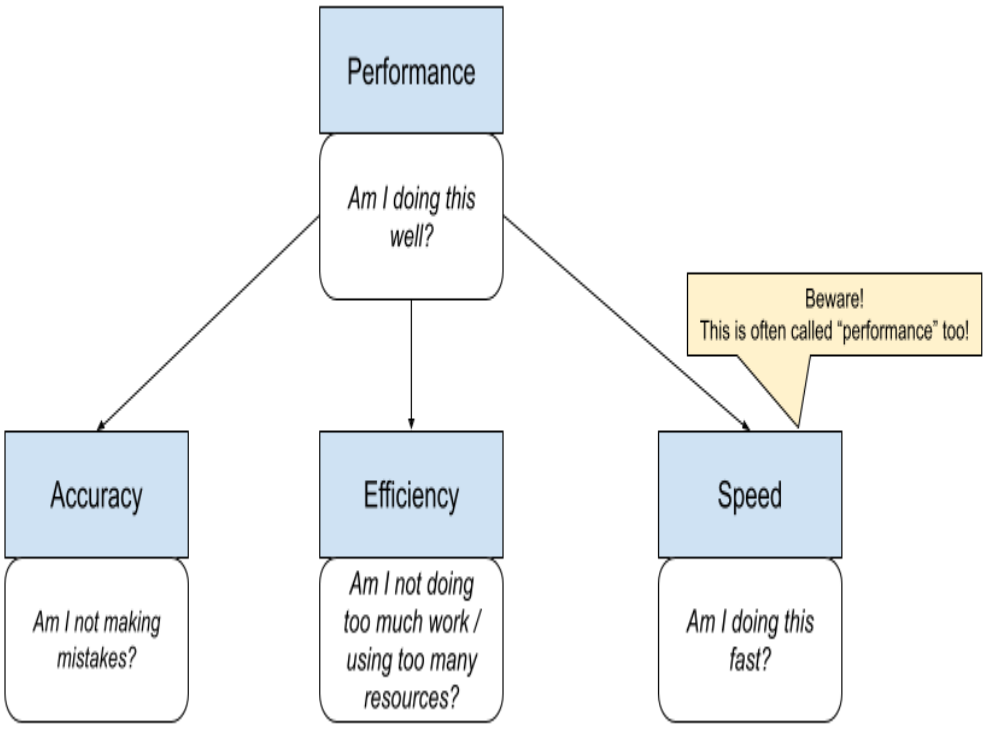


Figure 1-1. Performance Definition

In principle, software performance means "how well software runs" and consists of three core execution elements you can improve (or sacrifice):

Accuracy

> The number of errors you do make while doing the work to accomplish the task. It can be measured for software by the number of wrong results your application produces. For example, how many requests finished with non 200 HTTP status codes in a web system

Speed

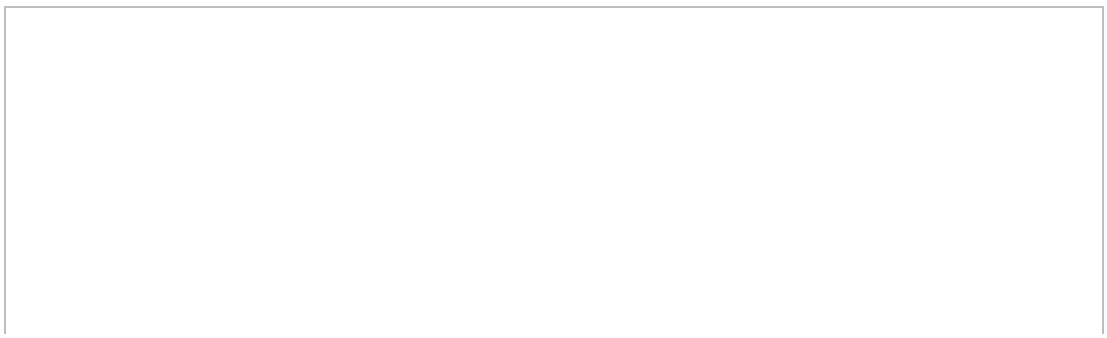
> How fast you do the work needed to accomplish the task, the timeliness of execution. It can be observed by operation latency or throughput. For example, compressing 1GB of data in memory typically takes around 10s (latency), allowing approximately 100MB per second throughput

Efficiency

The ratio of the useful energy delivered by a dynamic system to the energy supplied to it. In simpler words, it is the indicator of how many extra resources, energy, or work was used to accomplish the task. A waste. It is sometimes an easily measurable concept, quantified by the ratio of sound output to total helpful input. For instance, if our operation of fetching 64 bytes of valuable data from disk allocates 420 bytes on RAM, our memory efficiency would equal
$\text{energy output}/\text{energy used}*100\% = 15.23\%$. Note that this does not mean our operation is 15.23% efficient in total. We did not calculate energy, CPU time, heat and other efficiencies. For practical purposes, we tend to specify what efficiency we have in mind. Alternatively, when talking about overall program efficiency, we mean we don't waste significant effort.

$$\text{performance} = (\text{accuracy}*\text{efficiency}*\text{speed})$$

Improving any of those enhances the performance of the running application or system. It can help with reliability, availability, resiliency, overall latency, and more. Similarly, ignoring any of those can make our software less useful. Those three elements might also feel disjointed, but in fact, they are connected. For instance, notice that we can still achieve better reliability and availability without changing accuracy (not reducing the number of bugs). For example, with efficiency, reducing memory consumption decreases the chances of running out of memory and crashing the application or host operating system. This book focuses on knowledge, technics and methods, allowing you to increase the efficiency and speed of your running code without degrading accuracy.



IT'S NO MISTAKE THAT THE TITLE OF MY BOOK IS CALLED "EFFICIENT GO"

My goal is to teach you pragmatic skills allowing you to produce high quality, accurate, efficient and fast code with minimum effort. For this purposes, when I mention the overall efficiency of the code in my book (without saying a particular resource), I mean both speed and efficiency as shown in Figure 1-1. Trust me, this will help us to get through the subject effectively. You will learn more about why in "Efficiency: The Key to Pragmatic Code Performance".

Misleading use of the performance word might be the tip of the misconceptions iceberg in this area. We will now walk through many more serious stereotypes and tendencies that are causing the development and our software to worsen. In the best case, it results in more expensive to run or less useful programs. In the worse case, causing severe social and financial organizational problems.

Common Performance Misconceptions

The number of times when I was asked, on my code reviews or sprint planning, to ignore performance "for now" is staggering. And you have probably heard that too! I also rejected someone else's change-set for the same reasons numerous times. Perhaps our changes were dismissed at that time for good reasons, especially if they were micro-optimizations that added unnecessary complexity.

On the other hand, there were also cases where the reasons for rejection were based on common, factual performance misconceptions. Let's try to unpack five of the most damaging misunderstandings. Be cautious when you hear some of those generalized statements. Demystifying them might help you save enormous development costs long term.

Optimized Code is Not Readable

Undoubtedly, one of the most critical qualities of software code is its readability.

(...) it is more important to make the purpose of the code unmistakable than to display virtuosity. (...) The problem with obscure code is that debugging and modification become much more difficult, and these are already the hardest aspects of computer programming. Besides, there is the added danger that a too clever program may not say what you thought it said.

—Brian W. Kernighan and P. J. Plauger, The Elements of Programming Style (1978)

When we think about ultra-fast code, the first thing that sometimes comes to our minds is those clever, low-level implementations with a bunch of byte shifts, magic byte paddings, and unrolled loops. Or worse, pure assembly code linked to your application. Yes, low-level optimizations in this direction can make our code significantly less readable, but let's be honest, those are extreme and rare cases and should be applied ultra carefully.

Code optimizations might produce some extra complexity, increase cognitive load, and make our code harder to maintain. But such risk exists if we add any other functionality or change the code for different reasons. The problem is that engineers tend to connect optimization to complexity to the extreme and avoid performance optimization as fire. In their minds, it translates to an immediate negative readability impact. The point of this section is to show you that there are ways to make performance-optimized code clear. Efficiency and readability can coexist. Similarly, you can add a feature to your program that does not impact your ability to understand code. Refusing to write more efficient code because of fear of losing readability is like refusing to add vital functionality to avoid complexity. Of course, we can consider descoping it, but we should evaluate the consequences first.

For example, when you want to add extra validation to the input, you can naively paste a complex 50 code line's waterfall of `if` statements directly into the handling function, making the next reader of your code cry (Or yourself when you will revisit this code month later). Or, you can encapsulate everything to a single `func validate(input string)`

`error` function, which will add only slight complexity. Alternatively, to avoid modifying the handling block of code, you can design the code to validate it on the caller side or in middleware. We can also rethink our system design and move validation complexity to another system or component, thus not implementing this feature at all.

How are performance improvements in our code different from extra features? I would argue they are not. You can design efficiency optimizations with readability in mind in the same manner as you do with features. Both can impact readability in a variety of ways—from trashing your code to adding little complexity. Both can be entirely transparent to the readers if hidden under abstractions[8].

Yet, we tend to mark optimizations as the primary source of readability problems. The foremost damaging consequence of this and other misconceptions in this chapter is that it's often used as an excuse to ignore performance improvements completely. This often leads to something called "premature pessimization", the act of making the program less efficient, the opposite of optimization.

> *Easy on yourself, easy on the code: All other things being equal, notably code complexity and readability, certain efficient design patterns and coding idioms should just flow naturally from your fingertips and are no harder to write than the pessimized alternatives. This is not premature optimization; it is avoiding gratuitous [author: unnecesary] pessimization.*
>
> —H. Sutter and A. Alexandrescu, C++ Coding Standards: 101 Rules

Readability is essential. I would even argue that unreadable code is rarely efficient in the long haul. When software evolves, it's easy to break previously made too clever optimization because we misinterpret or misunderstand it. Similar to bugs and mistakes, it's easier to cause performance issues in tricky code. In [Link to Come] and [Link to Come], you will see examples of pragmatic efficiency, where the code stays maintainable and easy to read despite having better performance.

> **TIP**
>
> It's easier to optimize readable code than make heavily optimized code readable. If you can't achieve both, in most cases, default to readability.

Optimization often results in less readable code because we don't design good efficiency into our software from the beginning. If you refuse to think about efficiency now, it might be too late to optimize the code later without impacting readability. It's much easier to find a way to introduce a simpler and more efficient way of doing things in the fresh modules where we just started to design APIs and abstractions. As you will learn in Chapter 3, we can do performance optimizations on many different levels, not only via nit-picking and code tuning. Perhaps we can choose a more efficient algorithm, faster data structure, or chose a different system tradeoff. These will likely result in much cleaner, maintainable code and better performance than improving efficiency after releasing the software. Under many constraints, like backward compatibility, integrations, or strict interfaces, our only way to improve performance would be to introduce additional, often significant, complexity to the code or system.

So, if you add new code, don't sacrifice readability. Surprisingly, code after optimization can be more readable! Let's look at a few Go code examples. In Example 1-1, you will see a sub-optimal code, potentially a "pessimization", that I have personally seen hundreds of times when reviewing student or junior developer Go code:

Example 1-1. Simple calculation for the ratio of reported errors.

```
type ReportGetter interface {
   Get() []Report
}

func FailureRatio(reports ReportGetter) float64 { ❶
   if len(reports.Get()) == 0 {
      return 0
   }

   var sum float64
   for _, report := range reports.Get() {
```

```
        if report.Error() != nil {
            sum++
        }
    }
    return sum / float64(len(reports.Get()))
}
```

❶ This is a simplified example, but there is quite a popular pattern of passing a function or interface to get elements needed for operation instead of passing them directly. It is useful when elements are dynamically added, cached, or fetched from remote databases.

I think you would agree that code from Example 1-1 would work for most cases, is simple and readable. Still, I would most likely not accept such code mainly because of potential efficiency issues. I would suggest simple modification as in Example 1-2 instead:

Example 1-2. Simple, more efficient calculation for the ratio of reported errors.

```
func FailureRatio(reports ReportGetter) float64 {
    got := reports.Get() ❶
    if len(got) == 0 {
        return 0
    }

    var sum float64
    for _, report := range got {
        if report.Error() != nil {
            sum++
        }
    }
    return sum / float64(len(got))
}
```

❶ Notice that in comparison with Example 1-1 instead of calling method `Get` in three places, I do it once and reuse the result via the `got` variable.

Some developers could argue that the `FailureRatio` function is potentially used very rarely, it's not on a critical path, and the current `ReportGetter` implementation is very cheap and fast. They could argue that Example 1-1 is more readable and only a slower few nanoseconds due

to three times fewer function calls (`Get`). They could call my suggestion a "premature optimization".

However, I deem it is a very popular case of premature pessimization. It is a silly case of rejecting more efficient code that does not speed up things a lot right now but does not harm either. On the contrary, I would argue that in our example, Example 1-2 is superior in many aspects:

Example 1-2 code is more efficient

> Interfaces allow us to replace the implementation. They represent a certain contract between users and implementations. From the point of view of the `FailureRatio` function, we cannot assume anything beyond that contract. Most likely, we cannot assume that the `ReportGetter` 's `Get` code will always be fast and cheap[9]. Tomorrow, someone might swap the `Get` code with the expensive I/O operation against filesystem, implementation with mutexes or call to the remote database[10]. Users will most likely forget to optimize this function from Example 1-1 when this will happen.

Example 1-2 code is safer

> It is potentially not visible in plain sight, but the code from Example 1-1 has a considerable risk of introducing race conditions. We may hit a problem if the `ReportGetter` implementation is synchronized with other threads that dynamically change the `Get()` result over time. It's better to avoid races and ensure consistency within a function body. Race errors are the hardest to debug and detect, so it's better to be safe than sorry.

Example 1-2 code is more readable

> We might be adding one more line and extra variable, but at the end, the code Example 1-2 is explicitly telling us that we want to use the same result across three usages. By replacing three instances of the `Get()` call with a simple variable, we also minimize the potential side effects,

making our `FailureRatio` purely functional (except the first line). By all means Example 1-2 is thus more readable then Example 1-1.

> **"PREMATURE" OPTIMIZATION IS THE ROOT OF (READABILITY) EVIL**
>
> Such a statement might be accurate, but evil is in the "premature" part. Not every performance optimization is premature. Furthermore, such a rule is not a license for rejecting or forgetting about more efficient solutions with comparable complexity.

Another example of optimized code yielding clarity is visualized by code Example 1-3 and Example 1-4:

Example 1-3. Simple loop without optimization

```
func createSlice(n int) (slice []string) { ❶
   for i := 0; i < n; i++ {
      slice = append(slice, "I", "am", "going", "to", "take",
"some", "space") ❷
   }
   return slice
}
```

❶ Return named parameter called "slice" will create a variable holding an empty `string` slice at the start of the function call.

❷ We append seven `string` items to the slice and repeat that *n* times.

Example 1-3 shows how we usually fill slices in Go, and one would say nothing is wrong here. It just works. However, I would argue that this is not how we should append in the loop if we know exactly how many elements we will append to the slice upfront. Instead, in my opinion, we should always write it as in Example 1-4.

Example 1-4. Simple loop with pre-allocation optimization. Is this less readable?

```
func createSlice(n int) []string {
   slice := make([]string, 0, n*7) ❶
   for i := 0; i < n; i++ {
      slice = append(slice, "I", "am", "going", "to", "take",
"some", "space") ❷
   }
```

```
    return slice
}
```

❶ We are creating a variable holding the string slice. We are also allocating space (capacity) for `n * 7` strings for this slice.

❷ We append seven `string` items to the slice and repeat that `n` times.

We will talk about the efficiency optimizations like in Example 1-2 and Example 1-4 in [Link to Come] with the more profound Go runtime knowledge from Chapter 4. In principle, both allow our program to do less work. In Example 1-4 thanks to initial pre-allocation, internal `append` implementation does not need to extend slice size in memory progressively. We do it once at the start. Now, I would like you to focus on the following question: Is this code more or less readable?

Readability can often be objective, but I would argue the more efficient code from Example 1-4 is more understandable. It adds one more line, so we could say the code is a bit more complex, but at the same time, it is explicit and clear in the message. Not only does it help Go runtime to perform less work, but it also hints to the reader about the purpose of this loop and how many iterations we exactly expect.

If you have never seen raw usage of the built-in `make` function in Go, you probably would say that this code is less readable. That is fair. However, once you realize the benefit and start using this pattern consistently across the code, it becomes a good habit. Even more, thanks to that, any slice creation without such pre-allocation tells you something too. For instance, it could say that the number of iterations is unpredictable, so you know to be more careful. You know one thing before you even looked at the loop's content! To make such a habit consistent across Prometheus and Thanos codebase, we even added a related entry to Thanos Go style guide.

READABILITY IS NOT WRITTEN IN STONE. IT IS DYNAMIC.

The ability to understand certain software code can change over time, even if the code never changes. Conventions come and go as the language community tries new things. With strict consistency, you can help the reader to understand even more complex pieces of your program by introducing a new, clear convention.

Finally, to understand Knuth's "premature optimization is the root of all evil" quote, we need to apply a specific context. While we can learn a lot about general programming from the past, there are many things we improved enormously from what engineers had in 1974. For example, back then, it was popular to add information about the type of the variable to its name as showcased in Example 1-5[11].

Example 1-5. Example of System Hungarian Notation applied to Go code.

```
type structSystem struct {
   sliceU32Numbers []uint32
   bCharacter      byte
   f64Ratio        float64
}
```

Hungarian notation was useful because compilers and Integrated Development Environments (IDEs) were not very mature at that point. However, nowadays, on our IDEs or even repository websites like GitHub, we can hover over the variable to immediately know its type. We can go to the variable definition in milliseconds, read the commentary, find all invocations and mutations. With smart code suggestions, advanced highlighting, and dominance of object-oriented programming started in the mid-1990s, we have tools in our hands that allow us to add features and efficiency optimizations (so complexity) without significantly impacting the practical readability[12]. Furthermore, the accessibility and capabilities of observability and debugging tools have grown enormously, which we will explore in [Link to Come]. It still does not permit clever code but allow us to understand bigger codebases much quicker.

To sum up, performance optimization is like another feature in our software, and we should treat it accordingly. It can add complexity, but there are ways to minimize the cognitive load required to understand our code[13]. As we learned in this chapter, there are even cases when a more efficient program is often a side effect of the simple, explicit, and understandable code. All practical suggestions and code examples you will see in this book have readability in mind.

> **WARNING**
>
> We need performance, but our goal is to make the code both readable and efficient. Ideally, we should never sacrifice readability. Simplify, use abstractions, encapsulation, and commentaries to avoid surprises and hard-to-follow code.

You Aren't Going to Need It (YAGNI)

YAGNI is a powerful and popular rule that I use very often while writing or reviewing any software.

> *One of the most widely publicized principles of XP [Extreme Programming] is the You Aren't Going to Need It (YAGNI) principle. The YAGNI principle highlights the value of delaying an investment decision in the face of uncertainty about the return on the investment. In the context of XP, this implies delaying the implementation of fuzzy features until uncertainty about their value is resolved.*
>
> —Erdogmus, Hakan; Favaro

In principle, it means avoiding doing the extra work that is not strictly needed for the current requirements. It relies on the fact that requirements constantly change, and we have to embrace iterating rapidly on our software.

Let's imagine a potential situation where Katie, a senior software engineer, was assigned the task of creating a simple webserver. Nothing fancy, just an HTTP server that exposes some REST endpoint. Katie is an experienced developer who created probably a hundred similar endpoints in the past.

She went ahead, programmed functionality, and tested the server in no time. Given some time left, she decided to add extra functionality, a simple bearer token authorization (Simple token-based authorization technique) layer. Katie knows that such change is outside of the current requirements, but she had written hundreds of REST endpoints, and each of those had a similar authorization. Experience tells her it's highly likely such requirements will come soon, too, so she will be prepared. Do you think such a change would make sense and should be accepted?

While Katie had shown a good intention and solid experience, we should refrain from merging such change to preserve the quality of the webserver code and overall development cost-effectiveness. In other words, we should apply the YAGNI rule. Why? In most cases, we cannot predict the feature. Sticking to requirements allows us to save time and complexity. There is a risk that the project will never need an authorization layer. For example, if the server is running behind a dedicated authorization proxy. In such a case, the extra code Katie wrote can bring a high cost even if not used. It is an additional code to read, which adds up to the cognitive load. Furthermore, it will be harder to change or refactor such code when needed.

Now, let's step into a more grey area. We explained to Katie why we need to reject the authorization code. She agreed, and instead, she decided to add some critical monitoring to the server by instrumenting it with a few vital metrics. Does this change violate the YAGNI rule too?

If monitoring is part of the requirements, it does not violate YAGNI and should be accepted. If it's not, without knowing the full context, it's hard to say. Critical monitoring should be explicitly mentioned in requirements. Still, even if not, webserver observability is the first thing that will be needed when we will run such code anywhere. Otherwise, how will we know that it is even running? In this case, Katie is technically doing something important that is immediately useful. In the end, we should apply common sense and judgment and add or explicitly remove monitoring from the software requirements before merging such change.

After such a change, in her free time, Katie (who writes solid code very quickly!) decided to add a simple cache to the necessary computation that enhances the performance of the separate endpoint reads. She even wrote and performed a quick benchmark to verify the endpoint's latency and resource consumption improvements. Does that violate the YAGNI rule?

The sad truth about software development is that performance efficiency and response time are often missing from stakeholders' requirements. The target performance goal for an application is to "just work" and be "fast enough", without details on what that means. We will discuss on how to define practical software performance requirements in [Link to Come], commonly known nowadays as Service Level Objectives (SLO). For this example, let's assume the worst. There was nothing in the requirements list about performance. Should we then apply YAGNI and reject Katie's change?

Again, hard to tell without full context. Implementing a robust and usable cache is not trivial, so how complex is the new code? Is the data we are working on easily "cachable"?[14] Do we know how often such an endpoint will be used (is it a critical path)? How far should it scale? On the other hand, computing the same result for a heavily used endpoint is highly inefficient, so cache is a good pattern.

I would suggest Katie takes a similar approach as she did with monitoring change—consider discussing with the team to clarify the performance guarantees that web service should offer. That will tell us if the cache is required now or it is violating the YAGNI rule.

As the last change, Katie went ahead and applied a simple efficiency rule mentioned in Thanos Style Guide. For instance, in relevant places, she implemented the slice pre-allocation improvement you learned from example Example 1-4. Should we accept such a change?

I would be strict here and say yes. Maybe you noticed that when we talked about readability, I suggested optimizing Example 1-4 always when you know the number of elements upfront. Isn't that violating the core statement

behind the YAGNI rule? Even if something is generally applicable, you shouldn't do it before you are sure *You Are Going to Need It*?

I would argue that such small efficiency habits that do not reduce code readability (some even improve it) should generally be an essential part of the developer job, even if not explicitly mentioned in requirements. Similarly, no project requirements state basic best practices like code versioning, having small interfaces, or avoiding big dependencies.

The main takeaway here is that using the YAGNI rule helps, but it is not permission for developers to completely ignoring performance efficiency. As you will learn from Chapter 3, it's usually thousands of small things that make up excessively resource usage and latency of an application, not just a single thing we can fix later. Ideally, well-defined requirements help clarify your software's efficiency needs, but they will never cover all details and best practices we should try to apply nevertheless.

Hardware is Getting Faster and Cheaper

Undoubtedly hardware is more powerful and less expensive than ever before. We see technological advancement on almost every front every year or month. From single-core Pentium CPUs with a 200 MHz clock rate in 1995 to smaller, energy-efficient CPUs capable of 3-4 GHz speeds. RAM sizes increasing from dozens of MB in 2000 to 64 GB in personal computers 20 years later, with faster access patterns. In the past, small capacity hard disks, moving to SSD, then 7GB/s fast NVME SSD disks with few TB of space. Network interfaces achieving 100 Gigabits throughput. In terms of remote storage, I remember Floppy Disks with 1.44 MB of space, then read-only CD-ROMs with a capacity of up to 553 MB, next we had Blue-Ray, read-write capability DVDs, now it's easy to get SD cards with TB sizes.

Now let's add to the above facts the popular opinion that amortized hourly value of typical hardware is cheaper than the developer hour. With all of this, one would say that it does not matter if a single function in code takes 1 MB more or does excessive disk reads. Why should we delay features,

educate or invest in performance-aware engineers if we could buy bigger servers and overall pay less?

As you probably imagine, this is not that simple. Let's unpack this quite harmful argument descoping the efficiency and performance from the software development todo list.

First of all, stating that spending more money on hardware is cheaper than investing the expensive developer time into efficiency topics is very short-sighted. It is like claiming that we should buy a new car and sell an old one every time something breaks, only because we don't want to care, learn or pay for an automotive mechanics job. Sure, that can work, but it would not be a very efficient approach. In many cases, that is simply a misleading oversimplification.

Let's assume a software developer's annual salary oscillates around $100,000. With other employment costs, let's say the company has to pay $120,000 yearly, so $10,000 monthly. For $10,000 in 2021, you can buy a server with 1 TB of DDR4 memory, two high-end CPUs, 1 Gigabit network card, and 10 TB of hard disk space. Let's ignore for now the energy consumption cost. Such a deal means that our software can over-allocate terabytes of memory every month, and we would still be better off than hiring an engineer to optimize this, right? Unfortunately, it does not work like this.

First of all, terabytes of allocation are more common than you think, and you don't need to wait a whole month! Figure 1-2 shows a screenshot of the heap memory profile of a single replica (of six totally), of a single service (of dozens) running in a single cluster, and we run thousands of clusters at Red Hat. We will discuss how to read and use profiles in [Link to Come], but the particular screenshot in Figure 1-2 shows the total memory allocated since the last restart of the process five days before.

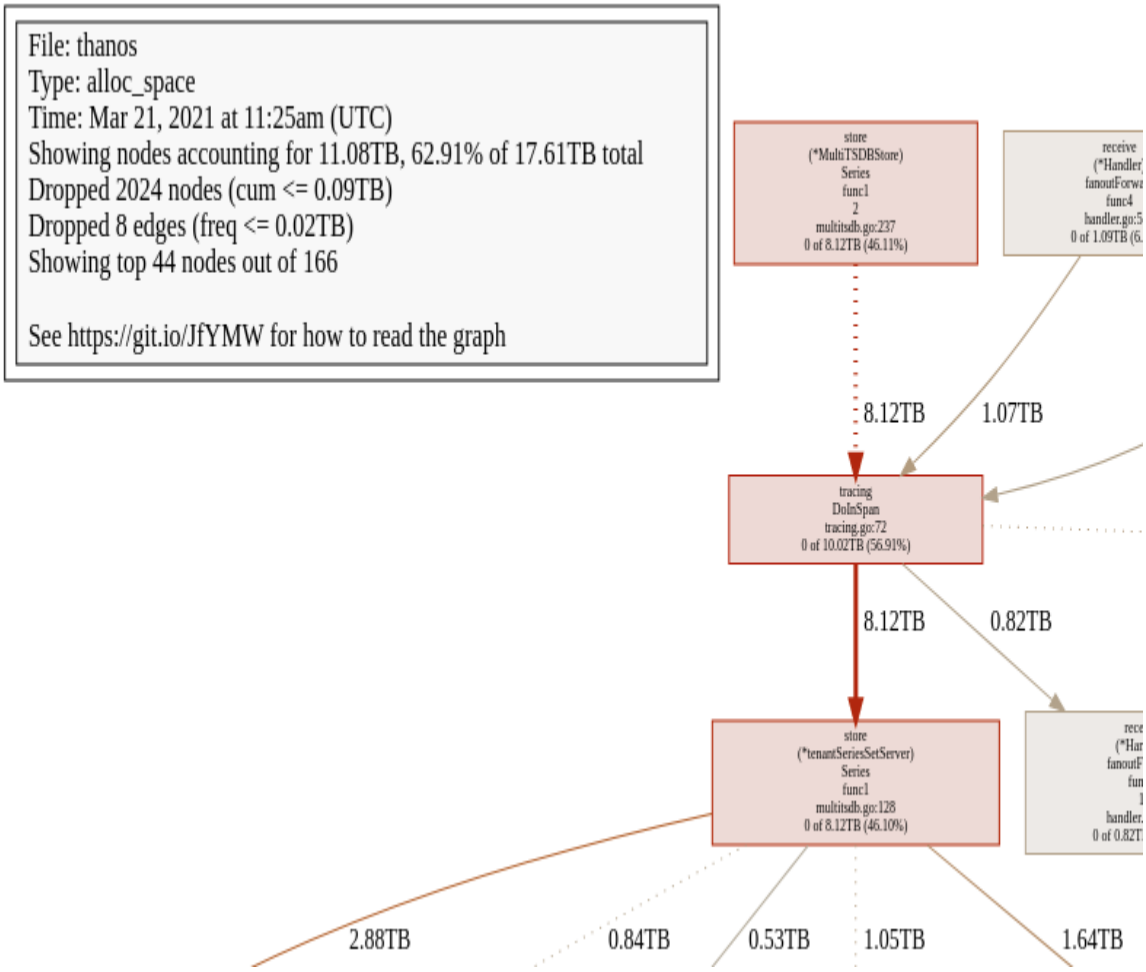


Figure 1-2. Snippet of memory profile showing all memory allocations within five days made by high traffic service. See the full profile here

Most of that memory was already released, but notice that this software from the Thanos project used 17.61 TB in total for only five days of running[15]. If you write desktop applications or tools instead, you will hit a similar scale issue sooner or later. Taking the previous example, if one function is over-allocating 1MB, it is enough to run it 100 times for critical operation in our application with only 100 desktop users to get to 10 TB wasted in total. Not in a month, but on a single run done by 100 users. As a result, slight inefficiency can create overabundant hardware resources quickly.

There is more. To afford an over-allocation of 10 TB, it is not enough to buy a server with that much memory and pay for energy consumption. The amortized cost, among other things, has to include writing, buying, or at

least maintaining firmware, drivers, operating system, and software to monitor, update and operate such server. Since for extra hardware, we need additional software, by definition, it requires spending money on engineers, so we are back where we were. We might have saved engineering costs by avoiding focusing on performance optimizations. In return, we would spend more on other engineers required to maintain over-used resources or pay a cloud provider which already calculated such extra cost, plus their profit into the cloud usage bill.

On the other hand, today, 10 TB of memory costs a lot, but tomorrow it might be a marginal cost due to technological advancements. What if we ignore performance problems and wait until server cost decreases or more users replace their laptops or phones with faster ones. Waiting is easier than debugging tricky performance issues!

Unfortunately, we cannot skip software development efficiency and expect hardware advancements to mitigate needs and performance mistakes. Hardware is getting faster and more powerful, yes. But, unfortunately, not fast enough. Let's go through three main reasons behind that this not intuitive effect.

First of all, there is a saying that "software expands to fill the available memory". This effect is known as Parkinson's law[16]. It states that no matter how many resources we have, the demands tend to match the supply. For example, Parkinson's Law is heavily visible in universities. No matter how much time the professor would give for the assignments or exam preparations, students will always use all of it and probably do most of it last minute[17]. We can see similar behaviour in software development too.

The second reason is that software tends to get slower more rapidly than hardware becomes faster. Niklaus mentions a "Fat Software" term that explains why there will always be more demand for more hardware.

> *Increased hardware power has undoubtedly been the primary incentive for vendors to tackle more complex problems (...). But it is not the inherent complexity that should concern us; it is the self-inflicted complexity. There are many problems that were solved long ago, but for the same problems, we are not offered solutions wrapped in much bulkier software.*
>
> —Niklaus Wirth, A Plea for Lean Software (1995)

Software is getting slower faster than hardware is getting more powerful because products have to invest in a better user experience to get profitable. Prettier operating systems, glowing icons, complex animations, high definition videos on websites, or fancy emojis that mimic your face expression, thanks to facial recognition techniques. It's a never-ending battle for clients, which brings more complexity, thus computational demands.

On top of that, rapid democratization of software thanks to better access to computers, servers, mobile phones, IoT devices, and any other kind of electronics. As Marc Andreessen said, "Software is eating the world". The COVID-19 pandemic that started in late 2019 accelerated digitalization even more as remote, internet-based services became the critical backbone of modern society. We might have more computation power available every day, but more functionalities and user interactions consume all of it and demand even more. In the end, I would argue that our overused 1MB in the aforementioned single function might become a critical bottleneck on such a scale pretty quickly.

If that feels still very hypothetical to you, just look at the software around you. We use social media, where Facebook alone generates 4 PB[18]] of data per day. We search online, causing Google to process 20 PB of data per day. However, one would say those are rare, planet-scale systems with billions of users. Typical developers don't have such problems, right? When I looked at most of the software co-created or used, they hit some performance issue related to significant data usage sooner or later. Prometheus UI page, written in React, was performing a search on millions of metric names or tried to fetch hundreds of MBs of compressed samples,

causing browser latencies and explosive memory usage. Single Kubernetes at our infrastructure, with low usage, are generating 0.5 TB of logs per day (most of those never used). The excellent grammar checking tool I used to write this book is making too many network calls when text has more than 20 000 words, slowing my browser considerably. Our simple script for formatting our documentation in markdown and link checking took minutes to process all elements. Our Go static analysis job and linting exceeded 4GB of memory and crashed our CI jobs. It used to take 20 minutes for my IDE to index all code from our mono-repo, despite doing it on a top-shelf laptop. I still haven't edited my 4K ultra-wide videos from GoPro because the software was too laggy. I could go forever with those examples, but the point is that we live in a really "Big Data" world.

> *When I started programming we not only had slow processors, we also had very limited memory — sometimes measured in kilobytes. So we had to think about memory and optimize memory consumption wisely.*
>
> —V. Simonov; https://va.lent.in/optimize-for-readability-first/[Optimize for readability first (2014)]

While we can acknowledge that hardware was more constrained in the past, this does not change the fact that we can now ignore optimizations. Because of big data, we have to optimize memory and other resources wisely, if not wiser than before. It also will be much worse in the future. Our software and hardware have to handle the data growing at extreme rates, faster than any hardware development. We are just on the edge of introducing 5G networks capable of transfers up to 20 Gigabits per second. We introduce mini-computers in almost every item we buy like TVs, bikes, washing machines, freezers, desk lamps, or even deodorants!. We call this movement "Internet of Things" (IoT). Data from those devices are estimated to grow from 18.3 ZB in 2019 to 73.1 ZB by 2025[19]. The industry can produce 8K TVs, rendering resolutions of 7,680 by 4,320, so approximately 33 million pixels. If you ever wrote any computer game, you are probably as scared as me. It will take a lot of efficient effort to render so many pixels in highly realistic games with immersive, highly destructive environments at 60+ frames per second. Modern cryptocurrencies and blockchain algorithms

also pose challenges in computational energy efficiencies, with Bitcoin energy consumption increased in February 2021 to roughly 130 Terawatt-hours of energy (0.6% of global electricity consumption).

The last reason, but not least, behind not fast enough hardware progression is that hardware advancement stalled on some fronts like CPU speed (clock-rate) or memory access speeds. We will cover some challenges of that situation in Chapter 4, but I believe every developer should be aware of the fundamental technological limits we are hiting right now.

It would be odd to read a modern book about efficiency that does not mention Moore's Law, right? You probably heard of it somewhere before. It was first stated in 1965 by former CEO and co-founder of Intel, Gordon Moore.

> *The complexity for minimum component costs [author: the number of transistors, with minimal manufacturing cost per chip] has increased at a rate of roughly a factor of two per year. (...) Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000.*
>
> —Gordon E. Moore, Cramming more components onto integrated circuits

Mr Moore's observation had a big impact on the semiconductor industry. But decreasing the transistors' size would not have been that beneficial if not for Rober H. Dennard and his team. In 1974 their experiment revealed that power use stays proportional to the transistor dimension (constant power density)[20]. This means that smaller transistors were more power-efficient. In the end, both laws promised exponential performance per watt growth of transistors. It motivated investors to continuously research and develop ways to decrease the size of MOSFET[21] transistors. We can also fit more of them on even smaller, more dense microchips, which reduced manufacturing costs. The industry continuously decreased the amount of space needed to fit the same amount of computing power, enhancing any

chip, from CPU, through RAM and Flash memory to GPS receivers and high definition camera sensors.

In practice, Moore's prediction lasted not ten years as he thought, but nearly 60 so far and still holds. We continue to invent tinier, microscopic transistors, currently oscillating around ~70nm. Probably we can make them even smaller. Unfortunately, as we can see on Figure 1-3 we reached the physical limit of Dennard's Scaling around 2006[22].

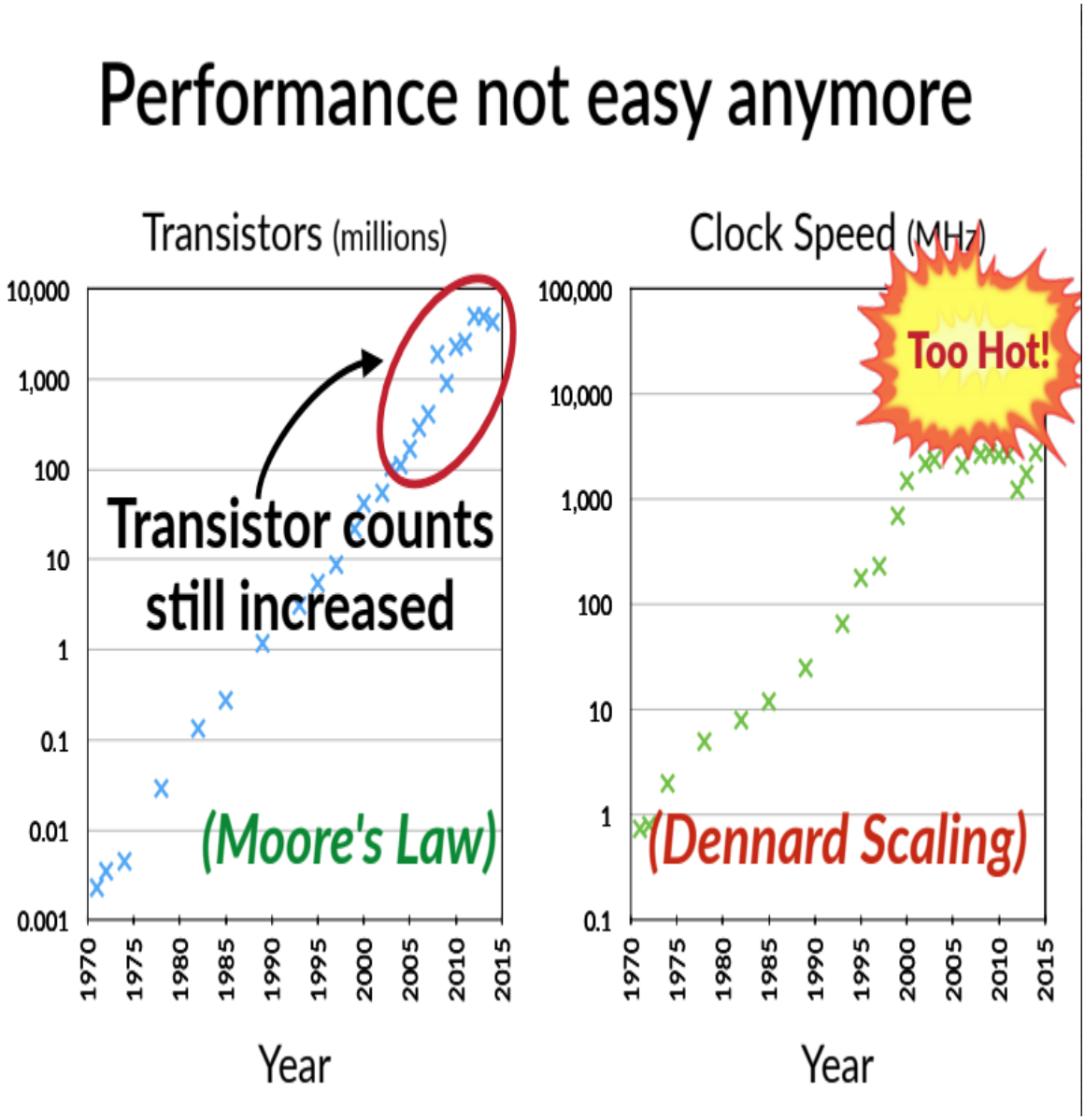


Figure 1-3. Image from Why we cannot have faster transistors: Moore's Law vs Dennard's Rule

While technically, power usage of the higher density of tiny transistors remains constant, such dense chips heat up quickly. Beyond 3-4 GHz of clock speed, it takes significantly more power and other costs to cool those transistors to keep them running. As a result, unless you plan to run software on the bottom of the ocean[23], you are not getting CPUs with faster instruction execution anytime soon. We only can have more cores.

So, what we have learned so far? Hardware speed is getting capped, the software is getting bulkier, and we have to handle continuous growth in data and users. Unfortunately, that's not the end. There is a vital resource we tend to forget about while developing the software: power. Every computation of our process takes electricity, which is heavily constrained on many platforms like mobile phones, smartwatches, IoT devices, or laptops. Non-intuitively there is a strong correlation between energy efficency and software speed and efficiency. I love the Chandler Carruth presentation, which explained this surprising relation well:

> *If you ever read about "power-efficient instructions" or "optimizing for power usage", you should become very suspicious. (...) This is mostly total junk science. Here is the number one leading theory about how to save battery life. Finish running the program. Seriously, race to sleep. The faster your software runs, the less power it consumes. (...) Every single general-usage microprocessor you can get today, the way it conserves power is by turning itself off. As rapidly and as frequently as possible.*
>
> —Chandler Carruth, Efficiency with Algorithms, Performance with Data Structures

To sum up, avoid the common trap of thinking about hardware as a continuously faster and cheaper resource that will save us from optimizing our code. It's a trap. Such a broken loop makes engineers gradually lower their coding standards in performance and demand more and faster hardware. Cheaper and more accessible hardware then creates even more mental room to skip efficiency and so on. There are amazing innovations like recent Apple's M1 Silicon[24], RISC-V standard[25] and more practical

Quantum computing appliances, which promise a lot. Unfortunately, as of 2021, hardware is growing slower than software efficiency needs.

If you are not yet convinced that we should take a little bit more thought on software efficiency, there is one more very important and, sadly, often forgotten point. Software developers are often "spoiled" and detached from the typical human reality. It's often the case that engineers create and test the software on premium, high-end laptop or mobile devices. We need to realize that many people and organizations are and will be utilizing older hardware or worse internet connections[26]. It might be the case that people will have to run your applications on slower computers. It might be worth considering efficiency in our development process to improve our software's overall accessibility and inclusiveness.

We Can Scale Horizontally Instead

As we learned in the previous sections, we expect our software to handle more data sooner or later. But it's unlikely your project will have billions of users from day one. We can avoid enormous software complexity and development cost by pragmatically choosing a much lower target number of users, operations, or data sizes to aim for at the beginning of our development cycle. We usually simplify the initial programming cycle by assuming a low number of notes in your mobile note-taking app, a lower number of requests per second in the proxy you build, or smaller files in the data converter tool your team is working on. It's ok to simplify things, but as you will learn in Chapter 3, it's also important to roughly predict performance requirements in the early design phase. Similarly, it's essential to find the expected load and usage in the mid to long term of software deployment. The software design that guarantees similar performance levels, even with increased traffic, is called "scalable". Generally, scalability is very difficult and expensive to achieve in practice.

> *Even if a system is working reliably today, that doesn't mean it will necessarily work reliably in the future. One common reason for degradation is increased load: perhaps the system has grown from 10,000 concurrent users to 100,000 concurrent users, or from 1 million to 10 million. Perhaps it is processing much larger volumes of data than it did before. Scalability is the term we use to describe a system's ability to cope with increased load.*
>
> —Martin Kleppmann, Designing Data-Intensive Application

Inevitability, while talking about performance, we might touch on some scalability topics in this book. For this chapter's purpose, we can distinguish the scalability of our software into two types.

The first and sometimes the simplest way of scaling our application is by running the software on hardware with more resources ("vertical" scalability). For example, we could introduce parallelism for software to use not one but three CPU cores. If the load would increase, we provide more CPU cores. Similarly, if our process is memory intensive, we might bump running requirements and ask for bigger RAM space. The same with any other resource like disk, network or power. Obviously, that does not go without consequences. In the best case, you have that room in the target machine. Potentially you can make that room by re-scheduling other processes to different machines (e.g. when running in the cloud) or closing them temporarily (useful when running on the laptop or a smartphone). In the worse case, you need to buy a bigger computer, a more capable smartphone or laptop. The latter option is usually very limited, especially if you provide software for customers to run on their non-cloud premises. In the end, the usability of resource-hungry applications or websites that scale only vertically is much lower. If you or your customers run your software in the cloud, the situation is a little bit better. You can "just" buy a bigger server. As of 2021, you can scale up your software on the AWS platform up to 128 CPU cores, almost 4 TB of RAM and 14 Gbps of bandwidth[27]. In extreme cases you can also buy an IBM mainframe with 190 cores and 40TB of memory.

It does not take much time to notice that vertical scalability has its limits on many fronts. Even in cloud or datacenters, we simply cannot infinitely scale up the hardware. First of all, giant machines are rare and expensive. Secondly, as we will learn in Chapter 4, bigger machines run into complex issues caused by many hidden single points of failures. Pieces like memory bus, network interfaces, NUMA nodes, and the operating system itself can be overloaded and too slow[28].

This is why engineers found a different way of fulfilling the demand. All thanks to the Internet and advancement in network technologies. Instead of a bigger machine, we might try to offload and share the computation across multiple remote, smaller, less complex, and much cheaper devices! How does it look in practice? To search for messages with the word "home" in a mobile messaging app, we could fetch millions of past messages (or store them locally in the first place) and run regex matching on each. Instead, we can design an API and call remotely a backend system that splits the search into 100 jobs matching 1/100 of the dataset. Instead of building "monolith" software, we could distribute different functionalities to separate components and move to a "microservice" design. Instead of running a game that requires expensive CPUs and GPUs on a personal computer or gaming console, we run it in a cloud and stream the input and output in high resolution.

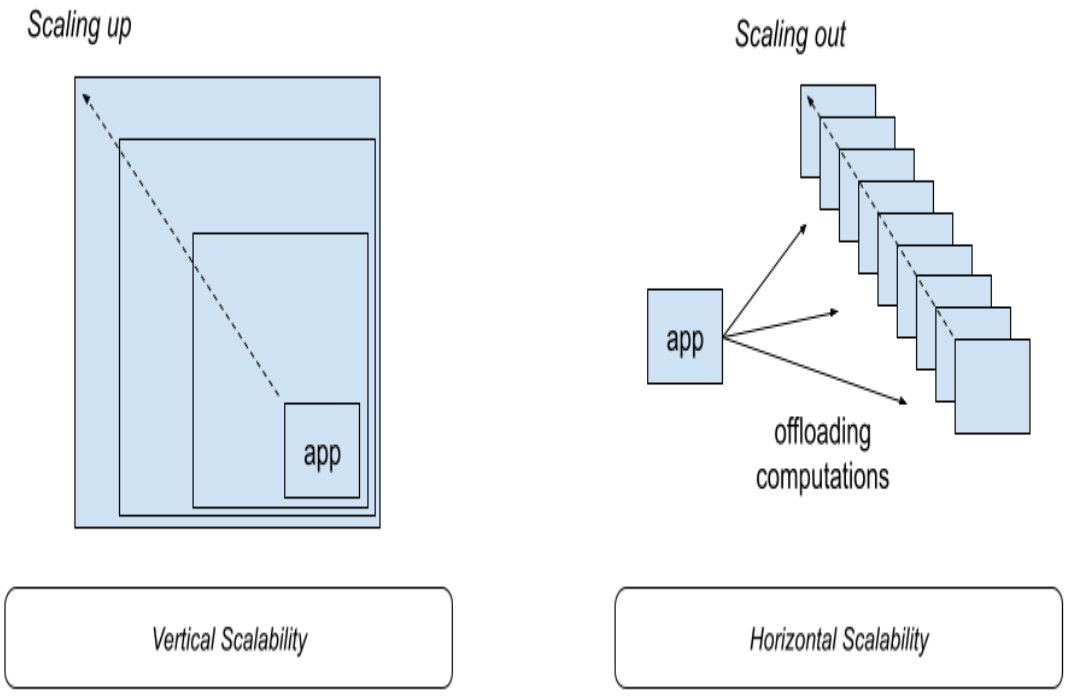


Figure 1-4. Vertical vs Horizontal scalability.

With horizontal and vertical scalability in mind, let me show you a specific scenario from the past. Many modern databases rely on the compaction process to store and look up the data efficiently. We can reuse many indices during this process, deduplicate the same data, and gather fragmented pieces into the sequential stream of data for quicker reads. At the beginning of the Thanos project, we decided to reuse a very naive compaction algorithm for simplicity. We calculated that, in theory, we don't need to make the compaction process parallel within a single block of data. Given a steady stream of 100+GB of eventually compacted data from a single source, we could rely on a single CPU and a minimal amount of memory and some disk space. The implementation was very naive and unoptimized at the very beginning, following good patterns D. Knuth mentioned. We wanted to avoid the complexity and effort of optimizing the project's reliability and functionality features. As a result, users who deployed our project quickly hit compactions problems being either too slow to cope with

incoming data or consuming hundreds of GB of memory per operations. The cost was the first problem, but not the most urgent. The bigger issue was that many Thanos users did not have bigger machines in their datacentres to scale the memory vertically.

At first glance, the compaction problem looked like a scalability problem. The compaction process was depending on resources that we could not just add up infinitely. As users wanted a solution fast, together with the community, we started brainstorming potential horizontal scalability techniques. We talked about introducing a compactor scheduler service that would assign compaction jobs to different machines or intelligent peer network using a gossip protocol. Without going into details, both solutions would add an enormous amount of complexity, probably doubling or tripling the complication of running the whole Thanos system. Luckily, it took a few days of brave and experienced developer time to redesign the code for efficiency and performance. It allowed the newer version of Thanos to make compactions twice faster and stream data directly from the disk, allowing minimal peak memory consumption. Three years later, the Thanos project still does not need to bring horizontal scalability for this operation.

It might feel funny now, but, personally, this story is quite scary. We were so close to bringing enormous, distributed system-level complexity. It would be fun to develop for sure, but it would also potentially kill the project's adoption. A similar situation repeated many times in my career in both open and closed source for smaller and bigger projects.

> *We follow two rules in the matter of optimization: Rule 1. Don't do it. Rule 2 (for experts only). Don't do it yet — that is, not until you have a clear and unoptimized solution. (...) optimization makes a system less reliable and harder to maintain, and therefore more expensive to build and operate.*
>
> —Michael A. Jackson, Principles of program design

Hopefully, previous misconceptions advised you to be careful with the strong "don't do optimization" words. Michael had an excellent intention of

deferring the unnecessary complexity. Unfortunately, as presented by the "lucky" Thanos compaction situation, if you follow that rule blindly, you can quickly end up with the premature "pessimization" that forces premature horizontal scalability. In other words, avoiding complexity can bring even bigger complexity. This appears to me as an unnoticed but critical problem in the industry. It is also one of the main reasons why I wrote this book. We might hit the need for scalability practices significantly quicker than we thought if we defer software performance and efficiency improvements. It is a massive trap because, with some optimization effort, we might even altogether avoid jumping into complications of the scalability methods.

The complications come from the fact that complexity has to live somewhere. We don't want to complicate code, so we have to complicate the system, which, if built from non-efficient components, wastes not only resources but also an enormous amount of developer or operator time. Horizontal scalability is complex. By design, it involves network operations. As we might know from CAP Theorem[29], we inevitably hit either availability or consistency issues as soon as we start distributing our process. Trust me, mitigating those elemental constraints, dealing with race conditions, and understanding the world of network latencies and unpredictability is a hundred times more difficult than extra complexity in our code hidden behind `io.Reader` interface.

It might feel to you that this section touches only infrastructure systems. That's not true. It applies to all software. If you write a frontend software or dynamic website, there might be a temptation to move small client computations to the backend. We should probably only do that if such computation depends on the load and grows out of userspace hardware capabilities. Moving it to the server prematurely might cost you in the complexity caused by extra network calls, more error cases to handle and server saturations causing DOS [30]. Another example comes from my experience. My master thesis was about "Particle Engine Using Computing Cluster". In principle, the goal was to add to the game in the Unity3D a particle engine. The trick was that such a particle engine was not supposed

to operate on client machines but offloading “expensive” computation to a nearby supercomputer in my University called “Tryton”[31]. Guess what? Despite the ultra-fast InfiniBand network[32], all particles I tried to simulate (realistic rain and crowd) was much slower and less reliable when offloaded to our supercomputer. It was not only less complex but also much faster to compute all locally.

Summing up, when someone says, “don’t optimize, we can just scale horizontally”, be very suspicious. Generally, it is simpler and cheaper to start from efficiency improvements before we escalate to a scalability level. On the other hand, there should also be a judgment that tells you when optimizations are becoming too complex, and scalability might be a better option. You will learn more about that in Chapter 3.

Time to Market is More Important

Time is expensive. One aspect of this is that software developer time and expertise cost a lot. The more features you want your application or system to have, the more time is needed to design, implement, test, secure and optimize the solution’s performance. The second aspect is that the more time a company or individual spends to deliver the product or service, the longer their “time to market” is, which can hurt the financial results.

> *Once time was money. Now it is more valuable than money. A McKinsey study reports that, on average, companies lose 33% of after-tax profit when they ship products six months late, as compared with losses of 3.5% when they overspend 50% on product development.*
>
> —Charles H. House and Raymond L. Price, The Return Map: Tracking Product Teams (1991)

It’s hard to measure such impact, but your product might no longer be pioneering when you are “late” to market. You might miss valuable opportunities or respond too late to a competitor’s new product. That’s why companies mitigate this risk by adopting agile methodologies or POC (Proof of Concept) and MVP (Minimal Viable Product) patterns.

Agile and smaller iterations help, but in the end, to achieve faster development cycles, companies try other things too: scale their teams (hire more people, redesign teams), simplify the product, do more automation or do partnerships. Sometimes they try to reduce the product quality. As Facebook's proud initial motto was "Move fast and break things"[33], it's very common for companies to descope software quality in areas like code maintainability, reliability and performance to "beat" the market.

This what our last fifth misconception is all about. Descoping your software's efficiency and performance quality to get to the market faster is not always the best idea. It's good to know the consequences of such a decision. Know the risk first.

> *Optimization is a difficult and expensive process. Many engineers argue that this process delays entry into the marketplace and reduces profit. This may be true, but it ignores the cost associated with poor-performing products (particularly when there is competition in the marketplace).*
>
> —Randall Hyde, The Fallacy of Premature Optimization (2009)

Bugs, security issues, and poor performance happen, but it might damage the company. Without looking too far, let's look at a game released in late 2020 by the biggest polish game publisher, CD Project. The "Cyberpunk 2077" game was known to be a very ambitious, open world, massive and high-quality production. Well marketed, from a publisher with a good reputation, despite the delays, excited players around the world bought 8 million pre-orders. Unfortunately, when released in December 2020, the otherwise excellent game had massive performance issues. It had bugs, crashes, and a low frame rate (slowness) on all consoles and most PC setups. The game on some older consoles like PS4 or Xbox One was claimed to be unplayable.

Interestingly, for me, the game was fabulous despite all the problems. I pre-ordered Cyberpunk 2077 and played it on PS4 pro from day 1. The game had some issues, but it was not making the product unplayable, as others were admitting. It was crashing once every second hour, it was sometimes

super slow and laggy, but it was a fully playable, fun and otherwise polished game. There were, of course, follow up updates with plenty of fixes and drastic improvements over the following months.

Unfortunately, it was too late. The damage was done. The issues, which for me felt somewhat minor, were enough to shake the CD Project's financial perspectives. Five days after launch, the company lost one-third of its stock value costing founders more than one billion dollars. Millions of players were asking for game refunds. Investors sued the CD Project over game issues by investors and famous lead developers were leaving the company. Perhaps the publisher will survive and recover. Still, one can only imagine the implications of broken reputation impacting future productions.

More experienced and mature organizations know well the critical value of software performance, especially the client-facing ones. Amazon found that if their website were loading one second slower, they would lose $1.6 billion annually. They also reported that "100ms of latency costs 1% of profit"[34]. Google realized that slowing down web search from 400ms to 900ms caused a 20% drop in traffic. For some businesses, it's even worse. It was estimated that if a broker's electronic trading platform is five milliseconds slower than the competition, it could lose 1% of its cash flow, if not more. If 10ms slower, this number grows to a 10% drop in revenue.

Realistically speaking, it's true that millisecond-level slowness might not matter in most software cases. For example, let's say we want to implement a file converter from PDF to DOCX. Does it matter if the whole experience last 4 seconds or 100 milliseconds? In many cases, it does not. However, when someone puts that as a market value and a competitor's product has a latency of 200 milliseconds, code efficiency and speed suddenly will be a matter of winning or losing customers. And if it's physically possible to have such fast file conversion, competitors will try to achieve that sooner or later. This is also why so many projects, even in the open-source, are very loud about their performance results. While sometimes it feels like a cheap marketing trick, this works because if you have two similar solutions with similar feature sets, you will pick the fastest one.

It’s not all about the speed too. During my experience as a consultant around infrastructure systems, I saw many cases when customers migrated away from solutions requiring a larger amount of RAM or disk storage, even if that meant some loss in functionalities[35].

To me, the verdict is simple. If you want to win the market, skipping efficiency and performance in your software might not be the best idea. Don’t wait with optimization for the last moment. On the other hand, time to market is critical, so the crucial aspect is to balance a good enough amount of performance work into your software development process. This book aims to help you find that healthy threshold and reduce the time required to improve the efficiency of your software.

At this point, you should be aware of five main misconceptions and why they can mislead us when planning software development. There is a single pattern to all of them - over simplifications. Let’s look at what that means.

Be Vigilant to Simplifications

All those misconceptions are not malicious in any form. It’s only natural for humans to oversimplify complex concepts and processes. Such generalization is often damaging, in the best case, causing unnecessary battles on which programming language or IDE is the best or if we should put this performance optimization ticket on the current sprint or not. In the worst case, simplifications are a source of more serious conflicts and stereotypes. Through the millenniums, we developed the tactic of reducing complex topics to a simple heuristic that can be intuitively executed by the “feeling” part of the brain when needed[36]. Such simplifications might feel helpful at first glance. However they can also be incredibly misleading.

We talked through five misconceptions that were excuses to avoid thinking about efficiency or speed. To reinforce this, let’s quickly unpack the simplification that was created based on Donald Knuth’s famous “premature optimization is the root of all evil” quote:

> *The full version of the [Donald's Knuth] quote is "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil." and I agree with this. It's usually not worth spending a lot of time micro-optimizing code before its obvious where the performance bottlenecks are. But, conversely, when designing software at a system level, performance issues should always be considered from the beginning. A good software developer will do this automatically, having developed a feel for where performance issues will cause problems. An inexperienced developer will not bother, misguidedly believing that a bit of fine-tuning at a later stage will fix any problems.*
>
> —Randall Hyde, The Fallacy of Premature Optimization (2009)

Interestingly enough, if one would read the whole article where Donald Knuth put the quote, one would notice that it was more in favour of optimizations than against. The article was basically about writing efficient but also readable programs without `goto` statements. Nevertheless, premature optimization can be evil. And the essence of that statement is to understand when efficiency and speed optimization are premature. We will discuss that balance in details in Chapter 3.

Nothing is purely white or black. There is always something in the middle. Simplifications are natural and will always be tempting. The only thing we can do is teach others and ourselves not to take things to the extreme. We cannot categorize performance focus as something incredibly wrong or good without full context.

Let's now look on what is the pragmatic way to think about performance.

Efficiency: The Key to Pragmatic Code Performance

In "Behind Performance", we learned that performance splits into accuracy, speed and efficiency. I mentioned that in this book, I use efficiency in replacement for both speed and efficiency. The reason is that there is a

practical suggestion hidden in that effort regarding how we should think about our code performing in the production. The lesson here is to stop thinking about the code performance regarding how fast it does things. Generally, for non-specialized software, speed only marginally matters. It is the waste and unnecessary consumption of resources, which translates to wasting time itself, which often stops us from achieving software goals like usability in a cost-effective way. Sadly, efficiency is often overlooked.

Let's say you want to travel from city A to city B across the river. You can grab a fast car and drive around through the nearby bridge to get to city B quickly. But if you jump into the water and slowly swim across the river, you will be much faster in the city. Slower actions can still be faster when done efficiently, for example, by picking a shorter route. One could say, to improve travel performance and beat the swimmer, we could get a faster car, improve the road surface to reduce drag, or even add a rocket engine. We could potentially beat the swimmer, yes, but those drastic changes might be more expensive than simply doing less work and renting a boat instead.

Similar patterns exist in software. Let's say our algorithm performs search functionality for certain words against specific input, and it performs slowly. Given that we operate on persistent data, the slowest operation is usually the data access, especially if our algorithm does them extensively. It's very tempting to not think about efficiency and instead find a way to convince users to use SSD instead of HDD storage. This way, we could potentially reduce latency even up to 10x. That would be improving performance by increasing the speed element of the equation. On the contrary, if we would find a way to improve the current algorithm to read data only a few times instead of a million, we could achieve even lower latencies. That would mean we do less work to achieve the same effect, improving efficiency.

I want to propose focusing our efforts on efficiency instead of mere execution speed. That is also why this book's title is "Efficient Go", not something more general and catchy[37] like "Ultra Performance Go".

It's not that speed is less relevant. It is important, and as you will learn in Chapter 3, you can have more efficient code that is much slower and vice versa. Sometimes it's a tradeoff you will need to make. Both speed and efficiency are essential. Both can impact each other. In practice, when the program is doing less work on the critical path, it will most likely have lower latency. In the HDD vs SDD example above, changing to a faster disk might allow you to remove some caching logic, which results in better efficiency: less memory and CPU time used. The other way around works sometimes too—as we will learn in "Hardware is Getting Faster and Cheaper", the faster your process is, the less energy it consumes, improving battery efficiency.

I would argue that we generally should focus on improving efficiency before speed as the first step when improving performance. In the context of this book and daily programming in Go, the efficiency element is typically the most impactful and essential. You might be surprised that sometimes after improving efficiency, you achieved desired latency! Let's go through some further reasons why efficiency might be superior:

It is much harder to make efficient software slow

> This is similar to the fact that readable code is easier to optimize. However, as I mentioned before, efficient code usually performs better simply because less work has to be done. In practice, this also translates to the fact that slow software is often inefficient.

Speed is more fragile

> The latency of the software process depends on a huge amount of external factors. One can optimize the code for fast execution in a dedicated and isolated environment, but it can be much slower when left running for a longer time. At some point, CPUs might be throttled due to thermal issues with the server. Other processes (e.g. periodic backup) might surprisingly slow your main software[38] by preempting it from the CPU core or exhausting memory bus (throughput between your CPU and RAM is limited too). The network might be throttled. There are tons of hidden unknowns to consider when we program for mere

execution *speed*. This is why *efficiency* is usually what we, as programmers, can control the most.

Speed is less portable

Like above, if we optimize only for speed, we cannot assume it will work the same when moving our application from the developer machine to a server or between various client devices. Different hardware, environment, operating systems can diametrally change the latency of our application. That's why it's critical to design software for efficiency. First of all, there are fewer things that can be affected. Secondly, if you make two calls to the database on your developer machine, the chances are that you will do the same number of calls, no matter if you deploy it to an IoT device in the space station or ARM-based mainframe.

You can't tell if the software is fast by looking at that code

Efficiency is easier to tell with just static analysis. You either do one iteration or thousands. You either allocate memory for an array of a hundred integers or billions. We can still be surprised about underlying inefficiencies, so we have to understand the runtime resource usage fully. However, we can find things and improve just faster than when improving for pure speed. As you will learn in Chapter 3, we cannot assess speed of our code with our own eyes. Experience helps, but we might be surprised by what's slow and fast in many cases. We need to spend time on benchmarking and load testing to be sure, and that takes time.

Generally, efficiency is something we should do right after or together with readability. We should start thinking about that from the very beginning of the software design. The healthy efficiency awareness, when not taken to the extreme, results in robust development hygiene. It allows us to avoid silly performance mistakes that are hard to improve on in later development

stages. Doing less work also often reduces the overall complexity of the code and improve code maintainability and extensibility.

Summary

I think it's very common for developers to start their development process with compromises in mind. We often sit down with the attitude to negotiate to compromise certain software qualities from the very beginning. We are taught that we have to sacrifice efficiency, readability, testability etc., to accomplish our goals.

In this chapter, I would like to encourage you to hold out and not sacrifice or compromise any quality until it was demonstrated that there is no reasonable way we can achieve all of our goals. Maybe it's not visible at the start, but many things they do have solutions. // *https://youtu.be/3WBaY61c9sE?t=2872*

Let's try to be greedy in software development. Hopefully, at this point, you are aware that we have to think about performance, ideally from the early development stages. We learned what performance consists of. In addition, we learned that there are many misconceptions mentioned in "Common Performance Misconceptions" that are worth reconsidering or challenging when appropriate.

We need to be aware of the risk of premature pessimization and premature scalability as much as we need to think about avoiding premature optimizations.

Finally, we learned that efficiency in the performance equation might give us a bit of an advantage. It is easier to improve performance by improving efficiency first. It helped my students and me many times to effectively approach the subject of performance optimizations.

In the next chapter, we will walk through a detailed and opinionated introduction to Go. Of course, programming for efficiency is essential, but the key to that is knowledge. Let's walk through things to help us achieve all our software quality goals!

1 One example could be an expensive software error causing Boeing 737 Max pilots to take control over the automation, causing two fatal crashes.

2 Out of memory is an undesired state of application where it uses more memory than the host have or the caller allocated for this process. We will learn about this mechanism more in Chapter 4

3 Donald Knuth stated this twice, once in *Structured Programming with goto Statements* (1974), second in *Computer Programming as an Art* (1974). There were debates if this quote should be attributed to C. A. R. Hoare instead, but more recent researches proved against that. The fact that people were trying to find the actual author of the quote speaks of the matter's importance.

4 In a practical sense, there are limits to how fast our software can be. H. J. Bremermann in 1962 suggested there is a computational physical limit that depends on the mass of the system. We can estimate that 1 kg of the ultimate laptop can process ~10^{50} bits per second, while the computer with the mass of the Earth planet can process at a maximum of 10^{75} bits per second. While those numbers feel enormous, even such a large computer would take ages to brute force all chess movements estimated to 10^120 complexity. Those numbers have practical use in cryptography to assess the difficulty of cracking certain encryption algorithms.

5 I even did a small experiment on Twitter, proving this point.

6 UK Cambridge Dictionary defines the performance noun as "How well a person, machine, etc. does a piece of work or an activity".

7 I would even recommend, with your changelog, sticking to common standard formats like *https://keepachangelog.com/en/1.0.0/*. This material also contains valuable tips on clean release notes.

8 It's worth to mention, that hiding features or optimization can sometimes lead to lower readability. Sometimes explicitness is much better and avoids surprises.

9 As the part of the interface "contract", there might be a comment stating that implementations should cache the result. Hence, the caller should be safe to call it many times. Still, I would argue that it's better to avoid relying on something not assured by a type system to prevent surprises.

10 All of those three examples of `Get` implementations could be considered costly to invoke. Input-output (I/O) operations against the filesystem are significantly slower than reading or writing something from memory. Something that involves mutexes means you potentially have to wait on other threads before accessing it. Call to database usually involves all of those, plus potentially communication over the network.

11 This type of style is usually referred to as Hungarian Notation, which used extensively used in Microsoft. There are two types of this notation, too: App and System. Literature indicates that App's Hungarian can still give many benefits.

12 It is worth highlighting that these days, it is recommended to write code in a way that is easily compatible with IDE functionalities, e.g. your code structure should be a "connected" graph. This means that you connect functions in a way that IDE can assist. Any dynamic dispatching,

code injection and lazy loading disables those functionalities and should be avoided unless strictly necessary.

13 Cognitive load is the amount of "brain processing and memory" a person must use to understand a piece of code or function.

14 "Cachability" is often defined as an ability for being cached. It is possible to cache (save) any information to retrieve it later faster. However, the data might be valid only for a short time or only for a tiny amount of requests. If the data depends on external factors (e.g. user or input) and changes frequently, it's not well cachable.

15 That is a simplification, of course. The process might have used more memory. Profiles do not show memory used by memory maps, stack, and many other caches required for modern application to work. We will learn more about this in Chapter 4

16 Cyril Northcote Parkinson was a British historian who articulated the management phenomenon, which is now known as Parkinson's law. Stated as "Work expands to fill the time available for its completion," it was initially referred to as the government office efficiency that highly correlates to the official's number in the decision-making body.

17 At least that's what my studying looked like. This phenomenon is also known as "Student Syndrome".

18 PB means petabyte. One petabyte is 1000 TB. If we assume an average two hours long 4K movie takes 100GB, this means with 1 PB, we could store 10,000 movies, translating to roughly 2,3 years of constant watching.

19 1 zettabyte is 1 million PB, one billion of TB. I won't even try to visualize this amount of data. (:

20 Dennard, Robert H.; "Design of ion-implanted MOSFET's with very small physical dimension" (October 1974)

21 MOSFET stands for "metal–oxide–semiconductor field-effect transistor", which is simply speaking an insulated-gate allowing to switch electronic signals. This particular technology is behind most memory chips and microprocessors produced between 1960 and now. It has proven to be highly scalable and capable of miniaturization. It is the most frequently manufactured device in history, with 13 sextillion pieces produced between 1960 and 2018, *https://en.wikipedia.org/wiki/MOSFET*

22 Funnily enough, marketing reasons led companies to hide the inability to reduce the size of transistors effectively by switching CPU generation naming convention from transistor gate length to size of the process. 14nm generation CPUs still have 70nm transistors, similar to 10, 7, and 5nm processes.

23 I am not joking, Microsoft has proven that running servers 40 meters underwater is a great idea that improves energy-efficiency.

24 M1 chip is a great example of interesting tradeoff, choosing speed and both energy and performance efficiency over the flexibility of hardware scaling

25 RISC-V is an open standard for the instruction set architecture allowing easier manufacturing compatible "reduced instruction set computer" chips. Such a set is much simpler and allows

more optimized and specialized hardware than general-usage CPUs

26 To ensure developers understand and emphasize with users that have a slower connection, Facebook introduced "2G Tuesdays" that turn on the simulated 2G network mode on the Facebook app

27 That option is not as expensive as we might think. Instance type `x1e.32xlarge` costs $26.6 per hour, so "only" $19,418 per month

28 Even hardware management has to be different for machines with extremely large hardware. That's why Linux kernels have special hugemem type of kernels that can manage up to 4 times more memory and ~8 times more logical cores for x86 systems.

29 CAP used to be core system design principle. Its acronym comes from Consistency, Availability and Partition Tolerance. It defines a simple rule that only two of those three can be achieved.

30 Denial of Service, is a state of the system that makes the system unresponsive, usually due to malicious attack. It can also be trigged "accidentally" by the unexpectedly large load.

31 Around 2015, it was the faster supercomputer in Poland, offering 530,5 TFLOPS and almost 2000 nodes, most of them with dedicated GPUs

32 InfiniBand is a high-performance network communication standard, especially popular before fabric optic was invented.

33 Funny enough, Mark Zuckerberg on F8 conference in 2014 announced a change of the famous motto to "Move fast with stable infra".

34 *http://radar.oreilly.com/2008/08/radar-theme-web-ops.html*

35 "One example I see often, in the cloud-native world is moving logging stack from Elasticsearch to simpler solutions like Loki. Despite the lack of configurable indexing, the Loki project can offer better logging read performance with a smaller amount of resources

36 There are many good books around split brain phenomenons and its consequences. Some of them I can recommend are: "Thinking Fast and Slow" by Daniel Kahneman and "Everything is F*ucked" by Mark Manson

37 There is also another reason. "Efficient Go" name is very close to one of the best documentation pieces you might find about the Go programming language: "Effective Go"! It might also be one of the first pieces of information I have read about Go. It's specific, actionable and I recommend reading it if you haven't.

38 This situation is often called a "noisy neighbour".

Chapter 2. Efficient Introduction to Go

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at mpotter@oreilly.com.

Go is efficient, scalable, and productive. Some programmers find it fun to work in; others find it unimaginative, even boring. (...) those are not contradictory positions. Go was designed to address the problems faced in software development at Google, which led to a language that is not a breakthrough research language but is nonetheless an excellent tool for engineering large software projects.

—Rob Pike, Go at Google: Language Design in the Service of Software Engineering (2012)

BEFORE WE START, A SMALL DISCLAIMER

I am a huge fan of Go. I think Go achieves an amazing balance between ease of reading or writing and staying relatively low-level, allowing some control of runtime efficiency. All while ensuring best practices to ensure our code works reliably. It might be the most practical, all-rounder programming language in the world.

The number of things developers around the world were able to achieve with Go is impressive. For a few years in a row, Go has stayed on the list of top 5 languages people love or want to learn. It is used in many businesses, including the biggest tech companies like Apple, American Express, Cloudflare, Dell, Google, Netflix, Red Hat, Twitch and more. Of course, as with everything, nothing is perfect. I would probably change, remove or add a couple of things to Go, but If you would wake me up in the middle of the night and ask to write quickly, reliable backend code, I would write it in Go. CLI? In Go. Quick, reliable script? In Go as well. The first language to learn as a junior programmer? Go. Code for IoT, robots and microprocessors? The answer is also Go[1]. Infrastructure configuration? As of 2021, I don't think there is a better tool for robust templating than Go too.[2].

Don't get me wrong, there are languages with a specialized set of capabilities or ecosystems that are superior to Go. For example, think about Graphic User Interfaces (GUIs), advanced rendering parts of the game industry or code running in browsers.[3] However, once you realize many advantages of the Go language, it is pretty painful to jump back to others.

In Chapter 1, we spent some time establishing a particular efficiency awareness for our software. As a result, we learned that our goal is to write efficient code with the least development effort and cost. This chapter will explain why the Go programming language can be a solid option to achieve this balance between performance and other software qualities.

We will start with a brief introduction to Go in "Basics You Should Know About Go" then with a set of more advanced features in "Advanced Language Elements". Both sections list the short but essential facts everyone should know, something I wish I had when I started my journey with Go in 2014. It will cover much more than just basic information related to efficiency and can be used to introduce yourself to Go. However, if you are entirely new to the language, I would still recommend reading those sections, then check other resources like "A Tour of Go", write your first program in Go and then get back to this book. On the other hand, if you consider yourself a more advanced user or expert, I would suggest not

skipping "Basics You Should Know About Go" and "Advanced Language Elements". I explain few less known facts about Go that you might find interesting (or controversial!).

Last but not least, we will finish with answering the tricky question about the overall. Go performance capabilities in "Is Go "Fast"?", compared to other languages.

Basics You Should Know About Go

Go is an open-source project maintained by Google within a distributed team called "Go Team". The project consists of the programming language specification, compilator, tooling, documentation and standard libraries.

To quickly understand Go basics and its characteristics in fast forward mode, let's go through some facts and basic best practices. While some advice here might feel opinionated, this is all based on my experience working with Go since 2014—a background full of incidents, past mistakes and lessons learned the hard way. I'm sharing them here, so you don't need to make those errors.

Imperative, Compiled and Statically Typed Language

The central part of the Go project is the general-purpose language with the same name, primarily designed for systems programming. As you notice in our Example 2-1 below, Go is an imperative language, so we have (some) control over how things are executed. In addition, it's statically typed, and compiled which means that the compiler can perform many optimizations and checks before the program has ever run. Those characteristics alone are an excellent start to make Go suitable for reliable and efficient programs.

Example 2-1. Simple program printing "Hello World" and exiting.

```
package main

import "fmt"

func main() {
```

```
    fmt.Println("Hello World!")
}
```

Both project and language are called `Go`, yet sometimes you can refer to them with `Golang`.

> **GO VS GOLANG.**
>
> As the rule of thumb, we should always use the `Go` name everywhere, unless it's clashing with the English word `go`, or an ancient game called `Go`. `Golang` came from the domain choice (*https://golang.org*) since `go` was unavailable to authors. So use `Golang` when doing web searches for resources about this programming language.

Go also has its own mascot, called "Go gopher", presented in Figure 2-1.

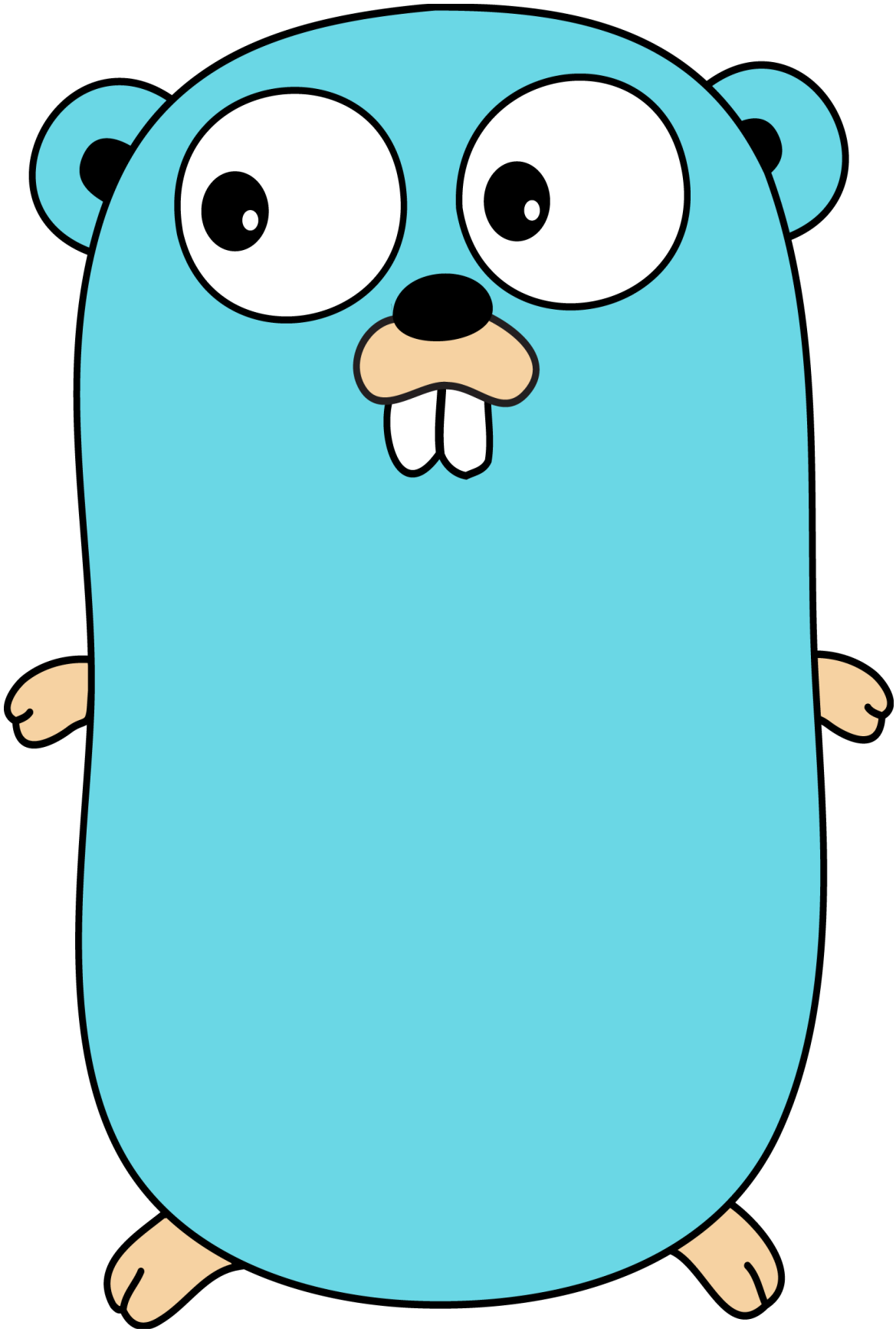


Figure 2-1. Go Gopher (credits: Renee French)

We see this cute Gopher in various forms, situations, and combinations, i.e., conference talks, blog posts, or project logos. Sometimes Go developers are called “gophers”, too!

Designed to Improve Serious Codebases

It all started when three experienced programmers from Google sketched the idea of the Go language around 2007.

Rob Pike

> Co-creator of UTF-8 and Plan 9 operating system. Co-author of many programming languages before Go e.g. Limbo for writing distributed systems and Newsqueak for writing concurrent applications in graphical user interfaces. Both were inspired by Hoare’s Communicating Sequential Processes (CSP).[4]

Robert Griesemer

> Among other work, he developed Sawzall language and did a doctorate with Niklaus Wirth. (The same Niklaus that was mentioned many times in Chapter 1).

Ken Thompson

> One of the original authors of the first Unix system. Sole creator of `grep` command-line utility. Ken co-created UTF-8 and Plan 9 together with Rob Pike. He wrote a couple of languages, too, e.g. Bon and B programming languages.

They aimed to create a new programming language that was meant to improve mainstream programming, led by `C++`, Java and Python at that point. After a year, it became a full-time project, with Ian Taylor and Russ Cox joining what was later referenced as “Go Team” in 2008. Go Team

announced the public Go project in 2009, with version 1.0 released in March 2012.

The main frustrations related to `C++` mentioned in the design of Go were:

- Complexity, many ways of doing the same thing, too many features.
- Ultra-long compilation times, especially for bigger codebases.
- Cost of updates and refactors in large projects.
- Not easy to use and prone to the errors memory model.

Those elements are why Go was born, from the frustration of existing solutions and ambition to allow more by doing less. The guiding principles were to make a language that does not trade off safety for less repetition yet allows simpler code. It does not sacrifice execution efficiency for faster compilation or interpreting yet ensures build times are quick enough. *https://talks.golang.org/2012/splash.article#TOC_5* [Go tries to compile as fast as possible, e.g. thanks to explicit imports]. Especially with caching enabled by default, only changed code is compiled, so build times are rarely longer than a minute.

YOU CAN TREAT GO CODE AS SCRIPT!

While technically, Go is a compiled language, you can run it like you would run Javascript, Shell or Python. It's as simple as invoking `go run <executable package> <flags>`. It works great because the compilation is ultra-fast. You can treat it like a scripting language while maintaining the advantages of compilation.

In terms of syntax, Go was meant to be simple, light on keywords, familiar. Basing syntax on C with type derivation (automatic type detection, like `auto` in `C++`), no forward declarations, no header files. Concepts are kept orthogonal, which allows easier combination and reasoning about them. Orthogonality for elements means that, for example, we can add methods to any type, any data definition (adding methods is separate to creating types). Interfaces are orthogonal to types too.

Governed by Google, Yet Open Source

Since announcing Go, all development is done in open, with public mailing lists and bug trackers. Changes go to the public, authoritative source code, held under the BSD style license. The Go team reviews all contributions. The process is the same if the change or idea is coming from Google or not. Project roadmap and proposals are developed in public too.

Unfortunately, the sad truth is that there are many open-source projects, but some projects are less open than others. Google is still the only company stewarding Go and have the last decisive control over it. Even if anyone can modify, use and contribute, projects coordinated by a single vendor risk selfish decisions or relicensing. While there were some controversial cases where the Go team decision surprised the community[5], overall, the project is very reasonably well-governed. Countless changes came from outside of Google, and especially the Go 2.0 draft proposal process has been well respected and community-driven. In the end, I believe consistent decision making and stewarding from the Go team brings a lot of benefits too. Conflicts and different views are inevitable, and having one consistent overview, even if not perfect, might be better than no decision or many ways of doing the same thing.

So far, this project setup is proven to work well for adoption and language stability. For our software efficiency goals, such alignment couldn't be better too. We have a big company invested in making sure each release does not bring any performance regressions. These days lots of internal Google software depends on Go, e.g. Google Cloud Platform. And many people rely on Googe Cloud Platform to be reliable. On the other hand, we have a vast Go community that gives feedback, finds bugs, contributes ideas and optimizations. And if that's not enough, we have open source code, allowing us, mere mortal developers, to dive into the actual Go libraries, runtime ("Go Runtime") etc., to understand the performance characteristics of the particular code.

Simplicity, Safety and Readability are Paramount

Robert Griesemer mentioned in GopherCon 2015 that they, first of all, when they first started building Go, they knew what things NOT to do. The main guiding principle was that simplicity, safety, and readability are paramount. In other words, Go follows the pattern of *Less is More*. This is a potent idiom that spans many areas. In Go, there is only one *idiomatic* coding style[6], and a tool called `gofmt` ensures most of it. In particular, code formatting (next to naming) is an element that is almost never settled among programmers. We spend time arguing about it and tuning it to our specific needs and beliefs. Thanks to a single style enforced by tooling, we save an enormous amount of time. As one of Go proverb goes, "*Gofmt's style is no one's favourite, yet Gofmt is everyone's favourite.*". Overall, the Go authors planned the language to be minimal so that there is essentially one way to write a particular construct. This takes away a lot of decision making when you are writing a program. There is one way of handling errors, one way of writing objects, one way of running things concurrently etc.

A huge number of features might be "missing" from Go, yet one could say it is more expressive than `C` or `C++`. Such minimalism allows maintaining simplicity and readability of Go code, which improves software reliability, safety and overall higher velocity towards application goals.

IS MY CODE IDIOMATIC?

The `Idiomatic` word is heavily overused in the Go community. Usually, it means Go patterns that are "often" used. Since Go adoption has grown a lot, there are many creative ways people have improved the initial "idiomatic" style. Nowadays, it's as clear as the "This is the way" saying from Mandalorian Series. Use this word with care and avoid it unless you can elaborate further.

Interestingly, the "Less is More" idiom can help our efficiency efforts for this book's purpose. As we learned in Chapter 1, if you do less work in runtime, it usually means faster, lean execution and less complex code. In this book, we will try to maintain this aspect while improving our code performance.

Packaging and Modules

Code is organized into packages and modules. Source code can be placed in different directories. A package is a collection of source files (with the `.go` suffix) in the same directory. The package name is specified with the `package` statement at the top of each source file, as seen in Example 2-1. All files in the same directory have to have the same package name[7] (the package name can be different from the directory name). Multiple packages can be part of a single Go Module. A module is a directory with `go.mod` file that states all dependent modules with their versions required to build the Go application. This file is then used by the dependency management tool Go Modules. Each source file in such a module can then import packages from the same module or external modules. Some packages can also be "executable". For example, if a package is called `main` and has `func main()` in some file, we can execute it. Sometimes such a package is placed in the `cmd` directory for easier discovery. Note that you cannot import the executable package. You can only build or run it.

Within the package, you can decide which functions, types, interfaces and methods are exported to package users and which are accessible only in package scope. This is important because it's better to export the minimal amount of API possible for readability, reusability, and reliability. Go does not have any `private` or `public` keywords for this. Instead, it takes a slightly new approach. As Example 2-2 shows, if the construct name starts with an upper case letter, any code outside the package can use it. If the element name begins with a lower case letter, it's private. It's worth noting that this pattern works for all constructs equally, e.g. functions, types, interfaces, variables etc. (orthogonality).

Example 2-2. Construct accessibility control using naming case.

```
package main

const privateConst = 1
const PublicConst = 2

var privateVar int
var PublicVar int
```

```
func privateFunc() {}
func PublicFunc()  {}

type privateStruct struct {
   privateField int
   PublicField  int ❶
}

func (privateStruct) privateMethod() {}
func (privateStruct) PublicMethod()  {} ❶

type PublicStruct struct {
   privateField int
   PublicField  int
}

func (PublicStruct) privateMethod() {}
func (PublicStruct) PublicMethod()  {}

type privateInterface interface {
   privateMethod()
   PublicMethod() ❶
}

type PublicInterface interface {
   privateMethod()
   PublicMethod()
}
```

❶ Careful readers might notice tricky cases of exported fields or methods on private type or interface. Can someone outside of the package use them if the struct or interface itself is private? This is quite rarely used, but the answer is yes, you can return a private interface or type in public function e.g. `func New() privateStruct { return privateStruct{}}`. All its public fields and methods are usable to the user of our package.

INTERNAL PACKAGES

You can name and structure your code directories as you want to form packages, but one directory name is reserved for special meaning. If you want to make sure only the given package can import other packages, you can create a package subdirectory named `internal`. Any package under the `internal` directory can't be imported by any other package than ancestor (and other packages in `internal`).

Dependencies Transparency by Default

In my experience, it was common to import some libraries, e.g. in C++, C# or Java in compiled form (e.g. JAR file), and use exported functions and structures/classes mentioned in some headers or documentation. Importing compiled code had some benefits. Notably, it was hiding hard to obtain, indirect dependencies, special compilation tooling or extra resources to build it. It was also easier to sell a coding library for close source purposes without exposing the source code.[8] In principle, this is meant to work well. Developers of the library maintain specific programmatic contracts (API), and users of such libraries do not need to worry about the complexities of implementations.

Unfortunately, in practice, this is rarely that perfect. Implementation can be broken or inefficient, the interfaces can mislead, and documentation can be missing. In such cases, access to the source code is invaluable, allowing us to understand implementation deeper. We can find issues based on code, not by guessing, and propose a fix to the library author or fork the package and use it immediately. We can extract the required pieces and use them to build something else.

Go assumes this imperfection by requiring each library's parts (in Go: module's packages) to be explicitly imported using a package URI called "import path". Such import is also strictly controlled, i.e. unused imports or cyclic dependencies are causing a compilation error. Let's see some of the different ways to declare those imports in Example 2-3.

Example 2-3. Portion of import statements from `github.com/prometheus/prometheus` module, `main.go` file.

```
import (
   "context" ❶
   "net/http"
   _ "net/http/pprof" ❷

   "github.com/oklog/run" ❸
   "github.com/pkg/errors"
   "github.com/prometheus/common/version" ❺
   "go.uber.org/atomic"
```

```
    "github.com/prometheus/prometheus/config" ❹
    promruntime "github.com/prometheus/prometheus/pkg/runtime"
    "github.com/prometheus/prometheus/scrape"
    "github.com/prometheus/prometheus/storage"
    "github.com/prometheus/prometheus/storage/remote"
    "github.com/prometheus/prometheus/tsdb"
    "github.com/prometheus/prometheus/util/strutil"
    "github.com/prometheus/prometheus/web"
)
```

❶ If the import declaration does not have a domain with a path structure, it means that the "standard" package is imported. Such a package has to have its source available in the Go version you have installed in your environment. This particular import, allows to use code from `$(go env GOROOT)/src/context/` directory with `context` reference e.g `context.Background()`.

❷ Package can be imported explicitly without any identifier. This means that we don't want to reference any construct from this package, but we want to have some global variables initialized. In this case, the `pprof` package will add debugging endpoints to the global HTTP server router. While allowed, in practice, we should avoid reusing global, modifiable variables.

❸ Non-standard packages can be imported using an import path in a form of an internet domain name and an optional path to the package in a certain module. `go` tooling integrates well with `https://github.com` so if you host your Go code in a git repository it will find a specified package for you. In this case, it's the `https://github.com/oklog/run` git repository with the `run` package in the `github.com/oklog/run` module.

❹ If the package is taken from the current module (in this case, our module is `github.com/prometheus/prometheus`), packages will be resolved from your local directory. In our example, `<module root>/config`.

❺ It might be confusing, but `github.com/prometheus/common` import is from a different module, so it's downloaded upstream.

This model focuses on open and clearly defined dependencies. It works exceptionally well with the open-source distribution model, where the community can collaborate on robust packages in the public git repositories. Of course, a module or package can also be hidden using standard version

control authentication protocols. Furthermore, at the current moment, the official tooling does not support distributing packages in binary form, so dependency source is highly encouraged to be present for compilation purposes.

The challenges of software dependency are not easy to solve. Go learnt by mistakes of `C++` and others, takes a careful approach to avoid long compilation times, but also an effect commonly called a "dependency hell".

> *Through the design of the standard library, great effort was spent on controlling dependencies. It can be better to copy a little code than to pull in a big library for one function. (A test in the system build complains if new core dependencies arise.) Dependency hygiene trumps code reuse. One example of this in practice is that the (low-level) net package has its own integer-to-decimal conversion routine to avoid depending on the bigger and dependency-heavy formatted I/O package. Another is that the string conversion package strconv has a private implementation of the definition of printable characters rather than pull in the large Unicode character class tables; that strconv honors the Unicode standard is verified by the package's tests.*
>
> —Rob Pike, Go at Google: Language Design in the Service of Software Engineering (2012)

Again, with efficiency in mind, potential minimalism in terms of dependencies and transparency bring enormous value. Fewer unknowns means we can quickly detect main bottlenecks and focus on the most significant value optimizations first. If we notice a potential room for optimization in our dependency, we don't need to work around it. Still, instead, we are usually welcome to contribute the fix directly to the upstream, which helps both sides!

Consistent Tooling

From the very beginning of Go, it had a powerful and consistent set tool as a part of its command-line interface tool called `go`. Let's enumerate a few utilities:

- `go bug` opens a new browser tab with the correct place where you can fill an official bug report (go repository on GitHub).
- `go build -o <output path> <packages>` builds given Go packages.
- `go env` shows all Go-related environment variables currently set in your terminal session.
- `go fmt <file, packages or directories>` formats given artefacts to the desired style, clean whitespaces, fix wrong indentations etc. Note that the source code does not need to be even valid and compilable Go code. You can also install an extended official formatter called
- `goimports` also cleans and formats your import statements.

> **TIP**
>
> For the best experience, set your programming IDE to run `goimports -w $FILE` on every file save, to not worry about the indentation anymore.

- `go get <package@version>` allows you to install the desired dependency with the expected version. Use the `@latest` suffix to get the latest version of `@none` to uninstall dependency.
- `go help <command/topic>` prints documentation about the command or given topic. For example, `go help environment` tells you all about possible environment variables Go uses.
- `go install <package>` similar to `go get` and install the binary if the given package is “executable”.
- `go list` lists Go packages and modules. It allows flexible output formatting using Go templates (explained later) e.g `go list -mod=readonly -m -f '{{ if and (not .Indirect)`

`(not .Main)}}{{.Path}}{{end}}' all` lists all direct non-executable dependant modules.

- `go mod` allows managing dependant modules.
- `go test` allows running unit, fuzz test and benchmarks. We will discuss the latter in detail in [Link to Come].
- `go tool` hosts a dozen of more advanced CLI tools. We will especially take a close look at `go tool pprof ` in [Link to Come] for performance optimizations.
- `go vet` runs basic static analysis checks.

In most cases, `go` CLI is all you need for effective Go programming.[9]

Single Way of Handling Errors

Errors are an inevitable part of every running software. Especially in distributed systems, they are expected by design, with advanced researches and algorithms for handling different types of failures.[10] Despite the commonality of errors, most programming languages do not recommend or enforce a particular way of failure handling. For example, in `C++` you see programmers using all means possible to return an error from a function:

- Exceptions
- Integer return codes (if the returned value is non zero, it means error)
- Implicit status codes[11]
- Other sentinel values (if the returned value is `null`, then it's an error)
- Returning potential error by argument
- Custom error classes
- Monads[12]

Each option has its pros and cons, but just the fact that there are so many ways of handling errors can cause severe issues. It causes surprises by potentially hiding the fact that some statements can return an error, introduce complexity and, as a result, can make our software unreliable.

Undoubtedly, the intention for so many options was good. It allows a developer choice. Maybe the software you create is non-critical, or it is its first iteration, so you want to make "a happy path" crystal clear. In such cases masking some "bad paths" completely sounds like a good short-term idea, right? Unfortunately, as with many shortcuts, it poses numerous dangers. Software complexity and demand for functionalities causes the code to never go out of the "first iteration", and non-critical code quickly becomes a dependency for something critical. This is one of the most important causes of unreliability or hard to debug software.

Go takes a unique path by treating the error as a first citizen language feature. It assumes we want to write reliable software, making error handling explicit, easy and uniform across libraries and interfaces. Let's see some examples in Example 2-4.

Example 2-4. Defining error flow.

```
func noErrCanHappen() int { ❶
   // ...
   return 204
}

func doOrErr() error { ❷
   // ...
   if shouldFail() {
      return errors.New("ups, XYZ failed")
   }
   return nil
}

func intOrErr() (int, error) { ❸
   // ...
   if shouldFail() {
      return 0, errors.New("ups, XYZ2 failed")
   }
   return noErrCanHappen(), nil
}
```

❶ The critical aspect here is that functions and methods define the error flow as part of their signature. In this case, the `noErrCanHappen` function states that there is no way any error can happen during its invocation.

❷ Just by looking at the `doOrErr` function signature, we know some errors can happen. We don't know what type of error yet though, we only know it is implementing a built-in `error` interface. We also know that if the error is nil, there was no error.

❸ The fact that Go functions can return multiple arguments is leveraged when they want to calculate some result in a "happy path". If the error can happen, it should be the last return argument (always). From the caller side, we should then only touch the result if the error is nil.

It's worth noting that Go also has an exception mechanism called `panics`. Panics are recoverable using the `recover()` built-in function. While useful or necessary for certain cases (e.g. initialization), you should never use panics for conventional error handling in your production code in practice. They are less efficient, hide failures and overall surprise the programmers. Having errors as part of invocation allows both the compilator and programmer to be prepared for error cases in the normal execution path. Example 2-5 shows how we can handle errors if they occur in our functions' execution path.

Example 2-5. Checking and handling errors.

```
import "github.com/pkg/errors" ❶

func main() {
   ret := noErrCanHappen()
   if err := nestedDoOrErr(); err != nil { ❷
      // handle error
   }
   ret2, err := intOrErr()
   if err != nil {
      // handle error
   }
   // ...
}

func nestedDoOrErr() error {
   // ...
```

```
    if err := doOrErr(); err != nil {
       return errors.Wrap(err, "do") ❸
    }
    return nil
}
```

❶ Notice that we did not import the built-in `error` package, but instead, we used external, open-source drop-in replacement `github.com/pkg/errors`. This allows a bit more advanced logic like wrapping errors you will see in <3>

❷ To tell if an error happened, we need to check if the `err` variable is nil or not. If an error occurs, we can follow with error handling. Usually, it means logging it, exiting the program, incrementing metrics, or even explicitly ignoring it.

❸ In some cases, it's appropriate to delegate error handling to the caller. If the function can fail from many errors, consider wrapping it with a `errors.Wrap` function to add a short context of what exactly is wrong.

ERROR WRAPPING

Notice that I recommended `errors.Wrap` (or `errors.Wrapf`) instead of the built-in way of wrapping errors. Go defines the` %w` identifier for the `Sprintf` type of functions that allows passing an error. Currently, I would not recommend `%w` because it's not type-safe and as explicit as `Wrap`, which was causing non-trivial bugs in the past.

The one way of defining errors and handling them is one of Go's best features. Interestingly, it is one of the language disadvantages due to verbosity and certain boilerplate involved. It sometimes might feel repetitive, but tools are allowing you to mitigate the boilerplate.

TIP

Some Go IDEs defines code templates. For example, in JetBrain's GoLand product, writing `err` and pressing the `tab` button will generate a valid `if err != nil` statement. You can also collapse/un-collapse error handling blocks for readability.

Another common complaint is that it can feel very "pessimistic" to write Go because the errors which may never occur are visible in plain sight. The programmer has to decide what to do with them at every step, which takes mental energy and time. Yet, in my experience, it's worth our work and makes our programs much more predictable and easier to debug.

> **NEVER IGNORE ERRORS!**
>
> Due to the verbosity of error handling, it's tempting to skip `err != nil` checks. Consider not doing it unless you know that a function will never return an error (and in future versions!). If you don't know what to do with the error, consider passing it to the caller by default. If you have to ignore the error, consider doing it explicitly with `_ = ` syntax. Also, always use liners, which will warn you about not checked errors.

Are there any implications of the error handling for general Go code runtime efficiency? Yes! Unfortunately, it's much more significant than developers usually anticipate. In my experience, error paths are frequently an order of magnitudes slower and more expensive to execute than happy paths. One of the reasons is we tend not to monitor or test error paths correctly (we will talk about the importance of it in [Link to Come]), so not many realize their impact. Another common reason is that construction errors often involves heavy string manipulation for creating human-readable messages. It can be costly, especially with lengthy debugging tags. We will touch on those elements in [Link to Come]. Understanding those implications and ensuring consistent and efficient error handling is essential in every software, and we will take a detailed look at that in the following chapters.

Strong Ecosystem

A commonly stated strong point of Go is that its ecosystem is exceptionally mature for such a "young" language. While things listed in this section are not mandatory for solid programming dialect, they improve the whole development experience. This is also why the Go community is so large and still growing.

First of all, Go allows the programmer to focus on business logic without necessarily reimplementing or importing third-party libraries for basic functionalities like YAML decoding or cryptographic hashing algorithms. Go standard libraries are high quality, robust, ultra backwards compatible and rich in features. They are well benchmarked, have solid APIs and good documentation. As a result, you can achieve most things without importing external packages. For example, running an HTTP server is dead simple as visualized by Example 2-6:

Example 2-6. Minimal code for serving HTTP requests (not recommended for production, though[13]).

```
package main

import  "net/http"

func handle(w http.ResponseWriter, _ *http.Request) {
   w.Write([]byte("It kind of works!"))
}

func main() {
   http.ListenAndServe(":8080", http.HandlerFunc(handle))
}
```

In most cases, the efficiency of standard libraries is good enough or even better than third party alternatives. For example, especially lower-level elements of packages, `net/http` for HTTP client and server code or `crypto`, `math`, `sort` parts (and more!), have a good amount of optimizations to serve most of the use cases. This allows developers to build more complex code on top while not worrying about the performance of basics like `sorting`. Yet, that's not always the case. Some libraries are meant for specific usage, and misusing them may result in significant resource waste. We will look at all things you need to be aware of in [Link to Come].

Another highlight of the mature ecosystem is a basic, official in-browser Go editor called Go Playground. It's a fantastic tool if you want to test something out quickly or share an interactive code example. It's also

straightforward to extend, so the community often publish variations of Go playground to try and share experimental language features like generics.

Last but not least, the Go project defines its own templating language called Go Templates. In some way, it's similar to Python's Jinja2 language. While it sounds like a side feature of Go, it's beneficial in any dynamic text or HTML generation. It is also often used in popular tools like Helm or Hugo. We will discuss how to use `Go Templates` efficiently in [Link to Come].

Unused Import or Variable Causes Build Error

If you define a variable in Go, but you never read any value from it or don't pass it to another function, compilation will fail. Similarly, if you added a package to the `import` statement, but you don't use that package in your file.

From what I see, Go developers got used to this feature and love it, but it might be surprising for newcomers. Failing on unused constructs can be frustrating if you want to play with the language quickly, e.g. create some variable without using it for debugging purposes. There are, however, ways to handle those cases explicitly! You can see a few examples of dealing with those usage checks in Example 2-7.

Example 2-7. Various examples of unused and used variables.

```
package main

func use(_ int) {}

func main() {
   var a int // error: a declared but not used ❶

   b := 1 // error: b declared but not used ❶

   var c int
   d := c // error: d declared but not used ❶

   e := 1
   use(e) ❷

   f := 1
```

```
    _ = f ❸
}
```

❶ Variables `a`, `b`, and `c` are not used, so they cause a compilation error.
❷ Variable `e` is used.
❸ Variable `f` is technically used for explicit no identifier (`_`). Such an approach is useful if you explicitly want to tell the reader (and compiler) that you want to ignore such value.

Similarly, unused imports will fail the compilation process, so tools like `goimports` (mentioned in "Consistent Tooling") automatically removes unused ones. Overall, failing on unused variables and imports effectively ensures that code stays clear and relevant. Note that only internal function variables are check. Elements like unused struct fields, methods or types are not checked.

Unit Testing and Table Tests

Tests are the mandatory part of every application, small or big. In Go, tests are a natural part of the development process—easy to write, focused on simplicity and readability. If we want to talk about efficient code, we need to have solid testing in place, allowing us to iterate over the program without worrying about regressions. Add a file with the `_test.go` suffix to introduce a unit test to your code within a package. You can write any Go code within that file, which won't be reachable from the production code. There are, however, four types of different functions you can add that will be invoked for different parts of testing. A certain signature distinguishes types, notably function name prefix: `Test`, `Fuzz`, `Example` or `Benchmark` and specific argument. Let's walk through the unit test type in Example 2-8. To make it more interesting, it's a table test. Examples and benchmarks are explained in "Code Documentation as a First Citizen" and [Link to Come].

Example 2-8. Example unit table test.

```
--- max_test.go ---
package max

import (
    "math"
    "testing"
```

```
    "github.com/efficientgo/tools/core/pkg/testutil"
)

func TestMax(t *testing.T) { ❶
    for _, tcase := range []struct { ❷
        a, b     int
        expected int
    }{
        {a: 0, b: 0, expected: 0},
        {a: -1, b: 0, expected: 0},
        {a: 1, b: 0, expected: 1},
        {a: 0, b: -1, expected: 0},
        {a: 0, b: 1, expected: 1},
        {a: math.MinInt64, b: math.MaxInt64, expected:
math.MaxInt64},
    } {
        t.Run("", func(t *testing.T) { ❸
            testutil.Equals(t, tcase.expected, max(tcase.a, tcase.b))
❹

        })
    }
}
```

❶ If function inside `_test.go` file is named with `Test` word and takes exactly `t *testing.T` it is considered a "unit test". You can run them through `go test` command.

❷ Usually, we want to test a specific function using multiple test cases (often edge cases) that define different input and expected output. This is where I would suggest using table tests. Define your input, output and run the same function in an easy to read loop.

❸ Optionally, you can invoke `t.Run`, which allows you to specify a subtest. It's a good practice to define those on dynamic test cases like in table tests. It will enable you to navigate to the failing case quickly.

❹ Go `testing.T` type gives useful methods like `Fail` or `Fatal` to abort and fail the unit test, or `Error` to fail but continue running and check other potential errors. In our example, I propose using a simple helper, called `testutil.Equals`, giving you a nice diff.[14]

I would recommend you writing tests often. It might surprise you, but writing unit tests for critical parts upfront will generally help you implement desired features much faster. This is why I would recommend following some reasonable form of Test-Driven Development, covered in "Efficiency-Aware Development Flow".

To sum up, the above information should give you a good overview of the language goals, strengths, and features before moving to more advanced features in "Advanced Language Elements" next.

Advanced Language Elements

Let's now discuss more advanced features of Go. Similarly to the basics mentioned in the previous section, it's crucial to overview core language capabilities before discussing efficiency improvements.

Code Documentation as a First Citizen

Every project at some point needs solid API documentation. For library-type projects, the programmatic APIs are the main entry point. Robust interfaces with good descriptions allow developers to hide complexity, bring value and avoid surprises. A code interface overview is essential for applications, too, allowing anyone to understand the codebase quicker. It's also not uncommon to reuse an application's Go packages in other projects.

Instead of relying on the community to create many potentially fragmented and incompatible solutions, the Go project developed a tool called `godoc` from the start. In some way, it behaves similarly to Python's Docstring and Java's Javadoc. `godoc` generates consistent documentation HTML website directly from the code and its comments.

The amazing part is that you don't have many special conventions that would make the code comments less readable from the source code directly. To use this tool effectively, you need to remember five things. Let's go through them using Example 2-9. The resulting HTML page, when `godoc` is invoked, can be seen in Figure 2-2.

Example 2-9. Example code with `godoc` compatible documentation.

```
--- block.go ---

// Package block contains common functionality for interacting with
TSDB blocks
// in the context of Thanos.
```

```
package block ❶

import ...

const (
   // MetaFilename is the known JSON filename for meta information.
❷
   MetaFilename = "meta.json"
)

// Download downloads directory that is meant to be block
directory. If any of the files
// have a hash calculated in the meta file and it matches with what
is in the destination path then
// we do not download it. We always re-download the meta file. ❷
// BUG(bwplotka): No known bugs, but if there was one, it would be
outlined here. ❸
func Download(ctx context.Context, id ulid.ULID, dst string) error
{
// ...

// cleanUp cleans the partially uploaded files. ❹
func cleanUp(ctx context.Context, id ulid.ULID) error {
// ...

--- block_test.go ---
package block_test

import ...

func ExampleDownload() { ❺
   if err := block.Download(context.Background(), ulid.MustNew(0,
nil), "here"); err != nil {
      fmt.Println(err)
   }
   // Output: downloaded
}
```

❶ Optional package level description must be placed right on top of the `package` entry with no intervening blank line. If many source files have those entries, `godoc` will collect them all.

❷ Any public construct should have a full sentences commentary starting with the name of the construct (it's important!), similarly right before its definition.

❸ Known bugs can be mentioned with `// BUG(who)` statements.

❹ Private constructs can have comments, but they will never be exposed in documentation anyway since they are private. Be consistent and start them with a construct name, too, for readability.

5. If you write a function named `Example<ConstructName>` in the test file e.g `block_test.go` (package name has to be different too e.g `block_test`), `godoc` will generate an interactive code block with desired examples. Since they are part of the unit test, they will be actively tested against, so your examples stay updated.

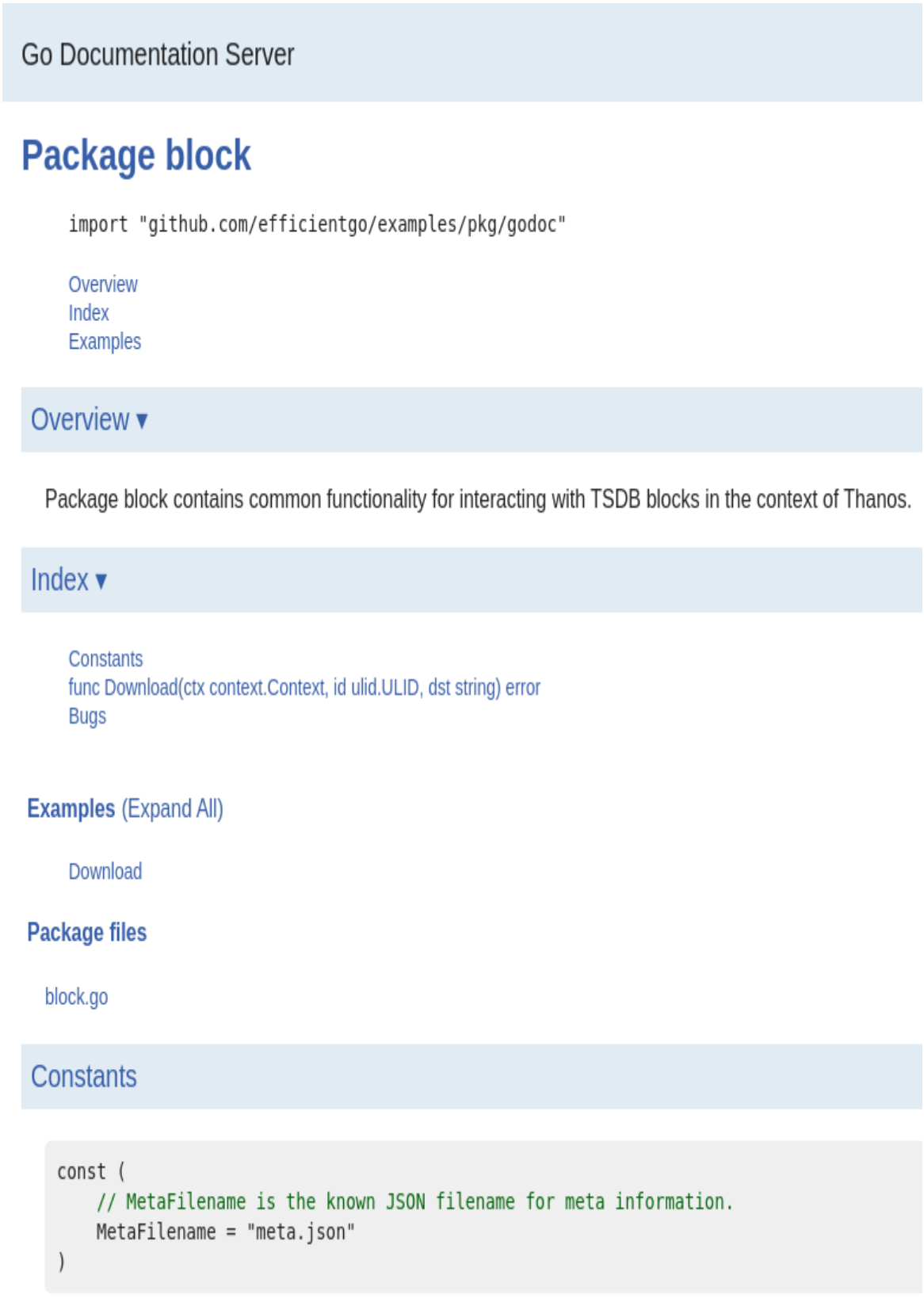


Figure 2-2. `godoc` output of Example 2-9

I highly recommend sticking to those five simple rules. Not only because you can manually run `godoc` and generate yourself documentation web page, but the additional benefit is that those rules make your Go code comments structured and consistent. Everyone knows how to read them and where to find them.

> **TIP**
>
> I recommend using whole English sentences in all comments, even if those will not appear in `godoc`. It will help you keep your code commentary self-explanatory and explicit. After all, comments are for humans to read.

Furthermore, Go Team maintains a public documentation website capable of scraping all public repositories called `https://pkg.go.dev`. Thus, if your public code repository is compatible with `godoc` it will be rendered correctly, and users can read the autogenerate documentation for every module/package version. For example, see our Example 2-9(TODO: Link once code examples are published) web page here.

Backwards Compatibility and Portability

Go has a strong take on backwards compatibility guarantees. This means that core APIs, libraries, and language spec should never break old code created for Go 1.0[15]. This was proven to be well executed. There is a lot of trust in upgrading Go to the latest minor or patch versions in practice. Upgrades are, in most cases, smooth and without significant bugs and surprises.

In terms of efficiency compatibility, it's hard to talk about any guarantees. For both the Go project and any library, there is (usually) no guarantee that the function that does two memory allocations now will not use hundreds in the next version. There have been surprises between the version in the efficiency and speed characteristics. The community is working hard on improving the compilation and language runtime (more in "Go Runtime" and Chapter 4). Since the hardware and operating systems are developed

too, the Go team is experimenting with different optimizations and features to allow everyone to have more efficient execution. Of course, we don't speak here about major performance regression, as those are usually noticed and fixed in the release candidate period. Yet if we want our software to be deliberately fast and efficient, we need to be more vigilant and aware of the changes Go is introducing.

Source code is compiled into binary code that is targeted to each platform. Yet Go tooling allows cross-platform compilation, so you can build binaries to almost all architectures and operating systems.

> **TIP**
>
> When you execute, Go binary which was compiled for a different operating system or architecture, it can return cryptic error messages. A common one is `Exec format error` when you try to run binary for Darwin (macOS) on Linux.

Speaking about portability, we can't skip mentioning the Go runtime and its characteristics.

Go Runtime

Many languages decided to solve portability across different hardware and operating systems using Virtual Machines. Typical examples are Java Virtual Machine (JVM) for Java bytecode compatible languages (e.g. Java or Scala) and Common Language Runtime CLR for .NET code, e.g. C#. Such a virtual machine allows building languages that do not need to worry about complex memory management logic (allocation and releasing), differences between hardware and operating systems, etc. JVM or CLR is interpreting the intermediate byte code and transfers program instructions to the host. Unfortunately, while making it easier to create a programming language, they also introduce some overhead and a significant amount of unknowns.[16] To mitigate the overhead, those virtual machines often use complex optimizations like Just in time (JIT) compilation to process chunks of specific virtual machine byte code to machine code on the fly.

Go does not need any "virtual machine". Our code and used libraries compile fully to machine code during the compilation time. Thanks to standard library support of large operating systems and hardware, our code, if compiled against particular architecture, it will run there with no issues.

Yet, something is running in the background (concurrently) when our program starts. It's the Go runtime logic that among other, minor features of Go is responsible for:

Managing Concurrency

> Go runtime implements one of the most known, unique features of Go. Robust and easy to use concurrency framework called "Go Routines". This framework allows you to run your code concurrently (not parallel![17]) while still maintaining a single process from the machine point of view. The unique part about it is its simplicity. The statement `go func() { /* some code */}()` starts a new routine. This is one of the key elements of how we can make our code run faster (not necessarily more efficiently in terms of resource usage), and we will explore it in detail in "Go Runtime Scheduler".

Memory Management with Garbage Collection

> Go wouldn't be as simple to write as it is now if runtime would not implement solid memory management. Every new construct in Go allocates some memory because it has to be stored somewhere for the program execution duration. Go follows the standard memory model and splits the process virtual memory into two areas stack and heap. The latter is the place for elements that lives longer than the duration of function execution. Go does not require (and allow) manually releasing memory for such items. Instead, Go runtime implements the Garbage Collection routine, which does that as efficiently as possible. This area impacts memory resource consumption and will be discussed in detail in "Garbage Collection".

Go runtime is similar to libc (C runtime) that performs similar duties for languages like C or C++. It is primarily written in Go itself too.[18] In fact, you can tell Go to be compiled with `cgo`, which then allows packages that use C functions and thus are depending and using C runtime.

Object Oriented Programming in Go

Undoubtedly Object-oriented programming (OOP) got enormous traction over the last decades. It was invented around 1967 by Alan Key, and it's still the most popular paradigm in programming.[19] In principle, it allows us to think about code as some objects with attributes (in Go `fields`) and behaviours (`Methods`) telling each other what to do. Most OOP examples talk about some high-level abstractions like an animal that exposes the `Walk()` method or car that allows to `Ride()`, but in practice, objects are usually less abstract, yet still helpful to be encapsulated and described by a class. As we can see in Example 2-10, you can have function `Compact` that takes a list of `Block` objects. `Compact` then translates that slice into `Sortable` object that implements sorting interface `sort.Sort` accepts. After sorting, `Compact` instantiates an object of the `Group` class and uses the `Group` `Add()` method to add all blocks. Since `Group` is also a `Block`, it returns the `Group` object as the compacted block. Object-Oriented Programming allows us to leverage advanced concepts like Encapsulation, Abstraction, Polymorphisms and Inheritance. Example 2-10 shows how we can achieve those vital programming mechanisms in Go.

Example 2-10. Example of the Object-Oriented Programming in Go Part 1.

```
type Block struct {
   id         string
   start, end time.Time
   // ...
}

func (b Block) String() string {  ❶
   return fmt.Sprint(b.id, ": ", b.start.Format(time.RFC3339), "-",
b.end.Format(time.RFC3339))
}

type Group struct {
```

```
    Block ❷
    // ...
}

func (g *Group) Add(b Block) { ❸
    if g.end.IsZero() || g.end.Before(b.end) {
        g.end = b.end
    }
    if g.start.IsZero() || g.start.After(b.start) {
        g.start = b.start
    }
    // ...
}

func Compact(blocks []Block) Block {
    sort.Sort(ToSortable(blocks)) ❹

    g := &Group{}
    for _, b := range blocks {
        g.Add(b)
    }
    return g.Block ❺
}
```

❶ In Go there is no separation between structures and classes. So, on top of basic types like `integer`, `string` etc., a `struct` type can have `methods` (behaviours) and `fields` (attributes). Use structures and methods to **encapsulate** more complex logic under a more straightforward interface, e.g. `String()` method on `Block`.

❷ If we add some struct, e.g. `Block` into another struct, e.g. `Group` without any `field` name, such a `Block` struct is considered `embedded`. Embedding allows us in a simple way to get the most valuable part of **inheritance**, so borrowing the embedded structure fields and methods. In this case, `Group` will have `Block's` fields and `String` method. This way, we can reuse a significant amount of code.

❸ There are two types of `methods` you can define in Go. One with `value` receiver (e.g `String()` method) and `Add()` with `pointer` receiver. “Receiver” is the variable after `func` of the type you add a method to, e.g. `(g *Group)`. It might sound convoluted, but the rule regarding which one to use is straightforward. Use pointer receiver (add `*`) if your method is meant to modify the local receiver state or if any other method does that (for consistency, don’t mix different receiver types within one struct). In our example, if the `Group` `Add()` method

would be a value receiver, we will not persist potentially injected `g.min` and `g.max` values. That's why we need to add a pointer receiver instead.

❹ Standard library sorting function used with our `Sortable` type explained in Example 2-12.

❺ This is the only thing missing for Go to support inheritance fully. Go does not allow casting specific types into another type unless it's an alias or strict single-struct embedding (shown in Example 2-12). You can only cast interface into some type. That's why here we need to specify embedded struct `Block explicitly`.

> **TIP**
>
> You can embed as many unique structures as you want within one `struct`. If the compilator can't tell which method to use, compilation will fail because of the name clash. Use type name to explicitly tell compilator what should be used. For example, taking Example 2-10, you can reference the id variable from `Group` 's embedded `Block` struct by writing `Group{}.Block.id`.

Go also allows defining interfaces that tell what methods struct has to implement to match it. Thus, there is no need to mark a specific struct explicitly that it implements a particular interface. It's enough just to implement specified methods. Let's see an example sorting interface exposed by the standard library in Example 2-11.

Example 2-11. Sorting interface from standard `sort` Go library.

```
// A type, typically a collection, that satisfies sort.Interface
can be
// sorted by the routines in this package. The methods require that
the
// elements of the collection be enumerated by an integer index.
type Interface interface {
   // Len is the number of elements in the collection.
   Len() int
   // Less reports whether the element with
   // index i should sort before the element with index j.
   Less(i, j int) bool
   // Swap swaps the elements with indexes i and j.
```

```
    Swap(i, j int)
}
```

In order to use our type in `sort.Sort` function, it has to implement all `sort.Interface` methods. Example 2-12 shows how `Sortable` type do it.

Example 2-12. Example of the Object-Oriented Programming in Go Part 2.

```
type Sortable []Block ❶

func ToSortable(blocks []Block) sort.Interface { ❷
    var s Sortable = blocks
    return &s
}

func (b *Sortable) Len() int           { return len(*b) }
func (b *Sortable) Less(i, j int) bool { return (*b)
[i].start.Before((*b)[j].start) }
func (b *Sortable) Swap(i, j int)      { (*b)[i], (*b)[j] = (*b)
[j], (*b)[i] }
```

❶ We can embed another type (e.g. slice of `Block's) as the only thing in our `Sortable` struct. This allows easy (but explicit) casting between `Sortable` and `[]Block`.

❷ This function shows that we can return a pointer to an instance of the `Sortable` type as the sorting interface itself, thanks to three implemented methods.

To sum up, those facts are what I found crucial when teaching others programming in Go, based on my own experience with the language. Moreover, it will be helpful when diving more into the runtime performance of Go in the following sections of this book.

However, if you never programmed in Go before, it's worth going through other materials like Tour of Go before jumping to the following sections and chapters of this book. Make sure you try writing on your own basic Go program, write a unit test, use loops, switches, concurrency mechanisms like channels and routines. Learn common types and standard library abstraction. As a person coming to the new language, we need to produce a program returning valid results before ensuring it executes fast and efficiently.

We learned about some basic and advanced characteristics of Go, so it's time to unwrap the efficiency aspects of the language. How is it easy to write high or good enough performance code in Go?

Is Go "Fast"?

Across the recent years, many companies have rewritten their products (e.g. from Ruby, Python, Java) to Go[20]. Two of the repeatedly stated reasons for the move to Go or starting a new project in Go were readability and excellent performance. Readability comes from simplicity and consistency (e.g. single way of error handling as you remember from "Single Way of Handling Errors"), and it's where Go excels, but what about performance? Is Go fast in comparison to other languages like Python, Java or C++?

In my opinion, this question is badly formed. Given time and room for complexities, any language can be as fast as your machine and operating system allow. This is because, in the end, the code we write is compiled to machine code that uses the exact CPU instructions. Additionally, most languages allow delegating execution to other processes, e.g. written in optimized Assembly. Unfortunately, all we sometimes use to decide if language is "fast" are raw, semi-optimized short program benchmarks that compare execution time and memory usage across languages. While it tells something, it effectively does not show practical aspects of it, e.g. how complex programming for efficiency was.[21]

Instead, we should look at programming language in terms of how hard and practical it is to write efficient code (not just fast) and how much readability and reliability such a process sacrifices. I believe the Go language has a superior balance between those elements while keeping it fast and trivial to write basic functional code.

One of the reasons for being able to write efficient code easier is the hermetic compilation stage, relatively small amount of unknowns in Go runtime (as mentioned in "Go Runtime"), ease to use concurrency framework and maturity of debugging, profiling and benchmarking tools (discussed in [Link to Come] and [Link to Come] accordingly).

Those Go characteristics did not appear from thin air. Not many know, but Go was designed on the shoulders of giants: C, Pascal and CSP.

> *In 1960, language experts from America and Europe teamed up to create Algol 60. In 1970, the Algol tree split into the C and the Pascal branch. ~40 years later, the two branches join again in Go.*
>
> —Robert Griesemer, The Evolution of Go (2015)

As we can see on Figure 2-3 many names mentioned in Chapter 1 are grandfathers of Go. Great concurrency language CSP created by Sir Hoare, Pascal declarations and packages created by Wirth and C basic syntax contributed to how Go looks today.

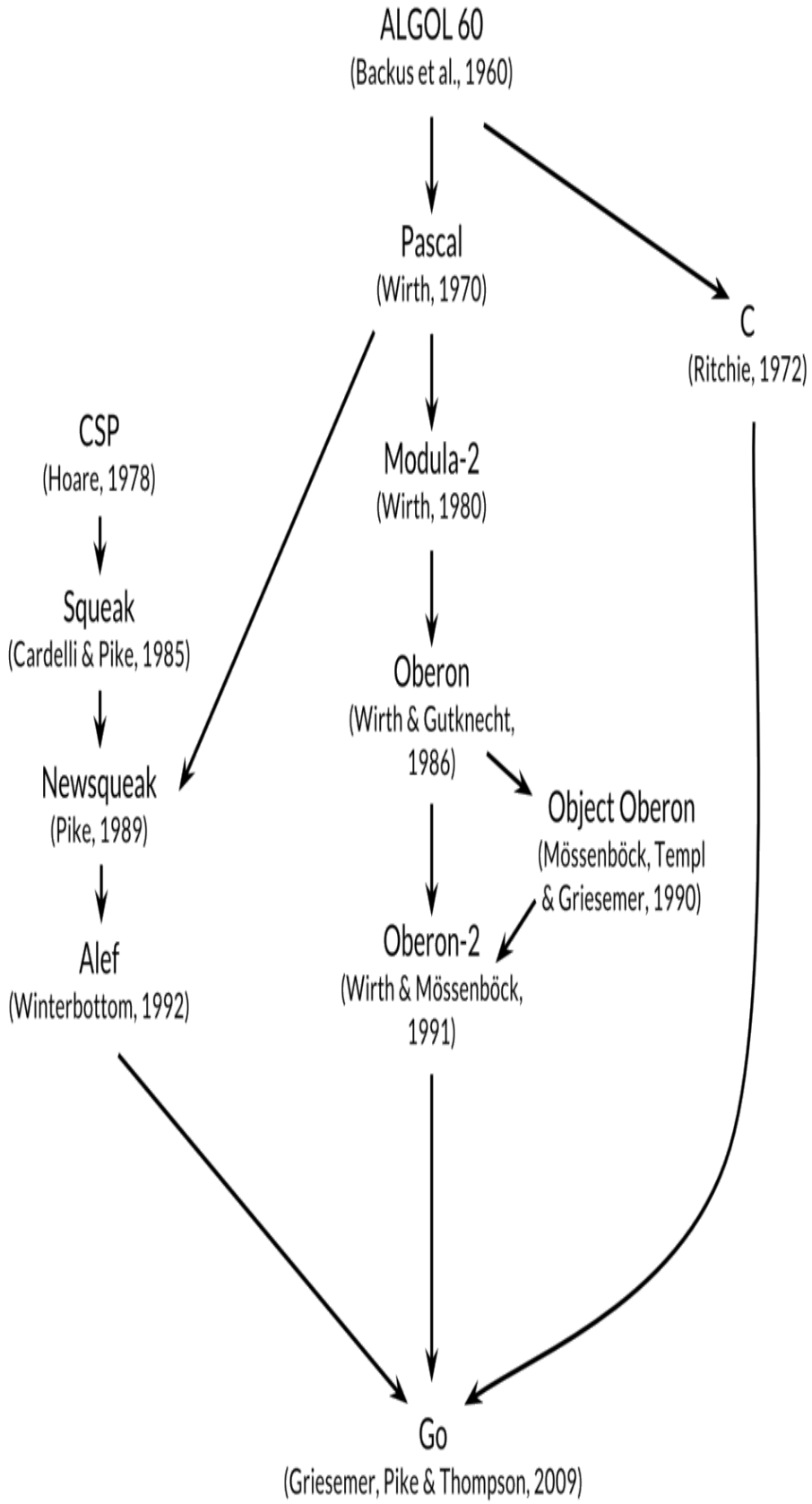

ALGOL 60
(Backus et al., 1960)
Pascal
(Wirth, 1970)
C
(Ritchie, 1972)
CSP
(Hoare, 1978)
Modula-2
(Wirth, 1980)
Squeak
(Cardelli & Pike, 1985)
Oberon
(Wirth & Gutknecht, 1986)
Newsqueak
(Pike, 1989)
Object Oberon
(Mössenböck, Templ & Griesemer, 1990)
Alef
(Winterbottom, 1992)
Oberon-2
(Wirth & Mössenböck, 1991)
Go
(Griesemer, Pike & Thompson, 2009)

Figure 2-3. Go genealogy

But not everything can be perfect. In terms of efficiency, Go has its own Achilles heel. As we will learn in [Link to Come], the Go memory model can sometimes be hard to control. Allocations in our program can be surprising (especially for new users), and the automatic memory release process through garbage collection has some overhead and eventual behaviour. Especially for data-intensive applications, it takes some effort to ensure memory or CPU efficiency. Similar for machines with strictly limited RAM capacities (e.g. IoT).

Yet, the decision to automate this process is highly beneficial, allowing the programmer to not worry about memory cleanup, which has proven to be even worse and sometimes catastrophic (e.g. deallocating memory twice). An excellent example of alternative mechanisms other languages use is Rust. It implements a unique memory ownership model, which replaces automatic, global garbage collection. Unfortunately, while more efficient, it turns out that writing code in Rust is much more complicated than in Go. That's why we see higher adoption of Go. This reflects the ease of use tradeoff the Go team picked in this element.

Fortunately, there are ways to mitigate the garbage collection mechanism's negative performance consequences in Go and keep our software lean and efficient. We will go through those in the following chapters.

Summary

Frankly speaking, I initially considered that perhaps I could be lazy and skip most of the Go introduction—that I could assume if you are reading my `Efficient Go` book that you know the Golang basics already. There are many resources that go into more details for elements I could spend only a subchapter.

However, in the end, I realized that the true power of the Go optimizations, benchmarking, and basic efficiency practices come when used in practice, in everyday programming. Therefore, I want to empower you to merry

efficiency with other good practices around reliability, abstractions or reliability for practical use. While sometimes, some fully dedicated approach has to be built (we will touch on this in [Link to Come]), the basic, but often good enough, efficiency comes from understanding simple rules and language capabilities. That's why I focused on giving you a better overview of Go and its features in this chapter.

With this knowledge, we can now move to Chapter 3 where we will learn how to start our journey that aims to improve the efficiency and overall performance of our program's execution.

---

1 New frameworks on tools for writing Go on small devices are emerging, e.g. GoBot and TinyGo

2 It's a controversial topic. There is quite a battle in the infrastructure industry for the superior language for configuration as code. For example, between HCL, Terraform, Go templates (Helm), Jsonnet, Starlark and Cue. In 2018, we even open-sourced one tool for writing configuration in Go, called "mimic" Arguably, the loudest arguments against writing configuration in Go are that it feels too much like "programming" and requires programming skills from sysadmins.

3 WebAssembly is meant to change this, though, but not soon

4 CSP is a formal language that allows describing interactions in concurrent systems. Introduced by C.A.R Hoare in Communications of the ACM (1978), it was an inspiration for the Go language concurrency system.

5 One notable example is the controversy behind dependency management work

6 Of course, there are some inconsistencies here and there, that's why the community created more strict formatters, linters or style guides. Yet the standard tools are good enough to feel comfortable in literally every Go codebase.

7 There is one exception, which are unit test files that has to end with `_test.go`, those files can have either the same package name or `<package_name>_test` name allowing to mimic external users of the package.

8 In practice, you can quickly obtain the C++ or Go code (even when obfuscated) from the compiled binary anyway, especially if you don't strip binary from debugging symbols.

9 While `go` is improving every day, sometimes you can add more advanced tools like `goimports` or `bingo` to improve the development experience further. `go` in some areas can't be opinionated and is limited by stability guarantees.

10 CAP Theorem mentions an excellent example of treating failures seriously. It states that you can only choose two from three system characteristics: consistency, availability, and partition.

As soon as you choose to distribute your system, you must deal with network partition (communication failure). As an error handling mechanism, you can either design your system to wait (loose availability) or operate on partial data (loose consistency).

11 `bash` has many methods for error handling, but the default one is implicit. Programmer can optionally print or check `${?}` that holds the exit code of the last command executed before any given line. An exit code of 0 means the command executed without any issues.

12 In principle, a monad is an object that holds some value optionally. For example some object `Option<Type>` with methods `Get()` and `IsEmpty()`. Furthermore, an "error monad" is an `Option` object that holds an error if the value is not set (sometimes referred to as `Result<Type>`).

13 The only things that would need to change is avoiding using global variables and checking all errors

14 This assertion pattern is also typical in other third party libraries like very popular `testify` package. However, I am not a fan of the `testify` package. Mainly because there are too many ways of doing the same thing (all you typically need is `Ok` and `Equals`), and the split of `require` and `assert` packages are tend to be inconsistently mixed and not used correctly.

15 *https://golang.org/doc/go1compat*

16 Since programs, e.g. in Java, compile to Java bytecode, many things happen before the code is translated to actual machine understandable code. The complexity of this process is too great to be understandable by a mere mortal, so machine learning "AI" tools we created to auto-tune JVM.

17 The difference between concurrency and parallelism is straightforward. Task runs in parallel when they are being executed simultaneously on two different CPU cores (CPU can only execute one instruction at a time). Concurrent tasks can run parallel from time to time, but they are executed one after another most of the time. The context switch is often so quick that it looks like tasks are run simultaneously from the outside.

18 This function starts up garbage collection loop in a separate routine.

19 Survey in 2020 shows that among the top ten used programming languages, two mandates object-oriented programming (Java, C#), six encourage it, two does not implement OOP. I personally almost always favour object-oriented programming for algorithms that have to hold some context larger than three variables between data structures or functions.

20 To name a few public changes, we seen salesforce case, AppsFlyer or Stream

21 For example, when we look on some benchmarks, we see Go sometimes faster, sometimes slower than Java. Yet if we look at CPU loads, every time Go or Java is faster, it's simply quicker because e.g. the implementation allowed less CPU cycles to be wasted on memory access. You can achieve that in any programming language, questions is how hard was it to achieve so? We don't usually measure how much time one spent to optimize code in each particular language, how easy it is to read or extend such code after optimizations etc. Only those metrics might tell us what programming language is "faster".

Chapter 3. Conquering Efficiency

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at mpotter@oreilly.com.

It's action time! In Chapter 1 we learned that software efficiency indeed matters. In Chapter 2, we studied the Go programming language and its basics and advanced features. Next, we discussed Go capabilities of being both easy to read and write. Finally, we mentioned that it could be an effective language to write efficient code too.

Undoubtedly achieving better efficiency of your program does not come without work. There are some cases where functionality you try to improve is already well-optimized, so any further optimization without system redesign might take a lot of time and only make a marginal difference. However, there might be other cases where current implementation is heavily inefficient. Removing instances of wasted work (e.g. unnecessary copies of big data structures) can improve the program's efficiency within only a few hours of developer time. The true skill as an engineer here is to tell, ideally after a short amount of research, in which situation you are

currently in. Firstly, do we need to improve anything on the performance side? If yes, is there a potential for the removal of wasted cycles? How much work is needed to reduce the latency of function X? Are there any suspicious over-allocations? Should we stop using so much network bandwidth and sacrifice memory space instead? This chapter will teach you about the tools and methodologies that will help you effectively answer those questions.

If you are struggling with those skills, don't worry! It's normal. The performance area is not trivial. Despite the demand, this space is still not mastered by many, and even major software players sometimes make poor decisions. It's surprising how many times a high-quality software is shipped with fairly apparent inefficiencies. For instance, at the beginning of 2021, the user of the popular game "Grand Theft Auto Online" optimized a loading time of from 6 minutes to 2 minutes without access to the source code! As mentioned in Chapter 1, this game cost a staggering ~$140 million and a few years to make. Yet, it had an obvious efficiency bottleneck with a naive JSON parsing algorithm and deduplication logic, which took most of the game loading time and worsened the game experience. This person's work is outstanding, but he used the same techniques you are about to learn. The only difference is that our job might be a bit easier—hopefully, you don't need to reverse-engineer the binary written in C++ code on the way!

In the above example, the company behind the game missed the apparent waste of computation that was impacting the performance of the loading game. It's unlikely that such a company did not have the resources to get an expert to optimize this part. It's instead a decision based on specific tradeoffs, where the optimization seemed not to be worth the investment since there might have been higher priority development tasks to do. In the end, one would say that an inefficiency like this did not stop the success of the "Grand Theft Auto" game. It did the job, yes, but, for example, my friends and I were never fans of the GTA Online game, literally because of loading times. I would argue that without this silly "waste", success might have been even bigger.

LAZINESS OR DELIBERATE EFFICIENCY DESCOPING?

There are other amusing examples of situations where a certain aspect of the efficiency of software could be descoped given circumstances. For instance, the amusing story about missile software developers who decided to accept certain memory leaks since the missile is destroyed at the end of the application run. Similarly, the story about "deliberate" memory leaks on low latency trading software that is expected to run only for very short durations.

One would say that the mentioned examples when the efficiency work was avoided and nothing tragically bad happen were pragmatic approaches. In the end, extra knowledge and work needed to fix leaks or slow-downs were avoided. Potentially yes, but what if those decisions were not really data-driven? We don't know, but there might have been made of laziness and ignorance without valid data points that this fix would indeed take too much effort. What if developers in each of those examples did not fully understand how the small effort was needed? What if they did not know how to even start optimizing the problematic parts of the software? Would they make better decisions otherwise? Take less risk? I would argue, yes.

In this chapter, I will introduce the topic of optimizations, starting with explaining the definition and initial approach in "Beyond Waste, Optimization is a Zero-Sum Game". In the next section, "Optimization Challenges", we will summarize the challenges that we have to overcome while attempting to improve the efficiency of our software.

In "Understand Your Goals" we will try to tame the tendency and temptation to maximize optimization effort by setting clear efficiency goals for our software. Practically speaking, we need only to be fast or efficient "enough", this is why setting the correct performance requirements from the start is so important. Next, in "Resource Aware Efficiency Requirements", I will propose a template and pragmatic process anyone can follow. Those efficiency requirements will be useful in "Got a Performance Problem? Keep Calm!" where I will teach you a professional flow for handling

performance issues you or someone else has reported. You will learn that the optimization process could be our last resort.

In "Optimization Design Levels" I will explain how to divide and isolate our optimization effort for easier conquering. Finally, in "Efficiency-Aware Development Flow" we will combine all pieces into a unified optimization process I always use and want to recommend to you—reliable flow, which applies to any software or design level.

Lots of learning ahead of us, so let's get started with understanding what optimization means.

Beyond Waste, Optimization is a Zero-Sum Game

It is not a secret that one of many weapons in our arsenal to overcome efficiency issues is an effort called "optimization". But what does optimization mean exactly? What's the best way to think about it and master it?

Optimization is not exclusively reserved for software efficiency topics. We tend to optimize many things in our life too, sometimes unconsciously. For example, if we cook a lot, we probably have salt in a well-accessible place. If our goal is to gain weight, we eat more calories. If we travel from early morning, we pack and prepare the day before. If we commute, we tend to use this time by listening to audiobooks. If our commute to the office is painful, we consider moving closer to a better transportation system. All of those are optimization techniques that are meant to improve our life toward a specific goal. Sometimes we need a significant change. On the other hand, small incremental improvements are often enough as they are magnified through repetition for a more substantial impact.

In engineering, the word "optimization" has its roots in mathematics, and it means finding the best solution from all possible solutions for a problem constrained by a set of rules. Typically in computer science, however, we use the word "optimization" to describe an act of improving the system or

program execution in a specific aspect. For instance, we can optimize our program to load a file faster or decrease peak memory utilization while serving a request in a web server.

> **WE CAN OPTIMIZE FOR ANYTHING.**
>
> Generally, optimization does not necessarily need to improve our program's efficiency characteristics if that is not our goal. For example, if we aim to improve security, maintainability or code size, we can optimize for that too. Yet, in this book, when we talk about optimizations, they will be on efficiency background (improving resource consumption or speed).

The goal of the efficiency optimization should be to modify code (generally without changing its functionality[1]), so its execution is either overall more efficient or at least more efficient in the categories we care about (and worse in others).

The important part is that, from a high-level view, we can perform the optimization by doing either of two things (or both):

- We can eliminate "wasted" resource consumption.
- We can trade off one resource consumption into another or deliberately sacrifice other software qualities.

Let me explain the difference between those two by describing the first type of change—reducing, so-called "waste".

Reasonable Optimizations

Our program consists of a code—a set of instructions that operates on some data and uses various resources on our machines (CPU, memory, disk, power etc.). We write this code so our program can perform the requested functionality. But all the things involved in the process are rarely perfect: our programmed code, compiler, operating systems and even hardware. As a result, we sometimes introduce a "waste". Wasted resource consumption represents a relatively unnecessary operation in our programs that takes

precious time, uses memory or CPU time, etc. Such waste might have been introduced as a deliberate simplification, by accident, tech debt, oversight or just unawareness of better approaches. For example:

- There might have been some debugging code we left accidentally that introduces massive latency in the heavily used function (e.g `fmt.Println` statements).
- We perform an expensive check that is not needed because the caller has already verified the input.
- We forgot to stop certain go-routines (concurrency paradigm we will explain in detail in "Go Runtime Scheduler"), which are no longer required, yet still running, which wastes our memory and CPU time[2].
- We used a non-optimized function from a third-party library, where an optimized one exists in a different, well-maintained library that does the same thing, just faster.
- We save the same piece of data saved a couple of times on disk, while it could be just reused and stored once.
- Our algorithm might perform checks too many times when it could have done less for literally free (e.g. naive search vs binary search on sorted data).

The operation performed by our program or consumption of specific resources is a "waste" if, by eliminating it, we don't sacrifice anything else. And "anything" here means anything we particularly care for, such as extra CPU time or other resource consumption or non-efficiency related qualities like readability, flexibility, or portability. Such elimination makes our software, overall, more efficient. You might be surprised how much waste every program has if we look closer. It just waits for us to notice it and take it back!

Optimization by reducing "waste" in our program is a simple yet effective technique. In this book, we will call it a reasonable optimization, and I would suggest doing it every time you notice such waste, even if you don't

have time to benchmark it afterwards. Yes. You heard me right. It should be part of coding hygiene, which we will discuss in [Link to Come]. Note that to treat it as "reasonable" optimization, it has to be obvious. As the developer, you need to be sure:

- That such optimization eliminates some additional work of the program.
- It does not sacrifice any other meaningful software quality or functionality.

Look for the things that might be "obviously" unnecessary. Eliminating such unnecessary work is easily obtainable and has no harm (otherwise, it's not waste).

BE MINDFUL ABOUT READABILITY

The first thing that usually gets impacted by any code modification is readability. If reducing some obvious waste meaningfully reduces readability, or you need to spend a few hours to experiment on readable abstractions for it, postpone such work. In such a case, it might be not obvious enough to consider this change a reasonable optimization. And that's fine. We can deal with that later! To impact readability, we need data to prove it's worth it.

Cutting "waste" is also an effective mental model that can be quickly learned. It will be helpful during almost every action in this book—optimizing, improving algorithms, systems or benchmarking. Like humans who are rewarded for being intelligently lazy, we also want to maximize the value our program brings with minimum runtime work.

One would say that reasonable optimization is an example of the anti-pattern often called premature optimization many have been warning against. And I cannot agree more that reducing obvious waste like this is indeed a premature optimization since we don't assess and measure its impact. But I would argue that if we are sure that such "premature" optimization deals no harm, other than a little bit of extra work, let's call it reasonable, do it and move on.

If we go back again to our commute to work example, if we notice we have a few stones in our shoes, of course, we pick them out, so we can walk without pain. We don't need to assess, measure or compare if removing stones improved our commute time or not. It is obvious that getting rid of stones will help us in some way, and it's not harmful to do so (we don't need to take stones with us every time we go)! (:

> *If you are dealing with something which is the noise, you don't deal with that right away because the payoff of investing time and energy is very small. But if you are walking through your codebase and you notice an opportunity for notable improvement (say 10% or 12%), of course, you reach down and pick it up.*
>
> —Scott Meyers, DConf Keynote

Initially, when you are new to programming or a particular language, you might not know which operations are unnecessary waste or if elimination of this potential waste will harm your program in any way. That's fine. The "obviousness" comes from practice, so don't guess here. If you are guessing, it means the optimization is not obvious. You will learn what's reasonable with experience, and we will practice this together in [Link to Come].

Reasonable optimizations yield consistent performance improvements and often simplifies or make our code more readable. However, we might want to take a more deliberate approach explained in the next section for bigger efficiency impacts, where the result might be less obvious.

Deliberate Optimizations

Beyond waste, we have operations that are critically important for our functionality. In such a case, we can say we have a zero-sum game[3]. This means we have a situation when we cannot eliminate certain operation that uses resources A (e.g. memory), without using more of resource B (e.g. CPU time) or some quality, e.g. readability, portability or correctness.

The optimizations that are not obvious or the ones that require us to make a certain tradeoff, we can call deliberate[4], since we have to spend a little bit

more time on them. Understand the tradeoff, measure or assess it and decide to keep it or throw it away.

Deliberate optimizations are not worse in any way. On the contrary, they often might have the most significant impact on resource consumption you want to cut. For example, if our request is too slow on a web server, we can consider optimizing latency by introducing a cache. Caching will allow us to save the result from expensive computation for requests asking for the same data. It saves CPU time and the need for introducing complex parallelization logic. Yet, we will sacrifice memory or disk usage during the server's lifetime and potentially introduce some code complexity. As a result, deliberate optimization might not improve the program's overall efficiency, but it can improve the efficiency of particular resource usage we care about at the moment. Depending on the situation, the sacrifice might be worth it.

However, the implication of having certain sacrifices means we have to perform such optimization in a separate development phase explained in "Efficiency-Aware Development Flow", isolated from the functionality one. The reason for it is simple. We have to be sure that we understand what we sacrifice and whether the impact is not too big. Humans are quite bad at estimating such impacts.

For example, a common way to reduce network bandwidth and disk usage is to compress the data before sending it or storing it. At the same time, it requires us to decompress (decode it) when receiving or reading it. The potential balance of the resources used by our software before and after introducing compression can be seen in Figure 3-1.

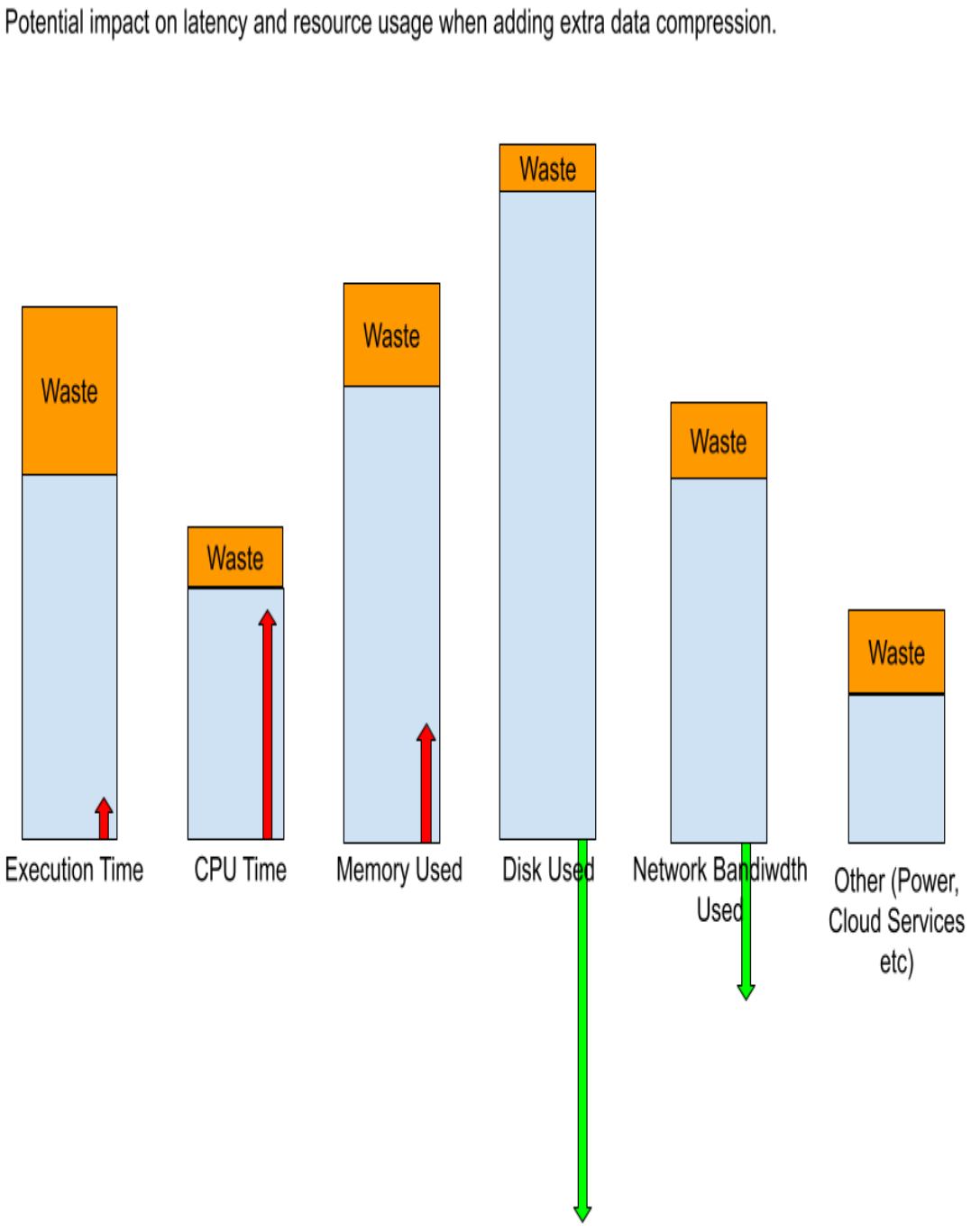


Figure 3-1. Potential impact on latency and resource usage if we compress the data before sending it over the network and saving it on disk.

The exact numbers will depend, but after compression addition, the CPU will be used more. Instead of simple data write operation, we have to go through all bytes and compress them. It takes some time, even for the best

lossless compression algorithms (e.g. `snappy` or `gzip`). All of those compression algorithms require some extra buffers, so some additional memory usage is expected too. More CPU time usually means more real-time is required to perform this operation.

On top of the caching and compression cases mentioned above, there are other examples too:

- We will learn in Chapter 4 that modern compilers perform many optimizations and extra checks that allow developers to care less about micro-optimizations and memory safety. It allows us to focus on readability and simple code. Of course, compiler work has to be generic and suitable for all major cases. It might sometimes add unnecessary extra expensive checks (e.g. bound checks) or miss opportunities for optimizations. We usually can't force the compiler to do something special directly. Still, we can write code differently for the correct compiler steps to kick in, or we can write assembly code on our own (very extreme optimization described in [Link to Come]). As you might imagine, this violates both rules of reasonable optimization: it's not an obvious improvement, and it is a tradeoff. We impact at least readability, maintainability if not portability, so it should be done with care and only if strictly necessary.
- Deliberate optimization is also when we remove a portion of functionality to have faster or more efficient execution (yes!). For example, our program was sorting results, but sorting results was never a requirement[5]. So we can potentially drop it to stream more and have more lean execution.
- There are often cases where you notice that your program loops through the same array many times to process various information. This feels like a "waste". It's very tempting for us to perform a loop fusion, so code that just loops once and does all work in a single main loop. This is a very common and harmful pitfall of premature optimization that looks reasonable at first glance. Yet more often than not, moving all into one loop might be worse, thus treated as deliberate

optimization and avoided if possible. Let me explain why in detail using Example 3-1 snippet.

Example 3-1. An example of tempting to optimize code, but such optimization should probably be postponed until we are sure this will yield meaningful efficiency improvements.

```
func ProcessInput(input []Element) {
   for _, i := range input {
      if needsChange(i) {
         thingsToChange = append(thingsToChange, i)
      }
      if needsToBeRemoved(i) {
         thingsToRemove = append(thingsToRemove, i)
      }
   }

   for _, i := range input {
      if isTypeA(i) {
         typeA++
         continue
      }
      if isTypeB(i) {
         typeB++
         continue
      }
      typeUnknown++
   }

   var x *Element
   for _, i := range input {
      if isX(i) {
         x = &i
         break
      }
   }

   sort.Slice(input, func(i, j int) bool { ❶
      return input[i].Something < input[j].Something
   })

   // Use processed information...
}
```

❶ Apart from the three loops above, sorting itself can be considered the 4th loop. In this case, it might be the most expensive one, due to

potential nested loops, due to the complexity of sorting being at best `O(n*logn)` (we will explain Big O notations in [Link to Come]).

In Example 3-1 we loop over the same array at least for times (three implicit ones, plus sorting). It might be tempting to assume this situation as wasted work and try to implement one big loop with all those processing, ideally including sorting. Such optimization is commonly known as Loop Fusion. However, in the case of Example 3-1, this is far from a reasonable optimization because if we would try to do that, we will risk the following problems:

- We would inevitably create more complex code which is harder to read (discussed in [Link to Come]) and maintain, simply because we would need to implement loop iteration that does many things at once (many is more complex than one).
- Putting all work in the single loop most likely won't help our CPU to do less work. It all depends on what you optimize for, but modern CPUs and compilers are designed to run loops and use data locality for advantage (as we will learn in Chapter 4). It would need more experiments with measurements to tell what's more efficient.
- The standard sorting function we use in Example 3-1 is already heavily optimized. The complexity or optimized sorting implementation is well hidden from us under the `sort.Sort` abstraction. An attempt to sort in place, in one big loop, would inevitably bring all of this complexity into our main function.
- Even if we make this function faster through loop fusion optimization, maybe `ProcessInput` is only executed a few times and only for small arrays? Since it requires work and heavy sacrifices, we need first to be sure it will make a difference in our overall program. Otherwise, it's not worth our time.

For all of the above reasons, such optimization has to be done deliberately. Only after we make sure the code is functional and readable and only if

needed. Yet, the temptation to optimize those loops (or other things we mentioned) straight away is a very commonly seen mistake—one I've personally made so many times. Don't make the same mistake. Don't optimize such cases unless you know it's valuable to do so.

To sum up, there are strong implications of categorizing optimization into reasonable and deliberate. If we see a potential efficiency improvement, we have to be aware of its unintended consequences. There might be cases where it's reasonable, easy to obtain optimization. We might have peeled some unnecessary operations from our program for free. As you will learn in "Efficiency-Aware Development Flow" we can do that straight away. But, more often than not, it's impossible to make our software efficient in every aspect. This is when we get ourselves into a zero-sum game, and we have to take a deliberate look at those problems. In this book and by practice, you will learn what situations you are in and how to predict those consequences.

Before we jump into bringing those two types of optimizations into our development flow, let's discuss the efficiency optimization challenges we have to be aware of. We will go through the most important ones in the next section.

Optimization Challenges

I would not need to write this book if optimizing our software was easy. It's not. The process can be time-consuming and prone to mistakes. This is why many developers tend to ignore this topic or learn at the later stages of their careers. But don't feel demotivated! Everybody can be an effective, performance aware developer after some practice. Knowing about the optimizations obstacles should give us a good indication of what skill gap we have yet to learn! Let's go through seven fundamental problems:

Programmers are bad at estimating what part is responsible for the performance problem

Let's be honest. We are really bad at guessing which part of the program consumes the most resources and how much. It's essential to do so because usually, the Pareto rule applies. It states that 80% of the time or resources consumed by our program comes only from 20% of operations our program performs. Since any optimization is time-consuming, we want to focus our efforts on that critical 20% of operations, not some noise. Fortunately, there are tools and methods for estimating this, which we will touch on in [Link to Come] and [Link to Come].

Programmers are notoriously bad at estimating exact resource consumption

Similarly to above, we often make wrong assumptions on whether certain optimization should help. Our guesses get better with experience (and hopefully after reading this book). Yet, it's best to **never trust your judgement** and always measure and verify all numbers after deliberate optimizations (discussed in depth in [Link to Come]). There are just too many layers in software executions with many unknowns and variables.

Maintaining efficiency over time is hard

The complex software execution layers mentioned above are constantly changing (new versions of operating systems, hardware, firmware, etc.), not to mention the program's evolution and future developers that might touch your code. We might have spent weeks optimizing one part, but it could be irrelevant if we don't guard against regressions. There are ways to automate or at least structure the benchmarking and verification process for the efficiency of our program because things change every day, discussed in [Link to Come].

Reliable verification of current performance is very difficult

As we will learn in "Efficiency-Aware Development Flow", the solution to the three above challenges is to benchmark, measure and validate the efficiency. Unfortunately, those are difficult to perform and prone to

errors. There are many reasons: inability to simulate the production environment close enough, external factors like noisy neighbours, lack of warm-up phase, wrong data sets or micro-benchmark accidental compiler optimizations. This is why we will spend the whole [Link to Come] on this topic.

Optimizing can easily impact other software qualities

Solid software is great at many qualities: functionality, compatibility, usability, reliability, security, maintainability, portability and efficiency. Each of those characteristics is non-trivial to get right, so they cause some cost to the development process. The importance of each of those can differ depending on your use cases too. However, there are safe minimums of each of the software qualities to be maintained for your program to be useful. This might be challenging when you add more features and optimization.

Specifically, in Go, we don't have strict control over memory management

As we learned in "Go Runtime" Go is garbage collected language. While it's life-saving for the simplicity of our code, memory safety and developer velocity, it has downsides that can be seen when we want to be memory efficient. There are ways to improve our Go code in order to use less memory, but things can get tricky since the memory release model is eventual. Usually, the solution is simply to allocate less. We will go through memory management in "Do we Have a Memory Problem?".

When our program is efficient "enough"?

In the end, all optimizations are never fully free. They require a bigger or smaller effort from the developer. Both reasonable and deliberate optimizations require some prior knowledge, spent time on implementation, experimentations, testing and benchmarking. Given that, we need to find justification for this effort. Otherwise, we can spend this time somewhere else. Should we optimize away this waste?

> Should we tradeoff the consumption of resource X for resource Y? Is such conversion useful for us? The answer might be “no”. And if “yes”, how much efficiency improvement is enough?

Regarding the last point, this is why it’s extremely important to know your goal. What things, resources, and qualities do you (or your boss) care about during the development? It can vary depending on what you build. In the next section, we will try to propose a pragmatic way of stating performance requirements for a piece of software.

Understand Your Goals

> *Before you proceed towards such lofty goals [program efficiency optimization], you should examine your reasons for doing so. Optimization is one of many desirable goals in software engineering and is often antagonistic to other important goals such as stability, maintainability, and portability. At its most cursory level (efficient implementation, clean non-redundant interfaces), optimization is beneficial and should always be applied. But at its most intrusive (inline assembly, pre-compiled/self-modified code, loop unrolling, bit-fielding, superscalar and vectorizing) it can be an unending source of time-consuming implementation and bug hunting. Be cautious and wary of the cost of optimizing your code.*
>
> —Paul Hsieh, Programming Optimization (2016)

By our definition, efficiency optimization improves our program resource consumption or latency. While it’s highly addictive to challenge ourselves and explore how fast our program can be [6], we need to understand that the aim of optimization is not to make our program perfectly efficient or “optimal” (as that might be simply impossible nor feasible), but rather sub-optimal enough. But what does “enough” means for us? When do you stop? What if there is no need even to start optimizing? Let’s explore what can help us decide here.

We discussed in Chapter 1 that sooner or later, as developers, we need to care about efficiency optimizations. Reasons for that can be summarized in a single sentence: we optimize our software to ensure a better user experience. We can make sure results are returned faster, so humans will save time. We can reduce the cost of running our software by the user, so it allows a bigger earning margin. By reducing the total memory usage needed for a single process, we can make sure smaller, more affordable and accessible machines can be used. With a smaller resource usage footprint, we might widen a range of devices (e.g. smartphones) that can be supported, thus reaching more clients.

This is great, but if more efficient software is always cheaper to run and improves user experience, shouldn't we try to optimize it to the maximum? In the end, our goal is to provide the maximum value to stakeholders, right?

Unfortunately everything costs, especially optimization effort. Practically speaking, there is no limit to how optimized software can be[7]. At some point, however, the benefit we gain from optimization versus the cost of finding and developing such optimization is impractical. The Figure 3-2 shows a typical correlation between efficiency, and different costs: runtime[8], optimizing and missed opportunities.

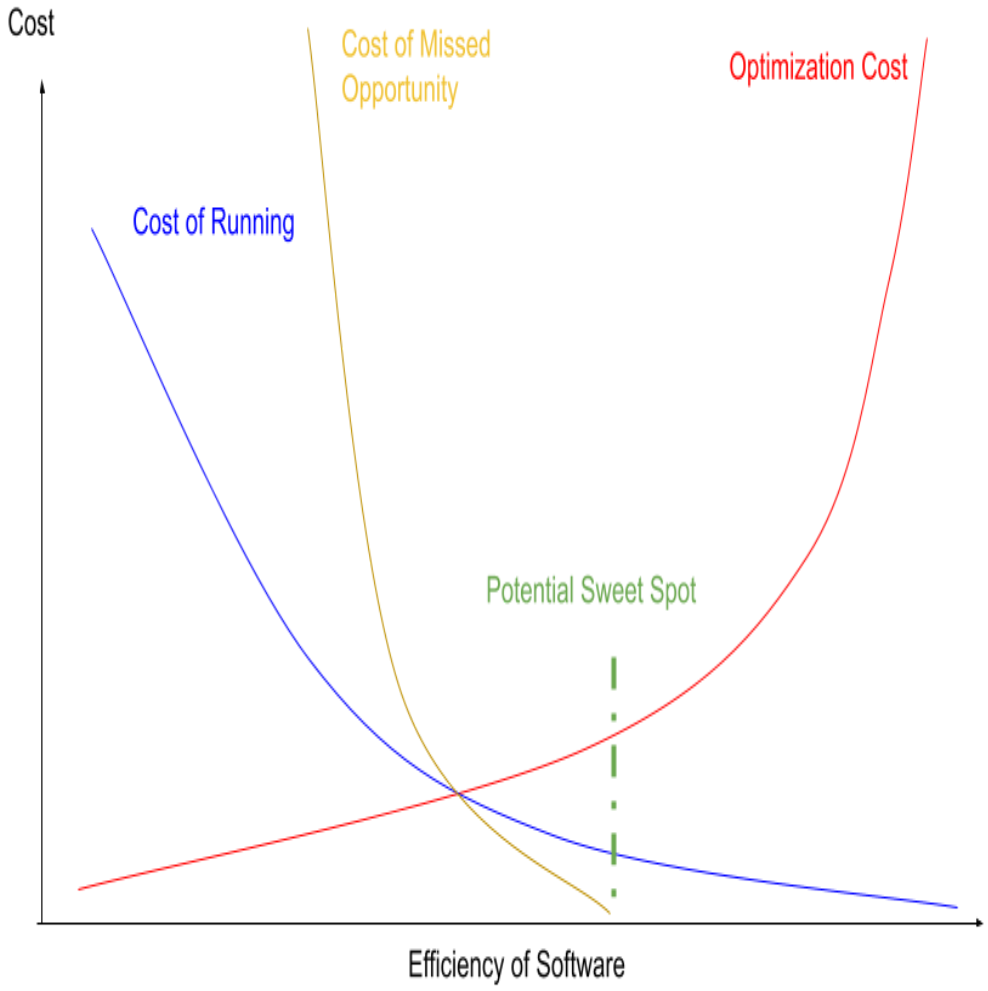


Figure 3-2. Beyond the certain Sweet Spot, the cost of gaining higher efficiency might get extremely high. This might quickly surpass the benefits we get from more lean software like computational cost and opportunities.

Figure 3-2 explains why at some optimization point it might be not feasible to invest more time and resources in software efficiency. Beyond some point, which we can call “Sweet Spot”, to gain more efficiency we might need to spend even more of the expensive developer time, we might need to

introduce clever, non-portable tricks, dedicated machine code, dedicated operating system, or even extreme, specialized hardware.

There is no single answer, where the Sweet Spot is. Typically, it's worth investing a bit more in optimization since it's a one-time cost, which improves the continuous cost of computing resources required by your software. Intuitively, the longer lifetime is planned for the software and the larger deployment of it is, the more such investment is worth it. If you plan to use your program only a few short times, your sweet spot might be in the very beginning of this diagram, which might even mean no optimization.

Of course, the computational cost is not everything that is improved if we make our software more efficient. There might be certain constraints on our software in terms of resource usage. If we require a large amount of memory or CPU for execution, it might mean we need to introduce distributed systems earlier which grows developer time and complexities exponentially. If our software requires a high-end GPU, it might limit the audience it can be useful and valuable. If the product is more expensive to run, it might lose the competition with other, more lean products.

While ideally, product owners should decide where the sweet spot is, it's often the developer role to advise on the cost of running and cost of making further optimization for the developed software. Let's dive into what can help us here.

In "Efficiency Requirements Should be Formalized" we explain why formal efficiency requirements are important. In "Resource Aware Efficiency Requirements" we will introduce lightweight formula for those. The "Acquiring and Assessing Efficiency Goals" touches on how to acquire and assess those efficiency requirements.

Efficiency Requirements Should be Formalized

As you probably already know, every software development starts with the **functional requirements gathering** stage (FR stage). An architect, product manager or yourself have to go through potential stakeholders, interview them, gather use cases and, ideally, write them down in some functional

requirement document. The development team and stakeholders are then reviewing and negotiating functionality details on this document. The FR document describes what input our program should accept and what behaviour and output a user expects. It also mentioned prerequisites, like what operating systems the application is meant to be running on. Ideally, we get formal approval on the FR document, and it becomes our "contract" between both parties. Having this is extremely important, especially when you are compensated for building the software:

- The FR document immediately tells us (developers) what functionality our software should have. It tells us what inputs should be valid and what things a user can configure. It dictates what we should focus on. Are we spending our time on something stakeholders paid for?
- Similarly it allows others to integrate with our software. For example, stakeholders might want to design or order further system pieces that will be compatible with our software. They can start doing this before our software is even finished!
- Ideally, it is written and formal. People tend to forget things, plus it's easy to miscommunicate. That's why we write it all down and ask stakeholders for review. Maybe we misheard something?

We do formal functional requirements for bigger systems and features. For a smaller piece of software, we tend to write them up on some issue in our backlog, e.g. GitHub or GitLab Issue and then document those. Even for our tiny scripts or little programs, we set some goals and prerequisites—maybe a specific environment (e.g. python version) and some dependencies (GPU on the machine). When you want others to use it effectively, you have to mention somewhere the functional requirements and goals of your software.

Defining and agreeing on functional requirements is well adopted in our software industry. Even if a bit bureaucratic, developers tend to like those specifications because it makes their life easier—requirements are stable and very specific.

Probably you know where I am going with this. Surprisingly, we often neglect to define similar requirements focused on more non-functional aspects of the software we are expected to build. For example, describing a required efficiency and speed of the desired functionality[9].

Such efficiency requirements were typically known to be part of Non-Functional requirement (NFR) documentation or specification. Its gathering process ideally should be similar to mentioned above FR process, but for all other qualities that requested software should have: portability, maintainability, extensibility accessibility, operability, fault tolerance and reliability, compliance, documentation, execution efficiency and so on. The list is long. Yet, in reality, NFRs were never fully adopted for most software we do, based on my experience and research. I found multiple reasons:

- Conventional NFR specification is considered bureaucratic and full of boilerplate. Especially if the mentioned qualities are not quantifiable and not specific, NFR for every software will look obvious and more or less similar. Of course, all software should be readable, maintainable, as fast as possible, using minimum resources and usable. This is not helpful.
- There are no easy to use, open and accessible standards for this process. The most popular ISO/IEC 25010:2011 standard costs around $200 to read. It has staggering 34 pages, and it was not changed from the last revision in 2017.
- NFRs are usually too complex to be applicable in practice. For example, the ISO/IEC 25010 standard mentioned above specifies 13 product characteristics with 42 sub-characteristics in total. It is hard to understand and just takes too much time to gather and walkthrough.
- As we will learn in "Optimization Design Levels", our software's speed and execution efficiency depend on more factors than just our code. The typical developer usually can impact the efficiency by optimizing algorithms, code and compiler. It's then up to the operator or admin to install that software, fit it into a bigger system, configure and provide operating system and hardware for that workload. When

developers are not in the domain of running their software on "production", it's hard for them to talk about runtime efficiency.

> **TIP**
>
> Site Reliability Engineering (SRE) introduced by Google is a role focused on marrying those two domains: software development and operators/admins. Such engineers have experience in both running and building their software on a large scale. With more hands-on experience, it's easier to talk about efficiency requirements.

- Last but not least, we are humans, and we are full of emotions. Because it's hard to estimate the efficiency of our software, especially in advance, it's not uncommon to feel humiliated when setting efficiency or speed goals. This is why we sometimes unconsciously refrain from agreeing to quantifiable performance goals. It can be uncomfortable, and that's normal.

> **NOTE**
>
> The NFR, so the Non-Functional Requirement name can be in some way misleading since many qualities, including efficiency, massively impact our software functionality. As we learned in Chapter 1, efficiency and speed are critical for user experience.

Okay, scratch that, we are not going there. We need something more pragmatic and easier to work with. Something that will state our rough goals for efficiency and speed of the requested software and will be a starting point for some contracts between consumers and the development team. Having such efficiency requirements on top of functional ones upfront is enormously helpful because:

- We can prioritize optimization work like any other functional work.
- We know how exactly much efficiency is enough. For instance, let's say we agree that a certain operation should use 1GB memory, 2 CPU seconds and take 2 minutes at maximum. If our tests show that it takes

2 GB of memory, 1 CPU second for 1 minute, then there is no point in optimizing latency.

- We know if we have room for a tradeoff or not. In the example above, we can precalculate or compress things to improve memory efficiency. We have still one CPU second to spare, and we can be slower for 1 minute or so.
- From a usage perspective, if we don't set any requirements, the user will implicitly create its expectation. Maybe for a certain input, our program is very fast. Users can assume this is by design, and they will depend on the fact that they can expect this performance for another type of request, which will take much more time. This can lead to poor experience and surprises[10].
- More often than not, your software will be a dependency for another software and form a bigger system. As discussed in [Link to Come], even a basic efficiency requirements document can tell system architects what to expect from the component. It can enormously help with further system performance assessments and capacity planning tasks.
- Last but not least, if users do not know what performance to expect from your software, you will have a hard time while supporting it over time. There will be lots of back-and-forths with the user on what is acceptable efficiency and what's not. Instead, with clear efficiency requirements, it is easier to tell if your software was underutilized or not, and as a result, the issue might be on the user side.

Let's summarize our situation. We know efficiency requirements can be enormously useful. We also know it can be tedious and full of boilerplate. Let's try to explore some options and see if we can find some sweet middle spot between our effort and the value it brings.

Resource Aware Efficiency Requirements

No one defined a good standard process for creating efficiency requirements, so let's try to define one! Of course, we want it to be as lightweight process as possible, but let's start with the ideal situation. What is the perfect set of information someone could put into some Resource Aware Efficiency Requirements (RAERs) document? Something that will be more specific and actionable than "I want this program to run adequately snappy". In Figure 3-3 we can see an example of potentially meaningful RAER for a single operation in some software.

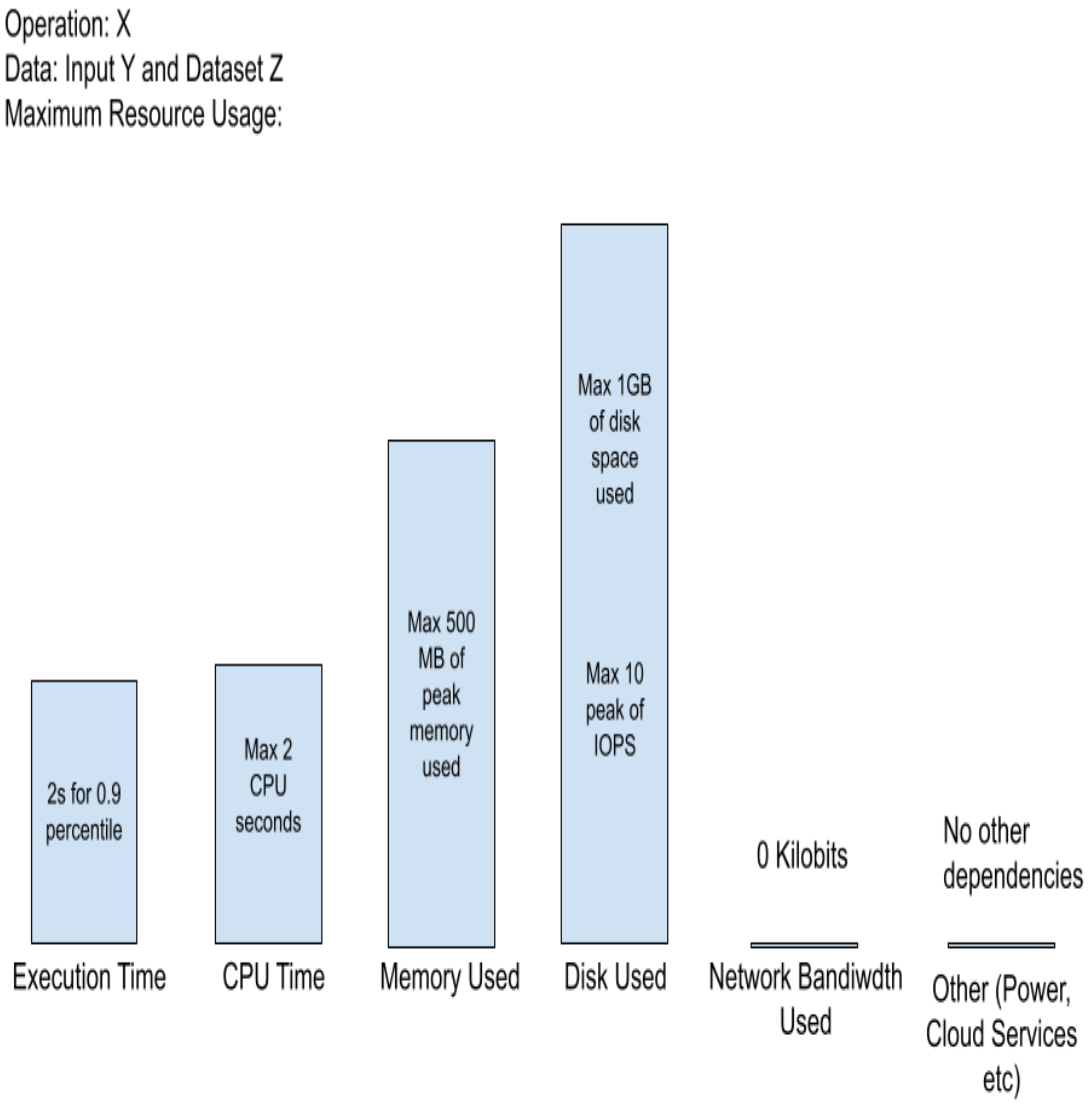


Figure 3-3. RAER entry for certain operation with certain input and dataset.

Ideally, such RAER is a set of records with efficiency requirements for certain operations. In principle, a single record should have information like:

- What operation, API, method, or function does it relate to.
- The size and shape dataset we operate on, e.g. input or data stored (if any).
- Maximum latency of the operation.
- The resource consumption budget for this operation on that dataset, e.g. memory, disk, network bandwidth etc.

Now, there are bad news and good news. The bad news is that strictly speaking, such records are unrealistic to gather in all details. This is because:

1. There are potentially hundreds of different operations our software does during its execution.
2. There is an almost infinite amount of shapes and sizes of data sets (e.g. imagine a SQL query being an input and stored SQL data being dataset - we have a near-infinite amount of options' permutations).
3. Modern hardware with an operating system has thousands of elements that can be "consumed" when we execute our software. Overall, CPU seconds and memory are common, but what about the space and bandwidth of individual CPU caches, memory bus bandwidth, number of TCP sockets taken, file descriptors used and thousands of other elements? Do we have to specify all that can be used?

The good news is that we don't need to provide all the small details. In fact, this is similar to how we deal with functional requirements. Do we focus on all possible user stories and details? No, just the most important ones. Do we define all possible permutations of valid inputs and expected outputs? No, we only define a couple of basic characteristics (e.g. input has to be a positive integer). Let's look at how we can simplify the level of details of the RAER entry:

- We can focus on the most utilized and expensive operations our software do first. This is because those will impact the software

resource usage the most.

- We don’t need to outline requirements for all tiny resources that might be consumed. But, again, we can start with those that have the highest impact and matter the most. Usually, it means specific requirements towards CPU time, memory space and storage (e.g. disk space). From there, we can iterate and add other resources that will matter in future. Maybe there are unique, expensive and hard to find resources our software needs worth mentioning (e.g. GPU). Maybe a certain consumption poses a limit to overall scalability, e.g. we could fit more processes on a single machine if our operation would use fewer TCP sockets or disk IOPS.
- Similar to what we do in unit tests when validating functionality, we can focus only on important categories of inputs and datasets. If we pick edge cases, we have high chances to provide resource requirements for the worst and best-case datasets. That is an enormous win already.
- Alternatively to above, there is a way to define the relation of input (or dataset) to the allowed resource consumption. We can then describe this relation in the form of mathematical functions. Even with some approximation, it’s quite an effective method. Our RAER for operation X in Figure 3-3 could then be described as seen in Figure 3-4.

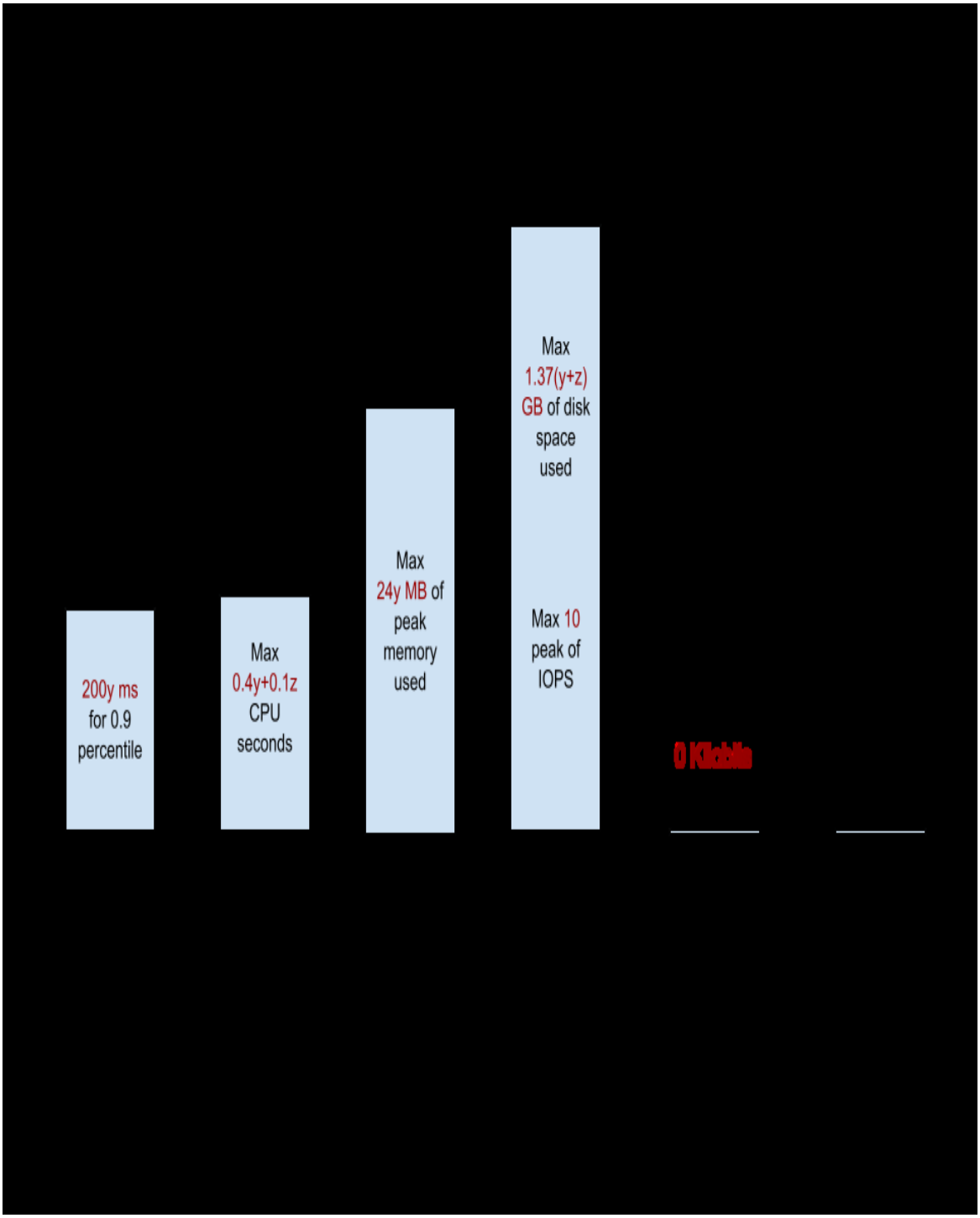


Figure 3-4. RAER entry for certain operation with certain input and dataset.

Overall, I would even go as far as proposing to include the RAERs in the functional requirement (FR) document mentioned previously. Put it in another section called “Efficiency Requirements”. After all, without rational speed and efficiency, our software can’t be called fully functional, does it?

To sum up, in this section, we defined the Resource Aware Efficiency Requirements specification that gives us approximations of needs and expected performance towards our software's efficiency. It will be extremely helpful for the further development and optimization techniques we will learn in this book. Therefore, I want to encourage you to understand the performance you aim for, ideally before you start developing your software, optimizing or adding more features to it. While this is best done in a formal RAER document, more lightweight versions would do too (better than nothing!).

Let's try to explain how we can possess or create such REARs ourselves for our system, application or function we aim to provide.

Acquiring and Assessing Efficiency Goals

Ideally, when you come to work on any software project, you have something like REAR specified. In bigger organizations, you might have dedicated people like project or product managers who will gather such efficiency requirements on top of functional requirements. They should also make sure the requirements are possible to fulfil. If they don't gather such REAR style of requirements, do not hesitate to ask them to provide such information. It's often their job to give those.

Unfortunately, in most cases, there are no specific efficiency requirements at all. Especially in smaller companies, community-driven projects or obviously, your personal projects. In those cases, we need to acquire those efficiency goals ourselves. How do we start?

This task is really really similar to functional goals. We need to bring value for users, so ideally, we need to ask them what they need in terms of speed and running cost. We go to stakeholders or customers and ask what they need in terms of efficiency and speed, what they are willing to pay for and what are the constraints on their side (e.g. cluster has only four servers or GPU has only 512 MB of internal memory). Similarly, with features, good product managers and developers will try to translate user performance needs into efficiency goals, which is not trivial if the stakeholders are not

from the engineering space. For example, the “I want this application to run fast” statement has to be translated into specifics.

Very often there are multiple personas of the system users too. For example, let’s imagine our company will run our software as a service for the customer and the service has already defined price. In this case, the user cares about the speed and correctness and our company will care about the efficiency of the software, as this translates to how much net profit the running service will have (or loss if the computation cost of running our software is too large). In this typical Software as a Service (SaaS) example, we have not one, but two sources of input for our RAERs.

DOGFOODING

Very often for smaller coding libraries, tools and our own infrastructure software, we are both developers and users of it. In this case, it’s much easier to set REARs from the user perspective. That is only one of the reasons why using your own software you create is a good practice. This approach is often called “eating your own dog food” (dogfooding).

Unfortunately, even if a user is willing to define us RAER, the reality is not so perfect. Here comes the difficult part. Are we sure that what was proposed from the user perspective is doable with the expected amount of time? We know the demand, but we have to validate it with the supply we can provide in terms of our team skillset, technological possibilities and time needed. Usually, even if some RAER is given, we need to perform our own diligence and define or assess RAER from an achievability perspective.

Defining and assessing complex RAER can get complicated. However, for practical use, we really need a very simple RAER to start with if you have to do it from scratch. It’s reasonable to start with potentially trivial, yet clear requirements.

Setting those requirements boils down to the user perspective. We need to find the minimum set of requirements that makes your software valuable in its context. Let’s say we need to create software that applies image

enhancements on top of a set of images in JPEG format. In RAER we can now treat such image transforming as an **operation** and set of image files and chosen enhancement as our **input**.

The second item on our RAER is the latency of our operation. From a user perspective, it is of course better to have it as fast as possible. Yet our experience should tell us, that there are some limits on how fast we can apply the enhancement to images (especially if large and many of those). But how to find a reasonable latency number requirement that would work for potential users, but also make it possible to achieve for our software?

It's not easy to agree on a single number, especially when we are new to the efficient world. We could potentially guess that 2 hours for a single image process might be too long, and 20 nanoseconds are not achievable, but it's hard to find the middle ground here. Yet as mentioned in "Efficiency Requirements Should be Formalized", I would really encourage you to try defining one number, as it would make your software much easier to assess!

DEFINING EFFICIENCY REQUIREMENT IS LIKE NEGOTIATING SALARY.

Agreeing to someone's compensations for their work is really similar to finding the requirement sweet spot for the latency or resource usage for our program. One side wants the salary to be the highest possible. As an employer, you don't want to overpay. It's also hard to assess what kind of value the person will be providing and how to set meaningful goals for such work. What works in salary negotiating works when defining RAER: Don't set too high expectations, look at other competitors, negotiate, and have trial periods!

Let's look at what can help define or assess the RAER latency here:

- Check competitors. They already got stuck in some kind of limits and framework for understanding the latency for their software work. You don't need to set that as your latency, but it can give some clue on what's possible.

- Figure out the initial, naive algorithm you want to try. Let's assume for our problem we want to read an image in JPEG format, decode it to memory, apply enhancement, encode it back and write to memory.
- Understand the input. Assume worst case first. Try to find the slowest part of your system and what input can trigger that. In our example, we can imagine the largest image we allow in our input (e.g. 8K resolution) is the slowest to process. The requirement of processing a set of images makes things a bit tricky. For now, we can assume the worst case and start negotiating with that. The worst case is that images are different and we don't use concurrency. This means our latency will be a function of x * N where `x` is the latency of the biggest image and `N` number of images in the set.
- If research, experience or competitors does not tell us what latency for the largest image might be, it's time to use our secret weapon—the **napkin math**. Let's do quick math together to check the minimum latency we expect to have in theory.

> **NOTE**
>
> Napkin Math sometimes referred to as the Back of the Envelope Calculations, is a technique of making rough calculations and estimations based on simple, theoretical assumptions. For example, we could assume latency for certain operations in computers e.g. sequential read of 8KB from SSD is taking approximately 10µs while writing 1 ms[11]. With that, we could calculate how long it takes to read and write 5 MB of sequential data. Then we can go from there and calculate overall latency if we make a few such reads in our system and so on etc.
>
> Napkin math is only an estimate, so we need to treat it with a grain of salt. Sometimes it can be intimidating to do since it all feels abstract. Yet such quick calculation is always an amazing test if our guesses and initial system ideas are correct. It gives amazing early feedback that is worth our time, especially around common efficiency requirements like latency, memory or CPU usage.

In terms of input, let's take an example image for an 8K screen in JPEG format. Most of the images I found, were around 4MB, so let's have this number represent our maximum input size. Using data from [Link to Come]

we can then calculate that such input will take at least 5ms to read and 0.5s to save it on a disk. Similarly encoding and decoding from JPEG format likely means at least looping through and allocating in memory up to 7680×4320 pixels. By looking on `image/jpeg` standard Go library each pixel is represented by three`uint8` numbers to represent colour in YCbCr format. That means approx 90 million unsigned 8 bytes integers. We need to fetch each element from memory (~5ns for sequential read from RAM) twice (one for decode, one for encode) which means 2 * 3ns to 90 million * 5 ns, so 0.9 seconds. As a result of this is quick math, we know now that without applying any enhancements or more tricky algorithms such operation for the single image will be no faster than let's say 0.9+0.5s, so 1.4 seconds.

Since napkin math is only an estimate, plus we did not account for actual enhancing operation, it would be safe to assume we are wrong up to 5 times. This means that the initial latency requirements could be 10 seconds for a single image. That means 10 * N seconds for N images. This could be our safe RAER latency we can now present to the user as a starting point. We assume worst cases and unoptimized algorithms, so there might be room for improvement here. But it might as well be fine for the user to wait 10 seconds on one image, and 100 seconds for ten images.

- To be more certain of the calculations we did, or if you are stuck in calculation we could perform a quick benchmark[12] for the critical, slowest operation in our system and compare without our napkin math results. Let's say we could write a single benchmark for reading, decoding, encoding and saving 8K images. It will take us a day but it will give us confirmation of our math was right. The Example 3-2 shows the result of the benchmark I did. The Example 3-3 summarize the result. It shows our calculation was quite accurate. It takes 1.56 seconds on average to perform a basic operation on an 8K image!

Example 3-2. Go microbenchmark results of reading, decoding, encoding and saving an 8K JPEG file.

```
goos: linux
goarch: amd64
```

```
pkg: github.com/efficientgo/examples/pkg/jpeg
cpu: Intel(R) Core(TM) i7-9850H CPU @ 2.60GHz
BenchmarkDecEnc-12            18        1341241537 ns/op
225826898 B/op        21 allocs/op
BenchmarkDecEnc-12            16        1758913063 ns/op
225818895 B/op        19 allocs/op
BenchmarkDecEnc-12            18        1365219728 ns/op
225821412 B/op        18 allocs/op
BenchmarkDecEnc-12            14        1629118092 ns/op
225815754 B/op        18 allocs/op
BenchmarkDecEnc-12            18        1721897886 ns/op
225815941 B/op        18 allocs/op
PASS
ok      github.com/efficientgo/examples/pkg/jpeg        175.120s
```

Example 3-3.

```
name       time/op
DecEnc-12  1.56s ±14%
name       alloc/op
DecEnc-12  226MB ± 0%
name       allocs/op
DecEnc-12   18.8 ±12%
```

Last, but not least item on our RAER is the rough resource consumption requirements. This is again depending on user scenarios, but we can identify a few typical cases which you can use as your starting points:

- If our image enhancing software is meant to run on a phone or laptop (client side), we need to set RAER according to that. We know it cannot exceed a single device capability. We can then safely assume the user won't be happy if our peak memory will be over 1GB memory and over 1 CPU time on a smartphone. For PCs and laptops we could soften the requirements to a few CPUs (let's say 4) and a few GBs or RAM if needed, (let's say 2GB).
- If our software runs in the cloud or servers, we have much more room to play here. In this case, typical resource usage requirements could be bound to the price we want to pay for compute power. Our RAER in principle would look like in Figure 3-5 and the price would then define requirements to each resource usages[13]

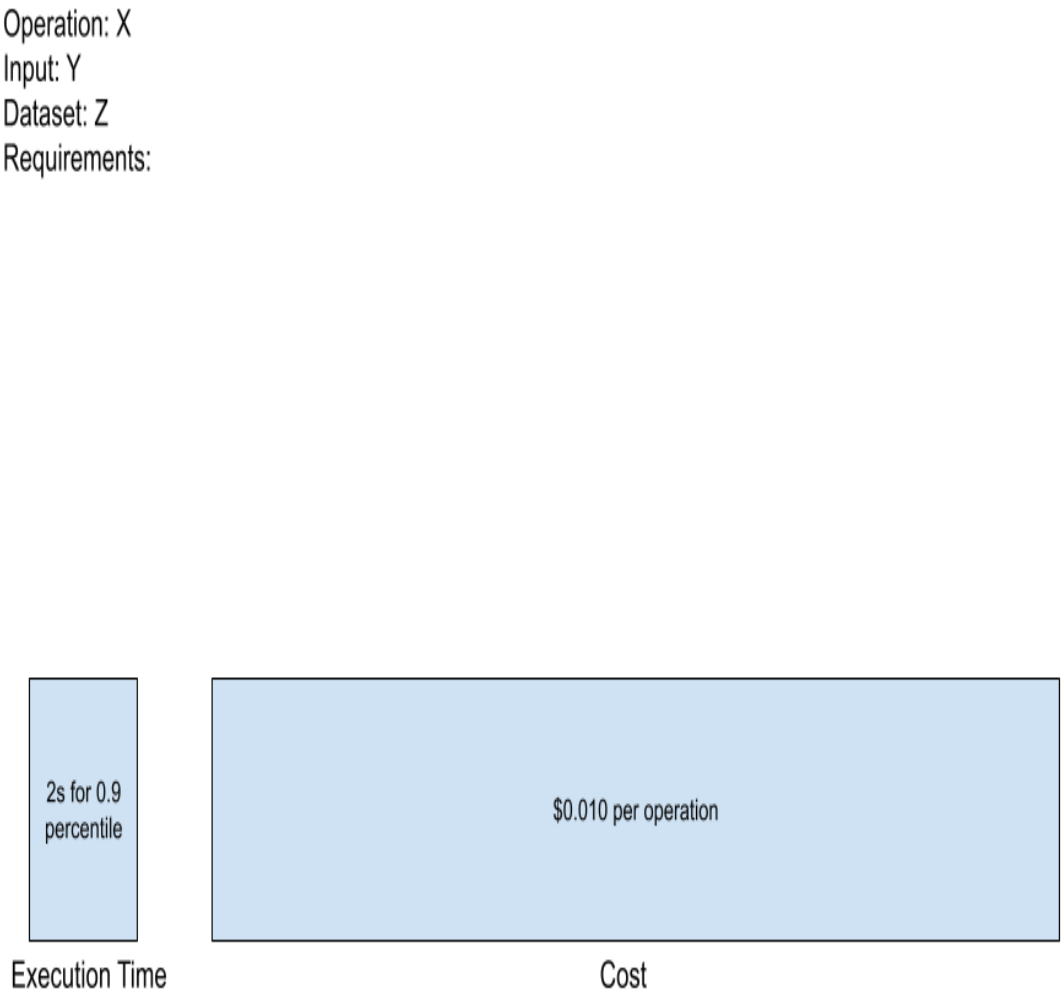


Figure 3-5. RAER with only latency and cost requirement.

Let's say we want our software to run on smartphones. We discussed we could pick peak memory of 1GB as our RAER requirement for memory. Both the napkin math and our test are a great way to validate if our naive algorithm and software idea is doable with that constraints. For example Example 3-3 shows that 226 MB of memory was allocated, without any optimization. With a spare 700 MB of memory for enhancement logic, we have a high chance of achieving this requirement if we use the knowledge in this book on how to keep our software lean and efficient enough!

Acquiring and assessing RAERs seems like a complex and unpopular thing, but once again, I recommend doing such exercise and getting those numbers before serious development. With benchmarking and Napkin Math we can quickly understand if RAERs are achievable with the rough algorithm we have in mind. The same process can be also used to tell if there is room for more, easy to achieve optimization in some fields, which we will discuss in [Link to Come].

With the ability to obtain, define and assess your RAERs, we can finally attempt to conquer some efficiency issues! In the next section, we will discuss steps I would recommend to handle such sometimes stressful situations professionally.

Got a Performance Problem? Keep Calm!

First of all, don't panic! We all have been there. We wrote a piece of code, tested it on our machine, and it worked great. Proud of it, we release it to others, and immediately someone reports performance issues. Maybe it can't run fast enough on other people machines. Perhaps it uses an unexpected amount of GBs of RAM with other users' datasets.

We have a couple of choices when facing efficiency issues in the program we build, manage, or own. But before you make any decisions, there is one critical thing you have to do. When issues happen, clear your mind from any negative emotions about yourself or the team you worked with. It's very common to blame yourself or others for mistakes. It is only natural to feel an uncomfortable sense of guilt when someone complains about our work. However, everyone (including us) must understand that the topic of performance is challenging to deal with. On top of that, non-efficient or buggy code happens every day, even for most experienced developers. There should be no shame in making mistakes.

Why do I write about emotions in a programming book? Because psychological safety is an important reason why developers take the wrong approach towards code efficiency. Procrastinating, being stuck, being afraid to try new things or scratch bad ideas are only some of the negative

consequences. From my own experience, if we start blaming ourselves or others, we won't solve any problems. Instead, we kill innovation, productivity and introduce anxiety, toxicity and stress. Those feelings can further prevent you from making a professional, reasonable decision on how to proceed with the reported efficiency issues or any other problems.

BLAMELESS CULTURE MATTERS

Highlighting blameless attitude is especially important during the "post-mortem" process, which the Site Reliability Engineers perform after incidents. For example, sometimes costly mistakes are triggered by a single person. While we don't want to discourage this person or punish them, it is crucial to understand the cause of the incident to prevent it in future. Furthermore, the blameless approach enables us to be honest about facts while respecting others, so everyone feels safe to escalate issues without fear.

We should stop worrying too much, and with a clear mind, we should follow a systematic, almost robotic process (yes, ideally, all of this is automated someday!). Let's face it, practically speaking, not every performance issue has to be followed by optimization. The effort of fixing efficiency issues is not free. It is time-consuming and often increases code complexity. The potential flow for the developer I would propose is presented on Figure 3-6. Note that the optimization step is not on the list yet!

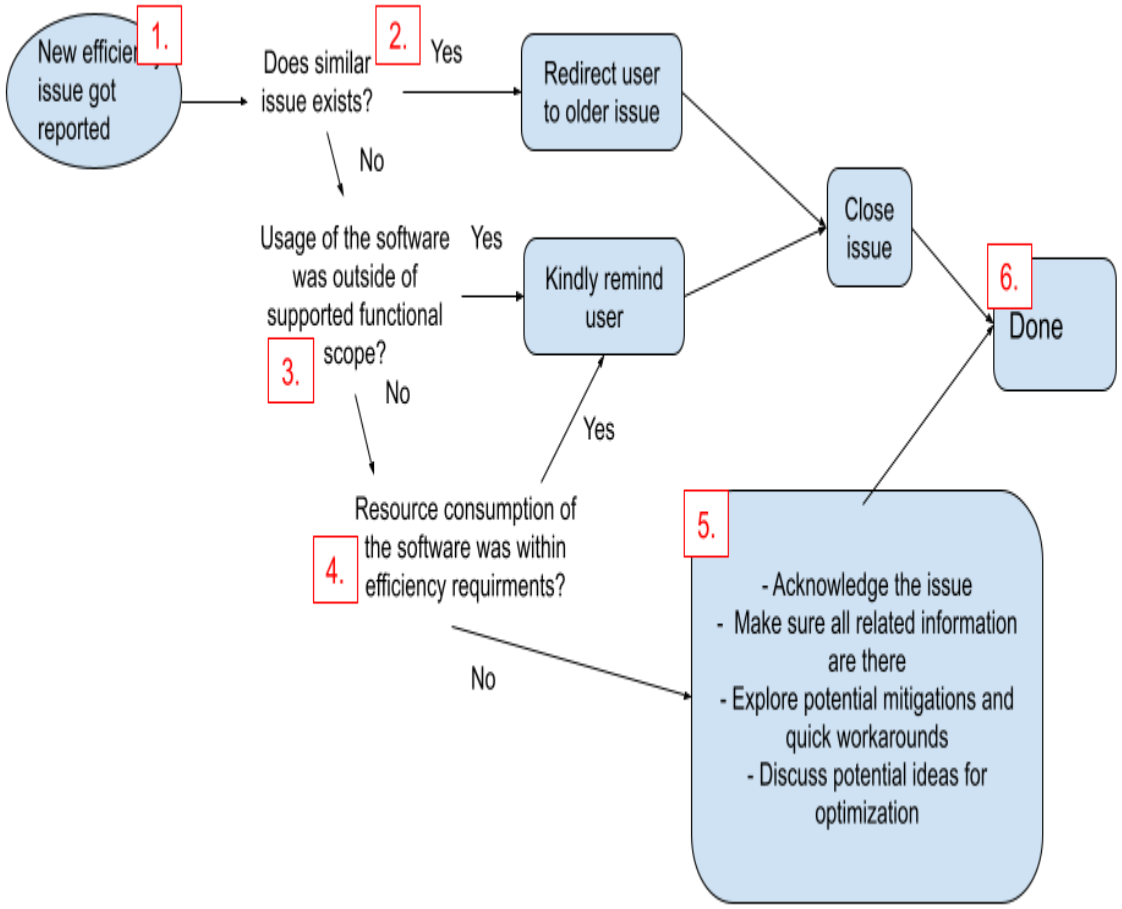


Figure 3-6. Recommended flow for efficiency issue triaging.

1: Efficiency issue was reported on our bug tracker

The whole process starts when someone reports an efficiency issue for the software we are responsible for. If more than one issue was reported, always begin the process shown in Figure 3-6 per every single issue (divide and conquer!). Note that going through this process and putting things through a bug tracker should be your way to go, even for small personal projects. How else would you remember all the things in detail you want to improve? Similarly, if you, a developer responsible for the software, find an efficiency issue—don't specially treat this, just file the issue and allow you or others to follow the same process!

Step 2: Check for duplicates

This might be trivial, but try to be organized. Combine multiple issues for a single, focused conversation. Save time. We are not yet at the stage where artificial intelligence can reliably find duplicates, unfortunately.

Step 3: Validate the circumstances against functional requirements

In this step, we have to ensure that the performance issue author has observed resulted from the supported functionality of our software. We design software for specific use cases defined in functional requirements. Due to the high demand for solving a variety of unique yet sometimes similar use cases, users often try to "abuse" our software for doing something it was never meant to do. Sometimes they are lucky, and things work, sometimes it ends with crashes, unexpected resources usage or slowdowns. For example, while maintaining Prometheus project, we were constantly facing situations where users tried to ingest unique events into Prometheus. The problem is that we designed the Prometheus as an efficient metric monitoring solution with a bespoke time-series database that assumes storing aggregated samples over time. If the ingested series were labelled with unique values, Prometheus slowly but surely was starting to use many resources (we call it a high-cardinality situation). In order to keep focused, we reminded the user of the issue about this requirement mismatch and closed the bug report. Similarly, we should do the same if the agreed prerequisites are not matched. For example, the unsupported, malformed request was sent, the software was deployed on a machine without GPU when it was clearly required in functional requirement agreement, and so on.

Step 4: Validate the situation against RAERs

Some expectations towards speed and efficiency cannot or does not need to be satisfied. This is where the formal efficiency requirements specification discussed in "Resource Aware Efficiency Requirements" is invaluable. If the reported observation (e.g. response latency for the

valid request) is still within the software agreed performance numbers, we should communicate that fact and move on[14]. Similarly, when the issue author deployed our software with HDD disk where SSD was required or the program was running on the machine with lower CPU cores than stated in the formal agreement.

> **FUNCTIONAL OR EFFICIENCY REQUIREMENTS CAN CHANGE!**
>
> There might also be cases where the functional or efficiency specification did not predict certain corner cases. As a result, the specification might need to be revised to match reality. Requirements and demand evolve, so should performance specification and expectations.

Step 5: Acknowledge the issue, note it for prioritization and move on

Yes, you read it right. After you check the impact and all the previous steps, it's often acceptable (and even recommended!) to do almost nothing about the reported problem at the current moment. There might be more important things that need our attention—maybe an important, overdue feature or another efficiency issue in a different part of the code. The world is not perfect. We can't solve everything. Exercise your assertiveness. Notice that this is not the same as ignoring the problem. We still have to acknowledge that there is an issue and ask follow-up questions that will help find the bottleneck and optimize it at a later date. Make sure to ask for the exact software version they are running. Try to provide a workaround or at least hints on what's going on so the user can help you find the root cause. Discuss ideas of what could be wrong. Write it all down on the issue. This will help yourself or another developer to have a great starting point later on. Communicate clearly that you will prioritize this issue with the team on the next prioritization session for the potential optimization effort.

6: Done, issue was triaged

Congratulations, the issue is handled. It's either closed or open. If it's open after all those steps, we can now consider its urgency and discuss the next steps with the team. Once we plan to tackle a specific issue, the efficiency flow in "Efficiency-Aware Development Flow" will tell you how to do it effectively. Fear not. It might be easier than you think!

THIS FLOW IS APPLICABLE FOR BOTH SAAS AND EXTERNALLY INSTALLED SOFTWARE.

The same flow is applicable for the software that is installed and executed by the user itself on their laptop, smartphone or servers (sometimes called "on-premise" installation) as well as when it's managed by our company "as a service" (Software as a Service—SaaS). As developers, we should still try to triage all issues in a similar, systematic way.

Before we move into this exciting optimization spree, one important thing to remember is that, generally, it is easier and more productive to do optimizations in one area at a time, in isolation from other work. The reason is simple—there are thousands of different little pieces in our computer architecture and operating system that allow us to execute our software. Optimizing just one element of our program in a way that will speed things up might sound straightforward. In reality, though, such improvement has to work reliably across different environments and cases, all while maintaining the correct output. Doing just that is a challenge in itself. If we try to achieve more wins, like optimizing part B and adding feature C and maybe fixing part D, we would add lots of unknowns that significantly decrease the chances of succeeding in a reasonable time. Dividing the problem into smaller pieces and conquering all in iterations is one of the most important patterns we will repeat in this book when discussing optimizations, benchmarking, measuring and development.

We divided optimizations into reasonable and deliberate. Let's not hesitate and make the next division. To simplify and isolate the problem of software efficiency optimizations, we can divide it into levels, which we can then design and optimize in isolation. We will discuss those in the next section.

Optimization Design Levels

Let's take our previous real-life example of the long commute to everyday work (we will use this example a couple of times in this chapter!) If such a commute makes you unhappy because it takes a considerable effort and is too long, it might make sense to optimize it. There are, however, so many levels we can do this on. We can start as small as buying more comfortable shoes for walking distances. Alternatively, we could buy an electric scooter or a car if that helps. We could plan the journey, so it takes less time or less distance to walk. We could buy an ebook reader and invest in a book reading hobby. Finally, we could move closer to the workplace or even change jobs. We could do one such optimization in those separate "levels" or all of them, but each optimization takes some investment, tradeoff (buying a car costs money) and effort. Ideally, we want to minimize the effort, while maximizing value and making a difference.

There is another crucial aspect of those levels: optimizations from one level can be impacted or devalued if we do optimization on a higher level. For instance, let's say we did many optimizations to our commute on one level. We bought a better car, organized car sharing to save on fuel, changed work time to avoid traffic etc. Imagine we would now decide to optimize on a higher level: move to an apartment within walking distance to our workplace. In such a case, any effort and investment in previous optimizations are now less valuable (if not fully wasted). Depending on the situation, they might have been worth it if it allowed us to survive until moving flat. Or the opposite could be true—if we knew we were moving flat, we could save optimization effort on the commute level. This is exactly the same in the engineering field. We should be aware of where we spend our optimization effort and when.

When studying computer science, one of the students' first encounters with optimization is learning theory about algorithms and data structures. They study how to optimize programs using different algorithms with better time or space complexities (explained in this book in [Link to Come]). While changing the algorithm we use in our code is an important optimization technique (covered in [Link to Come]), we have many more areas and

variables we can optimize to overall improve our software efficiency. Code itself does not run in the air (even if it's running in clouds!). To properly talk about the performance, there are more levels software depends on.

Figure 3-7 presents levels that take significant part in software execution. This list of levels is inspired by Jon Louis Bentley list made in 1982[15], and it's still very accurate.

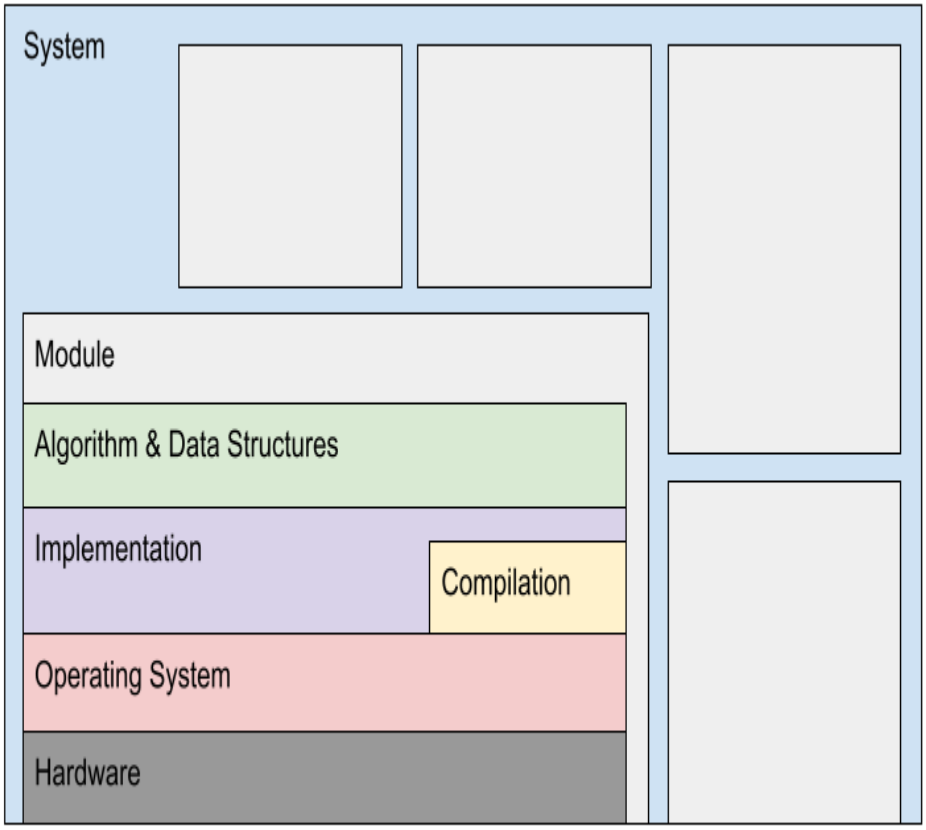


Figure 3-7. Levels that take part in software execution. We can provide optimization in each of those in isolation.

For this book, we can outline five optimization design levels, each with its optimization approaches and verification strategies. Let's dig into those, from the highest to lowest:

System level

In most cases, our software is part of some system. Maybe it's one of many distributed processes, and maybe it's a thread in the bigger monolith application. In all cases, we can say that system is structured around multiple modules. A module is a small software component that encapsulates certain functionality behind the method, interface, or other APIs (e.g. network API or file format) to be interchanged and modifier easier. In fact, each Go application, even the smallest, is an executable module that imports the code from other modules. As a result, your software depends on another system's component or depends on other components, or typically both. Optimizing on the system level means changing what modules are used, how they are linked together, who is calling which component and how often. In some way, we could say we are designing algorithms that work across modules and APIs, which are our data structures. It is non-trivial work but often brings enormous efficiency improvements. We will speak about approaches for system-level optimizations in [Link to Come].

Intramodule Algorithm and Data Structure level

Given a problem to solve, its input data and expected output, the module developer usually starts by designing two main elements of the procedure. First is the **algorithm**, so a finite number of computer instructions that operate on data and can solve our problem (e.g. produce correct output). You probably heard about many popular ones: binary search, quicksort, merge sort, map-reduce, and more. Any custom set of steps to do by your program can be called an algorithm too. The second element is **data structures**, often implied by a chosen algorithm. They allow us to store data on our computer, e.g. input, output or intermittent data. There are unlimited options here, too: arrays, hash maps, linked lists, stacks, queues, others, mixes or custom ones. A solid choice of the algorithms within your module is extremely important. They have to be revised for your specific goals (e.g. request latency) and characteristics of the input. We will speak about algorithm

level optimizations in [Link to Come] and how to assess them via tools like Big O-notation in [Link to Come].

Implementation (Code) level

Algorithms in the module do not exist until they are written in code and compilable to machine code. Developers have huge control here. We can have an inefficient algorithm implemented efficiently, which fulfils our RAERs. We can have an amazing, efficient algorithm implemented poorly that causes unintended system slowdowns. Optimizing on code level means taking a program written in higher level, practical language (e.g. Go) that implements a certain algorithm and producing a more efficient program in any aspect we want (e.g. latency) that yields the same, correct output. We discuss code level optimizations and verification techniques in [Link to Come].

> **NOTE**
>
> Some previous materials consider the compilation step as a separate level. I would argue that code level optimizations techniques have to embody compiler level ones. There is a deep synergy between your implementation and how the compiler will translate it to machine code. As developers, we have to understand this relation. We will explore Go compiler implications more in "Understanding Go Compiler".

Operating system level

These days our software is generally never executed directly on the machine hardware and is never running alone. Instead, we run operating systems that split each software execution into processes (then threads), schedule them on CPU cores, and provide other essential services like memory and IO management, device access and more. On top of that, especially in cloud-native environments, we have additional virtualization layers (virtual machines, containers) that we can put in the operating system level bucket. All those layers pose some overhead that can be optimized by those who control the operating system

development and configuration. In this book, I assume we, Go developers, can rarely impact this level. Yet, we can gain a lot by understanding the challenges and usage patterns that will help us achieve efficiency on other, higher levels. We will go through those in Chapter 4, mainly focusing on Unix operating systems and popular virtualization techniques. Note I assume in my book that device drivers and firmware also fit into this category.

Hardware level

Finally, at some point, a set of instructions translated from our code are executed by the computer CPU units, with internal caches that are connected to other essential parts in the motherboard: RAM, local disks, network interfaces, input and output devices and more. Usually, as developers or operators, we can abstract away from this complexity (which also varies across hardware products) thanks to the operating system level mentioned before. Yet, the performance of our applications is limited by hardware constraints. Some of them might be surprising. For example, were you aware of NUMA nodes existence for multi-core machines and how those can affect our performance?, did you know that memory buses between CPU and memory nodes have limited bandwidth? It's an extensive topic that may have an impact on our software efficiency optimization processes. We will explore this topic in Chapter 4 together with mechanisms Go employs to tackle those.

What are the practical benefits of dividing our problem space into levels? First of all, studies[16] show (and I can confirm that from my experience) that when it comes to application speed, it is often possible to achieve speedups with factors of ten to twenty at any of the mentioned levels if not more. The good news is that this implies the possibility to focus our optimizations on just one level to gain the desired system efficiency[17]. The bad news is that depending on the situation, you might not be able to change certain levels. For example, generally, as programmers, we don't have the power to change the compiler, operating system or hardware easily. Similarly, system

admins won't be able to change the algorithm the software is using. Instead, they can replace systems, configure or tune them.

Another implication is that if you would optimize your implementation already, let's say 10 to 20 times in one level, it might be hard to optimize this level further without significant sacrificing in development time, readability and maintainability. So you might have to look at another level to gain more.

Last but not least, as in our commuting example when we optimized for car transport but ended up walking, changing one level can break optimizations on the other levels. For instance, we started with one algorithm, which was memory bound (algorithm level), optimizing parts touching the memory and doing more work on the CPU (code level). By doing algorithm level optimization and changing it to be CPU bound, our work might need to be reverted. Everything is connected. This is why it's important to see the big picture when optimizing, even though we should do them in isolation.

We discussed what optimization is, we mentioned how to set performance goals, handle efficiency issues, and the design levels we operate in. Now it's time to hook everything together and inject this knowledge into the complete development cycle.

Efficiency-Aware Development Flow

> *The primary concerns of the programmer during the early part of a program's life should be the overall organization of the programming project and producing correct and maintainable code. Furthermore, in many contexts, the cleanly designed program is often efficient enough for the application at hand.*
>
> —Jon Louis Bentley, Writing Efficient Programs (1982)

Hopefully, at this point, you are aware that we have to think about performance, ideally from the early development stages. But as we mentioned in "Be Vigilant to Simplifications" there are risks. Let's say it loud. We don't develop code for it to be just efficient. We write programs

for certain functionality that will match the functional requirements we set or get from stakeholders. Our job is to get this work done effectively, so a pragmatic approach is necessary. How might the process of developing a working but efficient code might look from a high-level point of view?

We can simplify the whole development process to ten steps, as presented in Figure 3-8. For lack of better terms, let's call it **TFBO** flow—test, fix, benchmark, optimize.

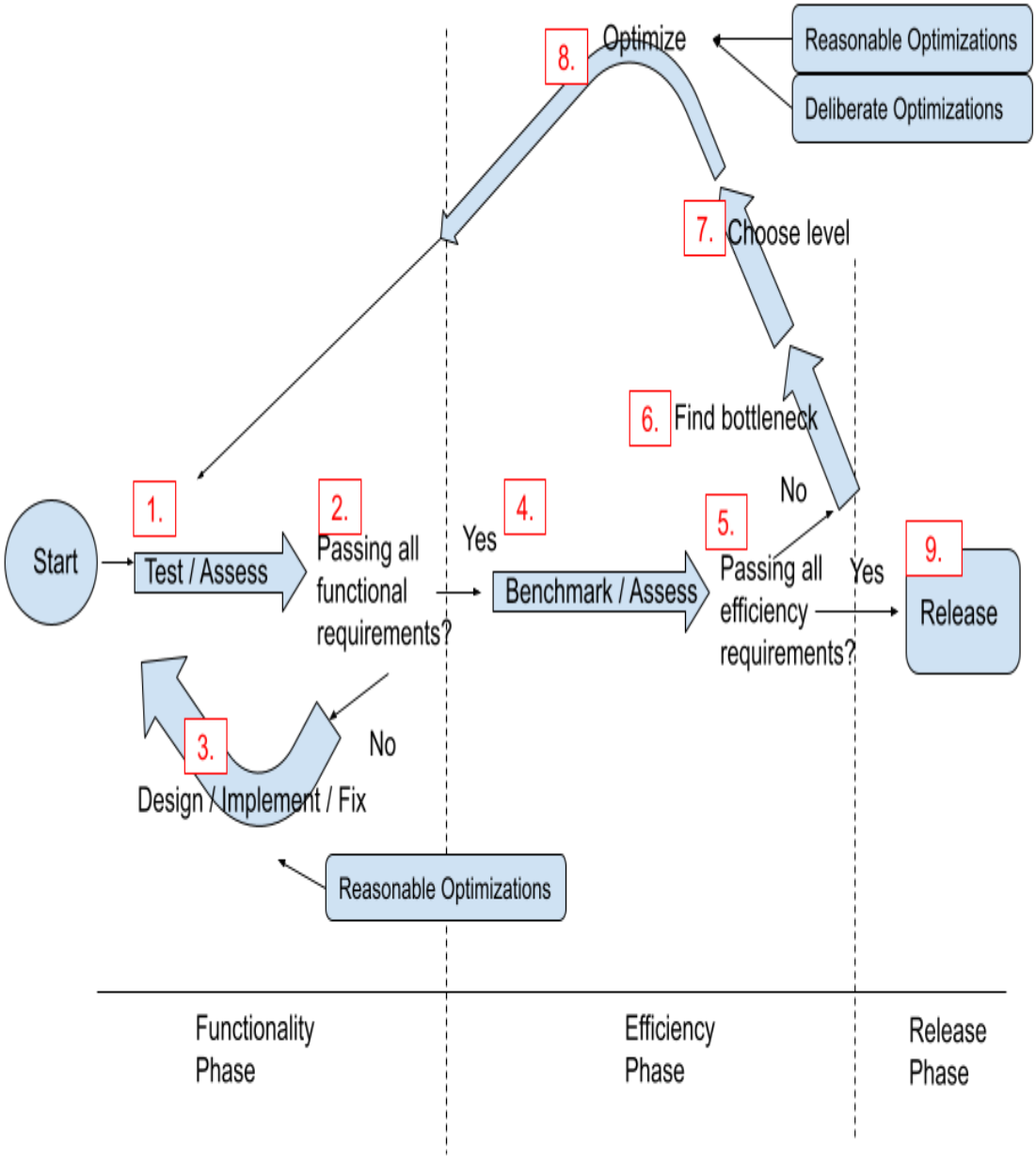


Figure 3-8. Efficiency aware development flow, applicable on any level.

The process is systematic and highly iterative. Requirements, dependencies and environments are changing, so we have to work in smaller chunks too. TFBO process can feel a little strict, but trust me, mindful and effective software development requires some discipline. It applies to cases when

you create new software from scratch, add a smaller feature, or change the code. TFBO should work for software written in any language, not only Go. It is also applicable for all levels mentioned in "Optimization Design Levels".

Let's now go through the TFBO steps presented in Figure 3-8.

Functionality Phase

> *It is far, far easier to make a correct program fast than it is to make a fast program correct.*
>
> —H. Sutter and A. Alexandrescu, C++ Coding Standards: 101 Rules

Always start with functionality first. No matter if we aim to start a new program, add new functionality or just optimize an existing one, we should always begin with the design or implementation of the functionality. Make it work, make it simple, readable, maintainable, secure etc., according to goals we have set, ideally in written form. Especially when you are starting your journey as a software engineer, focus on one thing at the time. With practice, we can add more reasonable optimizations early on as we discussed in "Reasonable Optimizations" to the process. Steps 1-3 dive into the rest of this phase in detail.

1. Test functionality first

It might feel counterintuitive for some, but you should almost always start with some sort of verification framework for the expected functionality in place. The more automated it is, the better. This also applies to when you have a blank page and start developing a new program. This development paradigm is called Test-Driven Development (TDD). It is particularly focused on code reliability and feature delivery velocity efficiency. In a strict form, on code level, it mandates a specific flow:

1. Write a test (or extend an existing one) such that it expects the feature to be implemented.

2. Make sure to run all tests and see the new tests failing for expected reasons. If you don't see the failure or you see other failures, fix those tests first.

3. Iterate with the smallest possible changes until all tests pass and the code is clean.

TDD eliminates many unknowns. Imagine if we would not follow TDD. We add a feature and we write a test. It's easy to make a mistake that makes the test always pass even without our feature. Similarly, let's say we add the test after implementation and it pass but other tests, previously added, fail. Most likely, we did not run a test before the implementation, so we don't know if everything worked before. TDD ensures you don't run into those questions at the end of your work, enormously improving reliability. It also reduces implementation time, allowing safe code modifications and giving you feedback early. Furthermore, what if the functionality we wanted to implement is already done and we did not notice? Writing a test first would reveal that quickly, saving us tons of time. Spoiler alert: we will use the same principles for Benchmark Driven Optimization in step 4 later on!.

TEST-DRIVEN DEVELOPMENT IN PRACTICE

The most popular excuse for skipping writing tests before (or even after) development I have noticed, was that tests would need to be very complex. For example, when code is integrated with many microservices and databases, a single unit test is not enough, we need integration tests with many mocks, fake stubs and the number of edge cases permutations grows. There are however things that help:

- Invest in simple frameworks that allow reusing tests[18].
- Run dependant systems in docker containers directly in unit test thanks to open source the Go e2e package we maintain.
- User your benchmark code to validate correctness too, thanks to `testOrBench` utility[19]

The TDD can be easily understood as a code-level practice, but what if you're designing or optimizing algorithms and systems? The answer is

actually quite simple—the flow is the same, but our testing strategy has to be applied on a different level, e.g. validating system design. We will discuss those techniques in detail for the main design levels in [Link to Come].

Okay, let's say we implemented a test or performed an assessment on what is currently designed or implemented. What's next?

2. Do we pass functional tests?

With the results from step 1, our work is much easier—we can perform data-driven decisions on what to do next! First, we should compare tests or assessment results with the functional requirements we agreed on. Is the current implementation or design fulfilling the specification? Great, we can jump to step 4. However, if tests are failing, or the functionality assessment shows some functionality gap, it's time to go to step 3 and fix this situation.

The problem is when you don't have those functional requirements stated anywhere formally. As discussed in "Efficiency Requirements Should be Formalized", this is why asking for functional requirements or defining them on your own is so important. Even the simplest bullet-point list of goals, written in project README, is better than nothing.

Now, let's explore what to do if the current state of our software does not pass functional verification.

3. If tests fail, we have to fix, implement or design missing parts

Depending on the design level we are at, in this step, we should design, implement or fix the functional parts to close the gap between the current state and functional expectation. As we discussed before in "Reasonable Optimizations", no optimization other than obvious, Reasonable Optimizations is allowed here. Focus on readability, design of modules and simplicity. For example, don't bother thinking if it's more optimal to pass an argument by pointer or value or if parsing integers here will be too slow unless it's obvious. Just do whatever makes sense from a functional and readability standpoint. We don't validate efficiency here yet, so let's forget about Deliberate Optimizations for now.

As you might have noticed in Figure 3-8, steps 1, 2, and 3 compose a small loop. This is to allow us to have an early feedback loop every time we change things in our code or design. Step 3 is like us steering the direction of our boat called "software" when sailing over the ocean. We know where we want to go and we sort of understand how to look at the sun or stars to go in the right direction. Yet without precise feedback tools like GPS, we can end up sailing to the wrong place and realizing that only after weeks. This is why it's beneficial to validate our sailing position in short intervals, for early feedback!

This is the same for our code. We don't want to work for months only to learn that we did not get closer to what we expected from the software. Leverage the functionality phase loop by making a small iteration of code or design change, going to step 1 (run tests), 2 and going back to step 3 to do another little correction[20]. This is proven to be the most effective development cycle engineers have found over the years. All modern methodologies like extreme programming, Scrum, Kanban and other Agile techniques are built on small iterations premise.

After potentially hundreds of iterations, we might have software or design that fulfils, in step 2, the functional requirements we have set for ourselves for this development session. Finally, it's time to ensure our software is fast and efficient enough! Let's look at that in the next section.

Efficiency Phase

Once we are happy with the functional aspects of our software, it's time to ensure it matches the resource consumption and speed we expect.

Splitting phases and isolating them from each seems like a burden at first glance, but it will actually organize your own developer workflow better. It gives us deep focus, ruling our early unknowns and mistakes and helps us to avoid expensive focus context switches.

Let's start our "Efficiency Phase" by performing initial (baseline) efficiency validation in step 4. Who knows, maybe our software is efficient enough without any changes?

4. If functional tests pass, we can move to the efficiency phase!

Here we employ, a very similar strategy to the one in Step 1 of the functionality phase, but towards efficiency space. We can define an equivalent of the TDD method explained in step 1. Let's call it Benchmark Driven Optimization (BDO). In practice, on the code level, step 4 look as the process below.

1. Write benchmarks (or extend existing ones) for all the operations from the efficiency requirements we want to compare against. Do it even if you know that the current implementation is not efficient yet. We will need that work later. It is not trivial, and we will discuss this aspect in detail in [Link to Come].
2. Ideally, you'd run all the benchmarks to make sure your changes did not impact unrelated operations. In practice, this takes too much time, so focus on one part of the program (e.g. one operation) you want to check and run benchmarks for that. Save the results for later. This will be our baseline.

Similar to step 1, the higher-level assessment might require different tools mentioned in [Link to Come]. Equipped with results from benchmarks or assessments, let's go to step 5.

5. Are we within RAERs?

In this step, we have to compare the results from step 4 with the RAERs we gathered. For example, is our latency within the acceptable norm for the current implementation? Is the amount of resources our operation consumes within what we agreed for? If yes, then there is no optimization needed!

Again, similar to step 2, we have to have requirements or rough goals established for efficiency. Otherwise, we have zero ideas if the numbers we see are acceptable or not. Again, refer to "Acquiring and Assessing Efficiency Goals" on how to define RAERs.

With this comparison, we should have a clear answer. Are we within acceptable thresholds? If yes, we can jump straight to the release process in

step 9. If not, there is exciting optimization logic ahead of us in steps 6, 7 and 8. Let's walk through those now.

6. Find the main bottleneck

Here we have to address the first challenge mentioned in "Optimization Challenges". We are typically bad at guessing which part of the operation causes the biggest bottleneck, and unfortunately, that's where our optimization should focus first.

The word bottleneck describes a place where the majority consumption of specific resources or software comes from. It might be a significant number of disk reads, deadlock, memory leak or a function executed millions of times during a single operation. A single program usually has only a few of those bottlenecks. To perform effective optimization, we have to first understand the bottleneck its consequences.

As part of this process, we need to first understand the underlying root cause of the problem we found in step 5. Let's say we see that operation is too slow. One of the reasons might be that we are actually making too many memory allocations, which takes time and increases the garbage collector's overhead in our program (we will dive more into Garbage Collection in "Garbage Collection"). While we care about the latency, it's the memory usage we should probably focus on improving.

Let's say in our above example situation we identified garbage collection taking too much CPU time, due to a large amount of memory allocation. That's the root cause of our problem, but it's now critical to find what pieces of code are allocating this memory. The important point is now that, not all statements that allocate some memory are bottlenecks and should be optimized if possible in random order. As mentioned Pareto rule in "Optimization Challenges", typically it's the 20% of statements responsible for 80% of memory allocations, so we have to find those that matters. Ideally, list them from those which allocates the most amount of memory in total and look for ways to reduce or optimize this situation one by one. This sixth step is exactly to find those critical places that if optimized, will mitigate our problem the most.

Finding such a place is not that hard in Go. The best way to find a bottleneck is to perform a process called profiling. Among other things, profiling allows us to check how many times a certain statement (e.g function) was executed during the operation or how much memory each statement allocates on the heap. These statistics give a clear indication of where the bottleneck might be. There are many ways you can gather profiles, but the most pragmatic way is to use native profiling tools Go uses, based on popular pprof format. The process of obtaining and understanding profiles is explained in detail in [Link to Come].

> **DON'T GUESS WHERE THE BOTTLENECK IS!**
>
> In theory, very experienced engineers can suspect certain places to be a programming bottleneck by carefully studying the code without any profiling. Don't get too confident, though! Doing quick profiling might be faster than designing and implementing optimization for part of the code that is rarely executed and repeated the whole process!

Let's say we found the set of functions that are executed the most or another part of a program that consumes the most amount of resources. What's next?

7. Choice of level

In step 7, we need to choose how we want to tackle the optimization. Should we make code more efficient? Think about improving the algorithm? Or maybe optimize on the system level? In extreme cases, we might want to optimize the operating system or hardware too! This non-trivial topic will be covered in [Link to Come]. The important part is to stick to single level optimization at one optimization iteration. Similar to the functionality phase, make short iterations and small corrections.

Once we know the level we want to make more efficient or faster, we are ready to perform optimization! Let's take a look at how it can work.

8. Optimize!

This is what everyone was waiting for. Finally, after all that effort, we know:

- What place in code or design to optimize for the most impact.
- What to optimize for—what resource consumption is too large.
- How much sacrifice we can make on other resources because we have RAER. There will be tradeoffs.
- On what level we are optimizing in.

Those elements make the optimization process much easier and often even make it possible, to begin with. Now we focus on the mental model we introduced in "Beyond Waste, Optimization is a Zero-Sum Game". We are looking for **waste**. We are looking for places where we can do **less work**. There are always things that can be eliminated, either for free (Reasonable Optimization) or by doing other work using another resource (Deliberate Optimization).

> **NOTE**
>
> The last part of this book, [Link to Come] is reserved for all kinds of optimization you can do in practice in Go and beyond. I hope you will find inspiration there for optimization you can apply in your program.

Let's say we found some ideas for improvement. This is the moment when you should implement it or design (depending on the level). But what's next? We cannot just release our optimization like this, simply because:

- We don't know that we did not introduce functional issues (bugs).
- We don't know if we improved any performance.

This is why we have to now perform the full cycle here (no exceptions!). It's critical to go to step 1 and test the optimized code or design. If they are problems we have to fix them or revert optimization (steps 2 and 3)[21].

> **WARNING**
>
> It is extremely tempting to ignore functional testing phase when iterating on optimizations. For example if you only reduce one allocation by reusing some memory, what can go wrong?
>
> I caught myself doing this and many times it was a painful mistake. When you find that you code cannot pass tests after a few iteration of optimizations it is hard to find what caused it. Usually you have to revert all and start from scratch. I encourage you to run scoped unit test every time after optimization attempt.

Once we are gain some confidence that our optimization did not break any basic functionality, it's crucial to check if our optimization actually improved the situation we want to improve. It's important to run **the same** benchmark, trying to make sure nothing changed except the optimization you did (step 4). This allows to reduce unknowns and iterate on our optimization in small parts.

With the results from this recent step 4, compare it with the baseline made in the initial visit in step 4. This is a crucial step that will tell us if we optimized anything or introduced performance regression. Again, don't assume anything. Let data speak for itself! Go has amazing tools for that, which we will talk about in [Link to Come]. You can see an example comparison in Figure 3-9. The optimization performed improved both latency and memory significantly.

name	old time/op	new time/op	delta
HandlerReceiveHTTP/typical_labels_under_1KB,_500_of_them/OK-12	1.45ms ± 5%	1.05ms ± 0%	-28.04%
HandlerReceiveHTTP/typical_labels_under_1KB,_500_of_them/conflict_errors-12	6.57ms ±10%	1.26ms ± 0%	-80.87%
HandlerReceiveHTTP/typical_labels_under_1KB,_5000_of_them/OK-12	15.7ms ± 5%	11.3ms ± 0%	-27.99%
HandlerReceiveHTTP/typical_labels_under_1KB,_5000_of_them/conflict_errors-12	63.5ms ± 5%	12.8ms ± 0%	-79.78%
HandlerReceiveHTTP/extremely_large_label_value_10MB,_10_of_them/OK-12	145ms ± 4%	97ms ± 0%	-33.18%
HandlerReceiveHTTP/extremely_large_label_value_10MB,_10_of_them/conflict_errors-12	1.52s ± 4%	0.10s ± 0%	-93.63%

name	old alloc/op	new alloc/op	delta
HandlerReceiveHTTP/typical_labels_under_1KB,_500_of_them/OK-12	1.75MB ± 0%	1.20MB ± 0%	-31.47%
HandlerReceiveHTTP/typical_labels_under_1KB,_500_of_them/conflict_errors-12	4.89MB ± 0%	1.36MB ± 0%	-72.07%
HandlerReceiveHTTP/typical_labels_under_1KB,_5000_of_them/OK-12	18.8MB ± 0%	13.2MB ± 0%	-29.65%
HandlerReceiveHTTP/typical_labels_under_1KB,_5000_of_them/conflict_errors-12	50.1MB ± 0%	14.9MB ± 0%	-70.37%
HandlerReceiveHTTP/extremely_large_label_value_10MB,_10_of_them/OK-12	343MB ± 0%	120MB ± 0%	-65.16%
HandlerReceiveHTTP/extremely_large_label_value_10MB,_10_of_them/conflict_errors-12	816MB ± 0%	120MB ± 0%	-85.32%

name	old allocs/op	new allocs/op	delta
HandlerReceiveHTTP/typical_labels_under_1KB,_500_of_them/OK-12	15.6k ± 0%	4.6k ± 0%	-70.55%
HandlerReceiveHTTP/typical_labels_under_1KB,_500_of_them/conflict_errors-12	35.6k ± 0%	7.6k ± 0%	-78.61%
HandlerReceiveHTTP/typical_labels_under_1KB,_5000_of_them/OK-12	155k ± 0%	45k ± 0%	-70.79%
HandlerReceiveHTTP/typical_labels_under_1KB,_5000_of_them/conflict_errors-12	355k ± 0%	75k ± 0%	-78.84%
HandlerReceiveHTTP/extremely_large_label_value_10MB,_10_of_them/OK-12	165 ± 0%	78 ± 0%	-52.73%
HandlerReceiveHTTP/extremely_large_label_value_10MB,_10_of_them/conflict_errors-12	434 ± 0%	201 ± 0%	-53.63%

Figure 3-9. Example output of the `benchstat` tool, showing a percentage comparison of memory usage and latency between two tests with six different cases.

If the new optimization does not have a better efficiency result, we simply try again and try different ideas until it works out. If the optimization has better results, we save our work and go to step 5 to check if it's enough. If not, we have to make another iteration. It's often useful to build another optimization on what we already did. Maybe there is something more to improve!

We repeat this cycle, and after a few (or hundreds), we hopefully have acceptable results in step 5. In this case, we can move to step 9 and enjoy our work!

9. Release and enjoy!

Great job! You went through the full iteration of the efficiency-aware development flow. Your software is now fairly safe to be released and deployed in the wild. The process might feel bureaucratic, but it's easy to build an instinct for it and follow it naturally. It might be the case that you use this flow without noticing already!

Summary

As we learned from this chapter, conquering efficiency is not trivial. However, there exist certain patterns that help to navigate through this process systematically and effectively.

The main takeaways are that optimizations typically consists of easy to remove the waste of effort our software does. Then after we peel that waste off, what is left is the opportunity to do more work in one area while sacrificing other resource consumption (e.g. do more computation, but using less memory). As we learned later, we can also divide optimizations into reasonable and deliberate. The former can be employed much earlier in the development process. We tend to do them more once we are more experience with the hardware, problem and programing language we work

with. Hopefully, once you read this book, you be able to consider plentiful reasonable optimizations!

In order to be mindful of the tradeoffs and our effort we discussed defining RAER, so we can assess our software towards a formal goal everyone understands.

Last but not least, we discussed easy to remember TFBO flow, which guides us through the practical development process.

To sum up, finding optimization can be considered a problem-solving skill. Noticing waste is not easy and it comes a lot with practice. This is somewhat similar to being good at programming interviews. At the end, what helps is the experience of seeing past patterns that was not efficient enough and how they were improved. Through the book we will exercise those skills and uncover many tools that can help us in this journey.

Yet before that, there is something more important to learn first. We can learn typical optimizations patterns by examples, but we will not be able to effectively remember them and apply them in unique contexts without understanding mechanisms that make those optimizations effective. This is why in the next chapter we will discuss first how Go interacts with the key resources in typical computer architecture.

---

1 There might be exceptions. There might be domains where it's acceptable to approximate results. Sometimes we can (and should) also drop nice-to-have features if they block critical efficiency characteristics we want.

2 Situation where resources are not cleaned after each periodic functionality due to leftover concurrent routine is often referred to as memory leak.

3 Zero-sum game comes from game and economic theory, that describes a situation where one player can only win X if other players in total lost exactly X.

4 I got inspired for dividing optimizations on reasonable and deliberate by the community-driven go-perfbook lead by Damian Gryski. In his book, he also mentioned the "dangerous" optimization category. I don't see a value in splitting categories further since there is a fuzzy borderline between deliberate and dangerous that depends on the situation and personal taste.

5 As we will discuss later in [Link to Come] early sorting is actually a good pattern that often enables more efficient, streamed execution for later computations.

6 No one said challenging ourselves is bad in certain situations. If you have time, playing with initiatives like Advent of Code is a great way to learn or even compete! This is however different to the situation where are paid to develop functional software and if we want to do that effectively.

7 Just imagine, with all resources in the world we could try optimizing the software execution to the limits of physics. And once we are there, we could spend decades on researches that push boundaries with things beyond physics we know. Practically speaking we never find the limit in our lifetime.

8 Runtime cost here, represents the cost of running the program while maintaining the same functionality and latency for the same amount of operations made by software. Note that I did not mention in Figure 3-2 other parts of Total Ownership Costs like maintenance.

9 I personally was never explicitly asked to create a non-functional specification, same with people around me.

10 Funnily enough, with enough program users, even with a formal performance and reliability contract, all observable behaviours of your system will be dependent on somebody. This is known as Hyrum's Law.

11 We use napkin math more often in this book and during optimizations, so I prepared a small cheat sheet for latency assumptions in [Link to Come].

12 We will discuss benchmarks in detail in [Link to Come].

13 We won't have time in this book to look closer into finding the potential maximum resource consumptions based on price, since this changes every month and it depends on the cloud provider and hardware used. This also is highly correlated to the scalability topic which constraints the resource usage further to fit a single server. This is explained in detail in books like Designing Data-Intensive Applications.

14 The reporter of the issue can obviously negotiate a change in the specification with the product owner if they think it's important enough or they want to pay additionally, etc.

15 "Writing Efficient Programs", Jon Louis Bentley

16 In "Multiplicative Speedup of Systems" by R. Reddy and A. Newell (1977) were elaborating on potential speedups of a factor of about ten for each software design level. What's even more exciting is that the fact, that for hierarchical systems the speedups from different levels multiples, which offers massive potential for performance boost when optimizing.

17 This is a quite powerful thought. Imagine you have your application returning a result in 10m, reducing it to 1m, by optimizing on one level (e.g. an algorithm) is a game-changer.

18 One example of such framework is what we implemented for Thanos object storage clients. Each client has to pass common interface acceptance tests. This reduces the amount of work to add new clients, but also assures correctness and same behaviour from an interface point of view!

19 We will expand on `testOrBench` usage in [Link to Come].

20 Ideally, we would have functionality checks for every code stroke or event of the code file saved. The earlier the feedback loop, the better. The main blocker for this is the time required

to perform all tests and their reliability. Fortunately, with technological improvements of the Continous Integration solutions, the feedback loop delay is reduced every year.

21 It might be surprising but optimizations can very often break functionality. The most often problem explored in [Link to Come] is with reusing the same memory. It's tempting to avoid allocating new memory when transforming existing objects. Sometimes however, same object will be modified by caller in subsequent code which can cause side effects and hard to catch bugs. Good functional tests can often detect this early.

Chapter 4. How Go Uses The CPU Resource (or Two)

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at mpotter@oreilly.com.

> *(...) one of the most useful abstractions we can make is to treat properties of our hardware and infrastructure systems as resources. CPU, memory, data storage, and the network are similar to resources in the natural world: they are finite, they are physical objects in the real world, and they must be distributed and shared between various key players in the ecosystem.*
>
> —Susan J. Fowler, Production-Ready Microservices (2016)

As you learned in "Behind Performance", software efficiency depends on how our program uses the hardware resources. If the same functionality uses fewer resources, our efficiency increases and the requirements and net cost of running such a program decrease. For example, if we use less CPU time (CPU "resource") or fewer resources with slower access time (e.g. disk), we reduce the latency of our software.

This might sound simple, but in modern computers, those resources interact with each other in a complex, non-trivial way. Furthermore, there are more processes than just one using those resources, so our program does not use them directly. Instead, those resources are managed for us by an operating system. If that wasn't complex enough, especially in cloud environments, we often "virtualize" the hardware further, so it can be shared across many individual systems in an isolated way. That means there are methods for "hosts" to give access to part of a single CPU or disk to a "guest" operating system that thinks it's all the hardware that exists. In the end, operating systems and virtualization mechanisms create layers between our program and the actual physical devices that store or compute our data.

To understand how to write efficient code or improve our program's efficiency effectively, we have to learn the characteristics, purpose and limits of the typical computer resources like CPU, different types of storage and network. There is no shortcut here. Furthermore, we can't ignore understanding how those physical components are managed by the operating system and typical virtualization layers.

In this and the next chapter, we will go through a pragmatic overview of all you need to know about modern computer resources to understand why your program is slow or uses more resources than expected. Knowing the root cause is the first step toward fixing such a situation.

> **NOTE**
>
> We won't discuss all types of computer architectures with all mechanisms of all existing operating systems as this would be impossible to fit in one book, never mind one chapter. Instead, this chapter will focus on a typical x86-64 CPU architecture with Intel or AMD CISC[1] together with the modern Linux operating system and, eventually, basic virtualization. Most developers deploy their programs to such a setup (e.g. ARM or MacOS). This should get you started and give you a jumping point if you will ever run your program on other, unique types of hardware or operating systems.

In this chapter, we will examine our program execution from the point of view of the Central Processing Unit (CPU). We will discuss how Go uses

CPUs for single and multiple core tasking.

We will start with "CPU in a Modern Computer Architecture" to understand how modern computers are designed, mainly focusing on the Central Processing Unit (CPU, also called "processor"). Then I will introduce "Assembly" language, which will help us understand how the CPU core executes instructions. After that, we will dig into "Understanding Go Compiler" to build awareness of what happens when we do `go build`. Furthermore, we will jump into "CPU and Memory Wall Problem", which will show you why modern CPU hardware is complex. This problem directly impacts writing efficient code on those ultra-critical paths. Finally, we will enter the realm of multi-tasking by explaining how "Operating System Scheduler" tries to distribute thousands of executing programs on out-numbered CPU cores and how "Go Runtime Scheduler" leverages that to implement an efficient concurrency framework for us to use. We will finish with an actionable, practical example of efficient concurrency in "Practical Example of Concurrency in Go".

> **HOW MUCH KNOWLEDGE IS ENOUGH?**
>
> This chapter might get overwhelming at first, especially if you are new to low-level programming. These mechanisms are abstracted away to focus on functionality in regular day-to-day work. Yet, awareness of what is happening will help us understand the optimizations, so focus on understanding high-level patterns and characteristics of each resource (e.g. how the Go scheduler works). We don't need to know how to write machine code manually or how to, blindfolded, manufacture the computer. Instead, let's treat this with curiosity about how roughly things work under the computer case.
>
> In other words, we need to have Mechanical Sympathy.

To understand how the CPU architecture works, we need to explain how modern computers operate. So let's dive into that in the next section.

CPU in a Modern Computer Architecture

All we do while programming in Go is construct a set of statements that tells the computer what to do, step by step. Given predefined language constructs like variables, loops, control mechanisms, arithmetic and I/O operations, we can implement any algorithms that interact with data stored in different mediums. This is why Go, like many other popular programming languages, can be called imperative—we, developers, have to describe how the program will operate. This is also how hardware is designed nowadays—it is imperative too. It waits for program instructions, optional input data, and the desired place for output.

Programming wasn't always so simple. Before general-purpose machines, engineers had to design fixed program hardware to achieve requested functionality, e.g. desk calculator. Adding a feature, fixing a bug or optimizing required changing the circuits and manufacturing new devices. Probably not the easiest time to be a "programmer" in!

Fortunately, around the 1950s, a few inventors worldwide figured out the opportunity for the universal machine that could be programmed using a set of predefined instructions stored in memory. One of the first people to document this idea was a great mathematician, John von Neumann and his team.

> *It is evident that the machine must be capable of storing in some manner not only the digital information needed in a given computation (...) and also the intermediate results of the computation (which may be wanted for varying lengths of time), but also the instructions which govern the actual routine to be performed on the numerical data. In a special-purpose machine, these instructions are an integral part of the device and constitute a part of its design structure. For an all-purpose machine, it must be possible to instruct the device to carry out whatsoever computation that can be formulated in numerical terms.*
>
> —A W. Burks, H. H. Goldstine

What's noteworthy is that most modern general-purpose computers, e.g. PCs, laptops and servers, are based on John von Neumann's design. This assumes that program instructions can be stored and fetched similarly to

storing and reading program data (instruction input and output). We fetch both the instruction to be performed (e.g. `add`) and data (e.g. addition operands) by reading bytes from a certain memory address in the main memory (or caches). While it does not sound like a novel idea nowadays, it established how general-purpose machines work. We call this Von Neumann Computer Architecture, and you can see its modern, evolved variation in Figure 4-1[2].

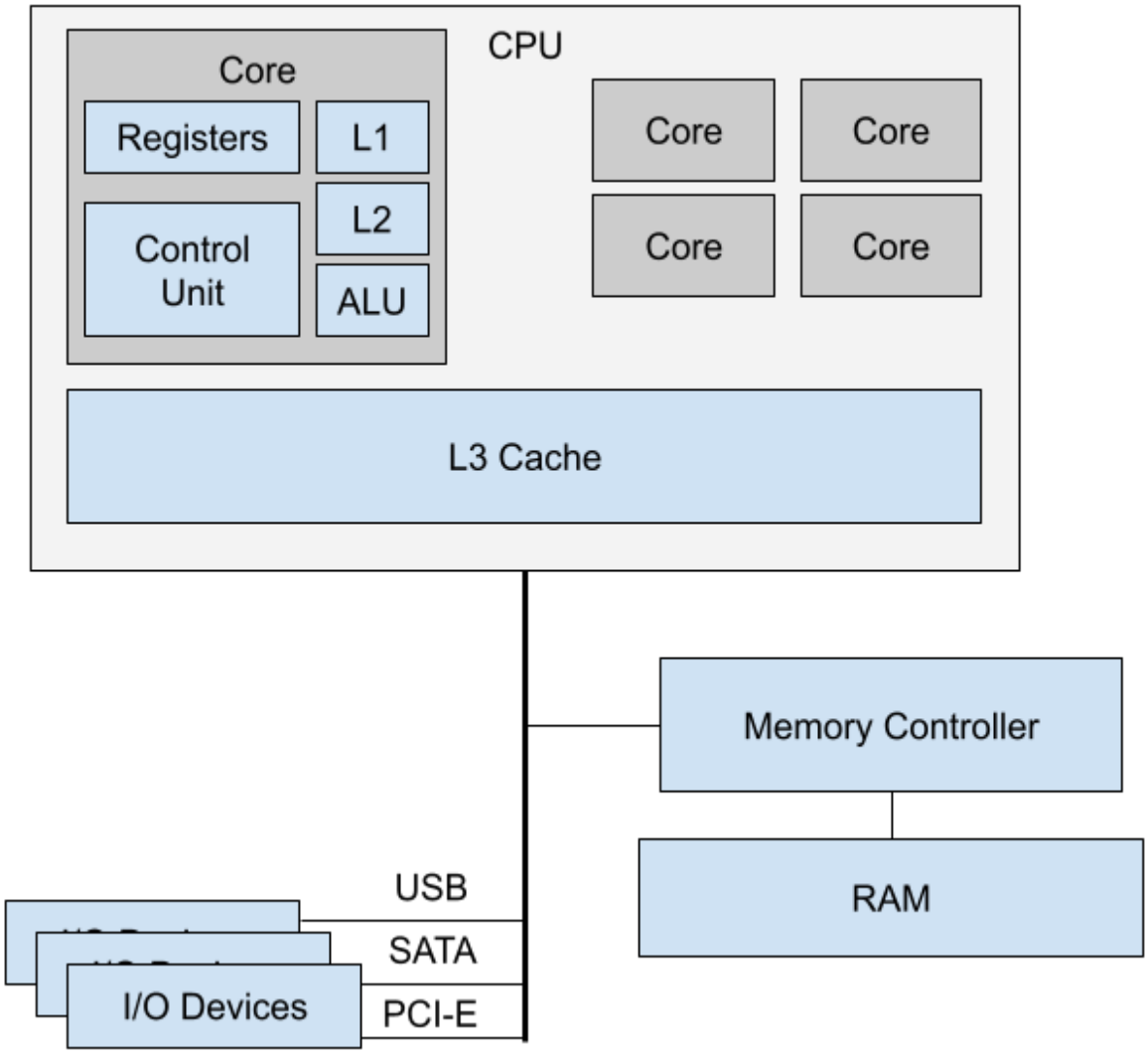


Figure 4-1. High-level computer architecture with a single, multi-core CPU and uniform memory access (UMA).

Modern architecture's heart is a Central Processing Unit (CPU) consisting of multiple cores (4-6 physical cores are the norm in the 2020s PCs). Each core can execute desired instruction with certain data saved in Random-Access Memory (RAM) or any other memory layers like registers or L-caches (sometimes even up to four layers). The RAM explained in "Do we Have a Memory Problem?" performs the duty of the main, fast, volatile memory that can store our data and program's code as long as the computer is powered. The memory controller makes sure RAM is supplied with a constant power flow to keep the information on RAM chips. Last but not least, the CPU can interact with a variety of external or internal Input / Output (I/O) Devices. From a high-level view, an IO device means anything that accepts sending or receiving a stream of bytes, for example, mouse, keyboard, speaker, monitor, HDD or SSD disk, network interface, GPU and thousands more.

Roughly speaking, CPU, RAM and popular I/O devices like disk and network interface are the most important parts of computer architecture. They might seem low-level, but those four are what typically matter in our RAERs, and this is what we are usually optimizing for in our software development.

In this chapter, we will focus on the brain of our general-purpose machines —the CPU. When should we care about CPU resources? Typically, from a performance standpoint, we should start looking at our Go process' CPU resource usage when any of those two occur:

- Our machine can't do other tasks because our process uses all the available CPU resource computing capacity.
- Our process runs unexpectedly slow, while we see higher CPU consumption.

There are many techniques to troubleshoot those severe consequences, but we must first understand the CPU's internal working and program execution basics. This is the key to efficient Go programming. Furthermore, it explains the numerous optimization techniques that might surprise us initially. For example, do you know why in Go (and other languages), we

typically should avoid using linked lists like structures if we plan to quickly iterate over them, despite their theoretical advantages like a quick insertion and deletion?

Before we learn why we must understand how the CPU core executes our programs. Surprisingly, I found that the best way to explain this is by learning how the Assembly language works. Trust me on this; it might be easier than you think!

Assembly

The CPU core, indirectly, can execute programs we write. For example, consider the simple Go code in Example 4-1.

Example 4-1. Simple function that reads numbers from a file and returns the total sum.

```
func Sum(fileName string) (ret int64, _ error) {
   b, err := ioutil.ReadFile(fileName)
   if err != nil {
      return 0, err
   }

   for _, line := range bytes.Split(b, []byte("\n")) {
      num, err := strconv.ParseInt(string(line), 10, 64)
      if err != nil {
         return 0, err
      }

      ret += num ❶
   }

   return ret, nil
}
```

❶ The main arithmetic operation in this function adds a parsed number from the file into a `ret` integer variable representing the total sum.

While such language is far from, let's say, spoken English, it is, unfortunately, still too complex and incomprehensible for the CPU. We call it a not machine-readable code. Thankfully every programming language has a dedicated tool called a compiler[3] that (among other things discussed

in "Understanding Go Compiler") translates our higher-level code to machine code. You might be familiar with a `go build` command that invokes a default Go compiler.

The machine code is a sequence of instructions written in binary format. In principle, each instruction is represented by a number (`opcode`) and followed by optional operands in the form of a constant value or address in the main memory. We can also refer to a few CPU core registers, which are tiny "slots" directly on the CPU chip that can be used to store intermediate results. For example, on AMD64 CPU, we have sixteen 64-bit general-purpose registers referred to as RAX, RBX, RDX, RBP, RSI, RDI, RSP, and R8-R15.

While translating to machine code, the compiler often adds additional code like extra memory safety bound checks. It automatically changes our code for known efficiency patterns for a given architecture. Sometimes this might be not what we expect. This is why it is useful to inspect the resulting machine code when troubleshooting some efficiency problems. Another advanced example of the need to read machine code by humans is when we need to reverse engineer programs without source code.

Unfortunately, it is impossible to directly read the machine code of, let's say, Example 4-1 compiled using, e.g. `go build` command, unless you are a genius. Yet it would be executed by our CPU just fine. There is, however, a great tool that we can use in such situations. We can compile Example 4-1 code not to machine code, but to Assembly language instead. We can also disassemble the compiled machine code to Assembly. The Assembly language represents the lowest code level that can be practically read and (in theory) written by human developers. It also well represents what will be interpreted by the CPU when converted to machine code.

DO I NEED TO UNDERSTAND ASSEMBLY?

We don't need to know how to program in Assembly to write efficient Go code. Yet a rough understanding of the Assembly and the decompilation process are essential tools in our pockets that can often reveal hidden, lower level computation waste.

Practically speaking, it's useful primarily for advanced optimizations when we have already applied all more straightforward optimizations. In addition, however, Assembly is beneficial for understanding the changes the compiler applies to our code when translating to machine code. Sometimes those might surprise us! Finally, it tells us how the CPU works.

It is worth mentioning that we can disassemble compiled code into various Assembly dialects. For example:

- To Intel syntax using standard Linux tool `objdump -d -M intel <binary>`.
- To AT&T syntax using similar command `objdump -d -M att <binary>`.
- To Go "pseudo" assembly language using Go tooling `go tool objdump -s <binary>`.

All three mentioned dialects are used in the various tools, and their syntax varies. To have an easier time, always ensure what syntax your disassembly tool uses. The Go Assembly is a dialect that tries to be as portable as possible, so it might not exactly represent the machine code. Yet it is usually consistent and close enough for our purposes. It can shows all compilation optimization discussed in "Understanding Go Compiler". This is why Go Assembly is what we will be using throughout this book. In Example 4-2 we can see tiny, disassembled part of the compiled Example 4-1 that represents `ret += num` statement.

Example 4-2. Addition part of code in Go Assembly language decompiled from the compiled Example 4-1.

```
// go tool objdump -s sum.test
ret += num
```

```
0x4f9b6d      488b742450    MOVQ 0x50(SP), SI   ❶
0x4f9b72      4801c6        ADDQ AX, SI         ❷
```

❶ The first line represents quadword (64-bits) MOV instruction that tells the CPU to copy the 64-bit value from memory under the address stored in register `SP` plus 80 bytes and put that into `SI` register[4]. The compiler decided that `SI` will store the initial value of the return argument in our function, so the `ret` integer variable for the `ret+=num` operation

❷ As a second instruction, we tell the CPU to add quadword value from the AX register to the SI register. The compiler used the `AX` register to store the `num` integer variable, which we parsed from the `string` in previous instructions (outside of this snippet).

Similar output to Example 4-2 can be obtained by compiling the source code to Assembly using `go build -gcflags '-S' <source>`.

To make things more complex, each distinct CPU implementation allows a different set of instructions, with different memory addressing etc. This is why the industry created Instruction Set Architecture (ISA) to specify a strict, portable interface between software and hardware. Thanks to ISA, we can compile our program, for example, to machine code compatible with ISA for x86 architecture and run it on any x86 CPU[5]. ISA defines data types, registers, main memory management, fixed set of instructions, unique identification, input/output model, etc. There are various ISAs for different types of CPUs. For example, both 32-bit and 64-bit Intel and AMD processors use x86 ISA, and ARM uses its ARM ISA (for example, new Apple M1 chips uses ARMv8.6-A).

As far as Go developers are concerned, ISA defines a set of instructions and registers our compiled machine code can use. To produce a portable program, a compiler can transform our Go code into machine code compatible with specific ISA (so architecture) and the type of the desired operating system. In the next section, let's look at how the default Go compiler works. On the way, we will uncover mechanisms to help the Go compiler produce efficient and fast machine code.

Understanding Go Compiler

The topic of building effective compilers can fill a few books. In this book, however, we will try to understand the Go compiler basics we, Go developers interested in the efficient code, have to be aware of. Generally, many things are involved in executing the Go code we wrote on the typical operating system, not only compilation. First, we need to compile it using a compiler, and then we have to use a linker to link different object files together, including potentially shared libraries. Those compile and link procedures, often called building, produce the executable ("binary") that the operating system can execute. During the initial start, called loading, other shared libraries can be dynamically loaded, too (e.g. Go plugins).

There are many code-building methods for Go code, designed for different target environments. For example, there is Tiny Go optimized to produce binaries for microcontrollers gopherjs that produces JavaScript for in-browser execution and android that produces programs executable on Android operating systems. However, this book will focus on the default and the most popular Go compiler and linking mechanism available in the `go build` command. The compiler itself is written in Go (initially in C). The rough documentation and source code can be found here.

The `go build` can build our code into many different outputs. We can build executables that require system libraries to be dynamically linked on startup. We can build shared libraries or even C compatible shared libraries. Yet the most common and recommended way of using Go is to build executables with all dependencies statically linked in. It offers a much better experience where invocation of our binary does not need any system dependency of a specific version in a certain directory. It is a default build mode for code with starting `main` function that can also be explicitly invoked using `go build -buildmode=exe`.

The `go build` command invokes both compilation and linking. While the linking phase also performs certain optimizations and checks, the compiler probably performs the most complex duty. Go compiler focuses on a single package at once. It compiles package source code into the native code that the target architecture and operating systems support. On top of that, it validates, optimizes that code and prepares important metadata for

debugging purposes. We need to "collaborate" with the compiler to write efficient Go and not work against it.

> *I tell everyone, if you're not sure how to do something, ask the question around what is the most idiomatic way to do this in Go. Because many of those answers are already tuned to being sympathetic with the operating system of the hardware.*
>
> —Bill Kennedy, Bill Kennedy on Mechanical Sympathy (Go Time)

To make things more interesting, `go build` also offers a special cross-compilation mode if you want to compile a mix of Go code that uses functions implemented in `C`, `C++` or even Fortran!. This is possible if you enable a mode called CGO, which uses a mix of `C` (or `C++`) compiler and Go compiler. Unfortunately, CGO is not recommened, and it should be avoided if possible. It makes the build process slow, the performance of passing data between C and Go is questionable, and non-CGO compilation is already powerful enough to cross-compile binaries for different architectures and operating systems. Luckily most of the libraries are either pure Go or are using pieces of Assembly that can be included in Go binary without CGO (more on including Assembly in [Link to Come]).

To understand the impact of the compiler on our code, see the existing stages the Go compiler performs in Figure 4-2. While `go build` includes such compilation, we can trigger just the compilation (without linking) alone using `go tool compile`.

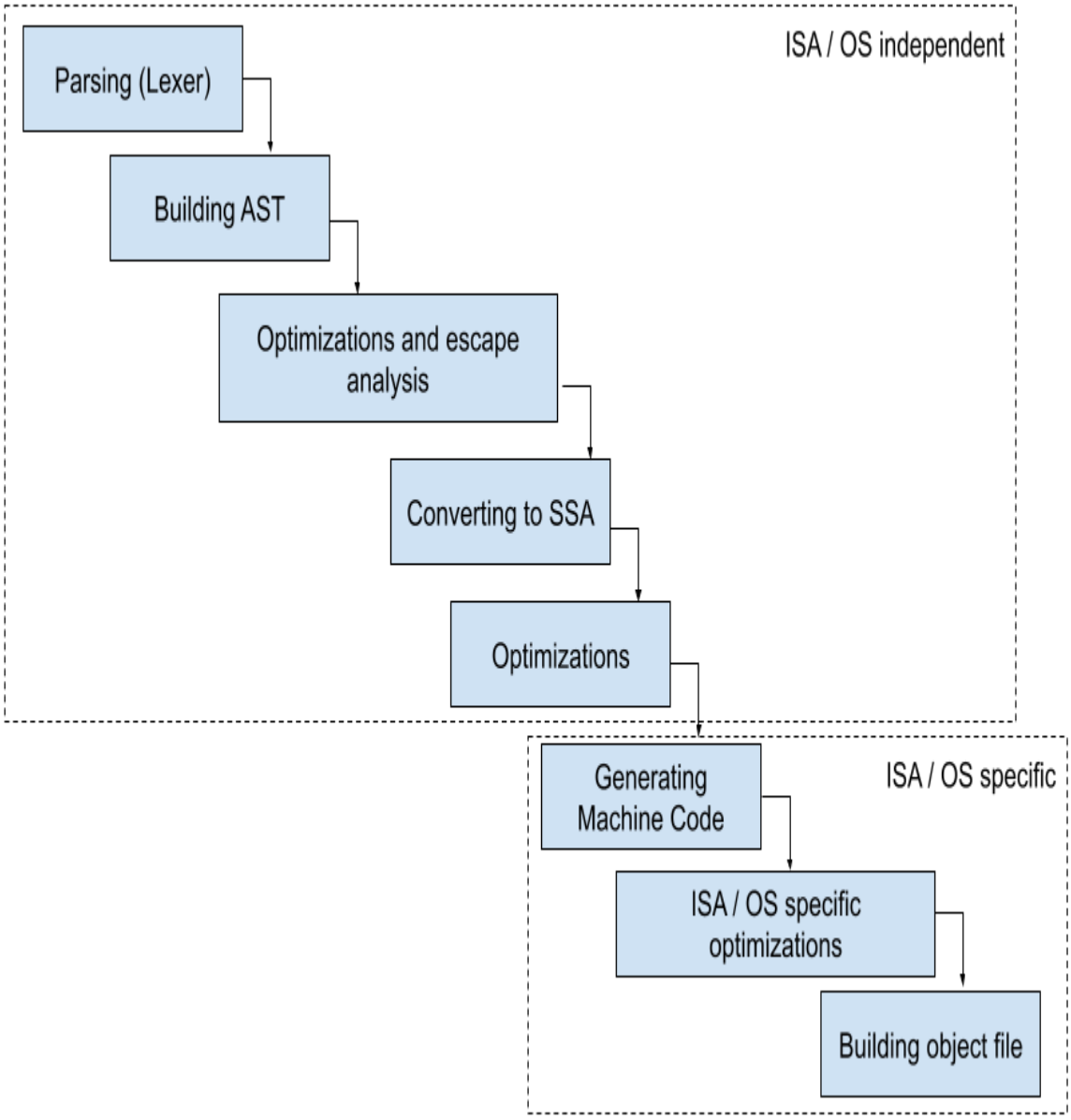


Figure 4-2. Stages performed by the Go compiler on each Go package.

As we mentioned previously, the whole process resides around the packages you use in your Go program. Each package is compiled in separation, allowing parallel compilation and separation of concerns. The compilation flow presented in Figure 4-2 works as follows:

1. The Go source code is first tokenized and parsed. The syntax is checked. The syntax tree references files and file positions to produce a meaningful error and debugging information.

2. Abstract syntax tree (AST) is built. Such a tree notion is a common abstraction that allows developers to create algorithms that easily transform or check parsed statements. While in AST form, code is initially type-checked. Declared but not used items are detected.

3. The first pass of optimization is performed. For example, the initial dead code is eliminated, so the binary size can be smaller and less code needs to be compiled. Then, escape analysis (discussed in "Do we Have a Memory Problem?") is performed to decide which variables can be placed on the stack and which have to be allocated on the heap. On top of that, in this stage, the function inlining occurs for simple and small functions.

FUNCTION INLINING

Functions in programming language allow us to create abstractions, hide complexities and reduce repeated code. Yet, the cost of calling execution is non-zero. For example, a function with a single argument call needs ~10 extra CPU instructions[6]. While the cost is fixed and typically at the level of nanoseconds, it can matter if we have thousands of those calls in the hot path and the function body is small enough that this execution call matters. This is why the compiler automatically substitutes some function calls with the exact copy of its body. This is called inlining or inline expansion. The logic is quite smart. For instance, from Go 1.9, the compiler can inline both leaf and mid-stack functions. There is also a second benefit of inlining—the ability for the compiler to apply other optimizations more effectively in code with fewer functions. Note that the structure methods, from a compiler perspective, are just functions, with the first argument being that structure, so the same inlining technique applies here.

MANUAL INLINING IS ALMOST NEVER NEEDED

It is tempting for beginner engineers to micro-optimize by inlining some of their functions manually. While developers had to do it in the early days of programming, this functionality is a fundamental duty of the compiler, which usually knows better when and how to inline a function. Use that fact by focusing on your code readability and maintainability with as many functions as you need. Rare cases of manual interventions are explored in [Link to Come], and they are focused more on simplifying functions than inlining them manually.

4. After early optimizations on AST, the tree is converted to the Static Single Assignment (SSA) form. This low-level, more explicit representation makes it easier to perform further optimization passes using a set of rules. For example, with the help of the SSA compiler can easily find places of unnecessary variable assignments[7].
5. Compiler applies further machine-independent optimization rules. So, for example, statements like `y := 0*x` will be simplified to `y :=0`. The complete list of rules is enormous and only confirms how complex this space is. Furthermore, some code pieces can be replaced by intrinsic function, so heavily optimized equivalent code (e.g. in raw Assembly).
6. Based on `GOARCH` and `GOOS` environment variables, the compiler invokes the `genssa` function that converts SSA to the machine code for desired architecture (ISA) and operating system.
7. Further ISA and operating system specific optimizations are applied.
8. Non-dead package machine code is built into a single object file (with the `.o` suffix) and debug information.

The final object file is compressed into a `tar` file called a Go archive, usually with .a file suffix. You can unpack it with `tar <archive>` or `go tool pack e <archive>` commands. Go archive typically contains the object file and package metadata in `__.PKGDEF file`. Such archive files for each package can be used by Go linker (or other linkers) to combine all into a single executable, commonly called a "binary file". Depending on the operating system, such a file follows a certain format telling the system how to execute and use it. Typically for Linux, it will be an Executable and Linkable Format (ELF). On Windows, it might be Portable Executable (PE).

The machine code is not the only part of such a binary file. It also carries the program's static data like global variables and constants. The executable file also contains a lot of debugging information that can take a considerable amount of binary size, like simple symbols table, basic type

information (for reflection) and PC-to-line mapping (address of the instruction to the line in the source code where the command was). That extra information enables valuable debugging tools to link machine code to the source code. Many debugging tools use it, for example, profiling (e.g. in `pprof` package explained in [Link to Come]) and the aforementioned `objdump` tool. For compatibility with debugging software like `Delve` or `GDB`, the DWARF table is also attached to the binary file[8].

On top of the already long list of responsibilities, the Go compiler must perform extra steps to ensure Go memory safety. For instance, the compiler can often tell during compile time that some commands will use a memory space that is safe to use (contains expected data structure and is reserved for our program). However, there are cases when this cannot be determined during compilation, so additional checks have to be done in runtime, e.g. extra bound checks or nil checks. We will discuss this in more detail in "Go Memory Management", but for our conversation about CPU, we need to acknowledge that such checks can take our valuable CPU time. While the Go compiler tries to eliminate those checks when unnecessary (e.g. in the Bound Check Elimination stage during SSA optimizations), there might be cases where we need to write code in a way that helps the compiler eliminate some checks.

There are many different configuration options for the Go build process. The first large batch of options can be passed through `go build -ldflags="<flags>"` which represents linker command options (the `ld` prefix traditionally stands for Linux linker, thus those are linker flags). For example:

- We can omit the mentioned DWARF table, thus reducing the binary size using `-ldflags="-w" ` (recommended for production build if you don't use debuggers there)
- We can further reduce the size with `-ldflags= "-s -w" `, removing DWARF and symbols table with other debug information. I would not recommend the latter option, as non-DWARF elements allow important runtime routines like gathering profiles.

Similarly, `go build -gcflags="<flags>"` represents Go compiler options (`gc` stands for `Go Compiler`, don't confuse it with GC which means garbage collection explained in "Garbage Collection"). For example:

- Afromentioned option `-gcflags="-S"` which prints Go assembly from the source code.
- `-gcflags="-N"` disables all compiler optimizations.
- `-gcflags="-m=<number>` builds the code while printing the main optimization decisions, where the number represents the level of detail. For example, see Example 4-3 for the automatic compiler optimizations made on our Example 4-1 code.

Example 4-3. Output of `go build -gcflags="-m=1" sum.go` on Example 4-1 code.

```
# command-line-arguments
./sum.go:10:27: inlining call to ioutil.ReadFile
./sum.go:15:34: inlining call to bytes.Split
./sum.go:9:10: leaking param: fileName
./sum.go:15:44: ([]byte)("\n") does not escape
./sum.go:16:38: string(line) escapes to heap
```

The compiler will print more details with an increased `-m` number. For example, `-m=3` will explain why certain decisions were made. This option is handy when we expect certain optimization (inlining or keeping variables on the stack) to occur, but we see still see an overhead while benchmarking in our TFBO cycle ("Efficiency-Aware Development Flow").

The Go compiler implementation is highly tested and mature, but there are limitless ways of writing the same functionality. There might be edge cases when our implementation confuses the compiler, so it does not apply certain naive implementations. Benchmarking if there is a problem, profiling the code and confirming with the `-m' option helps. More detailed optimizations can also be printed using further options. For example `-gcflags="-d=ssa/check_bce/debug=1"` prints all bound check elimination optimizations.

THE SIMPLER THE CODE, THE MORE EFFECTIVE COMPILER OPTIMIZATIONS WILL BE.

Too clever code makes it hard to read and maintain programmed functionality. But it also can confuse the compiler that tries to match patterns with their optimized equivalents. Using idiomatic code, keeping your functions and loops straightforward increases the chances the compiler applies the optimizations, so you don't need to!

Knowing compiler internals helps, especially when it comes to more advanced optimizations tricks, which among other things, help compilers to optimize our code. Unfortunately, it also means our optimizations might be a bit fragile regarding portability between different compiler versions. Go team reserves rights to change compiler implementation and flags since they are not part of any specification. This might mean that the way you wrote a function that allows automatic inline by the compiler might not trigger inline in the next version of Go Compiler. This is why it's even more important to benchmark and observe the efficiency of your program closely when you switch to a different Go version.

To sum up, the compilation process has a crucial role in offloading programmers from pretty tedious work. Without compiler optimizations, we would need to write more code to get to the same efficiency level while sacrificing readability and portability. Instead, if you focus on making your code simple, you can trust that the Go compiler will do a good enough job. Yet, if you need to increase efficiency for a particular hot path, it might be beneficial to double-check if the compiler did what you expected. For example, it might be the case that the compiler did not match our code with common optimization; there is some extra memory safety check that the compiler could further eliminate or function that could be inlined but was not. In very extreme cases, there might be even a value to write a dedicated Assembly code and import it from Go code (discussed in [Link to Come]).

Go building process constructs fully executable machine code from our Go source code. The operating system loads machine code to memory and writes the first instruction address to the program counter (PC) register when it needs to be executed. From there, the CPU core can compute each

instruction one by one. At first glance, it might mean that the CPU has a relatively simple job to do. But unfortunately, a memory wall problem causes CPU makers to continuously work on additional hardware optimizations that change how those instructions are executed. Understanding those mechanisms will allow us to control the efficiency and speed of our Go programs even better. Let's uncover this problem in the next section.

CPU and Memory Wall Problem

To understand the memory wall and its consequences, let's dive briefly into CPU core internals. The details and implementation of the CPU core change over time for better efficiency (usually getting more complex), but the fundamentals stay the same. In principle, a Control Unit visible on Figure 4-1 manages reads from memory through various L-caches (from smallest and fastest), decodes program instructions, and coordinates execution of them in the Arithmetic Logic Unit (ALU) and handles interruptions.

An important fact is that the CPU works in cycles. Most CPUs in one cycle can perform one instruction on one set of tiny data. This pattern is called the Single Instruction Single Data (SISD) in characteristics mentioned in Flynn's Taxonomy, and it's the key aspect of Von Neumann architecture. Some CPUs also allow Single Instruction Multiple Data (SIMD)[9] processing with special instructions like SSE, which allows the same arithmetic operation on four floating numbers in one cycle. Unfortunately, those instructions are not straightforward to use in Go, therefore quite rarely seen.

Meanwhile, registers are the fastest local storage available to the CPU core. Because they are small circuits wired directly into ALU, it takes only one CPU cycle to read their data. Unfortunately, there are also only a few of those (depending on CPU, typically 16 for general use), and their size is usually not larger than 64 bits. This means they are used as short-time variables in our program lifetime. Some of the registers can be used for our

machine code. Others are reserved for CPU use. For example, commonly seen PC register holds the address of the next instruction that the CPU should fetch, decode and execute.

Computation is all about the data. As we learned in Chapter 1, there is lots of data nowadays, scattered around different storage mediums—uncomparably more than what's available to store in a single CPU register. Moreover, a single CPU cycle is faster than accessing data from the main memory (RAM). On average, 100 times faster, as we read from our rough napkin math of latencies in [Link to Come] we will use through this book. As discussed in the misconception "Hardware is Getting Faster and Cheaper", technology allows us to create CPU cores with dynamic clock speed, yet the maximum is always around 4GHz. Funny enough, the fact we can't make faster CPU cores is not the most important problem since our CPU cores are already… too fast! It's a fact we cannot make faster memory, which causes the main efficiency issues in CPUs nowadays.

> *We can execute something in the ballpark of 36 billion instructions every second. Unfortunately, most of that time is spent waiting for data. About 50% of the time in almost every application. In some applications upwards of 75% of the time is spent waiting for data rather than executing instructions. If this horrifies you, good. It should.*
>
> —Chandler Carruth, Efficiency with Algorithms, Performance with Data Structures (CppCon 2014)

The problem mentioned above is often referenced as a Memory Wall problem. As a result, we risk wasting dozens, if not hundreds, of CPU cycles per single instruction since fetching that instruction and data (and then saving the results) takes ages.

This problem is so prominent that it has triggered recent discussions to revisit von Neumann's architecture as machine learning (ML) workloads (e.g. neural networks) for artificial intelligence (AI) use are more popular. Those workloads are especially affected by the Memory Wall problem because most of the time is spent performing complex matrix math calculations, which require traversing large amounts of memory[10].

The memory wall problem effectively limits how fast our programs do their job. It also impacts the overall energy efficiency that matters for mobile applications. Nevertheless, it is the best common general-purpose hardware nowadays. Industry mitigated many of those problems by developing a few main CPU optimizations we will discuss below: the hierarchical cache system, pipelining, out-of-order execution, and hyperthreading. Those directly impact our low-level Go code efficiency, especially in terms of how fast our program will be executed.

Hierachical Cache System

All modern CPUs include local, fast, small caches for often-used data. L1, L2, L3 (and sometimes L4) caches are on-chip static random access memory (SRAM) circuits. SRAM uses different technology for storing data faster than our main memory RAM but is much more expensive to use and produce in large capacities (main memory is explained in "Physical Memory"). Therefore, l-caches are touched as the first ones when the CPU needs to fetch instruction or data for an instruction from the main memory (RAM). The way the CPU is using L-caches is presented in Figure 4-3[11]. In the example, we will use a simple CPU instruction `MOVQ` explained in Example 4-2.

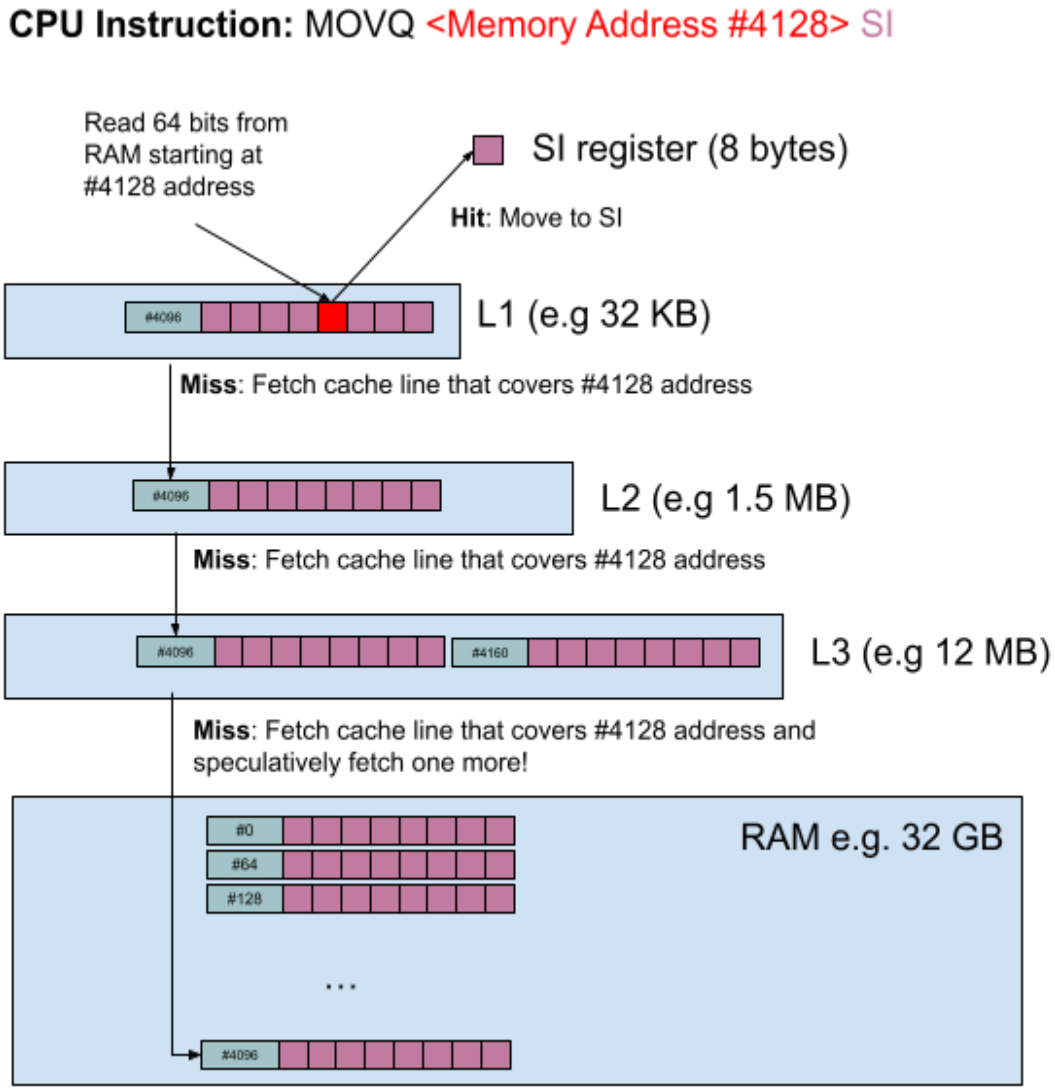


Figure 4-3. The "look up" cache method performed by the CPU to read bytes from the main memory through L-caches.

To copy 64 bits (`MOVQ` command) from a specific memory address to register `SI`, we must access the data which normally resides in the main memory. Since reading from RAM is slow, it uses L-caches to check for data first. The CPU will ask the L1 cache for those bytes on the first try. If the data is not there (cache miss), it visits a larger L2 cache, then the largest cache L3, then eventually main memory (RAM). In any of those misses, the CPU will try to fetch the complete "cache line" (typically 64 bytes, so eight times the size of the register), save it in all caches, and only use those specific bytes.

Reading more bytes at once (cache line) is useful as it takes the same latency as reading a single byte (explained in "Physical Memory"). Statistically, it is also likely that the next operation needs bytes next to the previously accessed area. L-caches partially mitigate the memory latency problem and reduce the overall amount of data to be transferred, preserving memory bandwidth.

The first direct consequence of having L-caches in our CPUs is that the smaller and more aligned the data structure we define (discussed in [Link to Come]), the better for efficiency. Such a structure will have more chances to fit fully in lower level caches and avoid expensive cache misses. The second result is that instructions on sequential data will be faster since cache lines typically would contain multiple items stored next to each other (explored in [Link to Come]).

Pipelining and out-of-order execution

If the data were magically accessible in zero time, we would have a perfect situation where every CPU core cycle performs a meaningful instruction, executing instructions as fast as CPU core speed allows. Since this is not the case, modern CPUs try to keep every part of the CPU core busy using cascading pipelining. In principle, the CPU core can perform many stages required for instruction execution at once in one cycle. This means we can exploit Instruction-Level Parallelism (ILP) to execute, for example, five independent instructions in five CPU cycles, giving us that sweet average of one instruction per cycle (IPC)[12]. For example, in an initial five-stage pipeline system (modern CPUs have 14-24 stages!), a single CPU core computes five instructions at the same time within one cycle as presented in Figure 4-4.

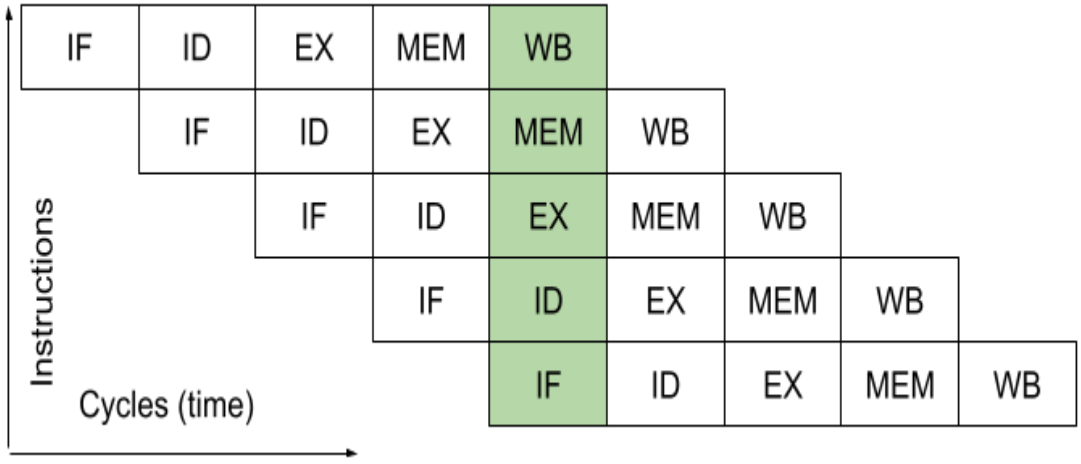


Figure 4-4. Example five-stage pipeline.

Classic five stages pipeline consists of 5 operations:

- IF: Fetch the instruction to execute.
- ID: Decode the instruction.
- EX: Start the execution of the instruction.
- MEM: Fetch the operands for the execution.
- WB: Write back the result of the operation (if any).

To make it even more complex, as we discussed in the L-caches section, it is rarely the case that even the fetch of the data (e.g. `MEM` stage) takes only one cycle. To mitigate this CPU core also employs a technique called out-of-order execution. In this method CPU attempts to schedule instructions in an order governed by the availability of input data and execution unit (if possible) rather than by their original order in the program. This book is enough to think about it as a complex, more dynamic pipeline that utilizes internal queues for more efficient CPU execution.

The resulting pipelined and out-of-order CPU execution is complex, but the above-simplified explanation should be all we need to understand two critical consequences for us developers. The first, trivial one is that every switch of instruction stream has a huge cost (e.g. in latency)[13], because the

pipeline has to reset and start from scratch, on top of the obvious cache trashing. We are not yet mentioning the operating system overhead that must be added on top. We often call this a "context switch", which is inevitable in modern computers since the typical operating systems use preemptive task scheduling. In those systems, the execution flow of the single CPU core can be preempted many times a second, which might matter in extreme cases. We will discuss how to influence such behaviour in "Operating System Scheduler".

The second consequence is that the more predictive our code is, the better. This is because pipelining requires the CPU cores to perform complex branch predictions to find instructions that will be executed after the current one. If our code is full of branches like `if` statements, `switch` cases or jump statements like `continue`, finding even two instructions to execute simultaneously might be impossible, simply because one instruction might decide on what instruction will be done next. This is called data dependency. Modern CPU core implementation goes even further by performing speculative execution. Since it does not know which instruction is next, it picks the most likely one and assumes that such a branch will be chosen. Unnecessary executions on the wrong branches are better than wasted CPU cycles doing nothing. Therefore, many branchless coding techniques emerged, which help the CPU predict branches and might result in faster code. Some methods are applied automatically by Go compiler, but sometimes, manual improvements have to be added.

Generally speaking, the simpler the code, with fewer nested conditionals and loops, the better for branch predictor (similarly, it's simpler than for the compiler to apply optimizations as mentioned in "Understanding Go Compiler"). This is why we often hear that the code that leans to the left is faster.

In my experience repeatedly [I saw] that code that wants to be fast, go to the left of the page. So if you [write] like a loop and the if, and the for and a switch, it's not going to be fast. By the way, the Linux kernel, do you know what the coding standard is? Eight characters tab, 80 characters line width. You can't write bad code in the Linux kernel. You can't write slow code there. (...) The moment you have too many ifs and decision points (...) in your code, the efficiency is out of the window.

—Andrei Alexandrescu, Speed is Found in The Minds of People (CppCon 2019)

The existence of branch predictors and speculative approaches in CPU has another consequence. It causes contiguous memory data structures to perform much better in pipelined CPU architecture with L-caches.

> **CONTIGUOUS MEMORY STRUCTURE MATTERS.**
>
> Practically speaking, on modern CPUs, developers in most cases should prefer contiguous memory data structures like arrays instead of linked lists in their programs. This is because typical linked-like list implementation (e.g., a tree) uses memory pointers to the next, past, child or parent elements. This means that when iterating over such a structure, the CPU core can't tell what data and what instruction we will do next until we visit the node and check that pointer. This effectively limits the speculation capabilities, causing in-efficient CPU usage.

We will discuss specific optimizations leveraging the knowledge of CPU pipelining in [Link to Come].

Hyper-threading

Hyper-threading is Intel's proprietary name for the CPU optimization technique called simultaneous multithreading (SMT)[14]. Other CPU makers implement SMT too. This method allows a single CPU core to operate in a mode visible to programs and operating systems as two logical CPU cores[15]. SMT prompts the operating system to schedule two threads onto the same physical CPU core. While a single physical core will never execute more than one instruction at a time, more instructions in the queue

help make the CPU core busy during idle times. Given the memory access wait times, this can utilize a single CPU core more without impacting the latency of process execution. In addition, extra registers in SMT enabled CPUs to allow for quicker context switches between multiple threads running on a single physical core.

SMT has to be supported and integrated with the operating system. You should see twice as many cores as physical ones in your machine when enabled. To understand if your CPU supports Hyper-threading, check the "thread(s) per core" information in the specifications. For example, using the `lscpu` Linux command on Example 4-4, my CPU has two, meaning hyper-threading is available.

Example 4-4. Output of `lscpu` command on my Linux laptop.

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          12
On-line CPU(s) list:             0-11
Thread(s) per core:              2
Core(s) per socket:              6
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           158
Model name:                      Intel(R) Core(TM) i7-9850H CPU @
2.60GHz
CPU MHz:                         2600.000
CPU max MHz:                     4600.0000
CPU min MHz:                     800.0000
```

The SMT is usually enabled by default but can be turned on demand on newer kernels. This poses one consequence when running our Go programs. We can usually choose if we should enable or disable this mechanism for our processes. But should we? In most cases, it is better to keep it enabled for our Go programs as it allows us to fully utilize physical cores when running multiple different tasks on a single computer. Yet, in some extreme cases, it might be worth dedicating full physical core to a single process to

ensure the highest quality of service. We will experiment with this in [Link to Come], but generally, a benchmark on each specific hardware should tell us.

All the above CPU optimizations and the corresponding programming techniques utilizing that knowledge tend to be used only at the very end of the optimization cycle and only when we want to squeeze the last dozen of nanoseconds on the critical path. While exciting, those techniques are not needed in the majority of practical cases[16]. Yet we can derive more practical lessons from that knowledge, something we can apply to our every day Go programming to achieve better efficiency and latency of our code before diving into low-level optimizations.

THREE PRINCIPLES OF WRITING A CPU EFFICIENT CODE.

- Use algorithms that do less work.
- Focus on writing low-complexity code that will be easier to optimize for compiler and CPU branch predictors. Ideally, separate “hot” from “cold” code.
- Prefer contiguous memory data structures.

With this brief understanding of CPU hardware dynamics, let’s dive deeper into the essential software types that allow us to run thousands of programs simultaneously on shared hardware—schedulers.

Schedulers

Scheduling generally means allocating necessary, usually limited, resources for a certain process to finish. For example, assembling the car parts must be tightly scheduled in a certain place at a certain time in a car factory to avoid downtime. We might also need to schedule a meeting among certain attendees with only certain time slots of the day free.

In modern computers or clusters of servers, we have thousands of programs that have to be running on shared resources like CPU, memory, network, disks etc. That's why the industry developed many types of scheduling software (commonly called schedulers) focused on allocating those programs to free resources on many levels.

In this section, we will discuss CPU scheduling. Starting from the bottom level, we have an operating system that schedules arbitrary programs on a limited number of physical CPUs. Operating System mechanisms should tell us how multiple programs running simultaneously can impact our CPU resources and, in effect, our own Go program execution latency. It will also help us understand how a developer can utilize multiple CPU cores simultaneously, in parallel or concurrently, to achieve faster execution.

Operating System Scheduler

As with compilers, there are many different Operating Systems (OS-es), each with different task scheduling and resource management logic. While most of the systems operate on similar abstractions (e.g. threads, processes with priorities), we will focus on the Linux Operating System in this book. Its core, called the kernel, has many important functionalities, like managing memory, devices, network access, security and more. It also ensures program execution using a configurable component called a scheduler.

> *As a central part of resource management, the OS thread scheduler must maintain the following, simple, invariant: make sure that ready threads are scheduled on available cores.*
>
> —The Linux Scheduler: a Decade of Wasted Cores (2016)

The smallest scheduling unit for the Linux Scheduler is called an OS thread. The thread (sometimes also referred to as "task" or "lightweight process") contains an independent set of machine code in the form of CPU instructions designed to run sequentially. While threads can maintain their execution state, stack and register set, they cannot run out of context.

Each thread runs as a part of the process. The process represents a program in execution and can be identified by its Process Identification Number (PID). When we tell Linux OS to execute our compiled program, a new process is created (for example, when fork system call is used). The process creation includes the assignment of a new PID, the creation of the initial thread with its machine code (our `func main()` in Go code) and stack, files for standard outputs and input and tons of other data (e.g. list of open file descriptors, statistics, limits, attributes, mounted items, groups, etc.). On top of that, a new memory address space is created, which has to be protected from other processes. All that information is maintained under the dedicated directory `/proc/<PID>` for the duration of the program execution.

Thread can create new threads (e.g. using clone syscall), which will have independent machine code sequence but will share the same memory address space. Threads can also create new processes (e.g. using fork) that will run in isolation and execute the desired program. Threads maintain their execution state: Running, Ready, and Blocked. Possible transformations of those states are presented in Figure 4-5.

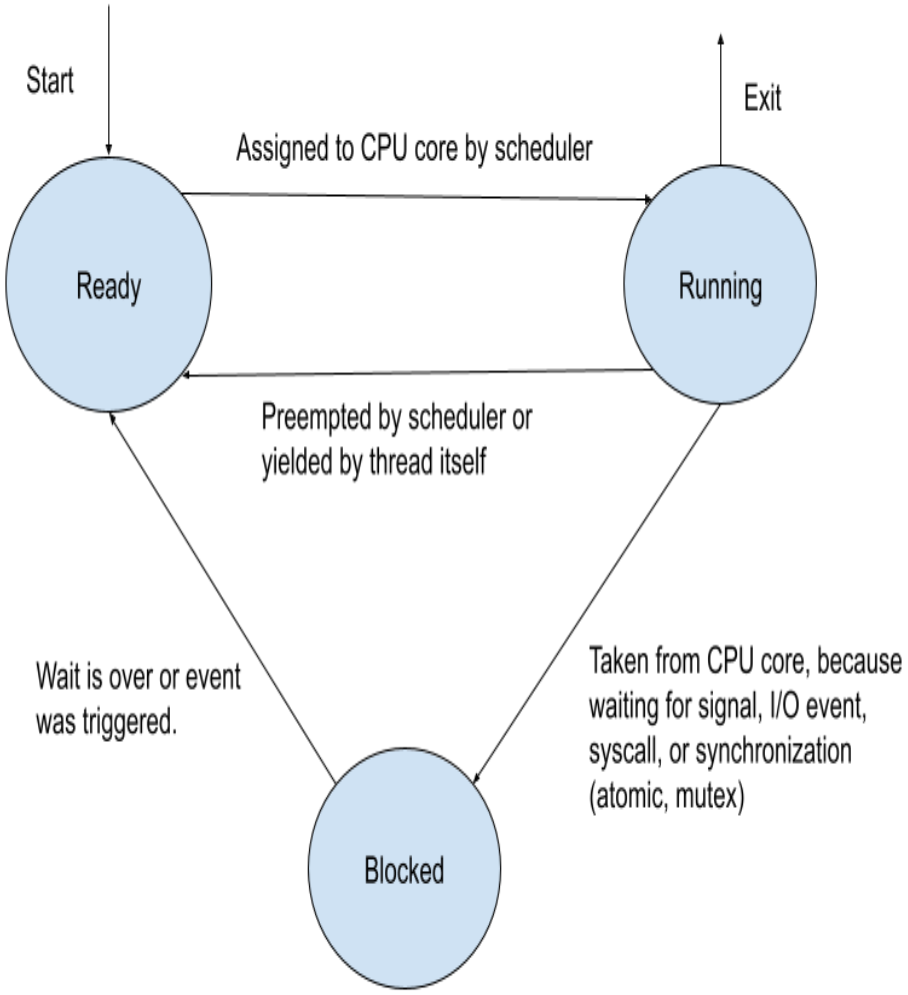


Figure 4-5. Thread states as seen by the Linux OS Scheduler.

Thread state tells the scheduler what thread is doing at the moment:

Running

Thread is assigned to the CPU core and doing its job.

Blocked

> Thread is waiting on some event that potentially takes longer than a context switch. For example, a thread reads from a network connection and is waiting for a packet or its turn on mutex lock. This is an opportunity for the scheduler to step in and allow other threads to run.

Ready

> Thread is ready for execution but is waiting for its turn.

As you might already notice, the Linux Scheduler does a preemptive type of thread scheduling. Preemptive means the scheduler can freeze a thread execution at any time. In modern OS, we always have more threads to be executed than available CPU cores, so the scheduler must run multiple "ready" threads on a single CPU core. The thread is preempted every time it waits for an I/O request or other events. The thread can also tell the operating system to yield itself (e.g. using sched_yield syscall). When preempted, it enters a "blocked" state, and another thread can take its place in the meantime.

The naive scheduling algorithm could wait for the thread to preempt itself. This would work great for I/O bound threads which are often in the "Blocked" state. For example, interactive systems with graphical interfaces or lightweight web servers working with network calls. But what if the thread is CPU bound, which means it spends most of its time using only CPU and memory? For example, doing some computation heavy jobs like linear search, multiplying matrixes or brute-forcing a hashed password. In such cases, the CPU core could be busy on one task for minutes, which will result in starving all other threads in the system. For example, imagine not being able to type in your browser or resize a window for a minute—it would look like a long system freeze!

This primary Linux scheduler implementation addresses that problem. It is called a Completely Fair Scheduler (CFS), and it assigns threads in short turns. Each thread is given a certain slice of the CPU time, typically

something up to time between 1 ms and 20 ms, which creates the illusion that threads are running simultaneously. It especially helped desktop systems, which must be responsive to human interactions. There are a few other important consequences of that design:

- The more threads want to be executed, the less time they will have in each turn. However, this can result in lower productive utilization of the CPU core, which starts to spend more time on expensive context switches.
- On the overloaded machine, each thread has shorter turns on the CPU core and can also end up having fewer turns per second. While none of the threads is completely starved (blocked), their execution can significantly slow down.

CPU OVERLOADING

Writing CPU efficient code means our program wastes significantly less amount of CPU cycles. Of course, this is always great, but the efficient implementation might be still doing its job very slowly if the CPU is overloaded.

An overloaded CPU or system means too many threads competing for the available CPU cores. As a result, the machine might be overscheduled, or a process or two spawns too many threads to perform some heavy task (we call this situation a noisy neighbour). If an overloaded CPU situation occurs, checking the machine CPU utilization metric (discussed in [Link to Come]) should show us CPU cores running at 100% capacity. Every thread will be executed slower in such a case, resulting in a frozen system, timeouts and lack of responsiveness.

CPU THROTTLING

A bit off-topic to the Linux Scheduler, the CPU throttling can also affect the effective execution of our software. Throttling happens when the CPU runs in energy efficiency mode or when the chip temperature is too high. This typically happens on PCs and laptops, prone to environmental circumstances, like harsh surrounding conditions or running on battery. Throttling happens much less in a server environment (clouds or datacenters) since it typically runs in special facilities focused on sustainable CPU speeds, stable energy provisioning, and cooling methods. The effect of CPU throttling is that your program also runs slower, as the CPU cores are slowing their clocks, despite there being enough CPU capacity in the system.

The load on the CPU and throttling is cirtical when we try to assess our program latency after optimization through benchmarking, e.g. in our TFBO in "Efficiency-Aware Development Flow". Especially when doing this on your developer machine (e.g. laptop), the same program can often run ten times slower or faster depending on the environment or your machine load, leading to wrong conclusions. We will discuss mitigations for this in [Link to Come].

- It is hard to rely on pure program execution latency (sometimes referred to as "wall time" or "wall-clock time") to estimate our program CPU efficiency. This is because modern OS schedulers are preemptive, and the program often waits for other I/O or synchronizations. As a result, it's pretty hard to reliably check if, after a fix, our program utilizes the CPU better than the previous implementation. This is why the industry defined the important metric to gather how long our program's process (all its threads) spent in the "Running" state on all CPU cores. It's called CPU time and can be represented as the number of clock ticks or seconds spent in the CPU cores since the process started. Sometimes it's also presented as process CPU utilization, so percentage CPU usage in a time window (e.g. collection cycle). For example, 120% CPU utilization means that our process used 1 and 1/5 of the single CPU core in a given time window, which tells us that the process has at least two threads scheduled on two different cores.

CPU TIME ON AN OVERLOADED MACHINE

Measuring CPU time is a great way to check our program's CPU efficiency. However, be careful when looking at the CPU time from some narrow window of process execution time. For example, lower CPU time might mean our process was not using much CPU during that moment, but it might also represent the situation of the overloaded CPU.

Overall, sharing processes on the same system has its problems. That's why in virtualized environments, we tend to reserve those resources (e.g. up to 200 milliseconds of CPU time)

- The final consequence of the CFS design is that it is too fair to ensure dedicated CPU time for a single thread. Linux scheduler has priorities, a user-configurable "niceness" flag and different scheduling policies. Modern Linux OS has even a scheduling policy that uses a special real-time scheduler in place of CFS for threads that need to be executed in the first order[17]. Unfortunately, even with a real-time scheduler, a Linux system cannot ensure that higher priority threads will have all the CPU time they need, as still, it will try to ensure low priority threads are not starved. Furthermore, because both CFS and real-time counterparts are preemptive, they are not deterministic and predictive. As a result, any task with hard real-time requirements (e.g. millisecond trading or airplane software) can't be guaranteed enough execution time before its deadline. This is why some companies develop their own schedulers or systems for strict real-time programs like Zephyr OS.

Despite the somewhat complex characteristics of the CFS scheduler, it remains the most popular thread orchestration system available in modern Linux systems. In 2016 the CFS was also overhauled for multi-core machines and NUMA architectures, based on findings from the famous research paper. As a result, threads are now smartly distributed across idle cores while ensuring migrations are not done too often and not among threads sharing the same resources.

With the basic understanding of OS scheduler, let's dive into why the Go scheduler exists and how it enables developers to program multiple tasks to run concurrently on single or multiple CPU cores.

Go Runtime Scheduler

The Go concurrency framework is built on the premise that it's hard for a single flow of CPU instructions (e.g. function) to utilize all CPU cycles due to the I/O bound nature of the typical workflow. While OS thread abstraction mitigates this by multiplexing threads into a set of CPU cores, Go language brings another layer—a "goroutine"--that multiplexes functions on top of a set of threads. The idea for goroutines is similar to coroutines, but since it is not the same (goroutines can be preempted) and since it's in Go language, it has the `go` prefix. Similarly to the OS thread, when the goroutine is blocked on a system call or I/O, the Go scheduler (not OS!) can quickly switch to a different goroutine which will resume on the same thread (or different if needed).

> *Essentially, Go has turned I/O-bound work [on the application level] into CPU-bound work at the OS level. Since all the context switching is happening at the application level, we don't lose the same 12k instructions (on average) per context switch that we were losing when using threads. In Go, those same context switches are costing you 200 nanoseconds or 2.4k instructions. The scheduler is also helping with gains on cache-line efficiencies and NUMA. This is why we don't need more threads than we have virtual cores.*
>
> —William Kennedy, Scheduling In Go : Part II - Go Scheduler

As a result, we have in Go very cheap execution "threads" in userspace (a new goroutine only allocates a few kilobytes for the initial, local stack), which reduces the number of competing threads in our machine and allows hundreds of goroutines in our program without extensive overhead. Just one OS thread per CPU core should be enough to get all work in our goroutines done[18]. This enables many readability patterns like event loops, map-

reduce, pipes, iterators or more without involving more expensive kernel multi-threading.

> **TIP**
>
> Using Go concurrency in the form of goroutines is an excellent way to:
>
> - Represent complex asynchronous abstractions (e.g. events).
> - Utilize our CPU to the fullest for I/O bound tasks.
> - Create a multi-threaded application that can utilize multiple CPUs to execute faster.

Starting another goroutine is very easy in Go. It is built in the language via a `go <func>()` syntax. See an example `function` that starts two goroutines and finishes its work in Example 4-5.

Example 4-5. A function that starts two goroutines.

```
func anotherFunction(arg1 string) { /*...*/ }

func function() {
   // ... ❶

   go func() {
      // ... ❷
   }()

   go anotherFunction("argument1") ❸

   return ❹
}
```

❶ The scope of the current goroutine.

❷ The scope of a new goroutine that will run concurrently any moment now.

❸ The `anotherFunction` will start running concurrently any moment now.

❹ When `function` terminates, two goroutines we started can still run.

It's important to remember that all goroutines have a flat hierarchy between each other. Technically, there is no difference when goroutine `A` started `B` or `B` started `A`. In both cases, both `A` and `B` goroutines are equal, and they don't

know about each other[19]. They also cannot stop each other unless we implement explicit communication or synchronization and "ask" the goroutine to shut down. The only exception is the main goroutine that starts with the `main()` function. If the main goroutine finishes, the whole program terminates, killing all other goroutines forcefully.

Regarding communication, goroutines, similarly to OS threads, have access to the same memory space within the process. This means that naturally, we can pass data between goroutines using shared memory. However, this is not so trivial because almost no operation in Go is atomic. This means that concurrent writing (or writing and reading) from the same memory can cause data races, leading to non-deterministic behaviour or even data corruption. To solve this, we need to use synchronization techniques like explicit atomic function (as presented in Example 4-6 or mutex (as shown in Example 4-7), so in other words, a lock.

Example 4-6. Safe multi-goroutine communication through dedicated atomic addition.

```
func sharingWithAtomic() (sum int64) {
   var wg sync.WaitGroup ❶

   concurrentFn := func() {
      atomic.AddInt64(&sum, randInt64())
      wg.Done()
   }
   wg.Add(3)
   go concurrentFn()
   go concurrentFn()
   go concurrentFn()

   wg.Wait()
   return sum
}
```

❶ Notice that while we use atomic to synchronize additions between `concurrentFn` goroutines, we use additional `sync.WaitGroup` (another form of locking) to wait for all those goroutines to finish. We do the same in the next example too.

Example 4-7. Safe multi-goroutine communication through mutex (lock).

```
func sharingWithMutex() (sum int64) {
   var wg sync.WaitGroup
   var mu sync.Mutex

   concurrentFn := func() {
      mu.Lock()
      sum += randInt64()
      mu.Unlock()
      wg.Done()
   }
   wg.Add(3)
   go concurrentFn()
   go concurrentFn()
   go concurrentFn()

   wg.Wait()
   return sum
}
```

The choice between atomic and lock depends on readability, efficiency requirements and what operation you want to synchronize. For example, if you want to concurrently perform a simple operation on a number like value write or read, addition, substitution or compare and swap, you can consider atomic package. Atomic is often more efficient than mutexes (lock) since the compiler will translate them into special atomic CPU operations that can change data under a single memory address in thread-safe way[20].

If, however, using atomic impacts the readability of our code, the code is not on a critical path, or we have a more complex operation to synchronize, we can use a lock. Go offers `sync.Mutex` that allows simple locking and `sync.RWMutex`, which allows locking for reads (`RLock()`) and for writes (`Lock()`). If you have many goroutines that do not modify shared memory, lock them with `RLock()` so there is no lock contention between those since concurrent read of shared memory is safe. Only when a goroutine wants to modify that memory, it can acquire a full lock using `Lock()` that will block all readers.

On the other hand, `lock` and `atomic` is not the only choice. Go language has another ace in its hand on this subject. On top of the coroutine concept,

Go also utilizes the C. A. R. Hoare's Communicating Sequential Process (CSP) paradigm can also be seen as a type-safe generalization of Unix pipes.

> *Do not communicate by sharing memory; instead, share memory by communicating.*
>
> —Effective Go

This model encourages sharing data by implementing a communication pipeline between goroutines using a channel concept. Sharing the same memory address to pass some data requires extra synchronization. However, suppose one goroutine sends that data to some channel, and another receives it. In that case, the whole flow naturally synchronizes itself and shared data is never accessed by two goroutines simultaneously, ensuring thread safety[21]. Example channel communication is presented in Example 4-8.

Example 4-8. An example of memory safe multi-goroutine communication through the channel.

```
func sharingWithChannel() (sum int64) {
   result := make(chan int64) ❶

   concurrentFn := func() {
      // ...
      result <- randInt64() ❷
   }
   go concurrentFn()
   go concurrentFn()
   go concurrentFn()

   for i := 0; i < 3; i++ { ❸
      sum += <-result ❹
   }
   close(result) ❺
   return sum
}
```

❶ Channel can be created in go with `ch := make(chan <type>, <buffer size>)` syntax.

❷ We can send values of a given type to our channel.

❸ Notice that in this example, we don't need `sync.WaitGroup` since we abuse the knowledge of how many exact messages we expect to

receive. If we did not have that information, we would need a waiting group or another mechanism.

4. We can read values of a given type from our channel.
5. Channels should also be closed if we don't plan to send anything through them anymore. This releases resources and unblocks certain receiving and sending flows (more on that later).

The important aspect of channels is that they can be buffered. In such a case, it behaves like a queue. If we create a channel with, e.g. buffer of three elements, a sending goroutine can send exactly three elements before it gets blocked until someone reads from this channel. If we send three elements and close the channel, receiving goroutine can still read three elements before noticing the channel was closed. A channel can be in three states. It's important to remember how the goroutine sending or receiving from this channel behaves when switching between those states.

Allocated, open channel

If we create a channel using `make(chan <type>)`, it's allocated and open from the start. Assuming no buffer, such a channel will block an attempt to send a value until another goroutine receives it or when we use the `select` statement with multiple cases. Similarly, the channel receive will block until someone sends to that channel unless we receive in a `select` statement with multiple cases or the channel was closed.

Closed channel

If we `close(ch)` allocated channel, a send to such channel will cause panic and receives will return zero value immediately. This is why it is recommended to keep responsibility for the closing channel in the goroutine that sends the data (sender).

Nil channel

If you define channel type (`var ch chan <type>`) without allocating it using `make(chan <type>)`, our channel is nil. We can also "nil" an allocated channel by assigning nil (`ch = nil`). In this

state, sending and receiving will block forever. Practically speaking, it's rarely useful to nil channels.

Go channels is an amazing and elegant paradigm that allows for building very readable, event-based concurrency patterns. However, in terms of CPU efficiency, they might be the least efficient compared to the `atomic` package and mutexes. Let that not discourage you! For most practical applications (if not overused!), channels can structure our application into robust and efficient concurrent implementation. We will explore some practical patterns of using channels in "Practical Example of Concurrency in Go".

Before we finish this section, it's important to understand how we can tune concurrency efficiency in the Go program. Concurrency logic is implemented by the Go scheduler in Go runtime package, which is also responsible for other things like "Garbage Collection", profiles or stack framing. The Go scheduler is pretty automatic. There are not many configuration flags. As it stands at the current moment, there are two practical ways developers can control concurrency in their code[22]:

A number of goroutines

As developers, we usually control how many goroutines we create in our program. Spawning them for every small workpiece is usually not the best idea, so don't overuse them. It's also worth noting that many abstractions from standard or 3rd party libraries can spawn goroutines, especially those that require `Close` or cancellation. Notably, common operations like `http.Do`, `context.WithCancel` and `time.After` create goroutines. If used wrongly, those goroutines can be easily leaked (leaving orphan goroutines), which typically wastes memory and CPU effort. We will explore ways to debug numbers and snapshots of goroutines in [Link to Come]

FIRST RULE OF THE EFFICIENT CODE

Always close or release the resources you use. Sometimes simple structures can cause colossal and unbounded waste of memory and goroutines if we forget to close those. We will explore common examples in [Link to Come].

GOMAXPROCS

This important environmental variable can be set to control the number of virtual CPUs you want to leverage in your Go program. The same configuration value can be applied via the `runtime.GOMAXPROCS(n)` function. This underlying logic on how the Go scheduler uses this variable is fairly complex[23], but it generally controls how many parallel OS thread executions Go can expect (internally called a "proc" number). Go scheduler will then maintain GOMAXPROCS/proc number of queues and try to distribute goroutines among those. The default value of GOMAXPROCS is always the number of virtual CPU cores your OS exposes, and that is typically what will give you the best performance. Trim GOMAXPROCS value down if you want your Go program to use fewer CPU cores (less parallelism) in exchange for potentially higher latency. We will explore the practical effect of the GOMAXPROCS setting in "Practical Example of Concurrency in Go".

RECOMMENDED GOMAXPROCS CONFIGURATION

Set GOMAXPROCS to the number of virtual cores you want your Go program to utilize at once. Typically we want to use the whole machine; thus, the default value should work.

For virtualized environments, especially using lightweight virtualization mechanisms like containers, use Uber's `automaxprocs` library, which will adjust GOMAXPROCS based on Linux CPU limits the container is allowed to use, which is often what we want.

Multi-tasking is always a tricky concept to introduce into language. I believe the goroutines with channels in Go are quite an elegant solution to this problem, which allows many readable programming patterns without sacrificing efficiency. Let's now explore practical concurrency for our efficiency aspect. Can we improve the latency of our Example 4-1 presented in this chapter by introducing concurrency? Let's dive into this in the next section.

Practical Example of Concurrency in Go

As with any efficiency optimization, the same classic rules apply when transforming a single goroutine code to a concurrent one. No exceptions here. We have to focus on the goal, apply the TFBO loop, benchmark early and look for the biggest bottleneck. As with everything, adding concurrency has trade-offs, and there are cases where we should avoid it. Let's summarize the practical benefits and disadvantages of concurrent code vs sequential.

Advantages

- Concurrency allows us to speed up the work by splitting it into pieces and executing each part concurrently. As long as the synchronization and shared resources are not a significant bottleneck, we should expect an improved latency.
- Because the Go scheduler implements an efficient preemptive mechanism, concurrency improved better CPU core utilization for I/O bound tasks, which should translate into lower latency, even with a `GOMAXPROCS=1` (so single CPU core).
- Especially in virtual environments, we often reserve a certain CPU time for our programs. Concurrency allows us to distribute work across available CPU time in a more even way.
- For some cases like asynchronous programming and event handling, concurrency represents well a problem domain, resulting in

improved readability despite some complexities. Another example is the HTTP server. Treating each HTTP incoming request as a separate goroutine not only allows efficient CPU core utilization but also naturally fits into how code should be read and understood.

Disadvantages

- Concurrency adds significant complexity to the code, especially when we transform existing code into concurrency (instead of building API around channels from day one). This hits readability since it almost always obfuscates execution flow, but even worse, it limits the developer's ability to predict all edge cases and potential bugs. This is one of the main reasons why I recommend postponing adding concurrency as long as possible. And once you have to introduce concurrency, use as few channels as possible for the given problem.
- With concurrency, there is a risk of saturating resources due to unbounded concurrency (uncontrolled amount of goroutines in a single moment) or leaking goroutines (orphan goroutines). This is something we need to care about additionally and test against (more on this later, in [Link to Come] and in [Link to Come]).
- Despite Go's very efficient concurrency framework, goroutines and channels are not free of overhead. If used wrongly, it can impact our code efficiency. Focus on providing enough work to each goroutine that will justify its cost. Benchmarks are a must-have.
- When using concurrency, we suddenly add three more non-trivial tuning parameters into our program. We have a GOMAXPROCS setting, and depending on how we implement things, we can control the number of goroutines we spawn and how large a buffer of the channel we should have. Finding correct numbers require hours of benchmarking and are still prone to errors.

- Concurrent code is hard to benchmark because it depends even more on the environment, possible noisy neighbours, multi-core settings, OS version and so on. On the other hand, sequential, single-core code has much more deterministic and portable performance, which is easier to prove and compare against.

As we see, using concurrency is not the cure for all performance problems. It's just another tool in our hands that we can use to fulfil our efficiency goals. We mentioned the clear need for concurrency to employ asynchronous programming or event handling in our code. We talked about relatively easy gains where our Go program does a lot of I/O operations. However, in this section, I would love to show you how to improve the speed of our Example 4-1 code using concurrency with two typical pitfalls.

As you might have noticed, the majority of the Example 4-1 code (after all bytes from the file are buffered) is CPU-bound—we work through in-memory content, split it into lines, parse and add a number in each line. For such a CPU-bound program, it will keep the OS thread busy as long as possible, so any additional goroutine within a single OS thread and CPU core will make things only worse. In this type of CPU-bound task, the only advantage in latency we could see from concurrency is when we will try to split the work and execute it on more than one CPU core, limiting the synchronization effort to a minimum. Let's explore three approaches we could do to optimize Example 4-1 with concurrency.

ADDING CONCURRENCY SHOULD BE ONE OF OUR LAST DELIBERATE OPTIMIZATIONS ON OUR LIST TO TRY.

As per our TFBO cycle, if you are still not meeting your RAERs, e.g. in terms of speed, make sure you try more straightforward optimization techniques (e.g. those from [Link to Come] before adding concurrency. The rule of thumb is to think about concurrency when our CPU profiler (explained in [Link to Come]) shows that our program spends CPU time only on things that are crucial to our functionality. Ideally, before we hit our readability limit, in the most efficient way we know.

The mentioned list of disadvantages is one reason, but the second is that our program's characteristics might differ after optimizations. For example, we thought our task was CPU-bound, but after improvements, we may find the majority of time is now spent waiting on I/O. Or we might realize we did not need those heavy concurrency changes after.

Let's try to add concurrency to our Example 4-1 code[24]. First, we have to benchmark our basic solution. (we will discuss benchmarking in detail in [Link to Come]). After measurements, I know that on my machine with four virtual CPU cores, it takes on average 78.4ms per operation on our test file with millions of numbers. This will be our baseline. CPU profiles show we spend the majority of time on `bytes.Split` and `strconv.ParseInt`. Can we calculate our sum faster if we do some of this work concurrently?

NOTE

The following examples are not fully optimized using simpler methods. Particularly, we can easily optimize both the `bytes.Split` and `strconv.ParseInt` functions. However, I decided to ignore those and focus our train of thoughts on concurrency patterns for clarity.

In [Link to Come], we will explore how to optimize Example 4-1 to the fullest by improving both of those known bottlenecks and adding concurrency for the best-combined effect.

The first thing we have to do is find computations we can do independently at the same time—computations that do not affect each other. Thanks to the

fact that the sum is commutative, it does not matter in what order numbers are added. Does it mean that naive, concurrent implementation could parse the integer from the string and add the result atomically to the shared variable? Let's explore this, rather simple solution, in Example 4-9.

Example 4-9. Naive concurrent implementation of Example 4-1 that spins a new goroutine for each line to compute.

```
func ConcurrentSum1(fileName string) (ret int64, _ error) {
   b, err := ioutil.ReadFile(fileName)
   if err != nil {
      return 0, err
   }

   var wg sync.WaitGroup
   for _, line := range bytes.Split(b, []byte("\n")) {
      wg.Add(1)
      go func(line []byte) {
         defer wg.Done()
         num, err := strconv.ParseInt(string(line), 10, 64)
         if err != nil {
            // TODO(bwplotka): Return err using other channel.
            return
         }
         atomic.AddInt64(&ret, num)
      }(line)
   }

   wg.Wait()
   return ret, nil
}
```

After unit tests, we can confirm the above implementation provides correct results. Benchmarks, however, show worse news. It takes about 303 milliseconds for Example 4-9 to perform single operation! Almost 4x more than the simpler, non-concurrent version in Example 4-1. Let's profile the CPU during the execution of our program. In that case, the flame graph shows clearly the impact of goroutine and scheduling overhead, presented in Figure 4-6 (see blocks called `runtime.schedule` and `runtime.newproc`).

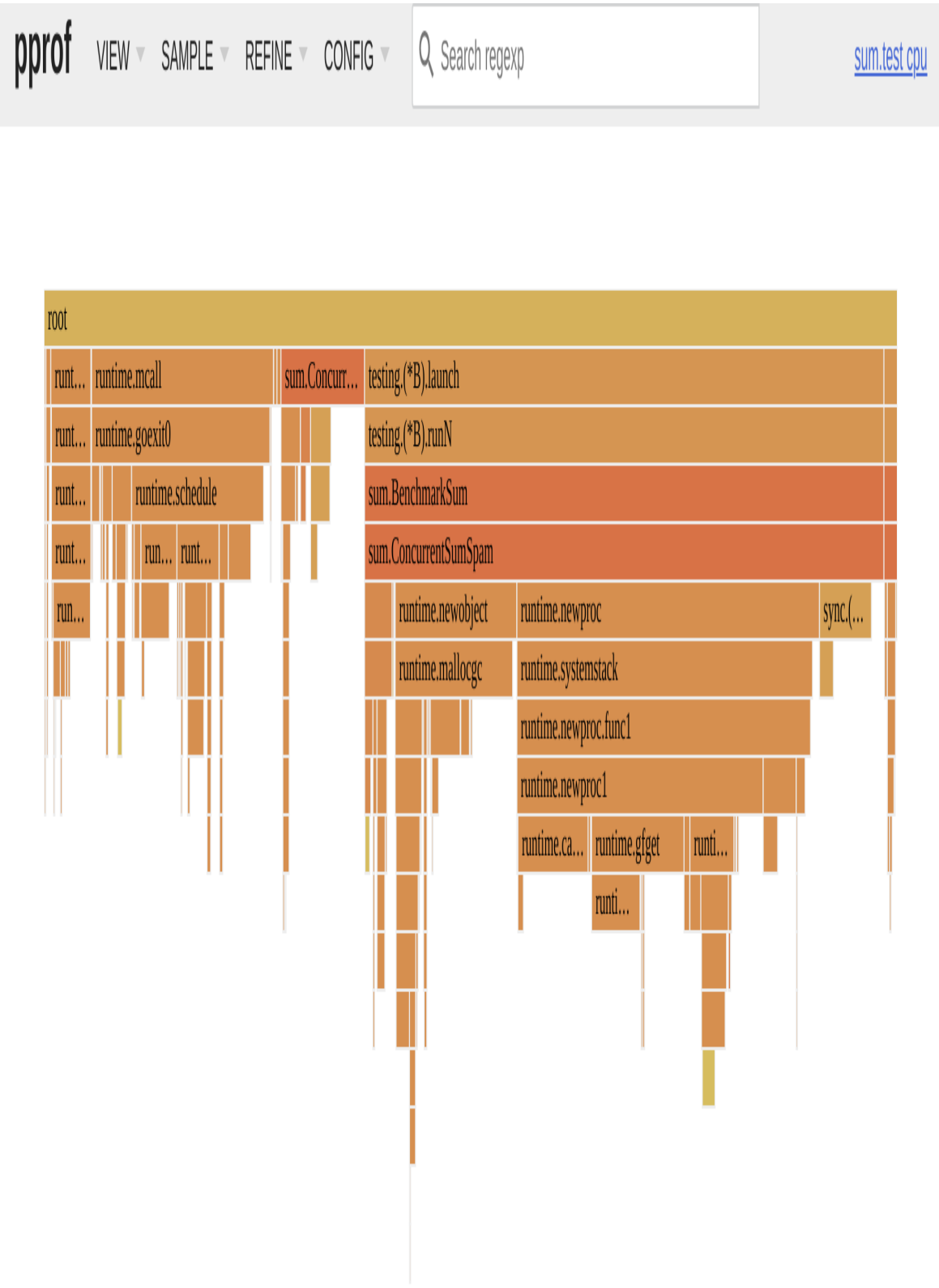


Figure 4-6. CPU profiler flame graph for Example 4-9 execution.

There are three main reasons why Example 4-9 is too naive and not recommended for our case:

1. The work we do concurrently (just parsing, really) is too fast to justify the goroutine overhead (both in memory and CPU usage). This explains the flame graph and the four times slower execution.
2. For larger datasets, we create millions of goroutines. While goroutines are relatively cheap and we can have hundreds of them, there is always a limit to that. So you can imagine the delay of the scheduler that tries to schedule millions of goroutines on only four CPU cores fairly.
3. Our program will have non-deterministic performance depending on the number of lines in the file. Plus, we potentially can hit a problem of unbounded concurrency since we will spam as many goroutines as the external file has lines (something outside of our program control).

That is not what we want, so let's improve our concurrent implementation. There are many ways we could go from here, but let's try to address all three problems we noticed.

We can solve problem number one by assigning more work to each goroutine. We can do that thanks to the fact that another property of addition is associativeness. We can essentially group work into multiple lines, parse and add numbers in each goroutine, and add partial results to the total sum. Doing that automatically helps with problem number two. Grouping work means we will schedule fewer goroutines. The question is, what is the best number of lines in a group? Two? Four? Hundred?

The answer most likely depends on the number of goroutines we want in our process and the number of CPUs available. There is also problem number three—unbounded concurrency. The typical solution here is to use a worker pattern (sometimes called goroutine pooling). In this pattern, we agree on a number of goroutines upfront, and we schedule all of them at once. Then we can create another goroutine that will distribute the work evenly. Let's see example implementation of that algorithm in Example 4-10. Can you predict if this implementation will be faster?

Example 4-10. Concurrent implementation of Example 4-1 that maintains a finite set of goroutines that computes a group of lines. Lines are distributed inefficiently using another goroutine.

```
func ConcurrentSum2(fileName string, workers int) (ret int64, _
error) {
   b, err := ioutil.ReadFile(fileName)
   if err != nil {
      return 0, err
   }

   var (
      wg     = sync.WaitGroup{}
      workCh = make(chan []byte, 10)
   )

   wg.Add(workers + 1)
   go func() {
      for _, line := range bytes.Split(b, []byte("\n")) {
         workCh <- line
      }
      close(workCh) ❶
      wg.Done()
   }()

   for i := 0; i < workers; i++ {
      go func() {
         var sum int64
         for line := range workCh {
            num, err := strconv.ParseInt(string(line), 10, 64)
            if err != nil {
               // TODO(bwplotka): Return err using other channel.
               continue
            }
            sum += num
         }
         atomic.AddInt64(&ret, sum)
         wg.Done()
      }()
   }
   wg.Wait()
   return ret, nil
}
```

❶ Remember, it's the sender that usually is responsible for the closing channel. Even if our flow does not depend on it, it's a good practice to always close channels after use.

Tests pass, so we can start benchmarking. Unfortunately, this implementation with eight goroutines takes 120ms to complete a single operation on average. Still almost twice slower than simpler, sequential Example 4-1. What's wrong? Let's investigate the CPU profile presented in Figure 4-7.

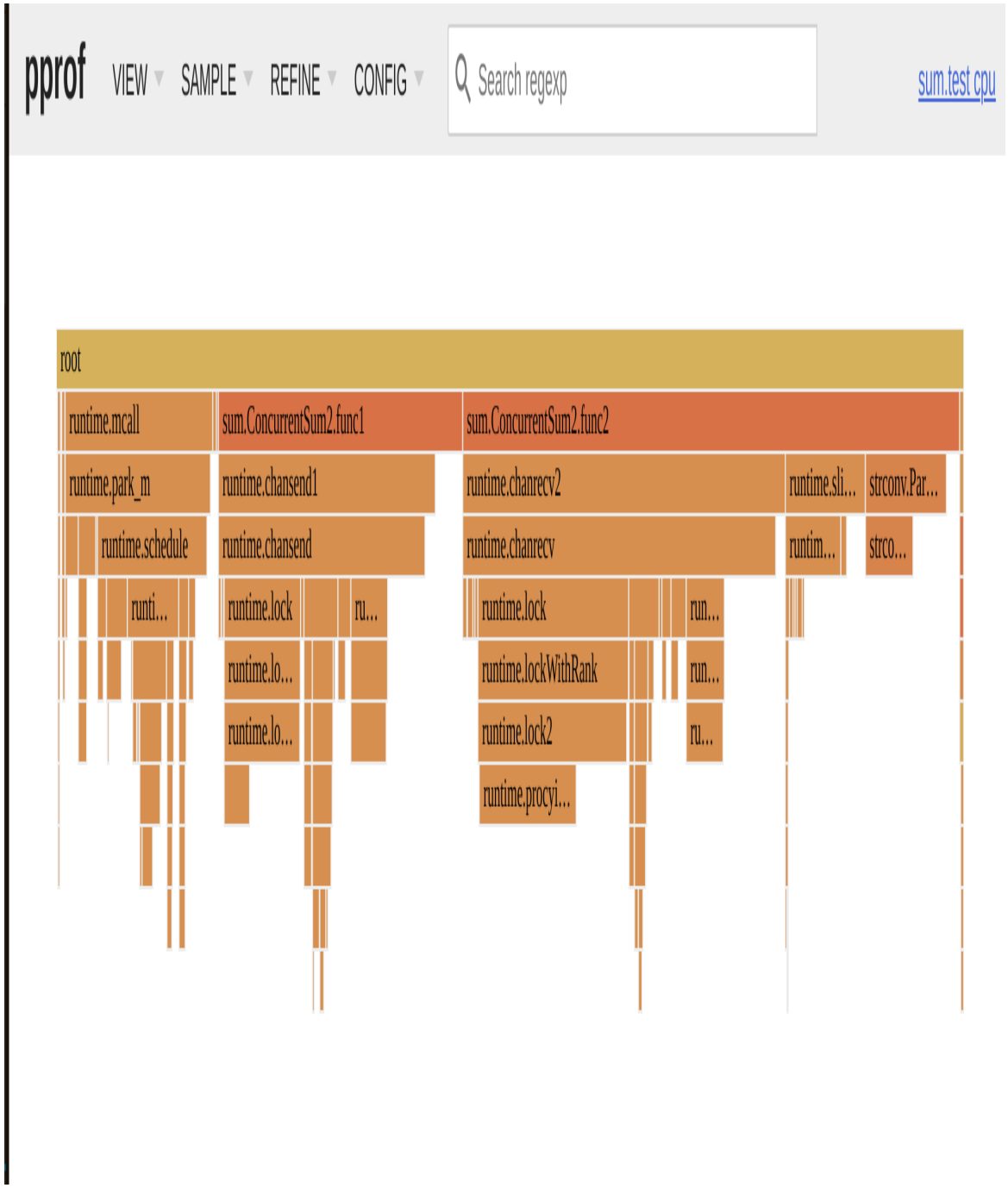


Figure 4-7. CPU profiler flame graph for Example 4-10 execution.

If you see a profile like this, it should immediately tell you that the concurrency overhead is just too large. It takes most of the CPU time of our application. We don't even see the actual work like parsing integers anymore since this work is out-numbered by Go scheduler overhead. The flame graph is full of `runtime.schedule`, locks from sending channels, and receiving[25].

The problem here is not with the distribution and worker pattern generally. Rather, our inefficient distribution algorithm sends line by line to each goroutine worker. The main issue is that the parsing and additions are faster than the communication overhead. Essentially coordinated distribution of work like this takes more CPU resources than work itself! This is why we see most CPU time spent in channels in Figure 4-7.

We have multiple options for improvement here. We can ensure more work in each goroutine, or we can sometimes delete the distribution work completely. Ideally, we could do both for more efficient execution! In our case, we can accomplish this via a coordination-free algorithm that will shard (split) the workload evenly across all goroutines. It's coordination-free because there is no communication to agree on which part of work is assigned to each goroutine. We can do that thanks to the fact that the size of the file is known upfront, so we can use some sort of hashing to assign each part of the file with multiple lines to each goroutine worker. Let's see how this could be implemented in Example 4-11.

Example 4-11. Concurrent implementation of Example 4-1 that maintains finite set of goroutines that computes group of lines. Lines are sharded without coordination.

```
func ConcurrentSum3(fileName string, workers int) (ret int64, _
error) {
   b, err := ioutil.ReadFile(fileName)
   if err != nil {
      return 0, err
   }

   var (
      bytesPerWorker = len(b) / workers
      resultCh       = make(chan int64)
   )
```

```
   for i := 0; i < workers; i++ {
      go func(i int) {
         // Coordination-free algorithm, which shards buffered file
deterministically.
         begin, end := shardedRange(i, bytesPerWorker, b) ❶

         var sum int64
         for _, line := range bytes.Split(b[begin:end],
[]byte("\n")) {
            num, err := strconv.ParseInt(string(line), 10, 64)
            if err != nil {
               // TODO(bwplotka): Return err using other channel.
               continue
            }
            sum += num
         }
         resultCh <- sum
      }(i)
   }

   for i := 0; i < workers; i++ {
      ret += <-resultCh
   }
   close(resultCh)
   return ret, nil
}
```

❶ `shardedRange` is not supplied for clarity. This function is not too complex, but it hides some complexity. You can see the full code here.

Test pass too, so we confirmed Example 4-11 is functionally correct. But is it faster? Full comparison of latencies and memory allocations are presented in Example 4-12.

Example 4-12. `benchstat` output for the base implementation Example 4-1 and three different concurrent iterations: Example 4-9, Example 4-10 and Example 4-11.

```
name (time/op)     Sum             ConcurrentSum1  ConcurrentSum2
ConcurrentSum3
Sum-4              78.4ms ± 4%     303.0ms ± 9%    119.6ms ± 4%
17.3ms ± 7%

name (alloc/op)    Sum             ConcurrentSum1  ConcurrentSum2
ConcurrentSum3
```

```
Sum-4                30.5MB ± 0%     102.3MB ± 0%     30.4MB ± 0%
30.5MB ± 0%
```

Finally, our effort paid off. Our CPU-bound task is now significantly (almost five times) faster! Assuming this is not fast enough, we could check if any concurrency overhead might be worrying. Let's look at our CPU profiler again, presented in Figure 4-8.

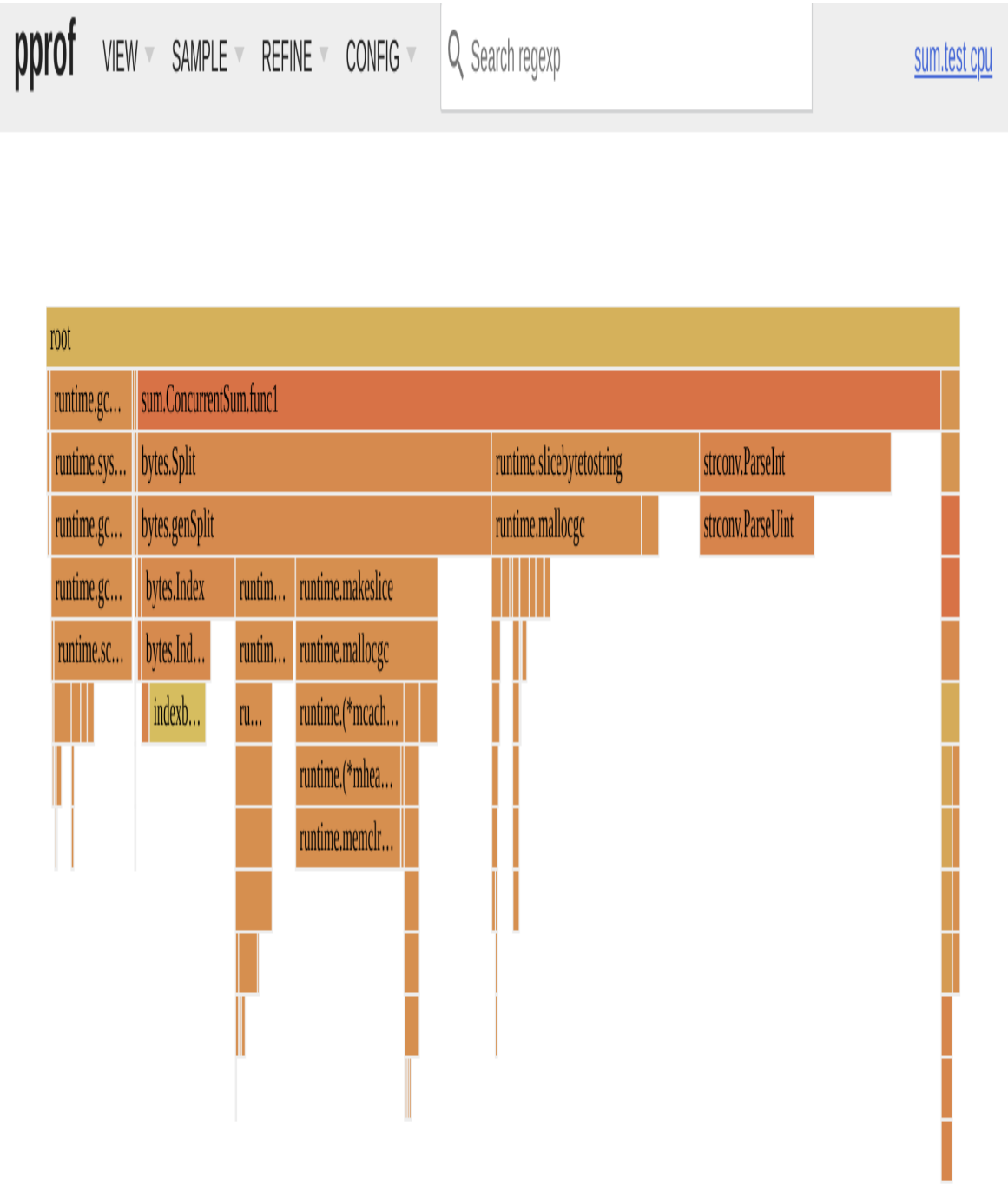


Figure 4-8. CPU profiler flame graph for Example 4-11 execution.

Our CPU profile looks perfect from the perspective of concurrency. The profile clearly shows the biggest CPU usage done by our two functions,

`bytes.Split` and `strconv.ParseInt`. We will optimize those in [Link to Come], which has the potential to reduce execution time to 5ms or less. On top of that, we can see `runtime.malloc` and `runtime.gc` involvement, which I will explain in the next chapter.

Hopefully, this section explained some possible concurrency patterns that allow utilizing our multi-core machines. We also presented how important benchmarking and profiling which we will discuss in (probably my favourite chapters) [Link to Come] and [Link to Come]! Sometimes results might surprise us, so always seek confirmation of your ideas.

Summary

The modern CPU hardware is a non-trivial component that allows us to run our software efficiently. With ongoing operating systems and Go language development and advancement in hardware, only more optimization techniques and complexities will arise to decrease running costs and increase processing power.

In this chapter, I hopefully gave you basics that will help you optimize your usage of CPU resources and, generally, our software execution speed. First, we discussed Assembly language and how it can be useful during Go development. Then, we explored Go compiler functionalities, optimizations and ways to debug its execution.

Later on, we jumped into the main challenge for CPU execution: the latency of memory access in modern systems. Finally, we discussed the various low-level optimizations like L-caches, pipelining, CPU branch prediction and HyperThreading.

Last but not least, we explored the practical problems of executing our programs in production systems. Unfortunately, our program is rarely the only process on our machine, so efficient execution matters. Finally, we discussed Go's concurrency framework with an actionable example of a step by step optimizations using benchmarks and profiling.

In practice, CPU resource is essential to optimize in modern infrastructure to achieve faster execution and the ability to pay less for our workloads. Unfortunately, CPU resource is only one aspect. Our optimization of choice might prefer using more memory to reduce CPU usage.

As a result, our programs typically use a lot of memory resources (plus I/O traffic through disk or network). While execution is tied to CPU, those resources like memory and I/O might be the first on our list of optimizations depending on what we want (e.g. cheaper execution, faster execution or both). Let's discuss those in the next chapter.

---

1 The most popular CPU types from AMD and Intel use Complex Instruction Set Computing (CISC). It's complex because some instructions can use multiple CPU cycles to perform CISC tasks. This is done to follow the "simple software, complex hardware" pattern. On the other hand, ARM type CPUs are gaining popularity, using Reduced Instruction Set Computing (RISC). RISC has fewer instructions to choose from but strictly performs a single cycle per instruction, reducing power consumption yet requiring smarter compilers. Nevertheless, most of the optimizations discussed in this book apply to CISC and RISC processing units.

2 To be technically strict, modern computers nowadays have distinct caches for program instructions and data, while both are stored the same on the main memory. This is so-called modified Harvard architecture. At the optimization levels we aim for in this book, we can safely skip this level of detail.

3 For scripted (interpreted) languages, there is no complete code compilation. Instead, there is an interpreter that compiles the code statement by statement. Another unique type of language is represented by a family of languages that use Java Virtual Machine (JVM). Such machine can dynamically switch from interpreting to just in time (JIT) compilation for runtime optimizations.

4 Note that in the Go Assembly register, names are abstracted for portability. Since we will compile to 64-bit architecture, `SP` and `SI` will mean RSP and RSI registers.

5 There can be incompatibilities, but mostly with special-purpose instructions like cryptographic or SIMD instructions, which can be checked in runtime if they are available before execution.

6 Function call needs more CPU instructions since the program has to pass argument variables and return parameters through the stack, keep the current function's state, rewind stack after the function call, adding new frame stack etc.

7 Go tooling allows us to check the state of our program through each optimization in SSA form thanks to the `GOSSAFUNC` environment variable. It's as easy as building our program with `GOSSAFUNC=<function to see> go build` and opening the resulting `ssa.html` file. You can read more about it at here.

8 However, there are discussions to remove it from the default building process.

9 On top of SISD and SIMD, Flynn's taxonomy also specifies MISD, which describes performing multiple instructions on the same data and MIMD, which describes full parallelism. MISD is rare and only happens when reliability is important. For example, four flight control computers perform exactly the same computations for quadruple error-check in every NASA space shuttle. MIMD, on the other hand, is more common thanks to multi-core or even multi-CPU designs.

10 This is why we see specialized chips (called Neural Processing Units, so NPUs) appearing in the commodity devices. For example, Tensor Processing Unit (TPU) in Google phones, A14 Bionic chip in iPhones and dedicated NPU in M1 chip in Apple laptops.

11 Sizes of caches can vary. Example sizes are taken from my laptop. You can check the sizes of your CPU caches in Linux by using the `sudo dmidecode -t cache` command.

12 If a CPU can in total performs up to 1 instruction per cycle (IPC $\Leftarrow$ 1), we call it a scalar CPU. Most modern CPU cores have IPC $\Leftarrow$ 1, but one CPU has more than one core which makes IPC > 1. This makes those CPUs superscalar. IPC had quickly become a performance metric for CPUs.

13 Huge cost is not an overstatement. Latency of context switch depends on many factors, but it was measured that in the best case, direct latency (including operating system switch latency) is around 1350 nanoseconds. 2200 nanoseconds if it has to migrate to a different core. This is only a direct latency, from the end of one thread to the start of another. The total latency that would include the indirect cost in the form of cache and pipeline warm-up could be as high as 10 000 nanoseconds (and this is what we see in our [Link to Come]). During this time, we could compute something like 40 thousand instructions.

14 In some sources, this technique is also called CPU threading (aka hardware threads). I will avoid this terminology in my book due to possible confusion with operating system threads.

15 Do not confuse Hyper-threading logical cores with virtual CPUs (vCPUs) referenced when we use virtualizations like virtual machines. Guest operating systems uses the machine's physical or logical CPUs depending on host choice, but in both cases, those are called vCPUs.

16 That's why we will explain those advanced code level optimizations near the end of our book, in [Link to Come]

17 There are lots of good materials about tuning up the operating system. Many virtualization mechanisms like containers with orchestrating systems like Kubernetes also have their notion of priorities and affinities (pinning processes to specific cores or machines). In this book, we focus on writing efficient code, but we must be aware that execution environment tuning has an important role in ensuring quick and reliable program executions.

18 Details around Go runtime implementing Go scheduling are pretty impressive. Essentially, go does everything to keep the OS thread busy (spinning the OS thread) so it's not moving to a blocking state as long as possible. If needed, it can steal goroutines from other threads, polls networks, etc., to ensure we keep the CPU busy, so OS does not preempt the Go process.

19 In practice, there are ways to get this information using debug tracing. However, we should not rely on the program knowing which goroutine is a parent goroutine for normal execution

flow.

20 Funny enough, even atomic operations on CPU requires, some kind of locking. The difference is that instead of specialized locking mechanisms like spinlock, atomic instruction can use faster memory bus lock.

21 Assuming the programmer keeps to that rule. There is a way to send a pointer variable (e.g.. `*string`) that points to shared memory, which violates the rule of sharing information through communicating.

22 I omitted two additional mechanisms on purpose. First of all, `runtime.Gosched()` exists, which allows yielding current goroutine, so others can do some work in the meantime. This command is less useful nowadays since the current Go scheduler is preemptive, and manual yielding has become impractical. The second interesting operation, `runtime.LockOSThread()`, sounds useful, but it's not designed for efficiency, but rather to pin goroutine to OS thread so we can read certain OS thread state from it.

23 I recommend watching this talk from GopherCon 2019 to learn low-level details around Go scheduler.

24 The full code with benchmarks is available in example repo.

25 You don't need to be familiar with those functions to get an idea of what they do (e.g. `runtime.chansend`), thanks do their meaningful names. And if you do, feel free to search for it on GitHub. We can find the code for each function to understand what it does.

Chapter 5. How Go Uses Memory Resource

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at mpotter@oreilly.com.

In the previous chapter, we looked under the hood of the modern computer. We discovered that improving efficiency by better CPU utilization (doing less work or multi-threading) is a key to faster executions of our programs. Efficient execution of instructions in the CPU is important, but the sole purpose of implementing those instructions is to modify the data. Unfortunately, the path of changing data is not always trivial. In Chapter 4, we learned that in von Neuman architecture (presented in Figure 4-1), we experience "CPU and Memory Wall Problem". The industry invented numerous technologies and optimization layers to overcome the Memory Wall and other challenges like memory safety or a need to store GBs of data in the main memory.

As a result, accessing eight bytes in our RAM might look like a simple `MOVQ <from address XYZ> <target register>` in the CPU, but the actual process to access that information from the physical chip

storing those bytes is very complex. We discussed mechanisms like "Hierachical Cache System", but there is much more. Memory controllers, operating systems, and Go runtime closely manage memory, making our programs more efficient.

In some way, those mechanisms are abstracted from programmers as much as possible. So, for example, we define a variable in Go code and rarely think about how much memory was used, where it got allocated, in how many L-caches it fits. In most cases, this is perfect enough, but as we learned in Chapter 1 sooner than later, our program will have some efficiency requirements, and this is when we start our TFBO flow ("Efficiency-Aware Development Flow"). For this purpose, we have to bring our Mechanical Sympathy on towards memory resource!

> **DISCLAIMER: WE WILL BE FOCUSING ON TYPICAL HARDWARE AND OPERATING SYSTEMS.**
>
> There are many different hardware types for memory and I/O devices. This book focuses on typical hardware, generally available for mobile devices, laptops, personal computers, servers, and computing clouds.
>
> For operating systems, we will focus on Linux, which has the most sophisticated and efficient memory and I/O management of all modern operating systems. The macOS and Windows systems are, however, in the majority of cases, sharing the knowledge and implementing similar mechanisms, so we can expect similar effects if our optimizations work for Linux systems.

This chapter will focus on understanding the RAM resource from the user perspective. We will start by exploring overall memory relevance in "Memory Relevance". Then we will set the context by understanding "Do we Have a Memory Problem?". After that, we will explain the patterns and consequences of each element involved in the memory access from bottom to top. The data journey for memory starts in "Physical Memory", so the hardware memory chips. Then we will move to operating system (OS) memory management techniques that allow managing limited physical memory space in multi-process systems: "Virtual Memory" and "OS Memory Mapping" with more detailed explanation of the "`mmap` Syscall".

With the lower layers of memory access explained, we can move to the key knowledge for Go programmers looking to optimize memory efficiency—the explanation of the "Go Memory Management". This includes the necessary elements like memory layout, what "Values, Pointers and Memory Blocks" mean, and the basics of "Go Allocator" with its measurable consequences. Finally, we will explore "Garbage Collection" mechanisms.

Last, we will look at pragmatic memory optimization patterns called "The Three R's Optimization Method" that I use when optimizations are needed.

We will go into many details about memory in this chapter, but the key aim is to build an instinct toward patterns and behaviour of Go programs when it comes to memory usage. For example, what problems can occur while accessing memory? How do we measure memory usage? What does it mean to allocate memory? How can we release it? Let's try to explore answers to those questions in this chapter.

Let's start this chapter by clarifying why RAM is relevant to our program execution. What makes it so important?

Memory Relevance

All Linux programs require more resources than just the CPU to perform their programmed functionalities. For example, let's take a web server like Nginx (written in C) or Caddy (written in Go). Those programs allow serving static content from disk or proxy HTTP requests, among other functionalities. They use the CPU to execute written code. However, a web server like this also interacts with other resources, for example:

- With RAM to cache basic HTTP responses.
- With disk to load configuration, static content, or write log lines for observability needs.
- With network to serve HTTP requests from remote clients.

As a result, the CPU resource is only one part of the equation. This is the same for most programs—they are created to save, read, manage, operate and transform data from different mediums.

One would argue that the "memory" resource, often called Random Access Memory (RAM)[1], sits at the core of those interactions. The RAM is the backbone of the computer because every external piece of data (bytes from disk, network or another device) has to be buffered in memory to be accessible to the CPU. So, for example, the first thing OS does to start a new process is to load part of the program's machine code and initial data to memory for the CPU to execute it.

Unfortunately, we must be aware of three main caveats when using memory in our programs:

- As mentioned in "CPU and Memory Wall Problem", RAM access is significantly slower than CPU operational speed.
- There is always a finite amount of RAM in our machines (typically from a few GB to hundreds of GB per machine), which forces us to care about space efficiency[2].
- Unless the persistent type of memory will be commoditized with RAM-like speeds, pricing and robustness, our main memory is strictly volatile. When the computer power goes down, all information is completely lost[3].

The ephemeral characteristics of memory and its finite size are why we are forced to add an auxiliary, persistent Input/Output (I/O) resource to our computer, i.e. a disk. These days we have relatively fast Solid-State Drive (SSD) disks (yet still around 10x slower than RAM) with a limited lifetime (~5 years). On the other hand, we have a slower and cheaper Hard-Disk Drive (HDD). While cheaper than RAM, the disk resource is also a scarce resource.

Last but not least, for scalability and reliability reasons, our computers rely on data from remote locations. Industry invented different networks and protocols that allow us to communicate with remote software (e.g.

databases) or even remote hardware (via iSCSI or NFS protocols). We typically abstract this type of I/O as a network resource usage. Unfortunately, the network is one of the most challenging resources to work with because of its unpredictable nature, limited bandwidth and bigger latencies.

While using any of those resources, we use it through the memory resource. As a result, it is essential to understand its mechanics. The problem is that accessing data to and from memory can be implemented differently. For example, even in Go, we can:

- Buffer all the data for computation purposes, or stream it in smaller or bigger chunks.
- Reuse buffers or allocate new ones every time we need them.
- Compress the data on the way.
- Encode that data in many different structures and alignments.
- In extreme cases, we can attempt to avoid object-oriented programming (OOP) and stick to data-driven design.

There are many things a programmer can do to impact the application's memory usage. But unfortunately, our implementations tend to be prone to inefficiencies and unnecessary waste of computer resources or execution time without proper education. This problem is amplified by the vast amount of data our programs have to process these days. This is why we often say that efficient programming is all about the data.

MOST EFFICIENCY PROBLEMS IN GO COME FROM MEMORY USAGE.

As mentioned in "Is Go "Fast"?", Go is garbage collected (GC) language. GC allows Go to be an extremely productive language, but it sacrifices some visibility and control over memory management (more on that in "Garbage Collection").

For cases where we need to process a significant amount of data, or we're under some resource constraints, we have to take more care with how our program uses memory. Unfortunately, this is not the most straightforward task, given the complexity of OS and Go memory management. Therefore, I would recommend reading this chapter with extra care since most first-level optimizations are usually around memory resources.

Once we start thinking about the memory efficiency of Go, we need to rethink our code and spend extra energy. Let's stay pragmatic, though. First, there are some reasonable lightweight optimizations we will learn in [Link to Come]. Then as TFBO flow teaches us, with benchmarking and profiling (explained in [Link to Come]), we can narrow down to concrete code parts that require more memory efficiency. Before all of this, in the next section, let's learn when to start the memory optimization process. A few common symptoms might reveal the low hanging fruits for memory optimizations.

Do we Have a Memory Problem?

It's useful to understand how Go uses the computer's main memory and its efficiency consequences, but we must also follow the pragmatic approach. We should refrain from optimizing memory until we know there is a problem. We can define a set of situations that should trigger our interest in Go memory usage and potential optimizations in this area:

- Our physical computer, virtual machine, container, or process crashed because of an out-of-memory signal (OOM), or our process is about to hit that memory limit[4].
- Our Go program is executing slower than usual, while the memory usage is higher than average. Spoiler: Our system might be under

memory pressure causing trashing or swapping explained in "OS Memory Mapping".

- Our Go program is executing slower than usual, with high spikes of CPU utilization. Spoiler: Garbage collection might slow our programs if an excessive number of short-living objects is created, which I'll explain in "Garbage Collection").

Suppose you encounter any of those situations, which occur, unfortunately, more often than we want. In that case, it might be a time to debug and optimize the memory usage of your Go program. On the other hand, one could use a more proactive approach—there are a set of signals that could trigger our suspicion over memory usage during development. There are cases where our program clearly could use less memory for its work, leading to future problems.

For example let's take our example of `Sum` function in Example 4-1. In "Practical Example of Concurrency in Go" we focus on improving latency of the `Sum` function. But memory is an essential resource too. How much memory does naive implementation of `Sum` use? We can perform a quick microbenchmark to tell (don't worry, I will explain how to perform benchmarks in [Link to Come]). Results presented in Example 5-1 reveals that in this form, Example 4-1 for a file with million lines (around 3.4 MB size) requires at least 30.5 MB of RAM space (in practice it needs more as we will learn in "Go Memory Management"). On top of that, if we double the input file size, we see Example 4-1 using twice as much memory.

Example 5-1. Benchmark allocation result for Example 4-1 with 1 million elements input and 2 million elements respectively.

```
name (alloc/op)     Sum1M          Sum2M
Sum-12              30.5MB ± 0%    60.9MB ± 0%

name (alloc/op)     Sum1M          Sum2M
Sum-12              818k ± 0%      1636k ± 0%
```

Generally, such results should trigger our suspicion toward memory consumption of the program in Example 4-1. Especially the relation of the

input size to the allocated memory often reveals if there is room for easy wins to reduce memory usage.

THREE CLEAR INDICATIONS WE WASTE MEMORY SPACE.

With experience, you can detect potential anomalies that might suggest there are trivial optimization for memory usage. For example:

- For simple functionality (e.g. to sort data), the program allocates more than, let's say, three times more the memory than the input (e.g. input file) data size.
- With time, the total memory used by the program constantly grows and never goes down. This most likely indicates a Memory Leak--a problem which is often easy to fix.
- The larger the input file, the more memory is required for the program functionality. This might be fine initially, but it indicates future memory resource problems if we try to scale our application[5].

In [Link to Come], I propose the algorithm and code improvements to Example 4-1 that allow it to use memory in numbers that are a fraction of the input file size while also improving the latency.

But before we go into optimizations, we need to understand the basics. The first input Example 4-1 allocated 30.5 megabytes of memory. But what does it mean? Where was that space reserved? Does it mean we used exactly 30.5 MB of physical memory or more? Was this memory released at some point? This chapter aims to give you awareness, allowing you to answer all of those questions. We will learn why memory is often the issue and what we can do about it. By the end of this chapter, you should be able to answer all of those questions.

As with the CPU resources, it's easier to figure out memory optimizations if we understand the foundations—the basics of memory management from the point of view of Hardware (HW), Operating System (OS) and Go runtime. Let's start with basic details about physical memory, which directly impacts our program execution. On top of that, this knowledge

might help you to understand better the specifications and documentation of modern physical memory!

Physical Memory

We store information digitally in the form of bits, the basic computer storage unit. A bit can have one of two values, 0 or 1. With enough bits, we can represent any information: integer, floating value, letters, messages, sounds, images, videos, programs, games, etc. Eight bits form a "byte". That number came from the fact that in the past, the smallest number of bits that could hold a text character was 8 bits[6]. That's why the industry standardized a "byte" as the smallest meaningful unit of information.

As a result, most hardware is byte-addressable. This means that, from a software programmer's point of view, there are instructions to access data in the form of individual bytes. If you want to access a single bit, you need to access the whole byte and use bitmasks to get or write the bit you want.

The byte addressability makes developer life easier when working with data from different mediums like memory, disk, network, etc. Unfortunately, that creates a certain illusion that the data is always accessible with byte granularity. Don't let that mislead you. More often than not, the underlying hardware has to transfer a much larger chunk of data to give you the desired byte.

For example, in "Hierachical Cache System" we learned that CPU registers are typically 64 bits (8 bytes) large, and the cache line is even bigger (64 bytes). Yet, we have CPU instructions that can copy a single byte from memory to the CPU register. However, an experienced developer will notice that to copy that single byte, in many cases, the CPU will fetch not one byte but at least a complete cache line (64 bytes) from physical memory.

From a high-level point of view, physical memory (RAM) can also be seen as byte-addressable, as presented in Figure 5-1.

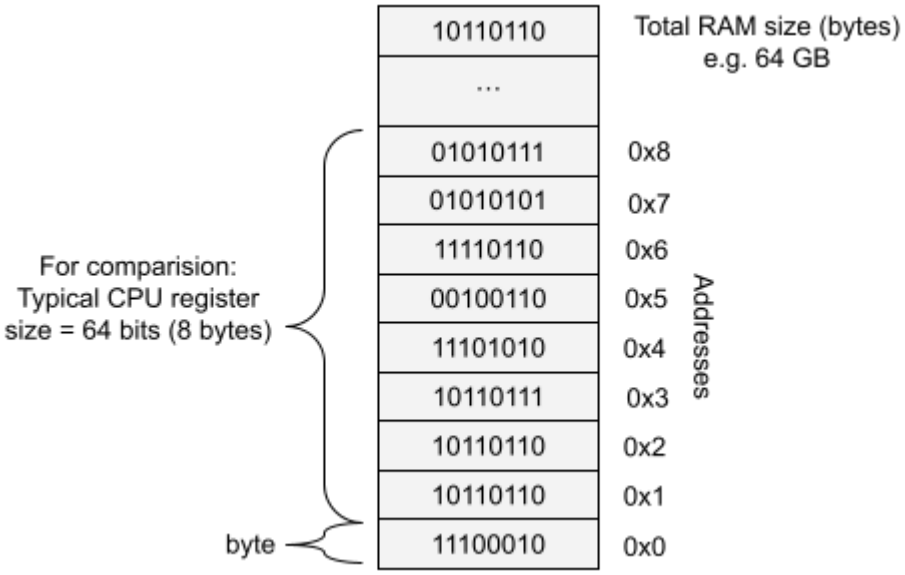


Figure 5-1. Physical memory address space.

Memory space can be seen as a contiguous set of 1-byte slots with a unique address. Each address is a number from 0 to the total capacity of memory in the system in bytes. For this reason, 32-bit systems that use only 32-bit integers for memory addresses typically could not handle RAM with more capacity than 4GB—the largest number we can represent with 32 bits is 2^{32}. This limitation was removed with the introduction of the 64-bit operating systems that use 64-bit (8 bytes)[7] integers for memory addressing.

We discussed in "CPU and Memory Wall Problem" that memory access is not that fast compared to, e.g. CPU speed. But there is more. Addressability, in theory, should allow fast, random access of bytes from the main memory. After all, this is why that main memory is called RAM (random access memory). Unfortunately, even if we look at our napkin math in [Link to Come], sequential memory access can be ten times (or more) faster than random access, which has a direct consequence on how we should use memory in our application. The first reason is that the CPU with L-caches (mentioned in "Hierachical Cache System") is fetching and caching bigger chunks of memory upfront.

The second reason is that, within the last couple of decades, we only improved the speed (bandwidth) of the sequential read. We did not improve

random access latency at all! The lack of improvement on the latency side is not a mistake. It is actually a very pragmatic and strategic choice. The internal designs of the modern RAM modules have to work against various requirements and limitations, for example:

Capacity

> There is a strong demand for bigger capacities of RAM (e.g., to compute more data or play more realistic games).

Bandwidth and latency

> We would like to wait less time on accessing memory while writing or often reading large chunks of data since memory access is the major slow down for CPU operations.

Voltage

> There is a demand for a lower voltage requirement for each memory chip, which would allow for running more of them while maintaining low power consumption and manageable thermal characteristics (more time on battery for our laptops and smartphones!)

Cost

> RAM is a fundamental piece of the computer required in large quantities; thus, production and usage costs must be kept low.

It is fascinating how much thought went into designing different generations of RAM hardware to fullfill the above requirements. Let's briefly talk about the basics.

The main physical memory that we use when we execute our programs (RAM) is based on Dynamic Random Access Memory (DRAM) chips are soldered into modules, often referred to as RAM "sticks". When connected to the motherboard, those chips allow us to store and read data bits as long as the DRAM is continuously powered.

DRAM contains billions of memory cells (as many cells as the number of bits DRAM can store). Each memory cell comprises one access transistor acting as a switch and one storage capacitor. Transistor guards the access to the capacitor, which is charged to store one or drained to keep 0 value. This allows each memory cell to store a single bit of information. This architecture is much simpler and cheaper to produce and use than Static RAM (SRAM), which is generally faster and used for smaller types of memory like registers and L-caches in the CPU.

At the moment of writing, the most popular memory used for RAM is the simpler, synchronous (clock) version in the DRAM family—SDRAM. Particularly the fifth generation of SDRAM called DDR4.

Let's summarize a few things worth remembering about modern generations of hardware for RAM like DDR4 SDRAM:

- Random access of the memory is relatively slow, and generally, there are not many good ideas to improve that soon. If anything, lower power consumption, larger capacity, and bandwidth only increase that delay.
- Industry is improving overall memory bandwidth by allowing us to transfer bigger chunks of adjacent (sequential) memory. This means that efforts to align Go data structures and knowing how those are stored in memory matters so that we can access them sequentially will matter.
- To leverage more complex SDRAM optimizations like DDR4 pages and bank groups, hardware controllers and operating systems must be more complex. This means more development (and runtime) overhead for those pieces.
- More memory capacity is great, and the industry is focused on bringing us bigger SDRAM chips for a lower price. However, the memory hardware access alone is extremely complex. As a result, the first thing we should focus on while optimizing our Go programs is

using less memory (instead of optimizing accessing that memory as the first thing).

We can learn many eye-opening facts if we would study the exact implementation of hardware and algorithms for accessing bits from DRAM memory cells. It explains:

- Why does the DDR4 SDRAM memory have the characteristics mentioned above?
- What does it mean for SDRAM to be both “dynamic” and “synchronous”?
- Why did the random access latency of DRAM not improve at all from the 1990s, and why do the numbers of MTs (Megatransfers per second) we see when we buy modern RAM sticks can be misleading?

If you want to learn answers to the above questions, I invite you to check the [Link to Come] where I provide a deep dive on SDRAM hardware and how it interacts with our CPU. In this section, I wanted to focus only on the basics of physical memory, which is a must-have to understand, before discussing further memory mechanisms.

These days programs never access physical memory directly—operating system (OS) manages the RAM space. This is probably for the best because every developer would otherwise have to understand low-level memory access details. But there are more important reasons why there has to be an OS between our programs and hardware. The next section will show you why by looking at how OS manages the memory and what it means for our Go programs.

OS Memory Management

The internal implementation of physical memory can be challenging to comprehend and analyze. The good news is that the modern operating systems do an excellent job of hiding that complexity, so it’s easier for programs to use it. The bad news is that operating systems add another

layer of complexity. Given that, it's useful for a programmer to understand its consequences from a high-level perspective.

Fear not! This section will try to expand on some basics of modern Linux OS memory management. This is a must-learn if we want to understand what it means when our Go program statement allocates, let's say, 100 MB of memory.

First, we need to understand the operating system's goals for RAM management. Hiding complexities of physical memory access is only one thing. The other, potentially even more important goal is to allow using the same physical memory simultaneously and securely across thousands of processes and their OS threads (both terms introduced "Operating System Scheduler"). The problem of multi-process execution on common memory space is non-trivial for multiple reasons:

Dedicated memory space for each process

> Programs are compiled assuming nearly full and continuous access to the RAM. As a result, the OS must track which slots from the physical memory from our Figure 5-1 address space belong to which process. Then we need to find a way to coordinate those "reservations" to the processes so only allocated addresses are accessed.

Avoiding external fragmentation

> Having thousands of processes with dynamic memory usage poses a great risk of waste in memory due to inefficient packing. We call this problem the external fragmentation of memory.

Memory isolation

> We have to ensure no process touches the physical memory address reserved for other processes running on the same machine (e.g. operating system processes!). This is because any accidental write or read from outside of process memory (out-of-bounds memory access) can crash other processes, malform data on persistent mediums (e.g.

disk), or crash the whole machine (e.g. if you corrupt the memory used by OS).

Memory safety

Operating systems are usually multi-user systems, which means processes can have different permissions to different resources (e.g. files on disk or other process memory space). This is why the mentioned out-of-bounds memory accesses have serious security risks[8]. Imagine a malicious process with no permissions reading credentials from other process memory, or causing a Denial of Service (DoS). This is especially important for virtualized environments, where a single memory unit can be shared across different operating systems and even more users.

Efficient memory usage

Programs never use all the memory they asked for at the same time. For example, instruction code and statically allocated data (e.g. constant variables) can be as large as dozens of MB. But for single-threaded applications, a maximum of a few kilobytes of data is used in a given second. Instructions for error handling are rarely used. Arrays are often oversized for worst-case scenarios.

To solve all those challenges, modern OS manages memory using three fundamental mechanisms we will learn about in this section: Paged virtual memory, memory mapping and hardware address translation. Let's start by explaining virtual memory.

Virtual Memory

The key idea behind virtual memory is that every process is given its own logical, simplified view of the RAM. As a result, programming language designers and developers can effectively manage process memory space as if they had an entire memory space for themselves. Even more, with virtual memory, the process can use a full range of addresses from 0 to 2^{64} for its

data, even if the physical memory has, for example, the capacity to accommodate only 2^{35} addresses (32 GB of memory). This frees the process from coordinating the memory between other processes, bin packing challenges, and other important tasks (e.g. physical memory defragmentation, security, limits, swap). Instead, all of those complex and error-prone memory management tasks can be delegated to the kernel (a core part of the Linux operating system).

There are a few ways of implementing virtual memory, but the most popular technique is called paging[9]. It means that OS divides physical and virtual memory into fixed-size chunks of memory. The virtual memory chunks are called pages, whereas physical memory chunks are called frames. Both pages and frames are typically 4KB large and can be individually managed.

THE IMPORTANCE OF THE PAGE SIZE

The sizing of the page is an important topic as that directly affects how efficiently memory is used by running software. The default page size might be 4 KB[10], but it can be changed to higher page sizes with respect to specific CPU capabilities[11]. It is also possible to use 4KB pages for normal workloads and have dedicated (sometimes transparent to processes!) memory space for `huge pages` of size from 2 MB to 1GB.

The 4KB number was chosen in the 1980s, and many say that it's time to bump this number up, given modern hardware and cheaper RAM (in terms of dollars per byte). Yet, the choice of page size is a game of trade-offs. Larger pages inevitably waste more memory space[12], which is often referred to as the internal memory fragmentation. On the other hand, keeping a 4KB page size or making it smaller makes memory access slower and memory management more expensive, eventually blocking the ability to use larger RAM modules in our computers.

The OS can dynamically map pages in virtual memory to specific physical memory frames (or other mediums like chunks of disk space), mostly transparently to the processes. The mapping, state, permissions and additional metadata of the page are stored in page entry in the many hierarchical page tables maintained by the OS[13].

To achieve an easy-to-use and dynamic virtual memory, we need to have a versatile address translation mechanism. The problem is that only the OS knows about the current memory space mapping between virtual and physical space (or lack of it). Our running program's process only knows about virtual memory addresses, so all CPU instructions in machine code use virtual addresses. Our programs will be even slower if we try to consult OS for every memory access to translate each address. We would waste even more CPU cycles. So the industry figured out dedicated hardware support for translating memory pages.

From the 1980s, almost every CPU architecture started to include the Memory Management Unit (MMU) used for every memory access. MMU translates each memory address referenced by CPU instructions to a physical address based on OS's page table entries. To avoid accessing RAM to search for the relevant page tables, engineers added the Translation Lookaside Buffer (TLB). TLB is a small cache that can cache a few thousand-page table entries (typically 4KB of entries). The overall flow looks then like in Figure 5-2.

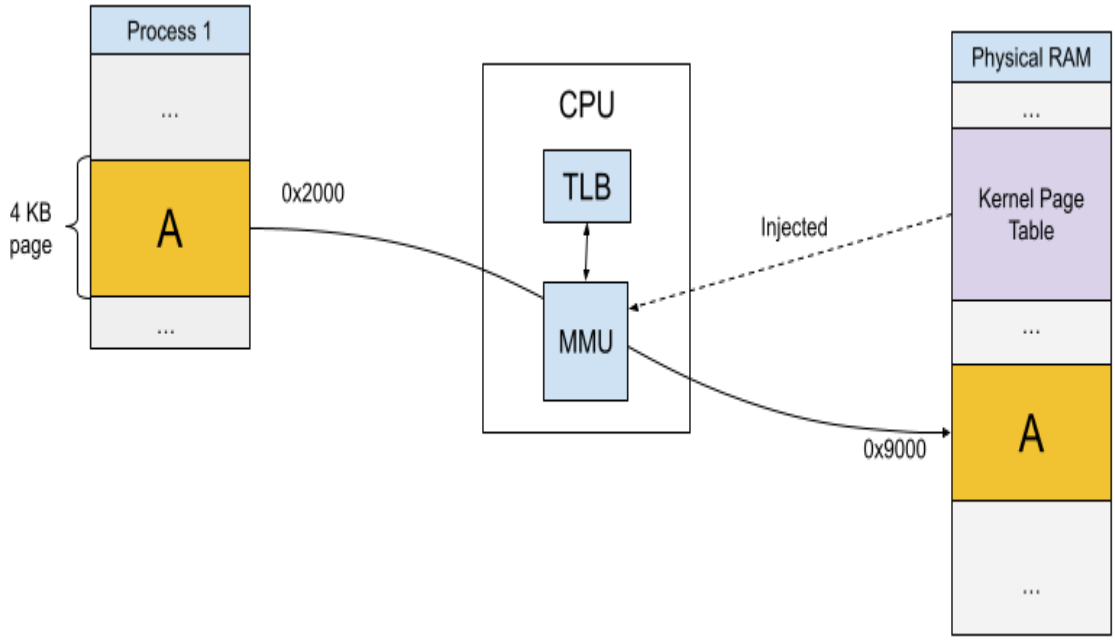


Figure 5-2. Address translation mechanism done by MMU and TLB in CPU. OS has to inject the relevant page tables so MMU knows what virtual addresses correspond to physical addresses.

TLB is very fast, but it has limited capacity. If MMU cannot find the accessed virtual address in the TLB, we have a TLB miss. This means that either CPU (hardware TLB management) or OS (software-managed TLB) has to walk through page tables in RAM, which causes significant latency (around 100 CPU clock cycles!).

It is essential to mention that not every "allocated" virtual memory page will have a reserved physical memory page behind it. In fact, most of the virtual memory is not backed up by RAM at all. As a result, we can almost always see large amounts of virtual memory used by the process (called `VSS` or `VSZ` in various Linux tools like `ps`). Still, the actual physical memory (often called `RSS` or `RES` from "residual memory") reserved for this process might be tiny. There are often cases where a single process allocates more virtual memory than is available to the whole machine! See an example situation like this on my machine in Figure 5-3.

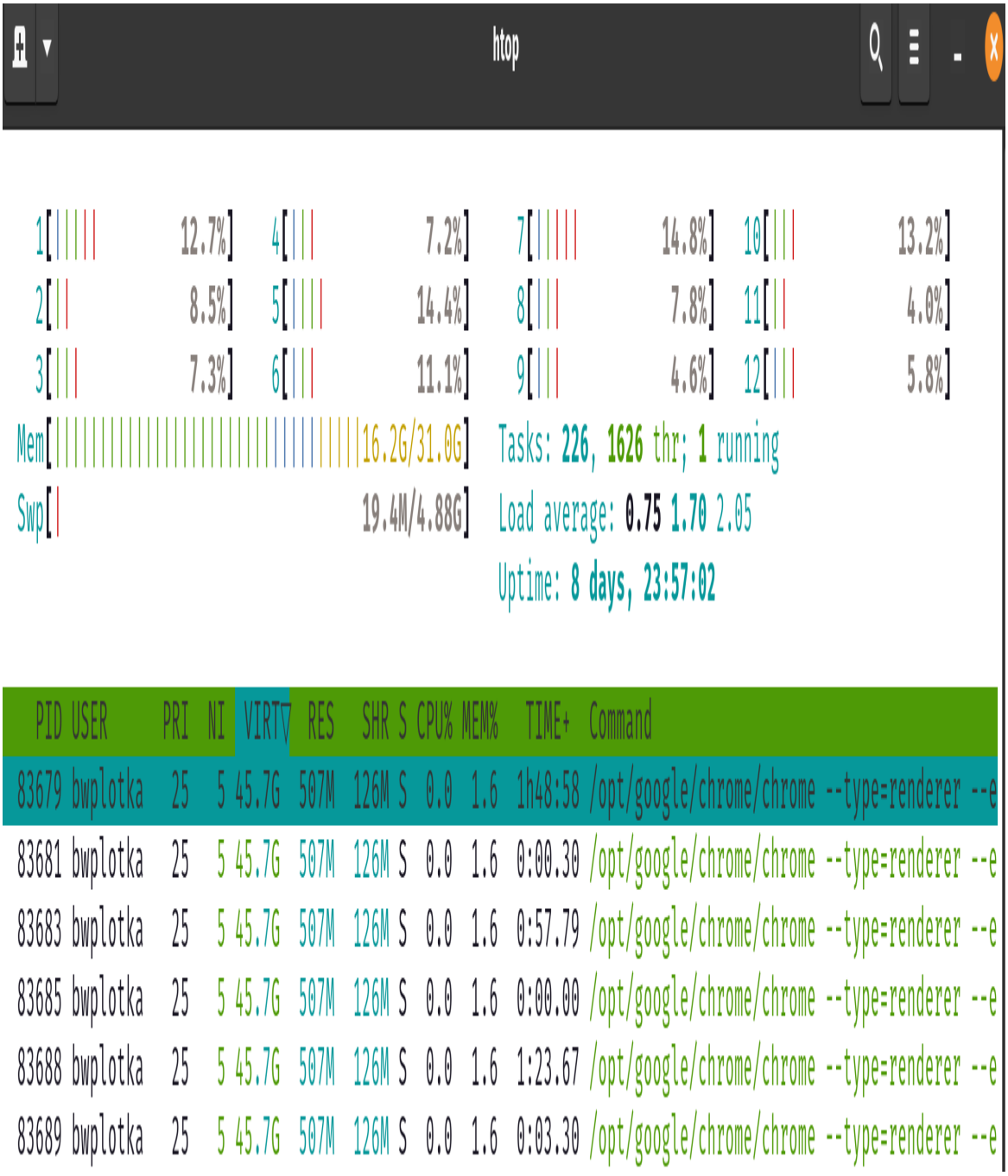


Figure 5-3. Few first lines of `htop` output, showing the current usage of a few Chrome browser processes, sorted by virtual memory size.

As we can see in Figure 5-3, my machine has 32 GB of physical memory, with 16.2 GB of current used. Yet we see Chrome processes using 45.7 GB of virtual memory each! However, if you look at the `RES` column, it has only 507 MB reserved, with 126 MB of it shared with other processes. So how this is possible? How can the process think that it has 45.7GB RAM

available, given the machine has only 32 GB and the system actually allocated just a few hundred MBs in RAM?

We can call such a situation a Memory Overcommitment, and it exists because of the very same reasons airlines often overbook seats for their flights. On average, many travellers cancel their trips at the last minute or do not show up for the flight. As a result, to maximize the plane's used capacity, it is more profitable for airlines to sell more tickets than seats in the airplane and handle the rare "out of seats" situations "gracefully" (e.g. by moving the unlucky customer to another flight). This means that the true "allocation" of seats happens when travellers actually "access" them during the flight onboarding process.

The OS performs the same overcommitment strategy by default[14] for processes trying to allocate physical memory. The physical memory is only allocated when our program access it, not when it "creates" a big object, for example, `make([]byte, 1024)` (you will see a practical example of this in "Go Allocator").

Overcommitment is implemented with the pages and memory mapping techniques. It allows the OS to map virtual memory pages to shared or private physical pages, pages on disk space or a mix of those. You can even have pages that are not mapped to anything yet (demand-zero page), which is the default initial state of the anonymous page.

Typically memory mapping refers to a low-level memory management capability offered with the mmap system call on Linux (and similar `MapViewOfFile` function in Windows). Go developers can utilize this explicitly in our Go programs for specific use cases. For instance, `mmap` is used extensively in almost every database software, e.g. in MySQL and PostgreSQL as well as those written in Go like Prometheus, Thanos and M3db projects. The `mmap` (among other memory allocation techniques) is also what Go runtime, and other programming languages use under the hood to allocate memory from OS, e.g. for the heap (discussed in "Go Memory Management").

Yet, using explicit `mmap` for most Go applications is not recommended. Instead, we should stick to the Go runtime's standard allocation mechanisms, which we will learn in "Go Memory Management". As our "Efficiency-Aware Development Flow" says, only if we see indications through benchmarking that this is not enough, we might consider moving to more advanced methods like `mmap`.

There is a reason why I explain `mmap` at the start of our journey with the memory resource. This is often overlooked (and that's why it is not much easy to use information about `mmap` behaviour), but the OS uses exactly the same memory mapping mechanism to manage all allocated pages in our system. So even if we don't use `mmap` explicitly, the data structures we use in our Go programs are indirectly saved to certain virtual memory pages, which are then `mmap`-like managed by OS. As a result, understanding the explicit `mmap` syscall will conveniently explain the on-demand paging and mapping techniques Linux OS uses to manage virtual memory.

Let's focus on Linux `mmap` syscall in the following subsection.

`mmap` Syscall

To learn about OS memory mapping patterns, I will explain `mmap` syscall functionality. I will start by going through an example of Go code that allows creating self-contained memory-mapped byte arrays backed by existing files on disk. We can cheat here and explain `mmap` before describing Go memory management in detail in "Go Memory Management" because the `mmap` syscall allows the programmer to escape from the standard Go memory mechanism.

The Example 5-2 shows a simplified abstraction that allows allocating a byte slice in our process virtual memory without Go memory runtime coordination. The content of the resulted byte slice maps 1:1 to the desired range of bytes from a file on disk.

Example 5-2. The adapted snippet of Linux specific Prometheus mmap abstraction that allows creating and maintaining read-only memory-

mapped byte arrays.

```
package mmap

import (
   "os"

   "golang.org/x/sys/unix"
)

type MemoryMappedFile struct {
   f *os.File
   b []byte
}

func OpenFile(path string, size int) (mf *MemoryMappedFile, _
error) {
   f, err := os.Open(path)
   if err != nil {
      return nil, err
   }

   b, err := unix.Mmap(int(f.Fd()), 0, size, unix.PROT_READ,
unix.MAP_PRIVATE) ❶
   if err != nil {
      _ = f.Close()
      return nil, err
   }
   return &MemoryMappedFile{f: f, b: b}, nil
}

func (f *MemoryMappedFile) Close() error {
   if err := unix.Munmap(f.b); err != nil { ❷
      _ = f.f.Close()
      return err
   }
   return f.f.Close()
}
```

❶ `unix.Mmap` is a Unix specific Go helper that uses the `mmap` syscall to create a direct mapping between bytes from the file on disk (between 0 and `size` address) and virtual memory allocated by returned `[]byte` array.

❷ `unix.Munmap` is one of the few ways to remove mapping and de-allocate mmap-ed bytes from virtual memory.

The returned byte slice in the `MemoryMappedFile` structure can be read as a regular byte slice acquired via typical means. However, since we marked this memory-mapped location as read-only (`unix.PROT_READ`), writing to such a slice will cause OS to terminate the Go process with the `SIGSEGV` reason (segmentation fault - memory address has accessed the process can't read that). Furthermore, a segmentation fault will also happen if we read from this slice after doing `Close` (`Unmap`) on it.

Before we move on, it's important to note that the Example 5-2 creates memory mapping with specific options: file descriptor of the file to map, read-only flag (`PROT_READ`) and shared flag (`MAP_SHARED`)[15]. Since `mmap` is used so widely, the syscall has many options, allowing for different memory map characteristics[16]. On top of reading or writing permissions and privacy options, one notable option is to create a memory mapping without any backed up file. This can be done by passing the `MAP_ANONYMOUS` flag and NULL file descriptor (0 in case of Go the `unix.Mmap` helper). Such virtual memory is not tied to any specific file, so it is empty from the start. When written to, it creates "anonymous" pages on physical memory. Yet it uses the same memory mapping techniques as the file-backed mapping the code from Example 5-2 allows. Anonymous map is heavily used by OS and programs to create scratch memory that is not meant to be persistent or read from a file, e.g., for heap and stack purposes.

At first glance, the `mmap-ed` byte array looks like a regular byte slice with extra steps and constraints. So what's unique about it? It's best to explain that using an example! Imagine we want to buffer a 600 MB file in the `[]byte` slice, so we can quickly access a couple of bytes on-demand from random offsets of that file. The 600 MB might sound excessive, but such a requirement is commonly seen in databases or caches where reading from a disk on demand might be too slow.

The naive solution without explicit `mmap` could look like in the Example 5-3. Every some instruction, we will look at what the OS memory statistics were telling me about the allocated pages on physical RAM.

Example 5-3. Buffering 600 MB from a file to access 3 bytes from three different locations.

```
f, err := os.Open("test686mbfile.out") ❶
if err != nil {
   return err
}

b := make([]byte, 600*1024*1024)
if _, err := f.Read(b); err != nil { ❷
   return err
}

fmt.Println("Reading the 5000th byte", b[5000])
fmt.Println("Reading the 100 000th byte", b[100000])
fmt.Println("Reading the 104 000th byte", b[104000]) ❸

if err := f.Close(); err != nil {
   return err
}
```

❶ Let's open 600+MB file. At this point, if you would run the `ls -l /proc/$PID/fd` (where `$PID` is the process ID of this executed program) command on a Linux machine, you would see a file descriptors, telling us that this process has used those files. One of the descriptors is a symbolic link to our `test686mbfile.out` file we just opened. The process will hold that file descriptor until the file is closed.

❷ Let's read 600MB into a pre-allocated `[]byte` slice. After the `f.Read` method execution, the RSS of the process shows 621 MB[17]. This means that we need over 600 MB of free physical RAM to run this program. The virtual memory size (VSZ) increased, too, hitting 1.3 GB.

❸ No matter what bytes we access from our buffer, our program will not allocate any more bytes on RSS for our buffer (however, it might need something marginal extra bytes for `Println` logic).

Generally, Example 5-3 proves that without explicit `mmap`, we would need to reserve at least 600MB of pages on physical RAM from the very beginning and keep all of them outside of other processes reach until we finish our desired functionality.

What would the same functionality look like with explicit `mmap`? Let me explain that with the code example presented in Example 5-4 that uses

Example 5-2 abstraction. This should tell you a lot about the memory mapping techniques used on modern computers.

Example 5-4. Memory mapping 600 MB from file to access 3 bytes from three different locations, using Example 5-2.

```
f, err := mmap.OpenFile("test686mbfile.out", 600*1024*1024) ❶
if err != nil {
   return err
}
b := f.Bytes() ❷

fmt.Println("Reading the 5000th byte", b[5000]) ❸
fmt.Println("Reading the 100 000th byte", b[100000]) ❹
fmt.Println("Reading the 104 000th byte", b[104000]) ❺

if err := f.Close(); err != nil { ❻
   return err
}
```

❶ Let's open our test file and memory map 600 MB of its content into the `[]byte` slice. At this point, similar to Example 5-3 we would see a related file descriptor for our `test686mbfile.out` file in the `fd` directory. More importantly, however, if you would execute the `ls -l /proc/$PID>/map_files` (again, $PID is the process ID) command, you would also have another symbolic link to our `test686mbfile.out` file we just referenced. This represents a file-backed memory map.

❷ After this statement, we have the byte buffer `b` with the file content. However, if we would check the memory statistics for this process, the OS did not allocate any page on physical memory for our slice elements[18]. So the total RSS is as small as 1.6 MB, despite having 600 MB of content accessible in `b`! The VSZ, on the other hand, is around 1.3 GB, which indicates the OS telling the Go program that it can access this space.

❸ After accessing a single byte from our slice, we would see an increase in RSS. Around 48-70KB worth of RAM pages for this mapping. This means that OS only allocated a few (10 or so) pages on RAM only when our code wanted to access a single, concrete byte from `b`.

❹ Accessing a different byte far away from already allocated pages triggers the allocation of extra pages. RSS reading would show 100-128KB.

5. If we access a single byte 4000 bytes away from the previous read, OS did not allocate any additional pages. This might be because of a few reasons[19]. For instance, when our program read the file's contents at offset 100 000, the OS already allocated a 4KB page with the byte we access here. Thus RSS reading would still show 100-128KB.
6. If we choose to remove the memory mapping, all the related pages for our will be eventually unmapped from RAM. This means our process total RSS number should be smaller[20].

> **AN UNDERRATED WAY TO LEARN MORE ABOUT YOUR PROCESS AND OS RESOURCE BEHAVIOUR.**
>
> Linux provides extreme statistics and debugging information (and more!) for the current process state. Everything is accessible in the form of special files inside `/proc/<PID>`. The ability to debug each memory, mapping detailed statistics and configuration, was eye-opening. Learn more about what you can do by reading proc (process pseudo-filesystem) documentation.

One of the main behaviours highlighted by the different observations we went through in Example 5-3 and Example 5-4 is called on-demand paging. When the process asks OS for any virtual memory (e.g. using `mmap`), OS will not allocate any page on RAM no matter how large. Instead, OS will only give the process the virtual address range. Further along, when the CPU performs the first instruction that accesses memory from that virtual address range (e.g. our `fmt.Println("Reading the 5000th byte", b[5000])` in Example 5-4), the MMU will generate a page fault. Page fault is a hardware interrupt that is handled by the OS kernel. The OS can respond in various ways:

Allocate more RAM frames

If we have free frames (physical memory pages) in RAM, OS can mark some of them as used and map these to the process that triggered the page fault. This is the only moment when OS actually "allocates" RAM (and increases the `RSS` metric).

De-allocate unused RAM frames and reuse them

If no free frame exists (high memory usage on the machine), OS can remove a couple of frames that belong to file-backed mappings for any process as long as the frames are not currently accessed. As a result, many pages can be unmapped from physical frames before OS has to reach more brutal methods. Still, this will cause other processes to generate another page fault potentially. If this situation happens very often, our whole OS with all processes will be seriously slowed down (memory trashing situation).

Triggering Out-of-memory (OOM) situation

If the situation worsens and all unused file-backed memory-mapped pages are freed, and we still have no free pages, the OS is essentially out of memory. Handling that situation can be configured in the OS, but generally, there are three options:

- OS can start unmapping pages from physical memory for memory mappings backed by anonymous files. To avoid data loss, a particular disk partition can be configured called swap partition (`swapon --show` command will show you the existence and usage in your Linux system). This disk space is then used to backup virtual memory pages from the anonymous file memory map. As you can imagine, this can cause a similar (if not worse) memory trashing situation and overall system slow down[21].
- A second option for OS is to simply reboot the system, generally known as the system level OOM crash.
- The last option is to recover from the OOM situation by immediately terminating a few lower priority processes (e.g. from user space). This is typically done by OS sending the SIGKILL signal. The detection of what processes to kill varies[22], but if we want more determinisms, the system admin can configure specific

memory limits per process or group of processes using, e.g. cgroups[23] or ulimit.

On top of the on-demand paging strategy, it's worth mentioning that OS never releases any frame pages from RAM at the moment of process termination or when it explicitly releases some virtual memory. Only virtual mapping is updated at that point. Instead, physical memory is mainly reclaimed lazily (on-demand) with the help of a page frame reclaiming algorithm (PFRA), we won't discuss in this book.

Generally, the `mmap` syscall seems complex to use and understand. Yet, you need to trust me that `mmap` is a fundamental piece of knowledge when considering Go program optimizations for more memory efficiency. So, let's now compose what we learned into the big picture of how OS manages the RAM and talk about the consequences we developers might observe when dealing with a memory resource.

OS Memory Mapping

The explicit memory mapping presented in Example 5-4 is just one example of the OS's possible memory mapping techniques. Beside rare file-backed mapping, there is almost no need to explicitly use such `mmap` syscall in our Go programs. However, as discussed before, there is a much bigger reason I showed you this: to manage virtual memory efficiently, the OS is transparently using the same technique of page memory mapping for nearly all the RAM!

No matter how the OS is asked to allocate memory on a process' virtual address space, it uses the memory mapping technique we learned in "`mmap` Syscall". The example memory mappings situation is presented in Figure 5-4, which composes a few common page mapping situations we could have in our machine into one picture.

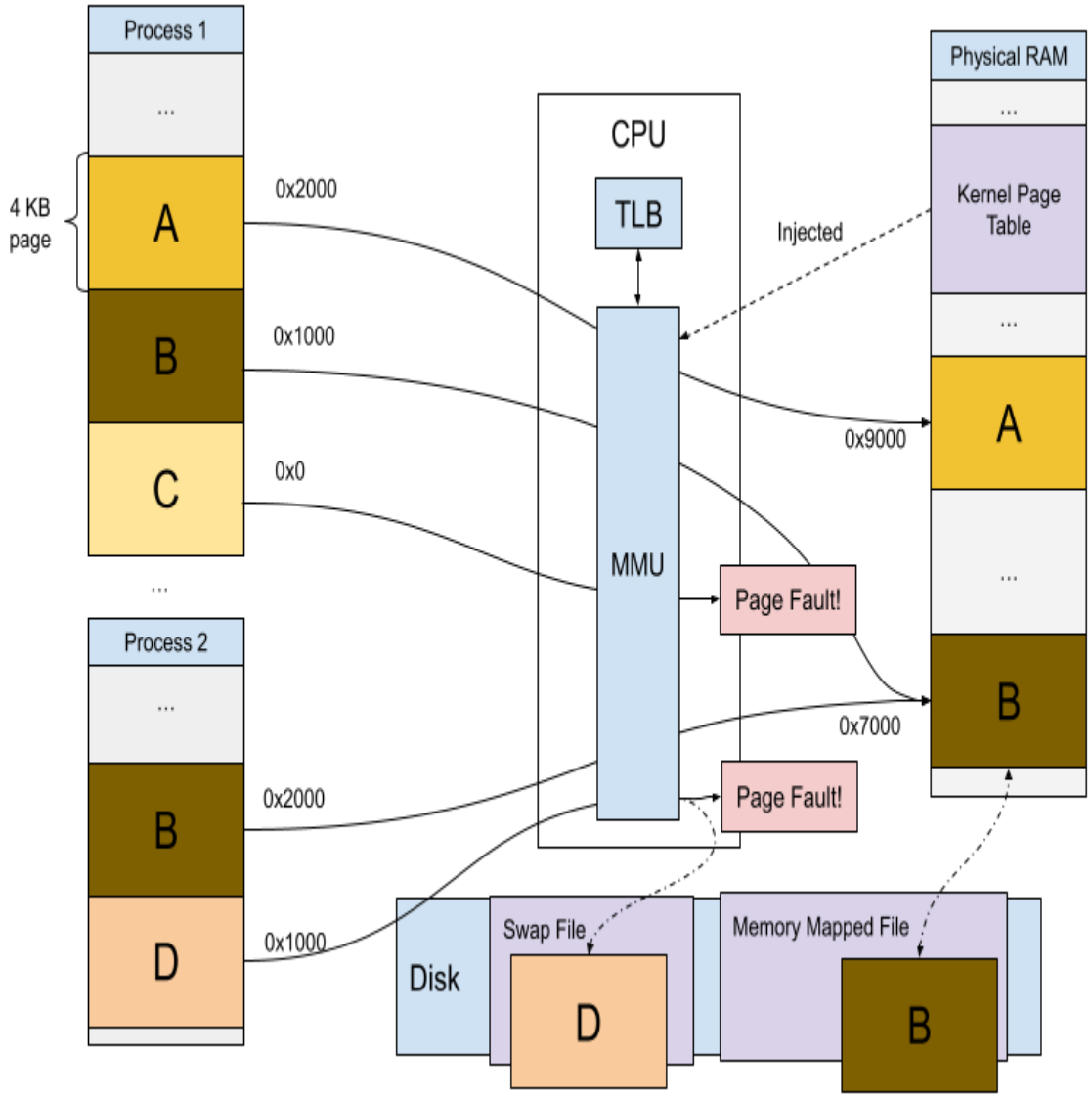


Figure 5-4. Example MMU translation of a few memory pages from two processes' virtual memory.

The situation presented in Figure 5-4 might look complicated, but we have already discussed all those cases. Let's enumerate them from the perspective of process 1 or 2:

Page A

> It represents the simplest case of anonymous file mapping that has already mapped frame on RAM. For example, if process 1 writes or reads a byte from an address between `0x2000` and `0x2FFF`, the MMU will translate the address to 0x9000, plus the required offset and the

CPU will be able to fetch or write it as a cache line to its L-caches and desired register.

Page `B`

It represents a file-based memory page mapped to a physical frame. This frame is also shared with another process since there is no need to keep two copies of the same data since both mappings map the same file on disk.

Page `C`

It is an anonymous file mapping that was not yet accessed. For example, if process 2 writes a byte to address `0x0` and `0xFFF`, a page fault hardware interrupt is generated by the CPU, and OS will need to find a free frame.

Page `D`

It is an anonymous page like `C`, but some data was already written to it. Yet OS seemed to unmap it from RAM because this page was not used for a long time by Process 2, or the system is under memory pressure. The OS backed the data to swap files in the swap partition to avoid data loss. Process 2 accessing any byte from a virtual address between `0x1000` and `0x1FFF` would result in a page fault, which will tell OS to find a free frame on RAM and read page `D` content from the swap file. Only then data can be available to Process 2. Note that such swap logic for anonymous pages is disabled by default on most operating systems.

You should have now a clearer view of OS memory management basics and virtual memory patterns. Let's now go through a list of important consequences those pose on Go (and any other programming language) developers:

Practically speaking, observing the size of virtual memory is never useful

On-demand paging is why we always see larger virtual memory usage (represented by Virtual Set Size—VSS) than residential memory usage (RSS) for a process (for example, the browser memory usage in Figure 5-3). While the process thinks that all pages it sees on virtual address space are in RAM, most of them might be currently unmapped and stored on disk (mapped file or swap partition). In most cases, you can ignore the VSS metric when assessing the amount of memory our Go program uses.

It is impossible to tell precisely how much memory a process (or system) is used in a given time

What metric can we use if the VSS metric does not help assess process memory usage? As we will learn in [Link to Come], for Go developers interested in the memory efficiency of their programs, knowing the current and past memory usage is essential information. It tells how efficient their code is or if their optimizations work as expected. Unfortunately, because of the on-demand paging and memory mapping behaviour we learned in this section, this is currently very hard—we can only roughly estimate. Even the Residential Set Size (RSS) is not entirely accurate. The industry is working on more accurate estimations like Proportional Set Size (PSS) or Working Set Size (WSS) metrics we will discuss in detail in [Link to Come]. Still, given the current OS memory management design, a truly accurate number is impossible to obtain. As a result, don't be surprised if the RSS metric shows a few kilobytes or even megabytes more or less than you expected. Our memory usage metric won't be precise, and we, Go developers, have to adapt to it.

OS memory usage expands to all available RAM

Due to lazy release and page caches, even if all processes and kernel recently used 10% of all physical memory, the system-level RAM usage metric (e.g. in `top` / `htop` tools) might show a high RAM

utilizations[24]. So don't be surprised, and be sure to look at the correct metrics (discussed in [Link to Come]).

Tail latency of our Go program memory access is much slower than just physical DRAM access latency

There is a high price to pay for using OS with virtual memory. In the worst cases, already slow memory access caused by DRAM design (mentioned in "Physical Memory") is even slower. If we stack up things that can happen like TLB miss, page fault, looking for a free page, or on-demand memory loading from disk, we have extreme latency, which can waste thousands of CPU cycles. The OS does as much as possible to make sure those bad cases happen rarely, so the amortized (average) access latency is as low as possible. As Go developers, we have some control to reduce the risk of those extra latencies happening more often. For example, we can use less memory in our programs or prefer sequential memory access (more on that later).

High usage of RAM might cause slow program execution

When our system executes many processes that want to access large quantities of pages, close to RAM capacity, memory access latencies and OS cleanup routines can take most of the CPU cycles. Furthermore, as we discussed, things like memory trashing, constant memory swaps and page reclaim mechanisms will slow the whole system. As a result, if your program latency is high, it is not necessarily doing too much work on the CPU or executing slow operations (e.g. I/O)--it might just use a lot of the memory!

Hopefully, you understand the impact of OS memory management on how we should think about the memory resource. As in "Physical Memory", I could not explain every detail of very complex memory management. This is because the kernel algorithms evolve, and different OS-es manage memory differently. But the information I provided should give you a rough understanding of standard techniques and their consequences. Such a

foundation should also give you a kickstart toward learning more from further materials like Understanding the Linux Kernel book or LWN.net site.

With that knowledge, let's discuss how Go has chosen to leverage the memory functionalities the OS and hardware offer. It should help us find the right optimizations we should try in our TFBO flow if we have to focus on the memory efficiency of our Go program.

Go Memory Management

The programming language task here is to ensure that developers who write programs can create variables, abstractions and operations that use memory safely, efficiently and (ideally) without fuss! So let's dig into how the Go language enables that.

Go uses a relatively standard internal process memory management pattern that other languages (e.g. `C`/`C++`) share, with some unique elements. As we learned in "Operating System Scheduler", when a new process starts, the operating system creates various metadata about the process, including a new, dedicated virtual address space. The OS also creates initial memory mappings for a few starting segments based on information stored in the program binary. Once the process starts, it uses `mmap` or `brk`/`sbrk`[25] to dynamically allocate more pages on virtual memory when needed. The example organization of the virtual memory in Go is presented in Figure 5-5.

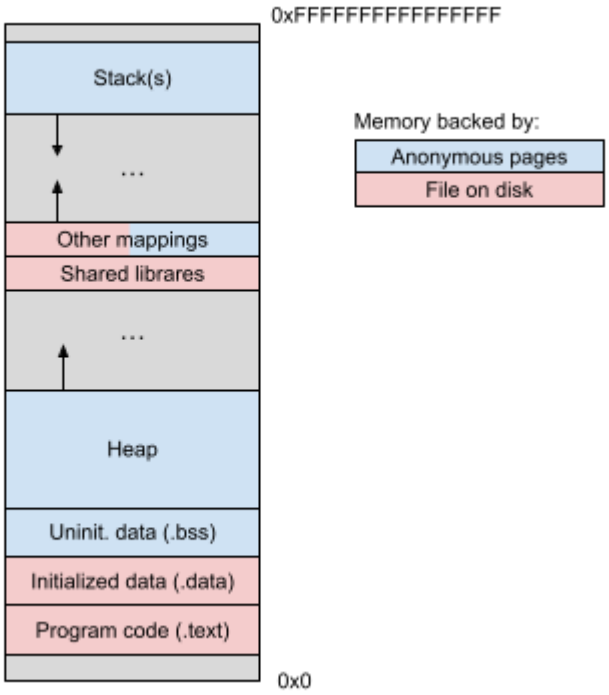


Figure 5-5. Memory layout of an executed Go program in virtual address space.

We can enumerate a couple of common sections:

`.text`, `.data` and shared libraries

Program code and all global data like global variables are automatically memory-mapped by OS when the process starts (whether it takes 1 MB or 100 GB of virtual memory). This data is read-only, backed up by the binary file. Additionally, only a small contiguous part of the program is executed at the time by the CPU, so the OS can keep a minimal amount of pages with code and data in the physical memory. Those pages are also heavily shared (more processes are started using the same binary, plus some dynamically linked shared libraries).

`.bss`

When OS starts a process, it also allocates anonymous pages for uninitialized data (`.bss`). The amount of space used by `.bss` is known in advance—for example, the `http` package defines `DefaultTransport` global variable. While we don't know the value

of this variable, we know it will be a pointer, so we need to prepare 8 bytes of memory for it. This type of memory allocation is called static allocation. This space is allocated once, backed by anonymous pages and is never freed (from virtual memory at least; if swapping is enabled, it can be unmapped from RAM).

Heap

The first (and probably the most important) dynamic segment in Figure 5-5 is the memory reserved for dynamic allocations, typically called the heap (do not confuse it with the data structure called the same). Dynamic allocations are required for program data (e.g. variables) that change based on dynamic input when the application is running. As a result, such allocations are unknown in advance and must be stored in memory for an unpredictable amount of time. The OS prepares the initial number of anonymous pages for the heap when the process starts. After that, the OS gives some control over that space to the process. It can then increase or decrease its size using the `sbrk` syscall or by preparing or removing extra virtual memory using `mmap` and `unmmap` syscalls. It's up to the process to organize and manage the heap in the best possible way, and different languages do that differently:

- C forces the programmer to manually allocate and free memory for variables (using `malloc` and `free` functions).
- `C++` adds smart pointers like std::unique_ptr and std::shared_ptr, which offers simple counting mechanisms to track object lifecycle (reference counting)[26].
- Rust has a powerful memory ownership mechanism, but it makes programming much harder for non-memory critical code areas[27]
- Finally, languages like Python, C#, Java, and more implement advanced heap allocators and garbage collector mechanisms. Garbage collectors periodically check if any memory is unused and can be released.

In this sense, Go is closer to Java with memory management than C. Go implicitly (transparently to the programmer) allocates memory that requires dynamic allocation on the heap. For that purpose, Go has its own unique components (implemented in Go and assembler), called Allocator and Garbage Collection (GC). We will discuss those below in "Go Allocator" and "Garbage Collection".

MOST OF THE TIME, IT'S ENOUGH TO OPTIMIZE THE HEAP USAGE

Heap is the memory that usually stores the largest amounts of data in physical memory pages. It is so significant that in most cases, it's enough to just look at heap size to assess the Go process memory usage. On top of that, the overhead of heap management with runtime garbage collection is significant too. Both make the heap our first choice when looking for efficiency problems.

Manual process mappings

Both Go runtime and developer writing Go code can manually allocate additional memory-mapped regions (e.g. using our Example 5-2 abstraction). Of course, it's up to the process what kind of memory mapping to use (private or shared, read or write, anonymous or file-backed), but all those have a dedicated space in the process's virtual memory presented in Figure 5-5.

Stack

The last section of the Go memory layout is reserved for function stacks. The stack is a simple yet fast structure allowing accessing values in Last In, First Out (LIFO) order. Programming languages use those to queue all the elements (e.g. variables) that can use automatic allocation. As opposed to dynamic allocations fulfilled by the heap, automatic allocations work well for local data like local variables, function input, or return arguments. Allocations of those elements can be "automatic" because the compiler can deduce their lifespan before the program starts.

Some programming languages might have a single stack or stack per thread. Go is a bit unique here. As we learned in "Go Runtime Scheduler", the Go execution flow is designed around goroutines. Thus Go maintains a single dynamically-sized stack per Go routine. This might even mean hundreds of thousands of stacks. Whenever the goroutine invokes another function, we can push its local variables and arguments to stack. As soon as we leave the function, we can pop those elements from the stack. If stack structures require more space than what's reserved on virtual memory, Go will ask the OS for more memory attributed to the stack segment, e.g. via the `mmap` syscall.

Stacks are incredibly fast as there is no extra overhead to figure out when memory used by certain elements must be removed (usage tracking). Thus ideally, we write our algorithms so that they allocate primarily on the stack instead of the heap. Unfortunately, in many cases, this is impossible due to stack limitations or when the variable has to live longer than the function's scope. Therefore, the compiler decides which data can be allocated automatically (on the stack) and which must be allocated dynamically (on the heap). This process is called escape analysis and will be discussed in [Link to Come].

How do I know whether a variable is allocated on the heap or the stack? From a correctness standpoint, you don't need to know. Each variable in Go exists as long as there are references to it. The storage location chosen by the implementation is irrelevant to the semantics of the language.

The storage location does have an effect on writing efficient programs.

—The Go Team, Go: Frequently Asked Questions (FAQ)

All the mechanisms discussed above (except manual mappings) are helping Go developers from caring functional aspects of allocating and releasing memory. That is a huge win. For example, when we want to make some HTTP calls, we might want to create an HTTP client using a standard library. Let's say we write a `client := http.Client{}` code

statement. As a result of Go's memory design, we can immediately start using `client`, focusing on our code's functionality, readability, and reliability. In particular:

- We don't need to ensure that OS have a free virtual memory page to hold the `client` variable. Likewise, we don't need to find a valid segment and virtual address for it. Both will be done automatically by the compiler (if the variable can be stored on the stack) or runtime allocator (dynamic allocation on the heap).
- We don't need to remember to release memory kept by the `client` variable when we stop using it. Instead, suppose the `client` would go beyond code reach (nothing references it). In that case, the data in Go will be released—immediately when stored on the stack or in the next GC execution cycle if stored on the heap (more on that in "Garbage Collection"). Such automation is much less error-prone to potential memory leaks ("I forgot to release memory for `client`") or dangling pointers ("I released memory for `client` but actually some code still use it").

TRANSPARENT ALLOCATIONS MEAN THERE IS A RISK OF OVERDOING THEM.

Allocations are implicit in Go, making coding much easier, but there are trade-offs. One is around memory efficiency—if we don't see explicit memory allocations and releases, it's easier to miss apparent high memory usage in our code[28].

It is similar to going shopping with cash versus a credit card. It is more likely you will overspend with a credit card than with cash since you don't see that money flowing. With a credit card, money spent is almost transparent to us—the same is with allocations in Go.

Go is a very productive language because, when programming, we don't need to worry about where and how the data held by our variables and abstractions are stored. Yet sometimes, when our measurements indicate efficiency problems, it's useful to have a basic awareness of the parts of our

program that might allocate some memory and how this occurs and how the memory is released. So let's uncover that in the next sub-sections.

Values, Pointers and Memory Blocks

Let's put this straight before we start—you don't need to remember what type of statements trigger memory allocation, where (on stack or heap) and how much memory was allocated. As we will learn in [Link to Come] and [Link to Come], many robust tools can tell us all those pieces of information accurately and quickly. In most cases, within seconds, we can learn what code line and roughly how much was allocated. Thus, there is generally a theme that everyone repeats—we should not guess that information because humans tend to guess wrong.

This is generally true, but there is no harm in building some basic allocation awareness that might make us more effective while using those tools to analyze the memory usage of different parts of our program. The aim is to build a healthy instinct on what pieces of code can potentially allocate the suspicious amount of memory and where we need to be careful. Let's go through a list of a few popular operations that will allocate memory in Go (either statically, on the stack or the heap):

- Variable declarations (e.g. using `var`).
- Explicit call to `new` or `make` built-in functions or implicit `struct` creation with assignments like `:=http.Client{}` and `:= &http.Client{}`.
- Creation of maps, slices and anonymous functions.
- Concatenating non-constant strings.
- Calling the built-in `append` on the slice that has fully used capacity.
- Adding a new key to the map with a full underlying hash table.
- Converting bytes to string or vice versa (exceptions apply!).

To understand our own code, we must contain and often individually name the pieces of data our program will use for operations. This is done using variables and struct fields that have human-readable names. To make it even easier for compiler and developers, we describe that data additionally using a Go type system. The example memory representation of three variables can be seen in Figure 5-6.

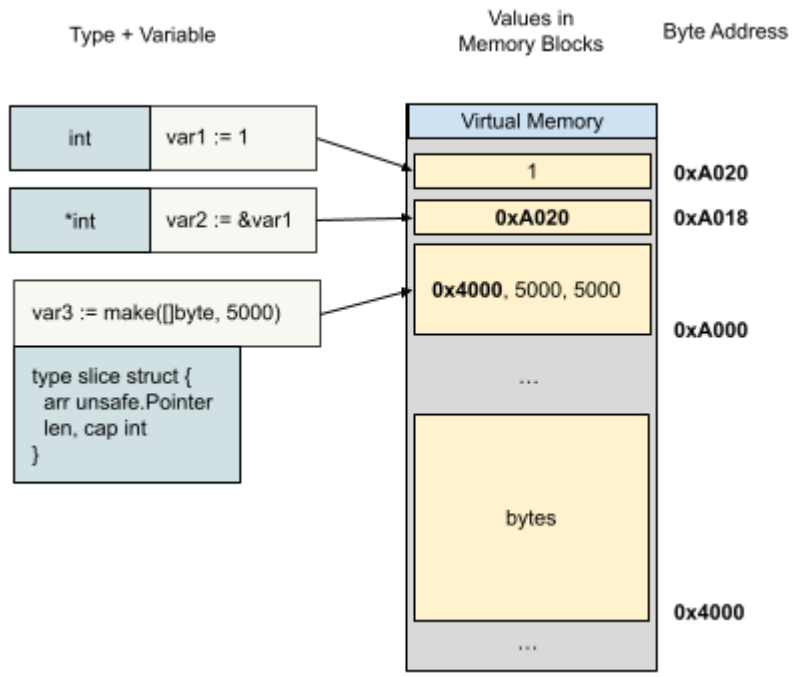


Figure 5-6. Representation of three variables allocated on the process's virtual memory.

Go is value oriented, rather than reference oriented as most managed runtime language. This is why I recommend thinking about Go variables (and also structure fields) as a simple label attached to a (typed) piece of data in memory which we often call "a value". There is no exception to this rule. If there is a label (variable), there is a corresponding value. Many labels can be attached to a single value.

Whenever we create a label (e.g. declare a variable), we have to allocate a memory block in virtual memory for that value. A memory block always holds a contiguous range of bytes representing the label's value (of the size of that label type). For instance, in Figure 5-6, we can see the integer value `1` connected to the `var1` label (variable). It requires a single memory block of integer size (64 bits, so 8 bytes).

On the other hand, the `var2` label is a pointer to the integer type. At first, it might be confusing that variables can contain some special value called a pointer if you put `*` next to its type when defining it. However, this is not that special—the `*int` type is just an `uintptr` or `unsafe.Pointer`, so simply a 64-bit unsigned integer representing a virtual memory address that points to the value holding `int` type[29]. The pointer can also be `nil` (Go's NULL value), a special value indicating that the pointer does not point to anything. In Figure 5-6, we can see that the `var2` label is attached to a normal memory block that contains a value, too—a memory address for the memory block referenced by `var1`.

This is also consistent with more complex types as seen in Figure 5-6. The `make([]byte, 5000)` creates a byte slice with pre-allocated 5000 elements, which is then assigned to the `var3` variable (following our thought process—a `var3` label was attached to this value).

MEMORY STRUCTURE FOR GO SLICE

Slice allows easy dynamic behaviour of the underlying array of a given type. A slice data structure requires a memory block that can hold three 64-bit integers to store a virtual memory address of the array and both length and capacity information for this slice[30]

The `var3` label references a value of the slice type with three fields, which fits in a 24 bytes memory block. When we look closer, there is nothing special in this type other than one of the fields (which is just another label!) attached to the pointer type value. That holds the memory address to another essential memory block containing a 5000 elements byte array we just asked for. To sum up, a variable always contains some direct value, which may indirectly reference other values—each value having its contiguous memory block!

STRUCTURE PADDING

The slice structure with three 64-bit fields requires 24 bytes long memory block. But the memory block size for a structure type is not always the sum of the size of its fields!

Smart compilers like in Go might attempt to align type sizes to the typical cache lines or the OS or internal Go allocator page sizes. For this reason, Go compilers sometimes add paddings between fields. We will explore optimizations that use that knowledge in [Link to Come].

The above information should give you some memory awareness and will be invaluable for the next, more practical chapters. We learned about Go values, pointers and memory blocks that might be stored statically, in manual mappings, stack or heap. From all of those, it's the heap that might be the most interesting to dive deeper into.

Most program memory allocations are only known in runtime; thus, dynamic allocation (in a heap) is needed. Therefore, when we optimize memory in Go programs, 99% of the time, we just focus on the heap. Go comes with two important runtime components, Allocator and Garbage Collection (GC), responsible for heap management. Those components are non-trivial pieces of software that often introduce certain waste in terms of extra CPU cycles by program runtime (so latency) and some memory waste. Given its non-deterministic and non-immediate memory release nature, it is worth discussing this in detail. So let's do that in the next two sections.

Go Allocator

It's far from easy to manage the heap as it poses similar challenges as the OS has towards physical memory. The OS has many different processes, and each wants a few (dynamically sized!) segments of its virtual memory to be allocated to physical memory efficiently. Go runtime (or the runtime of any other language) has a similar problem. Executed code we write requires someone to dynamically allocate and release various memory blocks of different sizes.

The Go Allocator is a piece of internal Go code maintained by the Go team. As the name suggests, it can dynamically (in runtime) allocate the memory blocks behind our program's data like variables, instances of structs, functions, types, slices or maps into the heap.

During compilation, the Go compiler performs a complex stack escape analysis to detect if the memory for objects can be automatically allocated (discussed in [Link to Come]). If yes, then it adds appropriate CPU instructions that store related memory blocks in the stack segment of the memory layout. However, the compiler can't avoid putting most of our memory on the heap in most cases. In these cases, it generates different CPU instructions that invoke the Go Allocator code.

The Go Allocator is responsible for bin packing the memory blocks in limited virtual memory space and asks for more space from OS if needed. Ideally, as seldom as possible because this operation is relatively slow. Generally, the Go developer can live without learning details about Go Allocator internals, so let's go through just high-level information about its algorithm. If nothing else, this should tell us why we see some of the surprising behaviour with Go memory characteristics. For example, the Allocator in Go 1.18:

- Is based on a custom Google `C++` `malloc` implementation called TCMalloc.
- Is OS virtual memory page aware, but it operates as if OS pages were 8KB large.
- It creates and manages spans (`mspan` entries) which holds one or multiple 8KB pages. Each span is created for a certain size class memory block. In Go 1.18, there are 67 different size classes (size buckets), the largest being 32 KB big.
- Memory blocks for objects that do not contain a pointer are marked with the `noscan` type, making it easier to track nested objects in the GC phase.

- Objects with over 32 KB memory block (e.g. 600MB byte array) are treated specially (allocated directly without using spans).
- Spans of different classes are stored and managed in a custom data structure called `mheap` in the complex hierarchy and linked list to be managed efficiently. This structure is also optimized for multi-threaded allocations (smart locking) since the goroutine needs to allocate objects simultaneously.
- If runtime needs more virtual space from OS for heap, it allocates a bigger chunk of memory at once (at least 1MB), which amortizes the latency of the syscall that resizes the heap. To achieve so, Go Allocator asks for more memory using `mmap` with private, anonymous pages, which are initialized by zero and allocated by OS only when accessed[31].

All of the above is constantly changing, with the open-source community and Go Team adding various small optimizations and features. The implementation has to be also heavily aware of the OS algorithms (pages, memory mapping, overcommitment) we discussed in "OS Memory Management". As you can imagine, if we combine all the layers of memory access (DRAM, L-caches, OS, and Go Allocator), the behaviour of how we see the physical memory used can be sometimes surprising.

They say one code snippet is worth a thousand words, so let's visualize and explain some of those allocation characteristics caused by a mix of Go, OS and hardware using an example. Example 5-5 shows the same functionality as in Example 5-4, but instead of explicit `mmap` we will rely on Go memory management. Similar to what we did in "`mmap` Syscall" examples, we will go step by step to check the memory resource usage after each statement[32].

Example 5-5. Allocation of large `[]byte` slice following by different access patterns.

```
b := make([]byte, 600*1024*1024) ❶
b[5000] = 1
b[100000] = 1
b[104000] = 1 ❷
for i := range b { ❸
```

```
    b[i] = 1
}
```

❶ `b` variable is declared as a `[]byte` slice. As we learned in "Values, Pointers and Memory Blocks", a slice requires 24 bytes for metadata and pointer (direct value), but let's not worry about that here. We might have a much bigger chunk of memory to manage. This is because the following `make` statement is tasked to create that slice with 600 MB of data (~600 million elements in the array). This memory block is allocated on the heap[33]. If we would analyze this situation closely, the Go Allocator seemed to create three contiguous anonymous mappings for that slice with different (virtual) memory sizes: 2MB, 598 MB, and 4MB. (The total size is usually bigger than the requested 600 MB because of the details about Go Allocator we mentioned above). Let's summarize the interesting statistics:

- The RSS for three memory mappings used by our slice: 548 KB, 0 KB, and 120 KB (far from VSS numbers).

- Total RSS of the whole process shows 21 MB. 99% of this comes from outside the heap.

- Go reports 600.15 MB of the heap size (despite RSS being significantly lower).

❷ Only after we start accessing the slice elements (either by writing or reading) will the OS start reserving actual physical memory surrounding those elements. Our statistics:

- The RSS for three memory mappings: 556 KB, (still) 0 KB, and 180 KB (only a few KB more than before accessing).

- Total RSS still shows 21 MB.

- Go reports 600.16 MB of the heap size (actually a few KB more, probably due to background go routines).

❸ After we loop over all elements to access it, we will see that OS mapped on-demand all pages for our `b` slice on physical memory. Our statistics prove this:

- The RSS for three memory mappings: 1.5 MB, (fully mapped) 598 MB, and 1.2 MB.

- Total RSS of the whole process shows 621.7 MB (finally same as heap size).

- Go reports, the same 600.16 MB of the heap size.

This example might feel similar to Example 5-3 and Example 5-4, but it's a bit different. Notice that in our example here in Example 5-5, there is no (explicit) file involved that could store some data if the page is not mapped. We also utilize Go Allocator to organize and manage different anonymous page mappings most efficiently, whereas in Example 5-4, Go Allocator is unaware of that memory usage.

Hopefully, going through Example 5-5 was insightful (it was for me!). It shows the somewhat surprising behaviour of the Go allocator and OS at first glance.

INTERNAL GO RUNTIME KNOWLEDGE VS OS KNOWLEDGE

Go Allocator tracks certain information that can be collected through different observability mechanisms discussed in [Link to Come].

Be mindful when using those. In the above example, we saw that the heap size tracked by Go Allocator was significantly larger than the actual amount of memory used on physical RAM (RSS)[34]! Similarly, the memory used by explicit `mmap` as in Example 5-4 is not reflected in any Go runtime metrics. This is why it's good to rely on more than one metric on our TFBO journey, as discussed in [Link to Come]

Probably the most important learning from this section is that we should refrain from trying to understand every tiny detail of heap and RSS changes in our programs. The behaviour of Go heap management backed up by on-demand paging tends to be indeterministic and fuzzy for both humans and the tools we will describe in [Link to Come]. We cannot control it directly either. For instance, if you would try to reproduce the Example 5-5 on your machine, you would most likely observe different mappings, more or less different RSS numbers (with a tolerance of few MBs) and different heap sizes. It all depends on the Go version you build a program with, the kernel version, the RAM capacity and model and the load on your system. This poses important challenges to the assessment step of our TFBO process, which we will discuss in [Link to Come].

DON'T BE BOTHERED BY A SINGLE MEMORY USAGE INCREASE FOR THE WHOLE PROCESS IN THE ORDER OF MEGABYTES OR LESS

Don't try to understand where every hundred bytes or kilobytes of your process RSS memory came from. In most cases, it is impossible to tell or control on that low level. Heap management overheard, speculative page allocations by both OS and the Go Allocator, dynamic OS mapping behaviour, and eventual memory collection (we will learn in the next section) make things indeterministic on such a "micro" kilobytes level. Even if you spot some pattern in one environment, it will be different in others unless we talk about bigger numbers like hundreds of megabytes or more!

The solution is that we have to adjust our mindset. There will always be a few unknowns. What matters is to explain bigger unknowns which contribute the most to the potential too high memory usage situation. First, we can find the function that uses the most of the memory based on profiles explained in [Link to Come]. Then, we can try measuring the exact allocations from that specific isolated code and compare across runs (explained in [Link to Come]). We can do that until we are happy with the memory.

So far, we have discussed how to efficiently reserve memory for our memory blocks through Go Allocator and how to access it. However, we cannot just reserve more memory indefinitely if there is no logic for removing the memory blocks our code does not need anymore. That's why it's critical to understand the second part of heap management responsible for releasing unused objects from the heap—the GC. Its important characteristics probably have the biggest implications on memory efficiency and (which might be surprising!) the CPU usage and latency of our Go programs! Let's explore that in the next section.

Garbage Collection

The second part of heap management is similar to vacuum cleaning your house. It is related to a process that removes the proverbial garbage—so unused objects from the heap. This process is generally more complex than it seems, and in this section, you will learn why. Let's start by describing the high-level algorithm.

Generally speaking, Garbage Collector (GC) is an additional background routine that executes "collection" at certain moments. The cadence of collections is critical:

- If the GC is running less often, we risk allocating a significant amount of new memory space without the ability to reuse memory used by unused objects (garbage).

- If the GC is running too often, we risk spending a majority of program time and CPU on GC work instead of moving our functionality forward. As we will learn later, the GC is relatively fast but can directly or indirectly impact the execution of other goroutines in the system, especially if we have a large number of objects in a heap (if we allocate a lot).

The interval of the GC runs is not based on time. It is currently defined by a single configuration variable called `GOGC`, which controls the "GC percentage". It is set to 100 by default, which means that the next GC collection will be done when the heap size expands to 100% of what it was at the end of the last GC cycle. GC has a pacing algorithm that estimates when that goal will be reached based on current heap growth. Programmers can also trigger another GC collection on demand by invoking the `runtime.GC()` function in their code.

> *Go famously offers a single option for tuning the Go garbage collector: GOGC (or runtime/debug.SetGCPercent). This option offers a direct trade-off between CPU and memory: it directly dictates the amount of memory overhead available to the garbage collector. Less memory overhead means more frequent garbage collection cycles and more CPU spent on GC. Conversely, more memory overhead means fewer regular cycles and more CPU for the application.*
>
> —Michael Knyszek, Proposal: Soft memory limit

Go implementation of the GC can be described as the concurrent, non-generational, tri-colour mark and sweep collector implementation. Whether invoked by the programmer or by the runtime-based "GOGC" option, the `runtime.GC()` implementation comprises a few phases. The first one is a Mark phase that has to:

1. Perform a "Stop the World" (STW) event to inject an essential Write Barrier (a lock on writing data) to all goroutines. Even though STW is relatively fast (10-30 microseconds in average case), it is quite impactful—it suspends the execution of all goroutines in our process for that time.

2. It tries to use 25% of the CPU capacity given to the process to concurrently mark all objects in the heap that are still in use.
3. Terminate marking by removing Write Barrier from the goroutines. This requires another STW event.

After the `Mark` phase, the GC function is generally completed. But there is another important element. The sweeping phase is then performed to release objects which were not marked as in-use. Interestingly this is not a background routine. Instead, sweeping is done lazily. Every time a goroutine wants to allocate memory through the Go Allocator, it must perform a Sweeping work first, then allocate. This is counted as an `allocation` latency, even though it is technically a garbage collection functionality—worth noting!

Generally speaking, the Go Allocator and Garbage Collection compose a sophisticated implementation of bucketed object pooling, where each pool of slots of different sizes are prepared for incoming allocations. When an allocation is not needed anymore, it is eventually released. The memory space for this allocation is not immediately released to the OS since it can be assigned to another incoming allocation soon (this is similar to the pooling pattern using `sync.Pool` we will discuss in [Link to Come]). When the number of free spans is big enough, Go releases memory to the OS. But even then, it does not necessarily mean that runtime deletes mapped regions straight away. On Linux, Go runtime typically "releases" memory through madvise syscall with `MADV_DONTNEED` argument by default[35]. This is because our mapped region might be needed again pretty soon, so it's faster to keep them just in case and ask the OS to take them back only if other processes require this physical memory.

> *Note that, when applied to shared mappings, `MADV_DONTNEED` might not lead to immediate freeing of the pages in the range. The kernel is free to delay freeing the pages until an appropriate moment. The resident set size (RSS) of the calling process will be immediately reduced, however.*
>
> —Linux Community, madvise(2) — Linux manual page

With the theory behind the GC algorithm, it will be easier for us to understand in Example 5-6 what happens if we will try to clean the memory used for the large, 600 MB byte slice we created in Example 5-5.

Example 5-6. Memory release (de-allocation) of large slice created in Example 5-5.

```
b := make([]byte, 600*1024*1024)
for i := range b { ❶
   b[i] = 1
}

b[5000] = 1 ❷
b = nil ❸
runtime.GC() ❹

// Let's allocate another one, this time 300 MB!
b = make([]byte, 300*1024*1024)
for i := range b { ❺
   b[i] = 2
}
```

❶ As we discussed in Example 5-5, the statistics after allocating a large slice and accessing all elements might look as follows:

- Slice is allocated in three memory mapping with the corresponding virtual memory size (VSS) numbers: 2MB, 598 MB, and 4MB.
- The RSS for three memory mappings: 1.5 MB, (fully mapped) 598 MB, and 1.2 MB.
- Total RSS of the whole process shows 621.7 MB.
- Go reports 600.16 MB of the heap size.

❷ After the last statement where data from `b` is accessed, even before `b = nil`, the `Mark` phase of GC would consider `b` as a "garbage" to clean. Yet, the GC has its own pace; thus, immediately after this statement, no memory will be released—memory statistics will be the same.

❸ In typical cases when you no longer use the `b` value, and function scope ends, or you will replace `b` content with a pointer to a different object, there is no need for `b = nil`--GC will know that the array pointed by `b` is garbage. Yet sometimes, especially on long-living functions (e.g. a goroutine that performs background job items delivered by the Go channel), it is useful to set the variable to `nil` to make sure the next GC run will mark it for cleaning earlier.

❹ In our tests, let's invoke the `GC` manually to see what happens. After this statement, the statistics will look as follows:

- All three memory mappings still exist, with the same VSS values (this proves what we mentioned about the Go Allocator is only advising on memory mappings, not removing those straight away)!
- The RSS for three memory mappings: 1.5 MB, 0 (RSS released), and 60 KB.
- Total RSS of the whole process shows 21 MB (back to the initial number).
- Go reports 159 KB of the heap size.

❺ Let's allocate another twice smaller slice. Memory statistics below prove the theory that Go will try to reuse previous memory mappings!

- Same three memory mappings still exist, with the same VSS values.
- The RSS for three memory mappings: 1.5 MB, 300 MB, and 60 KB.
- Total RSS of the whole process shows 321 MB.

- Go reports 300.1 KB of the heap size.

As we mentioned earlier, the beauty of GC is that it makes programmer life simpler thanks to care free allocations, memory safety and solid efficiency for most applications. Unfortunately, however, it also makes our life a bit harder when our program does violate our efficiency expectations (e.g. stated in the RAER discussed in "Efficiency Requirements Should be Formalized"), and the reason is not what you might think.

The main problem with the Go Allocator and GC pair is that they hide the root cause of our memory efficiency problems—in almost all cases, our code allocates too much memory!

> *Think of a garbage collector like a Roomba: Just because you have one does not mean you tell your children not to drop arbitrary pieces of garbage onto the floor.*
>
> —Halvar Flake, Twitter

Let's explore the potential symptoms we might notice in Go when we are not careful with the number and type of the allocations:

A CPU overhead

> First and foremost, the GC must go through all the objects stored on the heap to tell which ones are in-use. This can use a significant portion of the CPU resource, especially if there are many objects in heap[36]. This is also very visible if the objects stored on the heap are rich in pointer types, which forces GC to traverse them to check if they don't point to an object that was not yet marked as "in use". Given the limited CPU resources we have in our computers, the more work we have to do for GC, the less work we can perform towards the core program functionality, which translates to higher program latency. On top of that, the more we allocate, the `GOGC` percentile target makes GC run much more often, which multiplies that effect.

In platforms with garbage collection, memory pressure naturally translates into increased CPU consumption.

—Google Teams, Site Reliability Engineering

Additional increase in program latency

CPU time spent on GC is one thing, but there is more. First, the STW event performed twice slows down all goroutines. This is because the GC must stop all goroutines and inject (and then remove) a Write Barrier. It also prevents some goroutines which have to store some data in memory from doing any further work for the moment of GC marking.

There is also a second, often missed effect. The GC collection runs are destructive to the L-caches (discussed in "Hierachical Cache System") efficiency.

For your program to be fast, you want everything you're doing to be in the cache. (...) There are technical, physical reasons in the silicon why running around allocating memory, throwing it away and GC cleaning that for you, going to not only slow your program down, because GC is doing its work, but it slows the rest of your program down, because it kicked everything out of [CPU] cache.

—Bryan Boreham, Make your Go go faster!

Memory overhead

Last but not least, at the current moment, there is no way to set in GC any upper bound for the memory space used by Go[37]. This means that we have to often implement on our side checks against unbounded allocations (e.g. rejecting reading too big HTTP body requests). We will learn more about that in [Link to Come].

The main problem with the GC is that even if our allocations are bounded (i.e. from the code perspective, we only allocate a maximum of 200 MB in one loop iteration and then discard it before the next loop iteration), the collection phase is eventual. This means that we might be unable to release those memory blocks before new allocations come in.

This results in memory overhead. Changing the `GOGC` option to run GC less often only amplifies the problem but might be a good trade-off if you optimize the CPU and have spare RAM on your machines.

In extreme cases, our program might even leak memory if the GC is not fast enough to deal with all new allocations!

The GC can have sometimes surprising effects on our program efficiency. Hopefully, after this section, you will be able to notice when you are affected. Let's now discuss what we can do when we see in our profiles (discussed in [Link to Come]) that GC is being a CPU bottleneck or when we are running out of memory on our machines.

Before mentioning the more advanced method, let's highlight the superior way of dealing with problems in this space. Using the TFBO method, detect places where we allocate the most and try reducing the number and size of the allocations. Then benchmark to check if your assumptions were correct.

THE SOLUTION TO THE MOST MEMORY EFFICIENCY ISSUES.

Produce less garbage!

It's easy to over-allocate memory in Go. This is why the "three Rs" discussed in "The Three R's Optimization Method" method should be your first option.

I would personally recommend trying the 3R method first before doing anything else. Yet, in worst cases, we might be unable to reduce the number and size of the allocations in our code. For example, perhaps we use a third-party library we cannot change, or our problem domain requires much memory that cannot be balanced out with more CPU time due to latency requirements. In those cases, there are a few advanced techniques we can consider.

Optimize the structure of the allocated object

If we cannot reduce the number of allocations, perhaps we could reduce the number of pointers in our objects! It is often confusing for new Go developers what should be preferred: `[]int` or `[]*int`. The former option is superior from the GC perspective as it allows GC to spend much less CPU time traversing those pointers.

However, avoiding pointers is not always possible, given popular structures like `time.Time`, `string` or slices contain pointers. Especially `string` does not look like it, but it is just a special `[]byte`, which means it has a pointer to a byte array. In extreme cases, in certain conditions, it might be worth changing `[]string` into `offsets []int` and `bytes []byte` to make it a pointer-free structure!

Another widespread example where it's easy to get very pointer-rich structures is when implementing data structures that are supposed to be marshalled and unmarshalled to different byte formats like JSON, YAML or protobuf. It is tempting to use pointers for nested structures to allow optionality of the field (ability to differentiate if the field was set or not). Some code generation engines like Go protobuf generator put all fields as pointers by default. This is fine for smaller Go programs, but if you use a lot of such objects (which is common, especially if you use those messages over the network), you might consider trying to remove pointers from those data structures on the critical path.

This optimization is better for GC and can make the data structure more L-cache-friendly, decreasing the program latency. It also increases the chances that the compiler will put data structure on the stack instead of the heap.

Tuning of `GOGC` option

Adjusting the `GOGC` option from the default 100% value might positively affect your program efficiency. Moving the next GC collection to happen sooner or later (depending on what you need) might be beneficial. Unfortunately, it requires lots of benchmarking to

find the right number. It also does not guarantee this tuning will work well for all possible states of your applications. On top of that, this technique has poor sustainability if you change the critical path in your code a lot. Every change requires another tuning session. This is why some bigger companies like Google and Uber invest in automated tools that adjust `GOGC` automatically, in runtime!

Triggering GC and free OS memory manually

In extreme cases, you might want to experiment with manually triggered GC collections using `runtime.GC()`. For example, we might want to do that after the operation that requires a huge number of allocations and just before you will need more. Arguable for these cases, the 3R method explained in the below section might fit better. 3R recommends reusing the bytes from previous runs explicitly. Yet, if reuse is not possible, extra `runtime.GC()` call might help. Again, benchmarks are a must-have to double-check for negative side effects[38].

Memory ballasting

Ballasting allows stabilizing the GC by allocating (but never accessing) a large object. As a result, the GC will reuse the allocated memory for other memory blocks and will not release the underlying memory pages back to the OS. This is great for applications like memory-heavy proxies that, without traffic, require almost no memory but, during heavy traffic moments, require GBs of RAM. This helps with both operational aspects of our application (deterministic memory usage helping this capacity plannings) as well less GC overhead. This leverages the Go Allocator and OS allocation characteristic we see in Example 5-5.

Allocating objects off-heap

We mentioned trying to allocate objects on the stack first instead of the heap. The compiler does this, and we will discuss this in [Link to Come]. The `off-heap` allocation is something else. It means

allocating objects in a different segment of Go memory that is outside of Go runtime responsibility to manage. We can achieve that with the explicit `mmap` syscall we learned in "`mmap` Syscall". Some even tried calling C functions like `jemalloc` through the CGO. While possible, we need to acknowledge that doing this can be compared to reimplementing parts of the Go Allocator from scratch, not to mention dealing with the manual allocations and lack of memory safety. It is the last thing you might want to try for the ultimate high-performance Go implementation!

It's essential to benchmark and measure all the effects of the above optimizations before adding that to your production code. Some of those can be considered tricky to maintain an unsafe if used without extensive tests. On top of that, you should only make those changes to the critical areas (e.g. only for the code that allocates the most).

We learned about GC, and the existence of mode advanced optimization methods to mitigate some memory issues. Now, let's dive deeper into what every Go programmer should know: How to reduce allocations in the critical code parts effectively. Let's look at that in our last section for this chapter.

The Three R's Optimization Method

The three R's technique is an excellent method to reduce waste. It is generally applicable for all computer resources, but it is often used for ecology purposes to reduce literal waste. Thanks to those three ingredients: Reduce, Reuse and Recycle, we can reduce the impact we, humans, have on the Earth's environment and ensure sustainable living.

On FOSDEM 2018, I saw Bryan Boreham's amazing talk, where he described using this method to mitigate memory issues. Indeed, the three R's method is especially effective against memory allocations, which is the most common source of the memory efficiency and GC overhead problems. So, let's explore each " R " component and how they can help.

Reduce

There is almost always room to reduce allocations—look for the waste! Some ways to reduce the number of objects our code puts on the heap are obvious (reasonable optimizations like pre-allocations of slices we saw in Example 1-4). Other optimizations require certain trade-offs—typically more CPU time or less readable code (deliberate optimizations like streaming explained [Link to Come]). Some are more extreme, like string interning, where we avoid operating on the `string` type by providing a dictionary and using much smaller, pointer-free integers representing the ID of the string. Or perhaps explicit, unsafe conversion from `[]byte` to `string` (and vice versa) without copying memory, which potentially saves allocations, but if done wrongly can keep more memory in a heap (more on that in [Link to Come]). Finally, ensuring a variable does not escape to the heap can also be considered to reduce allocations.

There are unlimited different ways you could reduce those allocations. I already mentioned a few ideas; you will learn more in [Link to Come]. However, if you made it so far in this chapter, you are already in a better position to find optimization that will fit you, thanks to knowing how hardware, OS, and Go runtime work together. Another tip is to look for reducing allocations on all optimization design levels ("Optimization Design Levels"), not only code. In most cases, the algorithm must change first to get the decrease in allocations.

> *Attempting to directly affect the pace of collection has nothing to do with being sympathetic with the collector. It's really about getting more work done between each collection or during the collection. You affect that by reducing the amount or the number of allocations any piece of work is adding to heap memory.*
>
> —William Kennedy, Garbage Collection In Go: Part I - Semantics

Reuse

Reusing is also an effective technique. As we learned in "Garbage Collection", the Go runtime already reuses memory in some way. Still, there is a way to explicitly reuse objects like variables, slices or maps for repeated operations instead of recreating them in every loop. The main principle you have to learn is object pooling. It means keeping objects around in a special pool data structure that offers `Get` and `Put` methods. Instead of just creating an object, our code can just do `Get`. Then, rather than abandoning that object whenever the code does not need it, we give it back via the `Put` method.

Standard Go library offers a very simple `sync.Pool` implementation that is a good foundation, but we have to implement a couple of things on top on our side to make it truly efficient. For example, for slices, we might need a "bucketed" solution like the one from the Vitess project, which pools objects based on size, so we don't need to resize those slices or keep too big slices around. We will discuss pooling in detail in [Link to Come].

There are obviously other mechanisms to reuse memory. For example, on the algorithm level, the TCP protocol offers to keep connections alive for reuse, which also helps with the network latency required to establish a new connection.

DON'T REUSE EVERYTHING!

It's tempting to treat this tip literally and reuse every little thing including variables. However, as we mentioned in "Values, Pointers and Memory Blocks", variables are just labels to certain values. Variables alone do not allocate anything, so shadowing variables have zero effect on memory. On the contrary, variable shadowing is generally a terrible idea that can produce hard-to-find bugs in practice.

Reusing complex structures can also be very dangerous for two reasons[39]:

- It is often not easy to reset the state of a complex structure before using it a second time (instead of allocating a new one, which creates a deterministic, empty structure).
- We cannot concurrently use those structures, which can limit further optimizations or surprise us and cause data races.

Recycle

> Recycling is a pure minimum of what we must have in our programs. Fortunately, we don't need anything extra in our Go code, as it's the built-in GC's responsibility to recycle unused memory to the OS. In "Garbage Collection", we discussed how we could tune that process for better results.

Keep the three R's method in your memory, ideally in order. If you find a piece of code or algorithm that is the main offender in memory allocations, think if you can reduce those allocations (e.g. don't try to reuse objects that could be reduced first). If not, perhaps there is a way to pool that memory for some reuse. If not, tuning the GC or using off-heap is our only option, but this is very rare in practice. The `reduce` and `reuse` patterns are at the centre of the most popular Go memory optimization we will walk through in [Link to Come] and [Link to Come]. For example, in [Link to Come] I will present how Example 4-1 can be optimized to the point it allocates 4KB per operation on any kind of file (instead of 60 MB for 7 MB file and 4TB for 500GB file)!

Finally, it's important to reiterate—don't use the three R's method if you don't have any memory concerns or all the code in the memory-sensitive application. Optimizations take time and are harder to maintain. Do that only if you have to, and ideally, by sticking to the TFBO process discussed in "Efficiency-Aware Development Flow".

Summary

It was a long chapter, but you made it! Unfortunately, memory resource is one of the hardest to explain and master. Probably that's why there are so many opportunities to reduce the size or number of our program's allocations.

You learned the long, multi-layer path between our code that needs to allocate bits on memory and those bits landing on the DRAM chip. You learned about many memory trade-offs, behaviours and consequences on the OS level. Finally, you now know how Go uses those mechanisms and why memory allocations in Go are so transparent.

Perhaps you can already figure out the root causes of why Example 4-1 was using 60 MB of the heap for every single operation when the input file was 2 MB large. This knowledge will make it ultra-easy to optimize that case in [Link to Come] and most likely many problems with memory utilization in your code!

It is important to note that this space is evolving. Go compiler, Go garbage collector (GC) and Go Allocator are constantly improved, changed and scaled for the needs of Go users. For example, I am looking forward to GC's capabilities to run collection sooner thanks to Memory soft-limit parameter in Micheal's approved proposal. Yet most of the incoming changes will likely be only iterations of what we have now in Go.

Ahead of us is [Link to Come] and [Link to Come], which I consider the two of the most crucial chapters in the book. I have already mentioned many tools I used to explain the main concepts in past chapters: metric, benchmarking and profiling. It's time to learn those in detail!

1 In my book when I say “memory”, I mean RAM and vice versa. Other mediums offer “memorizing” data in computer architecture (e.g. L-caches), but we tend to treat RAM as the “main” memory resource.

2 Not only because of physical limitations like not enough chip pins, space and energy for transistors but also because managing large memory poses huge overhead as we will learn in “OS Memory Management”.

3 In some way, the RAM’s volatility can sometimes be treated as a feature, not a bug! Have you ever wondered why restarting a computer or process often fixes your problem? The memory volatility forces programmers to implement robust initialization techniques that rebuild the state from backup mediums, enhancing reliability and mitigating potential program bugs. In extreme cases, crash-only software with the restart as the primary way of the failure handling.

4 We can resolve that problem by simply adding more memory to the system or switching to the server (or virtual machine) with more memory resource. That might be a solid solution if we are willing to pay additionally if it’s not a memory leak and if such a resource can be increased (e.g. cloud has virtual machines with more memory). Yet, I suggest investigating your program memory usage, especially if you continuously have to expand the system memory. Then, there might be easy wins, thanks to trivially wasted space we could optimize.

5 Sometimes, there are relatively easy ways to change our code to use streaming or external memory algorithms that ensure stable memory usage. Knowing how to implement those is critical to modern Go development (explained in [Link to Come]).

6 Nowadays, popular encodings like UTF-8 can dynamically use from 1 up to 4 bytes of memory per single character (24 bits).

7 By just doubling the “pointer” size, we moved the limit to of how many elements we can address to extreme sizes. We could even estimate that 64 bit is enough to address all grains of sand from all beaches on the Earth!

8 Many Common Vulnerabilities and Exposures (CVE) issues exist due to various bugs that allow out of bounds memory access.

9 In the past, segmentation was used to implement virtual memory. This has proven to have less versatility, especially the inability to move this space around for defragmentation (better packing of memory). Still, even with paging, segmentation is applied to virtual memory by the process itself (with underlying paging). Plus, the kernel sometimes still uses non paged segmentation for its part of critical kernel memory.

10 You can check the current page size on the Linux system using `getconf PAGESIZE` command.

11 For example, typically, Intel CPUs are capable of hardware supported 4KB, 2MB or 1GB pages.

12 Even naive and conservative calculations indicate around 24% of total memory is wasted for 2MB large pages.

13 We won’t discuss the implementation of page tables since it’s pretty complex and not something Go developers have to worry about. Yet this topic is quite interesting as the trivial

implementation of paging would have a massive overhead in memory usage (what's the point of memory management that would take the majority of memory space it manages?). You can learn more here.

14 There is also an option to disable an overcommitment mechanism on Linux. When disabled, the virtual memory size (VSS) is not allowed to be bigger than the physical memory used by the process (RSS). You might want to do this, so the process will have generally faster memory access, but the waste of memory is enormous. As a result, I have never seen such an option used in practice.

15 `MAP_SHARED` means that any other process can reuse the same physical memory page if they access the same file. This is harmless if the mapped file does not change over time, but it has more complex nuances for mapping modifiable content.

16 A full list of options can be found in mmap documentation.

17 On Linux, you can find this information by doing `ps -ax --format=pid,rss,vsz | grep $PID`, where $PID is process ID.

18 How do I know? On Linux, we can have exact statistics for each memory mapping process is using thanks to the `/proc/<PID>/smaps` file. I wrote a small [Link to Come] on the debugging journey I had and how you can reproduce those experiments on your machine!

19 There are many reasons why accessing nearby bytes might not need allocating more pages on RAM in the memory-mapped situation. For example, the cache hierarchy (discussed in "Hierachical Cache System"), the OS and compiler deciding to pull more at once or such page being already in shared or private page because of previous accesses.

20 Note that physical frames for this file can be still allocated on physical memory by OS (just not accounted for our process). This is called `page cache` and can be useful if any process tries to memorize the same file. Page cache is stored on best-effort in the memory that would be otherwise not used. It can be released when the system is under high memory pressure.

21 Swapping is usually turned off by default on most machines.

22 Teaching the OOM killer article explains some problems in choosing what process to kill first. The lesson here is that global OOM killer is often hard to predict.

23 Exact implementation of memory controller can be found here

24 This is common for many OS-es. Not only Linux but also Android.

25 Remember, whatever type or amount of virtual memory OS is giving to the process - it uses the memory mapping technique. `sbrk` allows simpler resizing of the virtual memory section typically covered by the heap. However, it behaves like any other `mmap` using anonymous pages.

26 Of course no one blocks anyone from implementing external garbage collection on top of those mechanisms in `C` and `C++`

27 There reason why it's hard that ownership model in Rust requires programmer to be hyperaware of every memory allocation and what part owns it. Despite that, I am a huge fan of the Rust ownership model if we could scope this memory management only to a certain part of

your code. I believe it would be beneficial to bring some ownership pattern to Go, where a small amount of code could use that, whereas the rest would use GC. Wishlist for someday? (:

28 We will go through an example that proves this thesis in [Link to Come].

29 If you are more experienced with Go, you might recall two patterns of passing function arguments: by value (if we want to work on shallow copy) or by reference (if we want to share data). I found an amazing simplification that helps to understand pointers: "Go technically has only pass-by-value. When passing a pointer to an object, you're passing a pointer by value, not passing an object by reference".

30 See handy `reflect.SliceHeader` struct that represents a slice.

31 This is one of the reasons why in Go, every new structure has defined zero value or nil at the start, instead of random value.

32 Again, see [Link to Come] to learn about the debugging journey using our lovely /proc pseudo-filesystem, if you are interested.

33 We know that because `go build -gcflags="-m=1" slice.go` have `./slice.go:11:11: make([]byte, size) escapes to heap` line. More on that in [Link to Come].

34 This behaviour is often leveraged by more advanced Memory Ballasting technique explained in [Link to Come].

35 It's also possible to change Go memory release strategy by changing `GODEBUG` environment variable. For example, we can set `GODEBUG=madvdontneed=0`, so `MADV_FREE` will be used instead to notify the OS about not needed memory space. The difference between `MADV_DONTNEED` and `MADV_FREE` is precisely around the point mentioned in the quote above. For `MADV_FREE`, memory release is even faster for Go programs, but the resident set size (RSS) metric of the calling process might not be immediately reduced until the OS reclaims that space. This has proven to cause a massive problem on some systems (e.g. lightly virtualized systems like Kubernetes) that rely on `RSS` to manage the processes. This happened in 2019 when Go defaulted to `MADV_FREE` for a couple of Go versions. More on that is explained in my blog post.

36 To be strict, Go ensures that a maximum of 25% of the total CPU assigned for the process is used for the GC. This is, however, not a silver-bullet solution. By reducing the maximum CPU time used, we simply use the same amount just over longer periods. Thus I decided to skip this fact in our discussions.

37 There are active discussions to introduce soft limits though, so we might be able to set memory limit in a later version of Go. It is said to be internally evaluated as `SetMaxHeap` option.

38 For example, in the Prometheus project we removed, manual GC trigger when code condition changed a little. That decision was based on micro and macro benchmarks that we will discuss in [Link to Come].

39 See a nice blog post about those.

About the Author

Bartłomiej (Bartek) Płotka is a Principal Software Engineer at Red Hat, and the current technical lead of the CNCF SIG Observability group. He has helped to build many popular, reliable, performance- and efficiency-oriented distributed systems in Go with a focus on observability. He is a core maintainer of various open-source projects, including Prometheus, libraries in the gRPC ecosystem, and more. In 2017, together with Fabian Reinartz, he created Thanos, a popular open-source distributed time series database. Focused on cheap and efficient metric monitoring, this project went through hundreds of performance and efficiency focused improvements. Bartek's passion has always been to focus on the readability, reliability, and efficiency of Go. On the way, Bartek helped to develop many tools, wrote many blog posts, and created guides to teach others on writing pragmatic yet efficient Go applications.