CS263 runtime system project report

Shuyan Li

9398959

Project member:

Shuyan Li

**Objective:**

Go profiling and optimization

**Outline of this paper:**

1. Basic introduction

2. How to find the place to start profiling, with examples and results

3.a. Different types of imperfection we met in the program

3.b. The way we optimize these imperfections with examples and results

**1. Basic Introduction about profiling in Go**

The reason for program profiling is that even if the algorithms of the program look good, say running in linear time complexity, or even constant time complexity, the program is not perfect. Beside algorithms, we also need to save our memory. What’s more, sometime even the two approaches of a program have the same time complexity, the time consumption of the program varies a lot. As a result, we need to profile the program, characterize the program behavior, find out the unusual behaviors and improve the performance of the program.

Go is getting more and more popular these days. We mainly focus on go profiling in the project. There are bunches of tools we can use to profile go.

**pprof** is one of the most important and useful tool in go profiling. It is a package (library) that is easy to use and lightweight. The user need to add the flags in the code, like memory flag, CPU flag, allocation flag, etc. After the user run the program (no compile needed), by using the terminal (command line), the users can check the behavior of either the overall program or any specific function. There are many advantages of using pprof. Firstly, the pprof package can generate many diagrams for the structure of the programs and the time/memory consumption distribution. There are also lots of open source library online that can help converting these diagrams and statistics to many other diagrams, for example, Flame graph (will be shown in later part of the report). As a result, the programmers can know either the overall distribution of the time/memory consumption or any distribution among each function. This is very useful for the programmer to find out where to start profiling. What’s more, the programmer can control the pprof so that they can see anything they want by adding different kinds of flags in different places and run different terminal commands. The disadvantage is that the programmers need to keep repeating the process using terminal to check the performance. In another word, it’s requires lots of documents reading and testing.

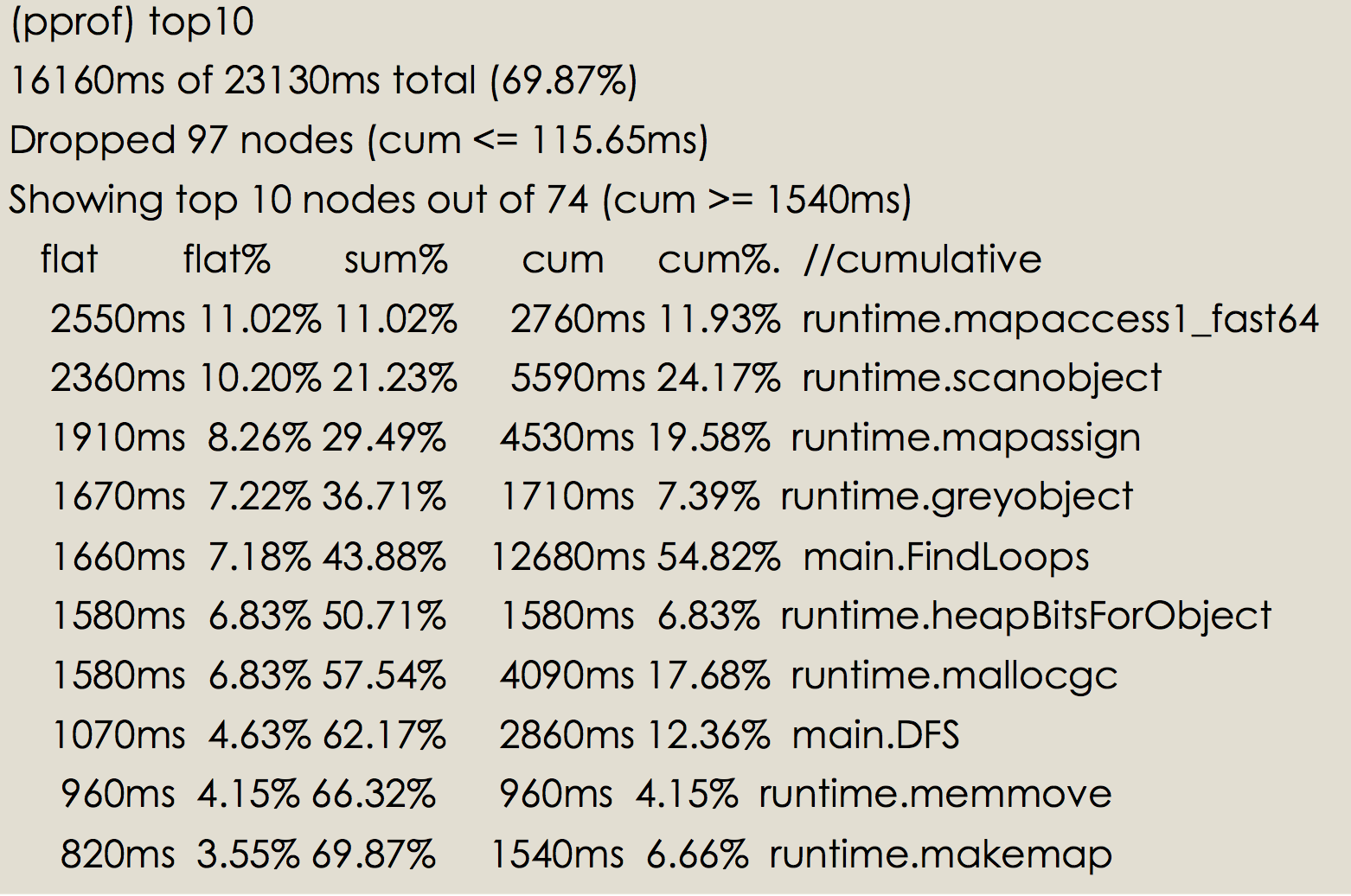
**Benchmark** is also a very useful tool in go profiling. It provides the user a way to test small pieces of code without running the whole test and programs. For example, in my project, there is a http handler test program. If we want to test the handler function, instead of using all pprof and flags and running the program in the localhost, we can simply use benchmark to run the program N times and get the performance without using all these flags. This is convenience when we want to find out the performance. We can then add the flag is the function needs more optimization. Benchmark can save lots of time during testing.

Another tool I researched is **stackImpect**. This is not required in our project but I just want to see if it can make things easier. stackImpect is a third-party platform. By adding the keys and agent number in the code, one can see the dynamic performance of the program on the dashboard of the stackImpect official website. stackImpect can also show the architecture of the program. The disadvantage, however, is that the programmer cannot control it. The architecture that the dashboard shows are all libraries/packages our function calls. In other word, if the programmer, like me, is not an expert in Go, it is difficult to use these information for profiling and optimization. However, it is a good tool for us to compare the performance of our function after any modification.

**2. How to find the place to start profiling, with examples and results**

The most important thing to start profiling is to find the “hot” function. There are two ways to find them: using pprof flag and using svg diagram.

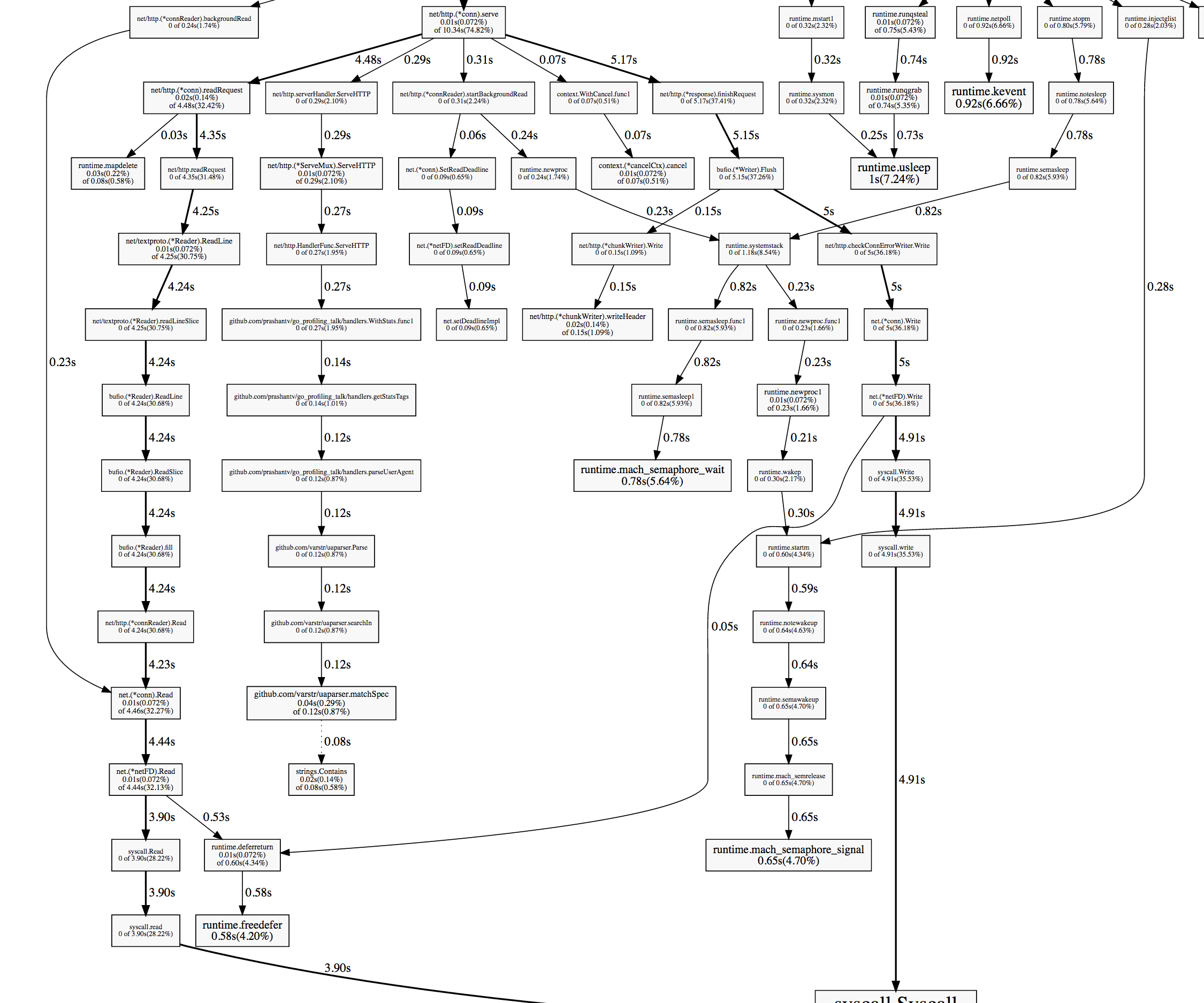
By adding the memory or CPU flag in pprof, one can check the consumption of all the functions:



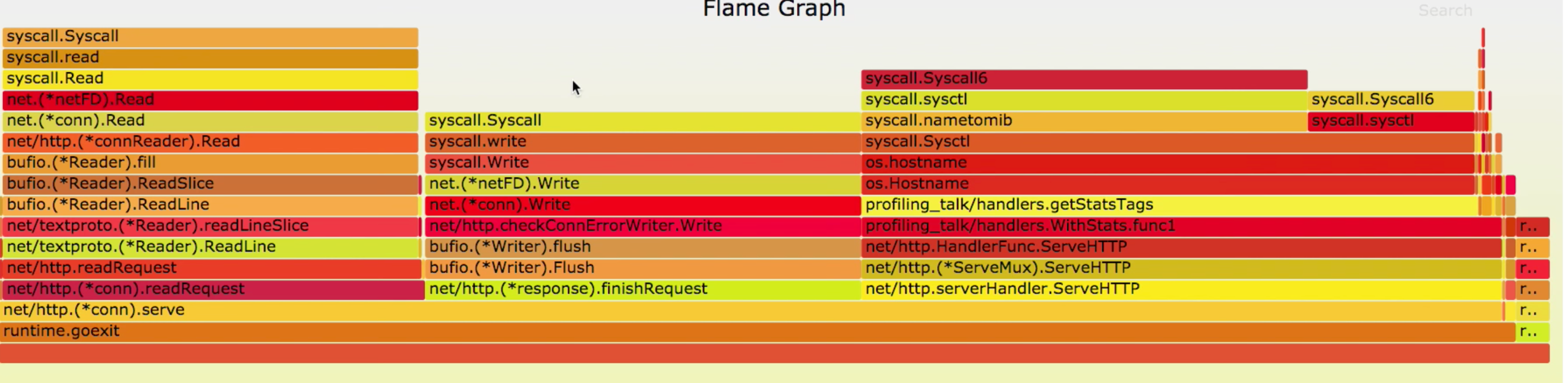
*figure 0: pprof results*

Here we show the top 10 results of the time consumption in the program. We can see that there are lots of runtime.func\_name which are packages, libraries or built-in functions. There are also main.func\_name that is writing by the programmer. These functions can be the starting points of our profiling process.

Another way for finding the hot function is checking the diagrams. Here are some examples for two kinds of file. The first diagram is the original svg diagram that is generated by the pprof and command line by typing “web” command, which is a flowchart(figure 1). The other one is an open source tool called flame graph, that will convert this flowchart to a block diagram(figure 2).



*Figure1: original web svg diagram*

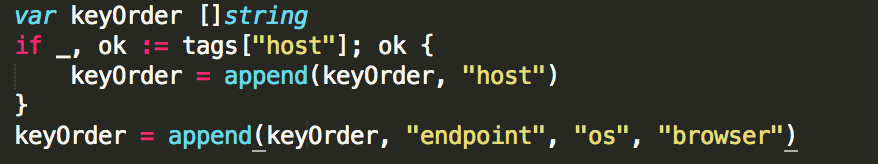


*figure 2: Flame graph*

**3. Different types of imperfection we met in the program and the way we fix them**

In this part, I will introduce 5 major problems in the code and the way to fix them. The code I use are all open source example code online. The algorithms of these codes are easy to understand, so they are all good resource for beginner like me to do the profiling practice.

Problem #1:intermediate objects



Here the author of this code creates a string slice, and appends a string called “host” into the string. The slice then calls another append to insert more string into it. This process seems very easy and naïve, but it introduces lots of intermediate objects. In Go, if we initialize a slice with smaller capacity and length, when we want to insert something later that is largest than this capacity or length, the runtime will not double the capacity and length, like vector in C++ and dynamic array in Java. Instead, the runtime will generate a new slice that has exactly the capacity and length after the insertion, and copy everything in the old slice to new slice. If we want to insert more, the runtime will keep generating the new slice and do the copy and paste. In other word, we initialize an A, then copy all A to a larger B, and then copy all B to a larger C.

The best way to fix this problem is to avoid these intermediate objects. Once we found that there are lots of memory consumption here, we can check the code. We found that all we need to do is inserting 5 string. Instead of declaring an empty slice with no length and capacity, we just need to define the slice with capacity equals 5.

Let’s see the improvement:

169-231-98-53:stats Shuyan$ go test -bench . -benchmem -cpuprofile prof.cpu

BenchmarkAddTagsToName-8     500000       **1214 ns/op**     220 B/op  **17 allocs/op**

169-231-98-53:stats shuyanli$ go test -bench . -benchmem -cpuprofile prof.cpu

BenchmarkAddTagsToName-8    300000 **580 ns/op**     144 B/o **10 allocs/op**

We can see that the time consumption for this function drop to its half and 7 allocations are eliminated.

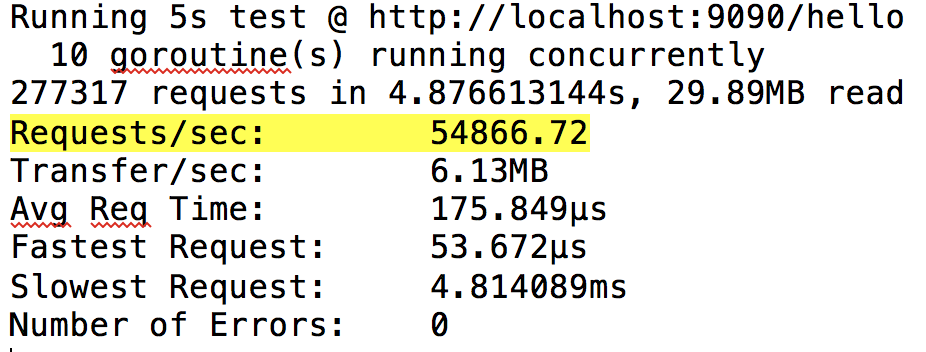
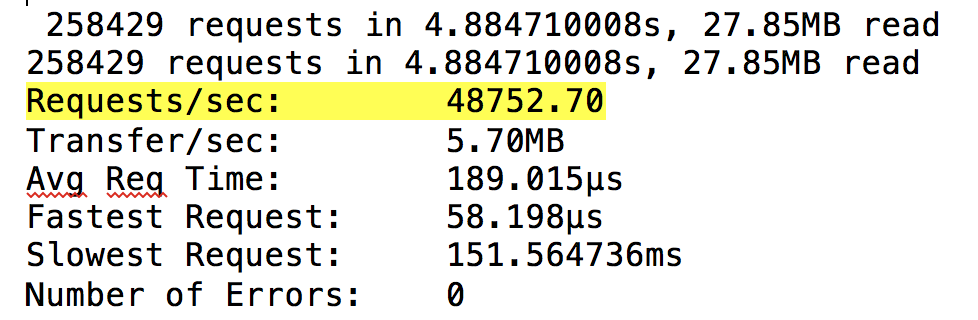
Problem # 2: cache

By checking either the benchmark, diagram or pprof that we just introduced in the previous chapter, we sometime find out that one function or one variable consume huge sums of time. When we go deep to that function, we might find out that there seems nothing we can improve this function. Now is the time to think about doing the cache. Sometime the function is looks good, but it maybe keeps calling some built-in library function during each iteration.

For example, when I found there is a large time consumption in my handler function, I checked the details of the code and realize that there is one line of code here in my http request function:

*Host, err := os.Hostname()*

The program gets the hostname from the server and uses it later. However, for our code, during thirty thousand of iterations of the requests, we don’t need to call this function to get the host name thirty thousand times. Instead, we can cache this. We write a new function that get the hostname once, then use a global variable to store this value. Every time the program need the variable, just use the global variable. There are also lots of other cache examples in my projects. By doing caching, we can significantly reduce the time consumption of the functions.



***Figure 3: before and after caching***

From the figure 3, we can see that after caching, we can handle 6000 more request per second.

Problem # 3: data structure

Sometimes even if the algorithms are the same, which means the big O of the function is the same, the performance of this function varies a lot due to different data structures that function use. For example, using either **map** or an array (slice in go) to store some data. However, in some cases, using an array instead of map, if possible, can reduce time the functions consume. This is because of the way Go implement the map. In GoLang, iterating over an array or slice is simple. Values are contiguous in memory. However, iterating over a map requires traversing the key space and doing lookups into the hash-table structure.

**Before: (u means time spent in user mode)**

# of loops: 76000 (including 1 artificial root node)

**26.02u** 0.24s 19.83r 1288896512kB

**After**

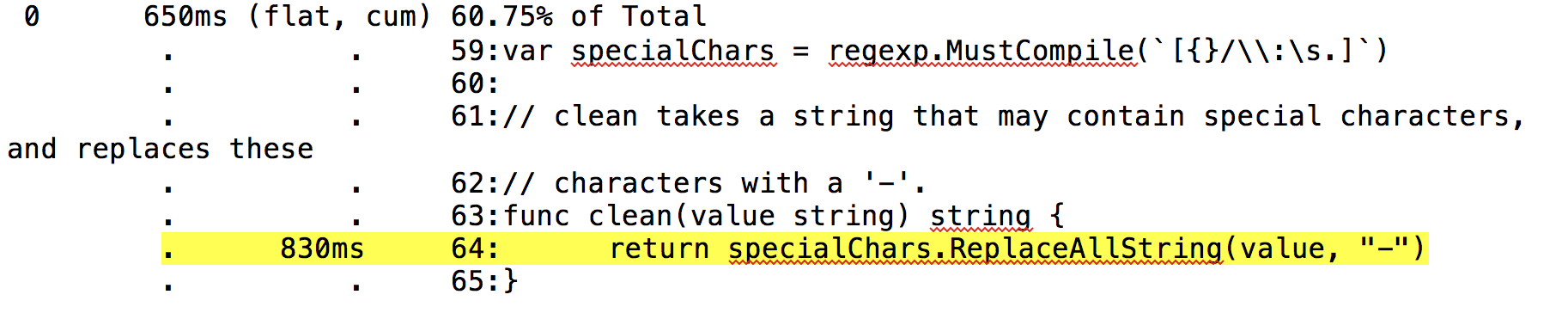
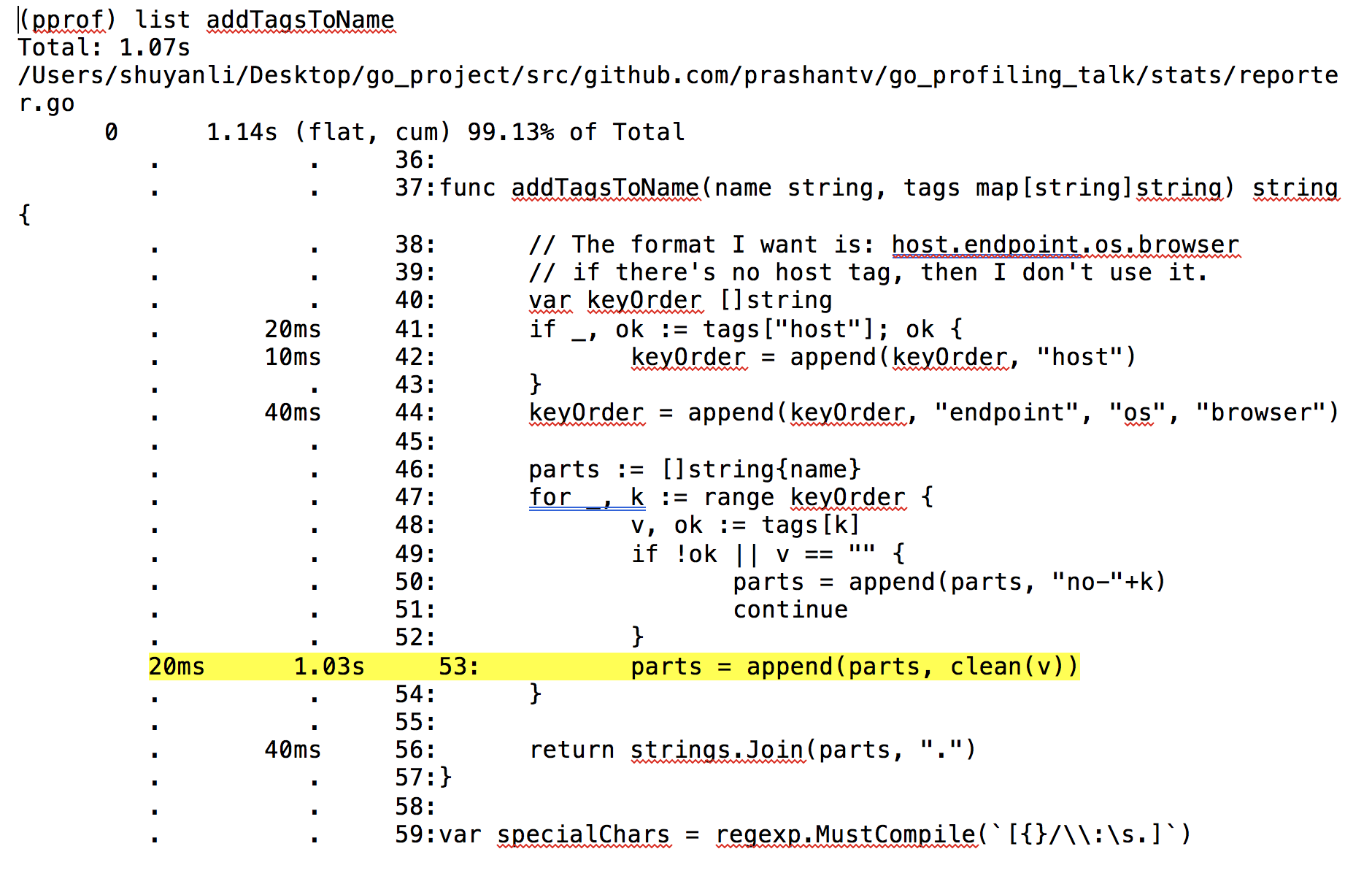
# of loops: 76000 (including 1 artificial root node)

**17.70u** 0.19s 12.34r 1227948032kB

This is the outcome of changing some data structure. Checking the data structure that involve look up and storing is always a good way to do this.

Problem #4: inefficient libraries or packages

Sometimes when we call the library functions, or some functions from packages, the performance is not good, even if the library is a well-performance, well-designed library. The reason is that most of the library are general. For some specific case, however, we might want to implement our own functions (or override the library functions) so that it can run better than the original library functions. Those override functions may not be widely-use, but they can be very fast in our situation.



***figure 4.1: pprof addTagsToName() function figure 4.2: pprof clean() function***

In figure 4.1, we can see that 1.03 out of 1.07 seconds of this “addTagsToName()” function spends its time in the highlighted line. It calls “clean” function. We then go to clean to check more details. We finally find out that all the time in clean function are consumed on this line of code. The clean function calls a Char library and replace all the selected char to “-”. This library function is very general, and it will generate lots of intermediate objects, like I introduced before. Since we already know that we want to replace char with another char, we can define our own replacement function without calling this library.

After I wrote my own function, using same time complexity, the time consumption per iteration drop from 2524ns to 1214 ns, and the allocation number reduces from 19 to 17. This indicates that not only the program is running faster, but also less memory the program consumed.

Problem #5: allocations

An allocation means a possible GC. This doesn’t mean GC is slow. GC is actually very fast. It’s just because every time we allocate, the runtime must do extra work to find some space, so allocations are not free. As a result, we want to minimize the allocation number. Check the path (hot path) where the number of allocation is high. Using the way introduced above to minimize the allocation number.

Example: In the adder function, there are four lines of code that is used to get the header and put them in a slice for later.

var html bytes.Buffer

err = templates.Execute(&html, structSum)

w.Header().Set("Content-Type", "text/html; charset=utf-8")

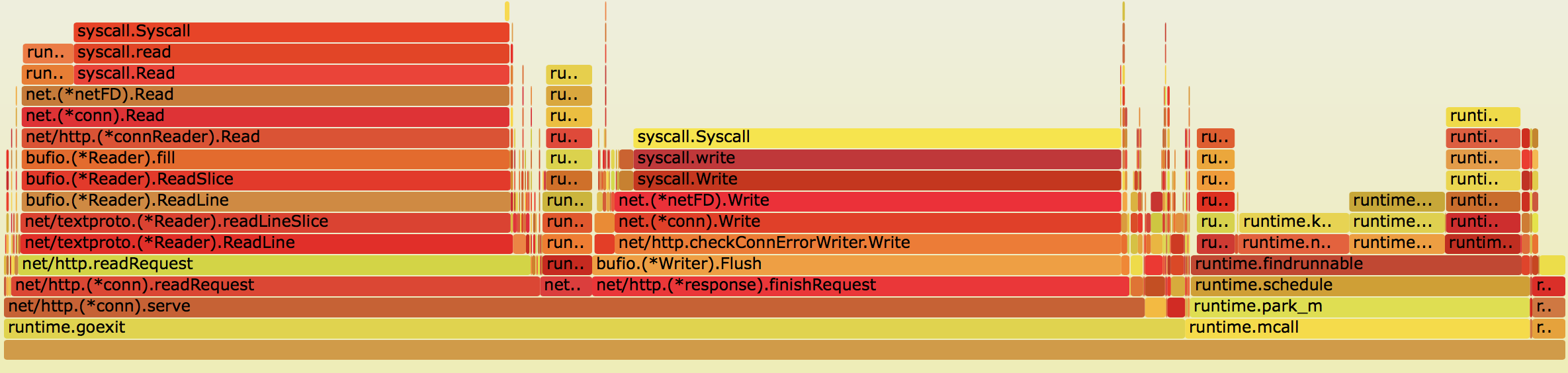
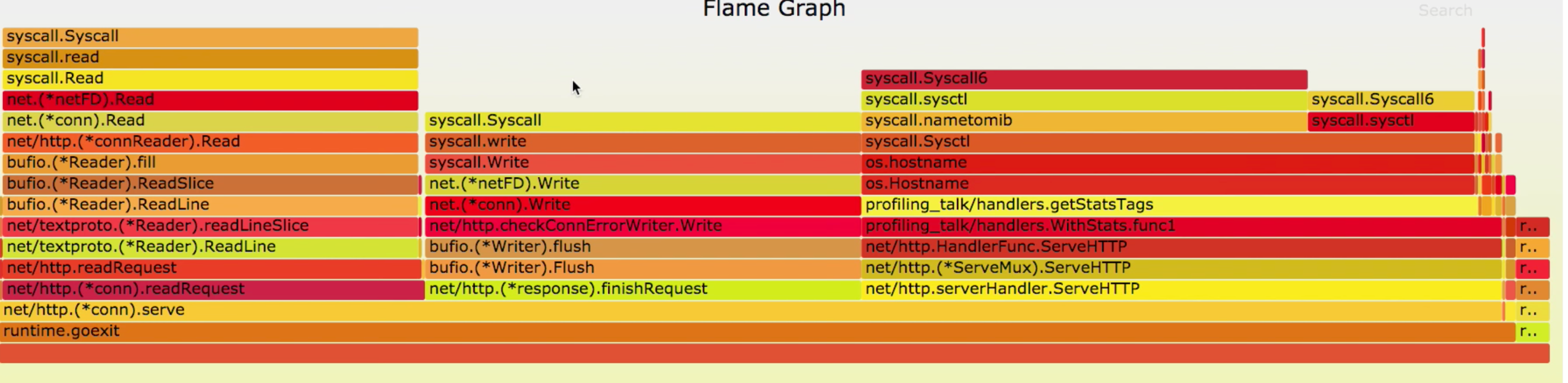
w.Write([]byte(html.String()))

Here, the author creates a Buffer called html, then passes this buffer to Execute to write in the output through IO, then creates a header and create template strings. The author calls write function to pass the buffer to a string and write them to a slice. We can see here that the code generates lots of intermediate objects so it has lots of allocations. Let’s get rid of all these intermediate objects: We don’t need the buffer, and we don’t need to call IO.write twice, instead, we simply pass in the IO.w in the Execute function so that we can get the output directly.

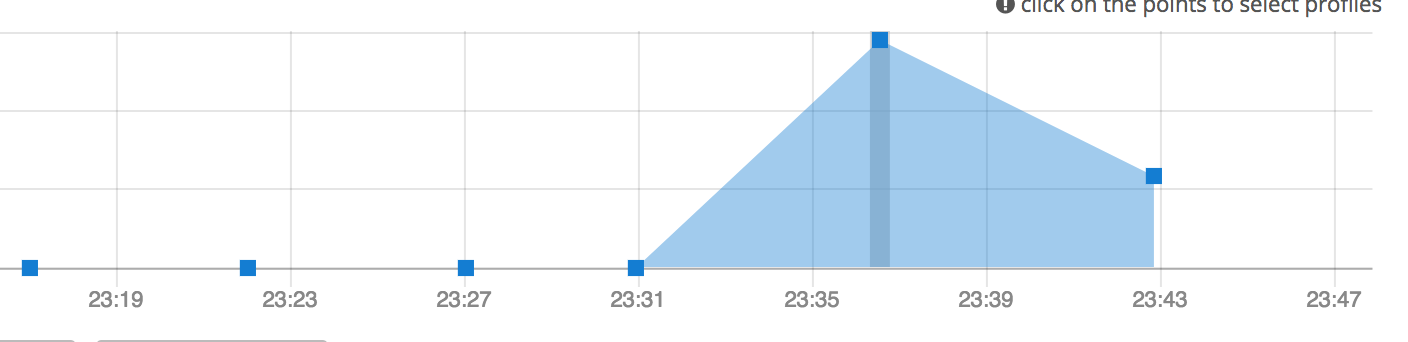
After the allocation elimination process, the output shows that the allocation number of this function drop from 21 to 17, and it runs slightly faster as well.

**Results and Conclusions:**

There are still lots of things I want to write in this report. But the report has already exceeded the limit, so I simply put some results here. In a nutshell, I learnt a lot in this project. Before this course, I simply believed that a good algorithm can determine the performance of a program. It was naïve. As we can see, there are still lots of things we can do if we want to optimize our program. Following are some results that show the performance before and after the profiling. We can see clearly that the program has a much better improvement and we can even make it better in the future.



***figure 5: before and after profiling. Read and write is not what we want to profile. We can clearly see that the function we want to profile, like all the handler, request functions that the author wrote before has significantly reduced it’s time and space consumption***

******

***figure 5: CPU performance of naïve and modified program. We can see that the program can run more than two times faster than the original program.***