After following the tutorial in week4 after the blog, we wanted to play something on our own to see if we can do the improvement for a given go program.

I found a simple web app online. It creates a request on port 9090 and there are two index, /hello and /simple. They all do a pretty simple thing, just print on the screen with “hello world”. The /hello operation records the response of the server so the run time of these two function should be different.

First thing first, I need to configure the path for go since when I was running the code, the system keeps giving me an error “cannot find the package”. As a C++ programmer on windows (just switched to mac), we use IDE like VS. As a result, I didn’t modify the path by my own before.

After reading some article, I set my local GOPATH by type in :

$sudo nano .bash\_profile

it shows me a blank document.

Then I add these in it:

# Setting for go programming language

export GOROOT="/usr/local/go"

export GOPATH="/Users/shuyanli/Desktop/go\_project"

export PATH="/Users/shuyanli/Desktop/go\_project/bin:$PATH"

I set the local path to my desktop and now I can import the local file to the main function.

Lets scan the two “hello world” function.

The simple function is nothing but fprint a string on the screen(guess there is not too much thing we can modify)

The hello function is way more complicate. It wraps the handler and the request with some latency tracking function

The difference of the function is that the hello function can return the states.

For example, if we type in the bash

$ go run main.go -printStats

when we refresh the simple function, nothing will be shown on the screen.

When we refresh the hello function, however, we can see the actual return states:

Starting server on :9090

IncCounter: handler.received.169-231-111-73.hello.Mac-OS.Chrome = 1

RecordTimer: handler.latency.169-231-111-73.hello.Mac-OS.Chrome = 46.884µs

IncCounter: handler.received.169-231-111-73.hello.Mac-OS.Chrome = 1

RecordTimer: handler.latency.169-231-111-73.hello.Mac-OS.Chrome = 31.994µs

IncCounter: handler.received.169-231-111-73.hello.Mac-OS.Chrome = 1

RecordTimer: handler.latency.169-231-111-73.hello.Mac-OS.Chrome = 42.787µs

IncCounter: handler.received.169-231-111-73.hello.Mac-OS.Chrome = 1

RecordTimer: handler.latency.169-231-111-73.hello.Mac-OS.Chrome = 32.152µs

We want to check the load test for the index. We need the tools first:

<https://github.com/adjust/go-wrk>

correction:

<https://github.com/tsliwowicz/go-wrk>

this is the right tool to use since the previous command cannot be found, no matter how I reinstall and recompile

following this link, we got the go-wrk tool, that can check the test load of a url.

Now, we open another command window and type:

$go-wrk -d 5 [http://localhost:9090/hello](http://localhost:8080/hello)

258429 requests in 4.884710008s, 27.85MB read

Requests/sec: 52905.70

Transfer/sec: 5.70MB

Avg Req Time: 189.015µs

Fastest Request: 58.198µs

Slowest Request: 151.564736ms

Number of Errors: 0

We just get the load test result as shown above(52k request/s)

Same for the simple:

Running 5s test @ http://localhost:9090/simple

  10 goroutine(s) running concurrently

275997 requests in 4.874995884s, 29.74MB read

Requests/sec: 56614.82

Transfer/sec: 6.10MB

Avg Req Time: 176.632µs

Fastest Request: 50.034µs

Slowest Request: 13.70035ms

Number of Errors: 0

If we check the heap, we can see how actually the program do the GC

# runtime.MemStats

# Alloc = 843720

# TotalAlloc = 27504491696

# Sys = 14850360

# Lookups = 118

# Mallocs = 289111011

# Frees = 289103025

…

…

# HeapReleased = 4276224

# HeapObjects = 7986

# Stack = 1179648 / 1179648

# MSpan = 46208 / 98304

# MCache = 9600 / 16384

# BuckHashSys = 1452356

# GCSys = 503808

# OtherSys = 5439476

# NextGC = 4194304

# NumGC = 9331

we can see that the number of garbage collection and there is a list for those garbages. According to our previous experience, this part is one of the main part that we can modify.

Result we got finally:

12.61s of 13.71s total (91.98%)

Dropped 153 nodes (cum <= 0.07s)

Showing top 10 nodes out of 79 (cum >= 0.16s)

      flat  flat%   sum%        cum   cum%

     8.74s 63.75% 63.75%      8.79s 64.11%  syscall.Syscall

     1.06s  7.73% 71.48%      1.06s  7.73%  runtime.usleep

     0.76s  5.54% 77.02%      0.76s  5.54%  runtime.kevent

     0.66s  4.81% 81.84%      0.66s  4.81%  runtime.mach\_semaphore\_wait

     0.59s  4.30% 86.14%      0.59s  4.30%  runtime.mach\_semaphore\_signal

     0.56s  4.08% 90.23%      0.56s  4.08%  runtime.freedefer

     0.07s  0.51% 90.74%      0.07s  0.51%  runtime.mach\_semaphore\_timedwait

     0.07s  0.51% 91.25%      0.07s  0.51%  runtime.memclrNoHeapPointers

     0.05s  0.36% 91.61%      0.22s  1.60%  net/http.(\*chunkWriter).writeHeader

     0.05s  0.36% 91.98%      0.16s  1.17%  runtime.mallocgc

Showing top 10 nodes out of 79 (cum >= 0.16s)

      flat  flat%   sum%        cum   cum%

     8.74s 63.75% 63.75%      8.79s 64.11%  syscall.Syscall

     1.06s  7.73% 71.48%      1.06s  7.73%  runtime.usleep

     0.76s  5.54% 77.02%      0.76s  5.54%  runtime.kevent

     0.66s  4.81% 81.84%      0.66s  4.81%  runtime.mach\_semaphore\_wait

     0.59s  4.30% 86.14%      0.59s  4.30%  runtime.mach\_semaphore\_signal

     0.56s  4.08% 90.23%      0.56s  4.08%  runtime.freedefer

     0.07s  0.51% 90.74%      0.07s  0.51%  runtime.mach\_semaphore\_timedwait

     0.07s  0.51% 91.25%      0.07s  0.51%  runtime.memclrNoHeapPointers

     0.05s  0.36% 91.61%      0.22s  1.60%  net/http.(\*chunkWriter).writeHeader

     0.05s  0.36% 91.98%      0.16s  1.17%  runtime.mallocgc

Important note:

When we first run go tool pprof -seconds 5 [http://localhost:9090/debug/pprof/profile](http://localhost:8080/debug/pprof/profile) and type in top10, we get a message that the profile is empty. We try run 10 seconds first then do the same step, but we keeps getting the same thing.

After research the next day, we figure out that (thank god) out program is actually run too “fast” so that the profile flag (we imported in the header as import \_ "net/http/pprof", don’t forget this \_ ! ) is not recording anything actually. We try to use the latest profile tools here: <https://github.com/pkg/profile>

But this is not good for a web request.

We followed this stack Overflow answer:

<http://stackoverflow.com/questions/30871691/cant-get-golang-pprof-working>

and finally find out that the only way that we can get the data is use the third terminal window

steps:

1: open window 1 and do $go run main.go

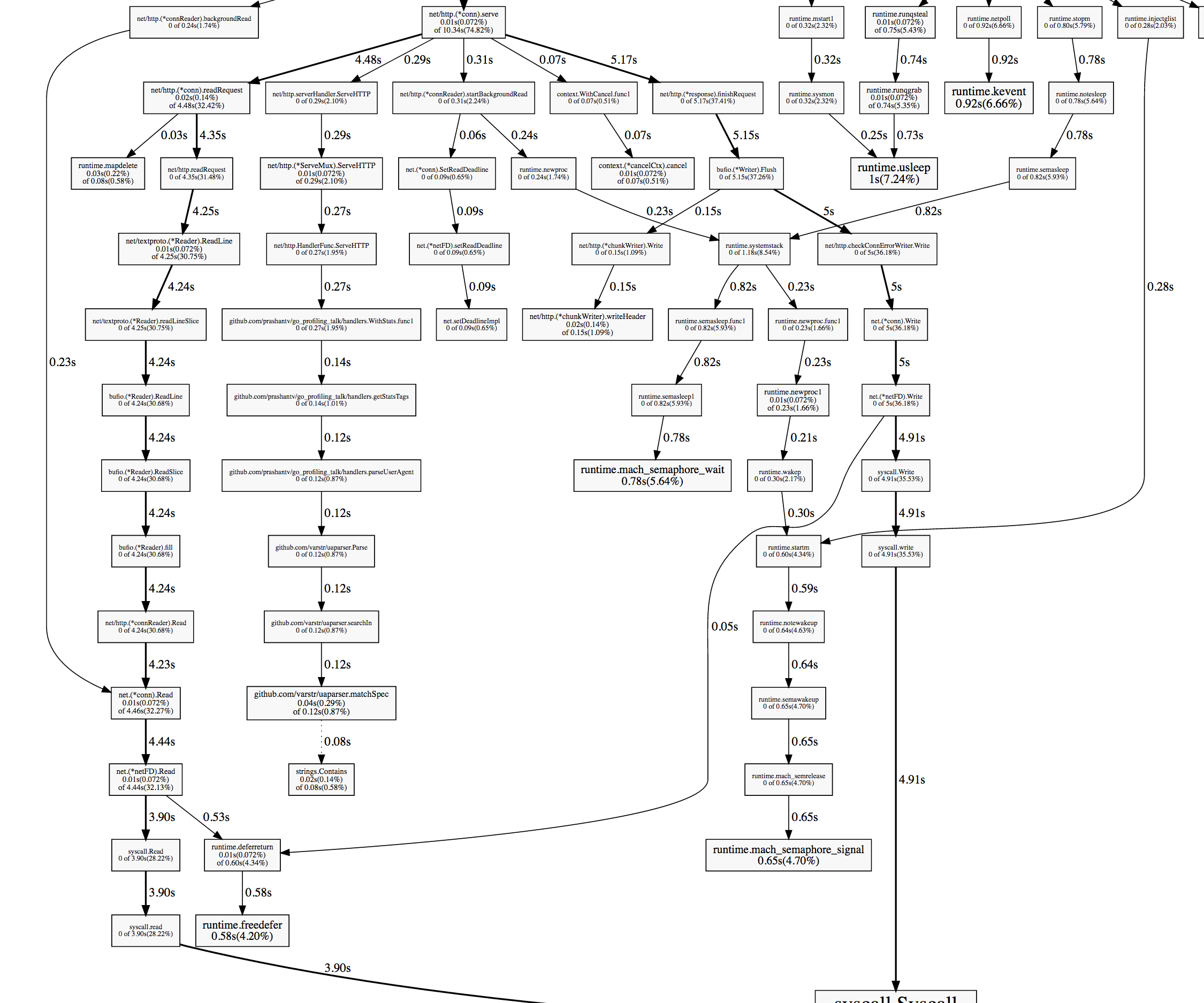
2: open the second window and do $ go-wrk -d 200 [http://localhost:9090/hello](http://localhost:8080/hello)

note that the duration should be longer then 10 seconds, otherwise there is no data in the profile.

3: open the third window simultaneously and type in $ go tool pprof -seconds 5 [http://localhost:9090/debug/pprof/profile](http://localhost:8080/debug/pprof/profile) .

only in this way can we record the data and get the result we want to see

Same as previous tutorial, after type in top 10 and web command, we can se the svg file that tells us the load of the program. According to the svg file, we can see that there are three main path for this program, read, handler and write.



According to the previous experience, read and run should be handled by the runtime, which indicates that we can only modify the handler if we want some improvement.

Zoom in the svg, we can see that the there are withstats and getstats following the handler function.

In the getstatsTag function

|  |
| --- |
| func getStatsTags(r \*http.Request) map[string]string { |
|  | userBrowser, userOS := parseUserAgent(r.UserAgent()) |
|  | stats := map[string]string{ |
|  | "browser": userBrowser, |
|  | "os": userOS, |
|  | "endpoint": filepath.Base(r.URL.Path), |
|  | } |
|  | host, err := os.Hostname() |
|  | if err == nil { |
|  | if idx := strings.IndexByte(host, '.'); idx > 0 { |
|  | host = host[:idx] |
|  | } |
|  | stats["host"] = host |
|  | } |
|  | return stats |
|  | } |

there is one line :

host, err := os.Hostname()

which get the hostname and return it to the host. Hostname, however, should not be changed for this case when the loop is running. The reason it makes the program slow is that in every loop we find the hostname and return it. This is not necessary. The solution is to cache this, by storing this hostname as a global variable so that we don’t need to return this for each loop

So we can do add a globel variable (var globalhostname = getHost()) at the beginning

…

and add a function:

func getHost() string{

|  |
| --- |
| host, err := os.Hostname() |
| if err == nil { |
| if idx := strings.IndexByte(host, '.'); idx > 0 { |
| host = host[:idx] |
| } |
| stats["host"] = host |
| } |
| return host |

}

Let’s see the improvement:

Running 5s test @ http://localhost:9090/hello

  10 goroutine(s) running concurrently

277317 requests in 4.876613144s, 29.89MB read

Requests/sec: 56866.72

Transfer/sec: 6.13MB

Avg Req Time: 175.849µs

Fastest Request: 53.672µs

Slowest Request: 4.814089ms

Number of Errors: 0

258429 requests in 4.884710008s, 27.85MB read

Requests/sec: 52905.70

Transfer/sec: 5.70MB

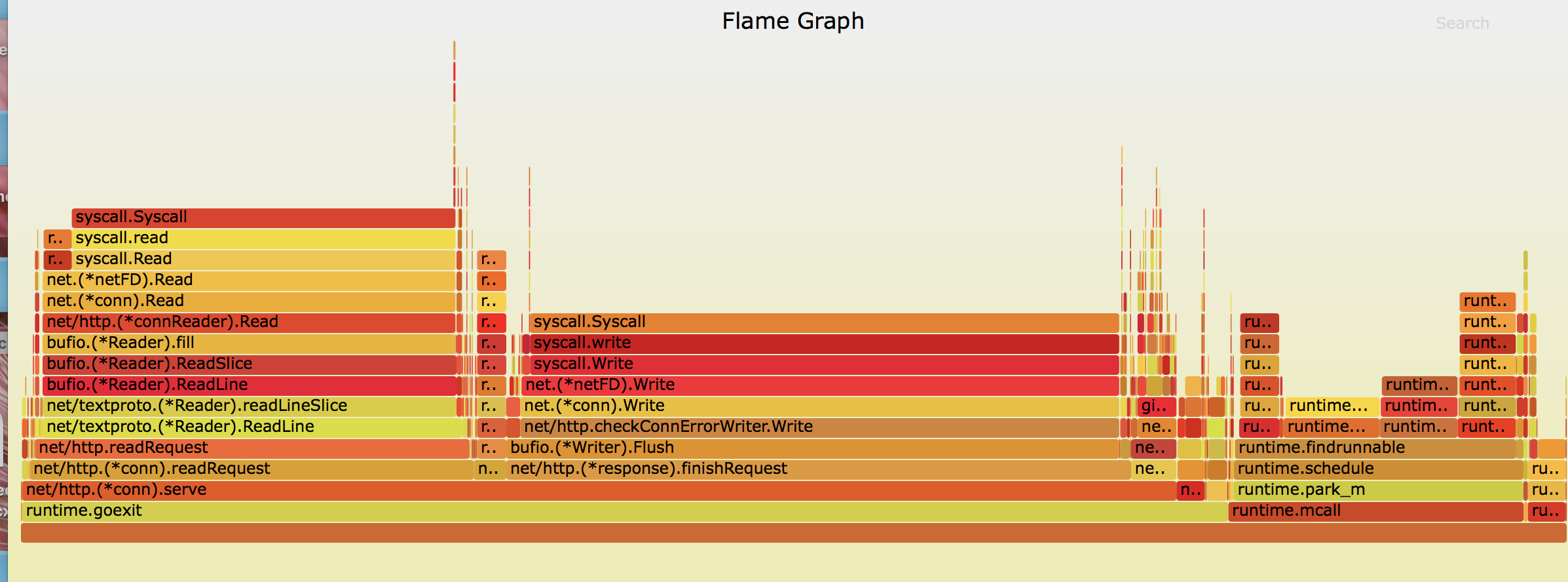
Avg Req Time: 189.015µs

Fastest Request: 58.198µs

Slowest Request: 151.564736ms

Number of Errors: 0

We can see that the requests/sec raise for 4 thousand/second simply because one cache.

Lest keep checking what else can we do. 

New technieque learnt: benchmark in go:

Updated:

Since we zoomed in and we want to check the performance of one of the root function, called nameTagsToName,we need some samples to test this function. However, just like the diagram shows, the percentage of this function for the overall time is quite small, as a result, the sample of this function is not enough to do pprof. As a result, we can use benchmark to test this specific function.

Benchmark are used to test the performance of the code. Example:

// from fib\_test.go

func BenchmarkFib10(b \*testing.B) {

// run the Fib function b.N times

for n := 0; n < b.N; n++ {

Fib(10)

}

}

typically just following this template and run a function b.N times, so we can now run the benchmark to test our function.

After scanning the svg graph, we see that all the stats.\*\* function all calling the addTagsToName function, for example the stats.incounter, stats.recordCounter. So we should see how our addTagsToName function work

func BenchmarkAddTagsToName(b \*testing.B) {

tags := map[string]string{

"host": "myhost",

"endpoint": "hello",

"os": "OS X",

"browser": "Chrome",

}

for i := 0; i < b.N; i++ {

addTagsToName("recv.calls", tags)

}

}

<http://stackoverflow.com/questions/20435382/test-memory-consumption>

following this answer, we use the –benchmem flag, which will record a memory consumption and an allocations count:

ShuyanmatoMacBook-Pro:stats shuyanli$ go test -bench . -benchmem

BenchmarkAddTagsToName-8     500000       2558 ns/op     528 B/op       19 allocs/op

So the function take 2.55 us to run per iteration and there are 19 allocations

Note: the more allocation it has, the slower the program is.

Creating a profile that only include the addTagsToName function like this(record only the addTagToNam function)

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

BenchmarkAddTagsToName-8     500000       2524 ns/op     528 B/op       19 allocs/op

The system will generate the stats.test and prof.cpu for me

Now we have a list that include this function:

169-231-98-53:stats shuyanli$ go tool pprof stats.test prof.cpu

Entering interactive mode (type "help" for commands)

(pprof) top10

1610ms of 1730ms total (93.06%)

Showing top 10 nodes out of 53 (cum >= 120ms)

flat  flat%   sum%        cum   cum%

     140ms 12.17% 12.17%      400ms 34.78%  regexp.(\*machine).tryBacktrack

     140ms 12.17% 24.35%      140ms 12.17%  regexp/syntax.(\*Inst).MatchRunePos

     140ms 12.17% 36.52%      290ms 25.22%  runtime.mallocgc

      90ms  7.83% 44.35%       90ms  7.83%  runtime.heapBitsSetType

      80ms  6.96% 51.30%      310ms 26.96%  runtime.growslice

      70ms  6.09% 57.39%       70ms  6.09%  regexp.(\*bitState).push

      60ms  5.22% 62.61%      510ms 44.35%  regexp.(\*machine).backtrack

      50ms  4.35% 66.96%       50ms  4.35%  regexp.(\*inputString).step

      40ms  3.48% 70.43%      780ms 67.83%  regexp.(\*Regexp).replaceAll

      40ms  3.48% 73.91%       40ms  3.48%  sync.(\*Mutex).Unlock

 (pprof) list addTagsToName

Total: 1.07s

/Users/shuyanli/Desktop/go\_project/src/github.com/prashantv/go\_profiling\_talk/stats/reporter.go

      0      1.14s (flat, cum) 99.13% of Total

         .          .     36:

         .          .     37:func addTagsToName(name string, tags map[string]string) string {

         .          .     38: // The format we want is: host.endpoint.os.browser

         .          .     39: // if there's no host tag, then we don't use it.

         .          .     40: var keyOrder []string

         .       20ms     41: if \_, ok := tags["host"]; ok {

         .       10ms     42: keyOrder = append(keyOrder, "host")

         .          .     43: }

         .       40ms     44: keyOrder = append(keyOrder, "endpoint", "os", "browser")

         .          .     45:

         .          .     46: parts := []string{name}

         .          .     47: for \_, k := range keyOrder {

         .          .     48: v, ok := tags[k]

         .          .     49: if !ok || v == "" {

         .          .     50: parts = append(parts, "no-"+k)

         .          .     51: continue

         .          .     52: }

         20ms      1.03s     53: parts = append(parts, clean(v))

         .          .     54: }

         .          .     55:

         .       40ms     56: return strings.Join(parts, ".")

         .          .     57:}

         .          .     58:

         .          .     59:var specialChars = regexp.MustCompile(`[{}/\\:\s.]`)

We can see that the highlight line takes majority of time, which invoke clean(v) function

As listed above, the left 20ms states that the function append(a,b) itself cost 20 ms, but the function the append calls(the clean function) takes 1.03 second, which takes lots of time.

lets see the clean function:

 0      650ms (flat, cum) 60.75% of Total

         .          .     59:var specialChars = regexp.MustCompile(`[{}/\\:\s.]`)

         .          .     60:

         .          .     61:// clean takes a string that may contain special characters, and replaces these

         .          .     62:// characters with a '-'.

         .          .     63:func clean(value string) string {

         .      830ms     64: return specialChars.ReplaceAllString(value, "-")

         .          .     65:}

return the replaceAllString function (that takes all char and replace with “-”) takes all the time. If we can improve this function, we can improve the performance of the addTagstoName

the function ReplaceAllString belong to package regexp

According to some online resource, regexp is slow since the function is generous. In out program, however, we can handle this specific case much faster by writing a little function by our own instead of using the built-in function in the library.

var specialChars = regexp.MustCompile(`[{}/\\:\s.]`)

// clean takes a string that may contain special characters, and replaces these

// characters with a '-'.

func clean(value string) string {

return specialChars.ReplaceAllString(value, "-")

}

in this function, we want to replace all the specialChars (that include (`[{}/\\:\s.]`)) by “-”. As a result, the output string should be the same length. We can simple use a function in time complex O(n) to finish this function.

Rewrite the clean function as follow:

func clean(buf \*bytes.Buffer, value string) {

for i := 0; i < len(value); i++ {

switch c := value[i]; c {

case '{', '}', '/', '\\', ':', ' ', '\t', '.':

buf.WriteByte('-')

default:

buf.WriteByte(c)

}

}

}

this function just read each char, if they are specialChars, we replace it with “-”, pretty standard O(n) algorithm

Let’s see how it work:

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

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

Compare to the previous naïve algorithm:

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

BenchmarkAddTagsToName-8     500000       2524 ns/op     528 B/op       19 allocs/op

We can see that the speed now is 1.21us/operation and the allocation even drop from 19 to 17.