GO PROFILING AND OPTIMIZATION

Shuyan Li
CS263

In the last lecture: Why profiling

- **To improve performance, track performance regressions**
- -find out where time is being spent

- **To characterize program behavior**
- -runtime
- -program test

- **To capture specific or unusual behavior**

In my presentation

- **Where shall we start profiling?**
- **What shall we profiling?**
- **How do we do the profiling?**

Introduction: tools we can use

- 1: pprof
- -terminal (command line)
- -multiple flag(-memflag, -cpuflag), diagrams, programmer friendly
- -**benchmark**
- -different ways to check performance
- -controllable

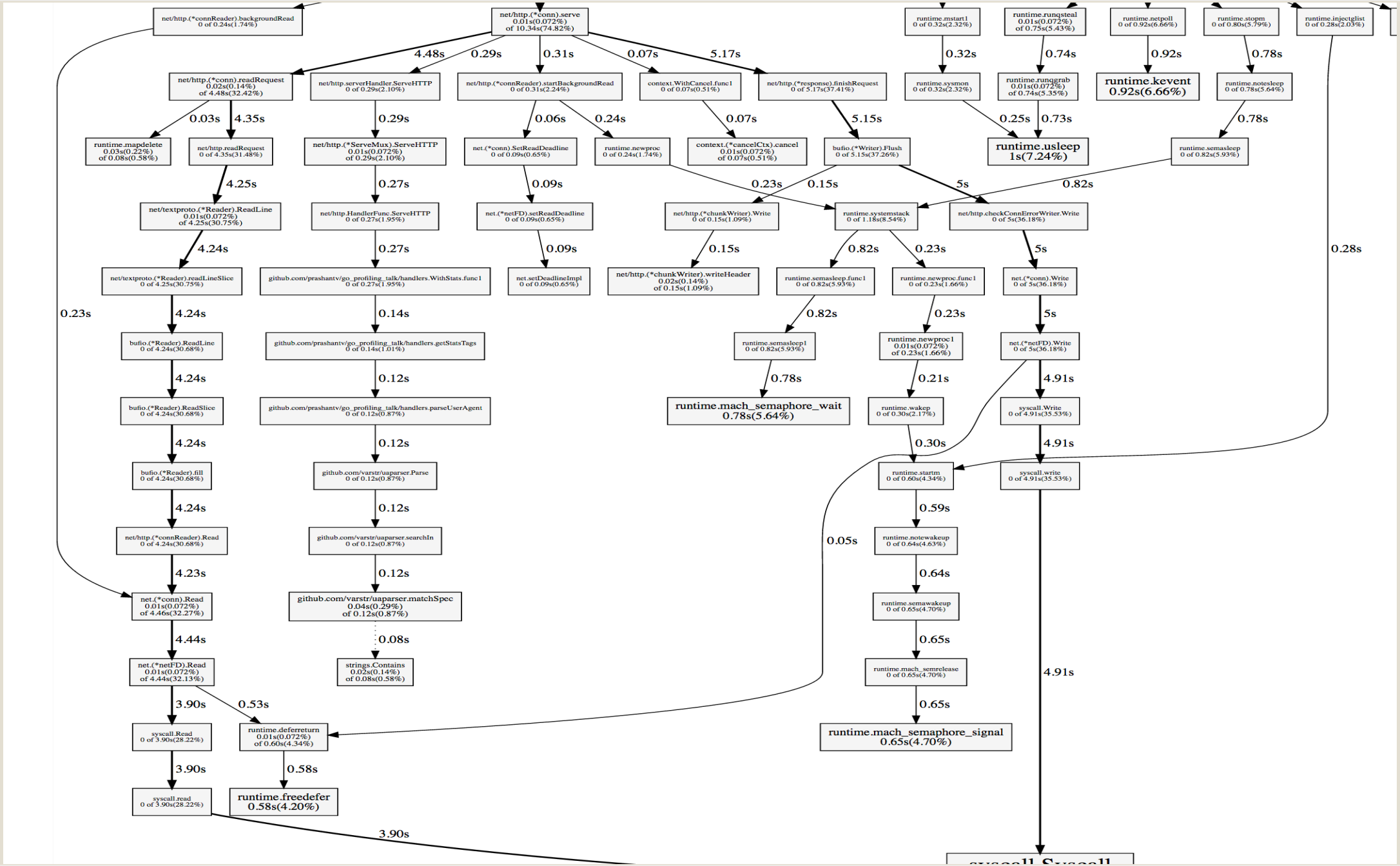
- 2: stackImpect
- -third-party platform
- -show the results dynamically on the dashboard
- -one can see the in-time performance
- -un-controllable

1. Where shall we start?

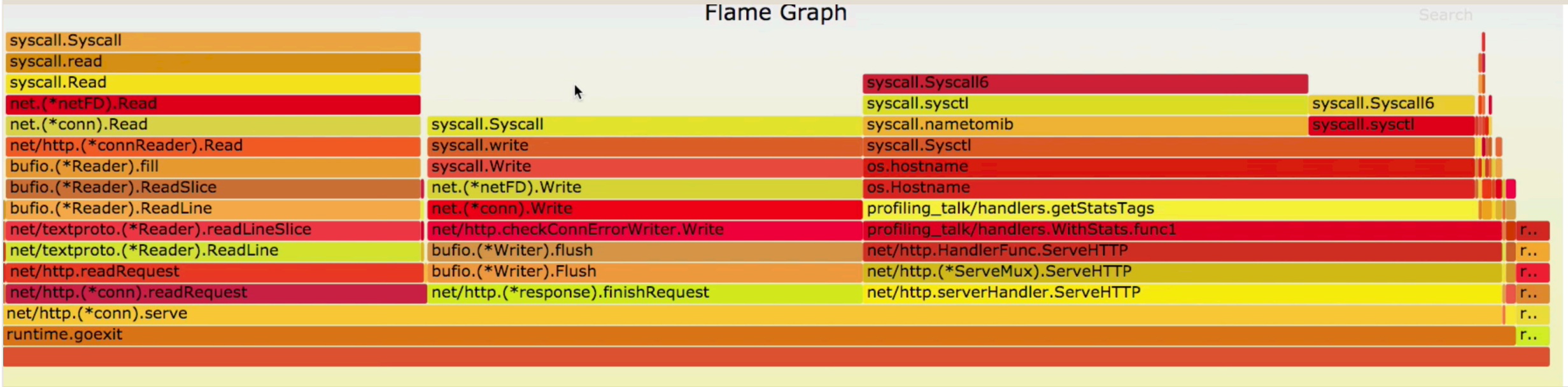
- **1) add the CPU flag and memory flag in code and use pprof in terminal to get the performance of the program:**
- (pprof) top10
- 16160ms of 23130ms total (69.87%)
- Dropped 97 nodes (cum <= 115.65ms)
- Showing top 10 nodes out of 74 (cum >= 1540ms)
- flat flat% sum% cum cum%. //cumulative
- 2550ms 11.02% 11.02% 2760ms 11.93% runtime.mapaccess1_fast64
- 2360ms 10.20% 21.23% 5590ms 24.17% runtime.scanobject
- 1910ms 8.26% 29.49% 4530ms 19.58% runtime.mapassign
- 1670ms 7.22% 36.71% 1710ms 7.39% runtime.greyobject
- 1660ms 7.18% 43.88% 12680ms 54.82% main.FindLoops
- 1580ms 6.83% 50.71% 1580ms 6.83% runtime.heapBitsForObject
- 1580ms 6.83% 57.54% 4090ms 17.68% runtime.mallocgc
- 1070ms 4.63% 62.17% 2860ms 12.36% main.DFS
- 960ms 4.15% 66.32% 960ms 4.15% runtime.memmove
- 820ms 3.55% 69.87% 1540ms 6.66% runtime.makemap

1. Where shall we start?

- 2) getting the svg diagram(shown in next slice)

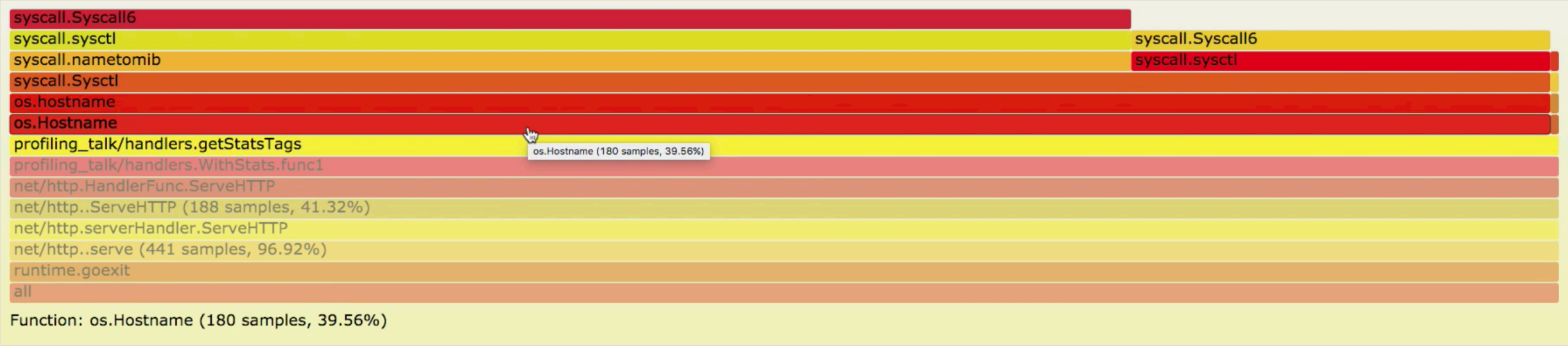

net/http.(*connReader).backgroundRead
0 of 0.24s(1.74%)
net/http.(*conn).serve
0.01s(0.072%)
of 10.34s(74.82%)
runtime.mstart1
0 of 0.32s(2.32%)
runtime.runqsteal
0.01s(0.072%)
of 0.75s(5.43%)
runtime.netpoll
0 of 0.92s(6.66%)
runtime.stopm
0 of 0.80s(5.79%)
runtime.injectglist
0 of 0.28s(2.03%)
4.48s
0.29s
0.31s
0.07s
5.17s
0.32s
0.74s
0.92s
0.78s
net/http.(*conn).readRequest
0.02s(0.14%)
of 4.48s(32.42%)
net/http.serverHandler.ServeHTTP
0 of 0.29s(2.10%)
net/http.(*connReader).startBackgroundRead
0 of 0.31s(2.24%)
context.WithCancel.func1
0 of 0.07s(0.51%)
net/http.(*response).finishRequest
0 of 5.17s(37.41%)
runtime.sysmon
0 of 0.32s(2.32%)
runtime.runqgrab
0.01s(0.072%)
of 0.74s(5.35%)
runtime.kevent
0.92s(6.66%)
runtime.notesleep
0 of 0.78s(5.64%)
0.03s
4.35s
0.29s
0.06s
0.24s
0.07s
5.15s
0.25s
0.73s
0.78s
runtime.mapdelete
0.03s(0.22%)
of 0.08s(0.58%)
net/http.readRequest
0 of 4.35s(31.48%)
net/http.(*ServeMux).ServeHTTP
0.01s(0.072%)
of 0.29s(2.10%)
net.(*conn).SetReadDeadline
0 of 0.09s(0.65%)
runtime.newproc
0 of 0.24s(1.74%)
context.(*cancelCtx).cancel
0.01s(0.072%)
of 0.07s(0.51%)
bufio.(*Writer).Flush
0 of 5.15s(37.26%)
runtime.usleep
1s(7.24%)
runtime.semasleep
0 of 0.82s(5.93%)
4.25s
0.27s
0.09s
0.23s
0.15s
5s
0.82s
net/textproto.(*Reader).ReadLine
0.01s(0.072%)
of 4.25s(30.75%)
net/http.HandlerFunc.ServeHTTP
0 of 0.27s(1.95%)
net.(*netFD).setReadDeadline
0 of 0.09s(0.65%)
net/http.(*chunkWriter).Write
0 of 0.15s(1.09%)
runtime.systemstack
0 of 1.18s(8.54%)
net/http.checkConnErrorWriter.Write
0 of 5s(36.18%)
0.28s
4.24s
0.27s
0.09s
0.15s
0.82s
0.23s
5s
net/textproto.(*Reader).readLineSlice
0 of 4.25s(30.75%)
github.com/prashantv/go_profiling_talk/handlers.WithStats.func1
0 of 0.27s(1.95%)
net.setDeadlineImpl
0 of 0.09s(0.65%)
net/http.(*chunkWriter).writeHeader
0.02s(0.14%)
of 0.15s(1.09%)
runtime.semasleep.func1
0 of 0.82s(5.93%)
runtime.newproc.func1
0 of 0.23s(1.66%)
net.(*conn).Write
0 of 5s(36.18%)
0.23s
4.24s
0.14s
0.82s
0.23s
5s
bufio.(*Reader).ReadLine
0 of 4.24s(30.68%)
github.com/prashantv/go_profiling_talk/handlers.getStatsTags
0 of 0.14s(1.01%)
runtime.semasleep1
0 of 0.82s(5.93%)
runtime.newproc1
0.01s(0.072%)
of 0.23s(1.66%)
net.(*netFD).Write
0 of 5s(36.18%)
4.24s
0.12s
0.78s
0.21s
4.91s
bufio.(*Reader).ReadSlice
0 of 4.24s(30.68%)
github.com/prashantv/go_profiling_talk/handlers.parseUserAgent
0 of 0.12s(0.87%)
runtime.mach_semaphore_wait
0.78s(5.64%)
runtime.wakep
0 of 0.30s(2.17%)
syscall.Write
0 of 4.91s(35.53%)
4.24s
0.12s
0.30s
4.91s
bufio.(*Reader).fill
0 of 4.24s(30.68%)
github.com/varstr/uaparser.Parse
0 of 0.12s(0.87%)
runtime.startm
0 of 0.60s(4.34%)
syscall.write
0 of 4.91s(35.53%)
4.24s
0.12s
0.59s
0.05s
net/http.(*connReader).Read
0 of 4.24s(30.68%)
github.com/varstr/uaparser.searchIn
0 of 0.12s(0.87%)
runtime.notewakeup
0 of 0.64s(4.63%)
4.23s
0.12s
0.64s
net.(*conn).Read
0.01s(0.072%)
of 4.46s(32.27%)
github.com/varstr/uaparser.matchSpec
0.04s(0.29%)
of 0.12s(0.87%)
runtime.semawakeup
0 of 0.65s(4.70%)
4.44s
0.08s
0.65s
4.91s
net.(*netFD).Read
0.01s(0.072%)
of 4.44s(32.13%)
strings.Contains
0.02s(0.14%)
of 0.08s(0.58%)
runtime.mach_semrelease
0 of 0.65s(4.70%)
3.90s
0.53s
0.65s
syscall.Read
0 of 3.90s(28.22%)
runtime.deferreturn
0.01s(0.072%)
of 0.60s(4.34%)
runtime.mach_semaphore_signal
0.65s(4.70%)
3.90s
0.58s
syscall.read
0 of 3.90s(28.22%)
runtime.freedefer
0.58s(4.20%)
3.90s

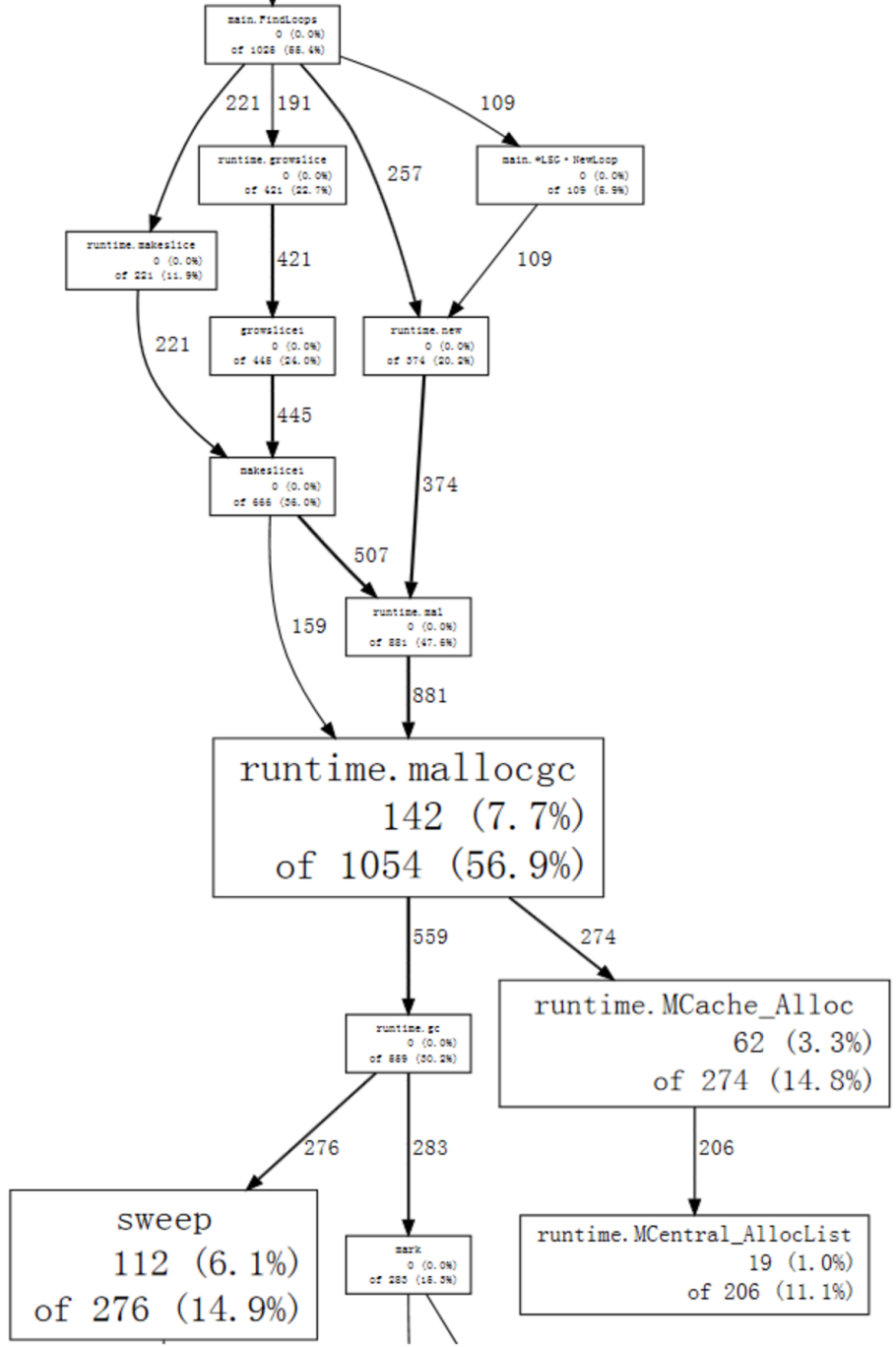
Flame Graph

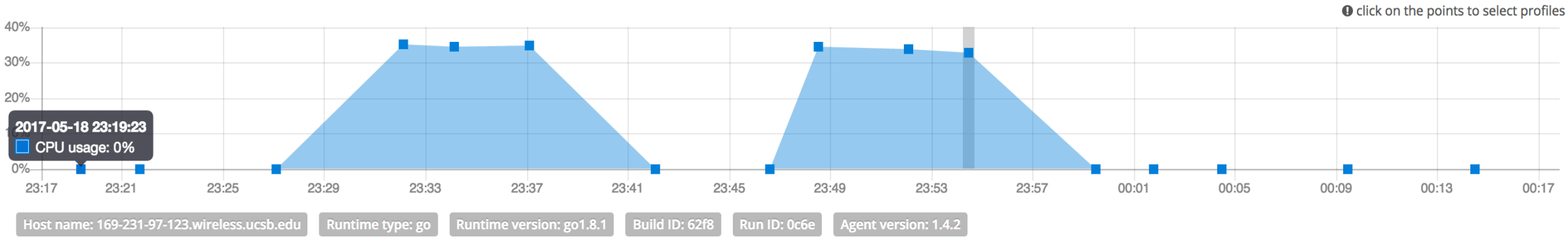

Flame Graph
Search
syscall.Syscall
syscall.read
syscall.Read
net.(*netFD).Read
net.(*conn).Read
net/http.(*connReader).Read
bufio.(*Reader).fill
bufio.(*Reader).ReadSlice
bufio.(*Reader).ReadLine
net/textproto.(*Reader).readLineSlice
net/textproto.(*Reader).ReadLine
net/http.readRequest
net/http.(*conn).readRequest
net/http.(*conn).serve
runtime.goexit
syscall.Syscall
syscall.write
syscall.Write
net.(*netFD).Write
net.(*conn).Write
net/http.checkConnErrorWriter.Write
bufio.(*Writer).flush
bufio.(*Writer).Flush
net/http.(*response).finishRequest
syscall.Syscall6
syscall.sysctl
syscall.nametomib
syscall.Sysctl
os.hostname
os.Hostname
profiling_talk/handlers.getStatsTags
profiling_talk/handlers.WithStats.func1
net/http.HandlerFunc.ServeHTTP
net/http.(*ServeMux).ServeHTTP
net/http.serverHandler.ServeHTTP
syscall.Syscall6
syscall.sysctl
r..
r..
r..
r..
r..
r..

1. Where shall we start?

- 3) Based on the diagrams, find out the hot program and the place to start our profiling:




main.FindLoops
0 (0.0%)
of 1028 (66.4%)
221
191
257
109
runtime.growslice
0 (0.0%)
of 421 (22.7%)
main.*LSC + NewLoop
0 (0.0%)
of 109 (6.9%)
runtime.makeslice
0 (0.0%)
of 221 (11.9%)
421
109
221
growslice1
0 (0.0%)
of 445 (24.0%)
runtime.new
0 (0.0%)
of 374 (20.2%)
445
makeslice1
0 (0.0%)
of 666 (36.0%)
374
507
159
runtime.mal
0 (0.0%)
of 881 (47.6%)
881
runtime.mallocgc
142 (7.7%)
of 1054 (56.9%)
559
274
runtime.gc
0 (0.0%)
of 869 (50.2%)
runtime.MCache_Alloc
62 (3.3%)
of 274 (14.8%)
276
283
206
sweep
112 (6.1%)
of 276 (14.9%)
mark
0 (0.0%)
of 283 (15.3%)
runtime.MCentral_AllocList
19 (1.0%)
of 206 (11.1%)



Host name: 169-231-97-123.wireless.ucsb.edu | Runtime type: go | Runtime version: go1.8.1 | Build ID: 62f8 | Run ID: 0c6e | Agent version: 1.4.2

net/http.(*conn).serve (/usr/local/go/src/net/http/server.go:1830)	13.625%	545 samples	↓ 1%
net/http.(*conn).serve (/usr/local/go/src/net/http/server.go:1763)	10.4%	416 samples	↑ 7%
runtime.mcall (/usr/local/go/src/runtime/asm_amd64.s:269)	5.25%	210 samples	↓ 22%
net/http.(*conn).serve (/usr/local/go/src/net/http/server.go:1825)	0.975%	39 samples	↓ 17%
net/http.(*conn).serve (/usr/local/go/src/net/http/server.go:1815)	0.875%	35 samples	↑ 9%
runtime.mstart (/usr/local/go/src/runtime/proc.go:1149)	0.5%	20 samples	↓ 17%
net/http.(*connReader).backgroundRead (/usr/local/go/src/net/http/server.go:656)	0.5%	20 samples	↑ 11%
runtime.gcBgMarkWorker (/usr/local/go/src/runtime/mgc.go:1537)	0.2%	8 samples	↓ 38%
net/http.(*conn).serve (/usr/local/go/src/net/http/server.go:1826)	0.075%	3 samples	↓ 25%
runtime.morestack (/usr/local/go/src/runtime/asm_amd64.s:398)	0.075%	3 samples	↓ 50%
runtime.bgsweep (/usr/local/go/src/runtime/mgcsweep.go:59)	0.05%	2 samples	↑ 100%
runtime._System (/usr/local/go/src/runtime/proc.go:3173)	0.05%	2 samples	↑ 100%
net/http.(*conn).serve (/usr/local/go/src/net/http/server.go:1827)	0.05%	2 samples	↑ 100%
net/http.(*conn).serve (/usr/local/go/src/net/http/server.go:1766)	0.025%	1 sample	
net/http.(*conn).serve (/usr/local/go/src/net/http/server.go:1854)	0.025%	1 sample	↑ 0%
runtime.gcBgMarkWorker (/usr/local/go/src/runtime/mgc.go:1581)	0.025%	1 sample	↓ 83%

2: What and how do we profile?

- **A) intermediate objects:** one simple example

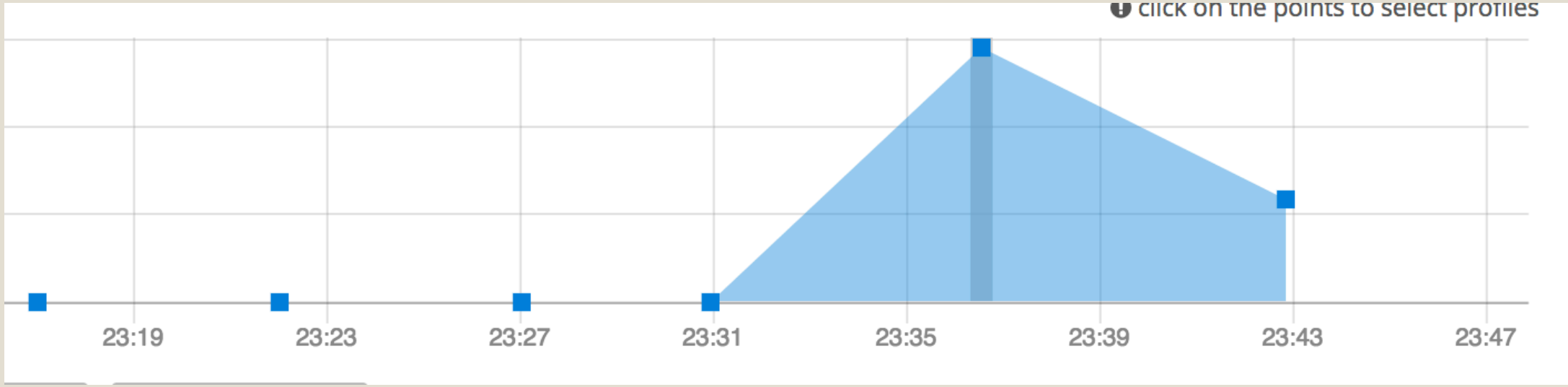
```
var keyOrder []string
if _, ok := tags["host"]; ok {
    keyOrder = append(keyOrder, "host")
}
keyOrder = append(keyOrder, "endpoint", "os", "browser")
```

```
//var keyOrder []string
var keyOrder := make([]string,0,4)
if _, ok := tags["host"]; ok {
    keyOrder = append(keyOrder, "host")
}
keyOrder = append(keyOrder, "endpoint", "os", "browser")
```

Result after

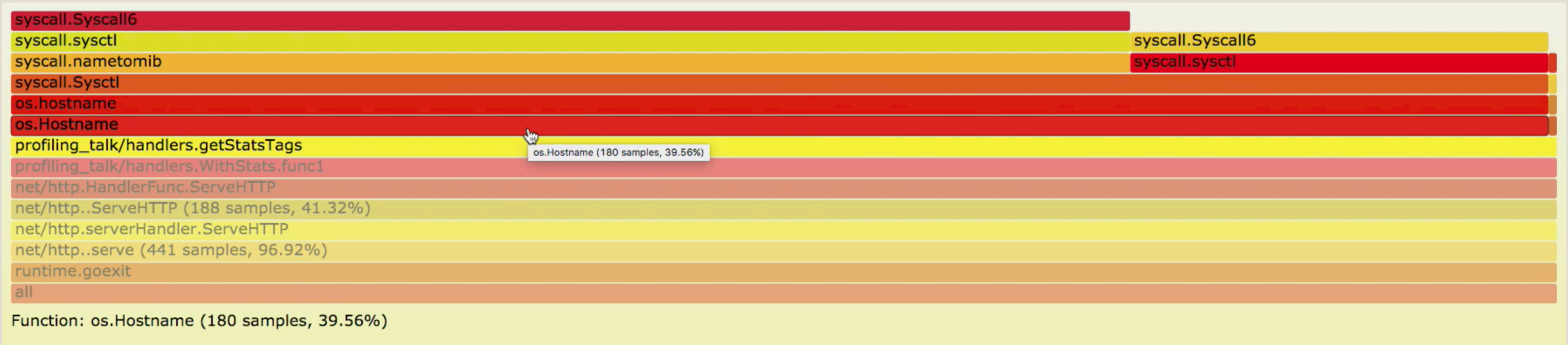
- 169-231-98-53:stats shuyanli$ go test -bench . -benchmem -cpuprofile prof.cpu
- BenchmarkAddTagsToName-8 500000 **1214 ns/op** 220 B/op
- **17 allocs/op**
-
- **and now we have:**
- ShuyanmatoMacBook-Pro:stats shuyanli$ go test -bench . -benchmem -cpuprofile prof.cpu
- BenchmarkAddTagsToName-8 300000. **580 ns/op** 144 B/o
- **10 allocs/op**

CPU performace after profiling



2: What and how do we profile?

- **B) cache**



```go
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":     _host,   //cache by calling the function just once
    }

    //=========by caching this hostname, we can improve the performance of this function
    // host, err := os.Hostname()
    // if err == nil {
    //  if idx := strings.IndexByte(host, '.'); idx > 0 {
    //      host = host[:idx]
    //  }
    //  stats["host"] = host
    // }
    return stats
}

//cache the host
func getHost() string{
    host, err := os.Hostname()
    if err == nil {
        if idx := strings.IndexByte(host, '.'); idx > 0 {
            host = host[:idx]
        }
    }
    return host

}
```

2: What and how do we profile?

- B) Cache: improvement

```
 258429 requests in 4.884710008s, 27.85MB read
258429 requests in 4.884710008s, 27.85MB read
Requests/sec:         48752.70
Transfer/sec:         5.70MB
Avg Req Time:         189.015µs
Fastest Request:      58.198µs
Slowest Request:      151.564736ms
Number of Errors:     0
```

```
Running 5s test @ http://localhost:9090/hello
  10 goroutine(s) running concurrently
277317 requests in 4.876613144s, 29.89MB read
Requests/sec:         54866.72
Transfer/sec:         6.13MB
Avg Req Time:         175.849µs
Fastest Request:      53.672µs
Slowest Request:      4.814089ms
Number of Errors:     0
```

2: What and how do we profile?

- **C) 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.
- **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

2: What and how do we profile?

- **D) built-in library 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. They may not be widely-use, but they can be very fast.

```
(pprof) list addTagsToName
Total: 1.07s
/Users/shuyanli/Desktop/go_project/src/github.com/prashantv/go_profiling_talk/stats/reporte
r.go
      0      1.14s (flat, cum) 99.13% of Total
         .          .     36:
         .          .     37:func addTagsToName(name string, tags map[string]string) string
{
         .          .     38:      // The format I want is: host.endpoint.os.browser
         .          .     39:      // if there's no host tag, then I 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.]`)
```

```
 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:}
```

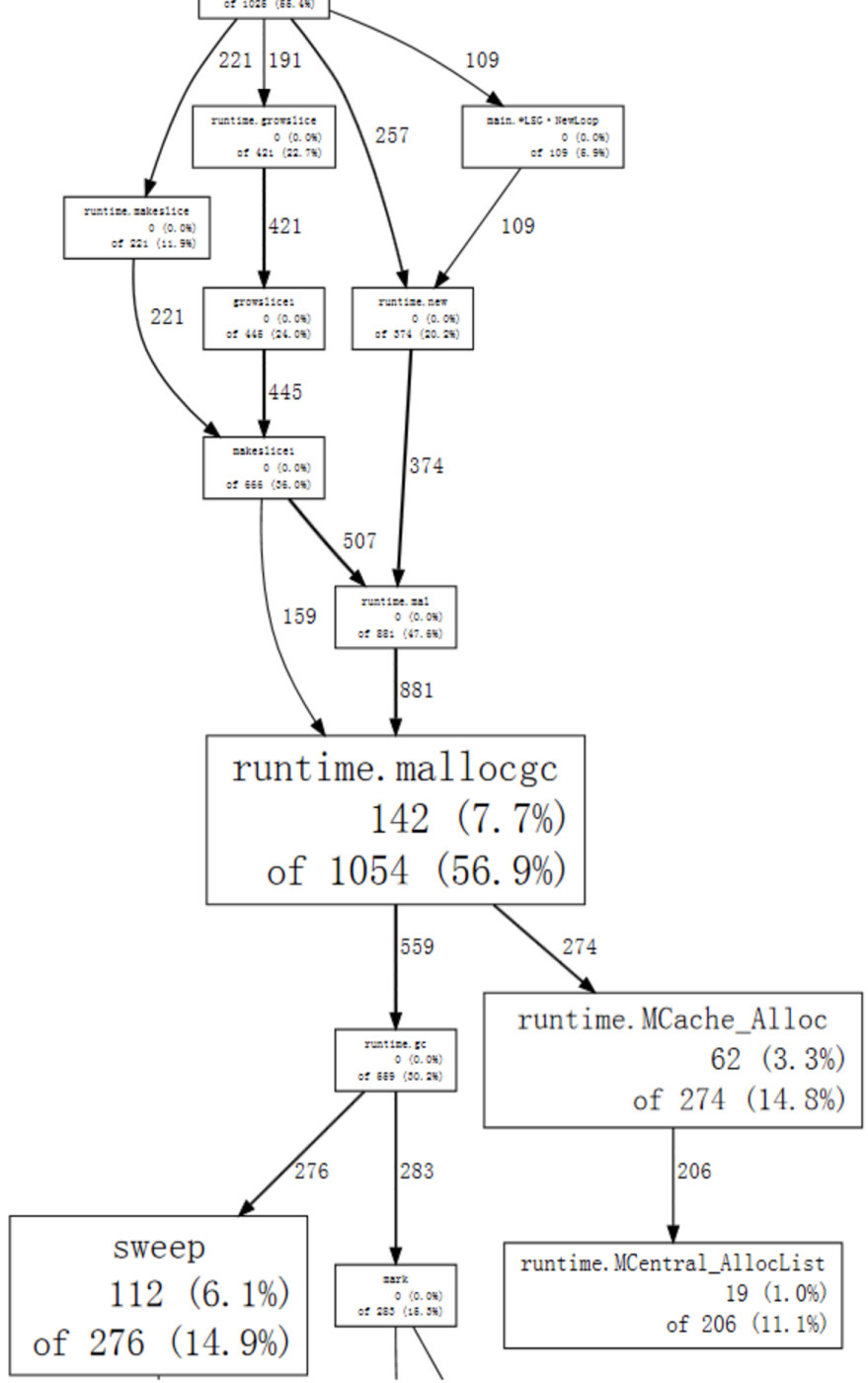
```go
//second implementation:
//write out to the buffer directly instead of creating intermediate object
func clean(buff *bytes.Buffer, value string) {

    //res := make([]byte, len(value ))
    for i := 0; i < len(value); i++ {
        switch c := value[i]; c {
        case '{', '}', '/', '\\', ':', ' ', '\t', '.':
            buff.WriteByte('-')
            //res[i] = '-'
        default:
            buff.WriteByte(c)
            //res[i] = c
        }
    }

    //return string(res)
}
```

Result:

- 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

2: What and how do we profile?

- **E) avoid allocation**
- An allocation means a possible GC. This doesn't mean GC is slow. GC is actually pretty fast. It's just because every time we allocate, the run time has to 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.


221
191
109
257
runtime.growslice
0 (0.0%)
of 421 (22.7%)
main.*LSC * NewLoop
0 (0.0%)
of 109 (5.9%)
runtime.makeslice
0 (0.0%)
of 221 (11.9%)
421
109
growslice1
0 (0.0%)
of 446 (24.0%)
runtime.new
0 (0.0%)
of 374 (20.2%)
221
445
makeslice1
0 (0.0%)
of 666 (36.0%)
374
507
runtime.mal
0 (0.0%)
of 881 (47.6%)
159
881
runtime.mallocgc
142 (7.7%)
of 1054 (56.9%)
559
274
runtime.gc
0 (0.0%)
of 559 (30.2%)
runtime.MCache_Alloc
62 (3.3%)
of 274 (14.8%)
276
283
206
sweep
112 (6.1%)
of 276 (14.9%)
mark
0 (0.0%)
of 283 (15.3%)
runtime.MCentral_AllocList
19 (1.0%)
of 206 (11.1%)

Brief idea of where allocation come from

```
var keyOrder []string
if _, ok := tags["host"]; ok {
    keyOrder = append(keyOrder, "host")
}
keyOrder = append(keyOrder, "endpoint", "os", "browser")
```

- we first create a slice called KeyOrder, which in this case is an empty slice that doesn't have any capacity and length. We then use append to add the host string to this slice. Now there is a re-allocation since we allocate a new slice, copy over all the elements (actually there is nothing we can copy form keyOrder) to the keyOrder. Then we call the append again to append the slice. In other words, we did 2 more allocation, which was a waste of time and space.

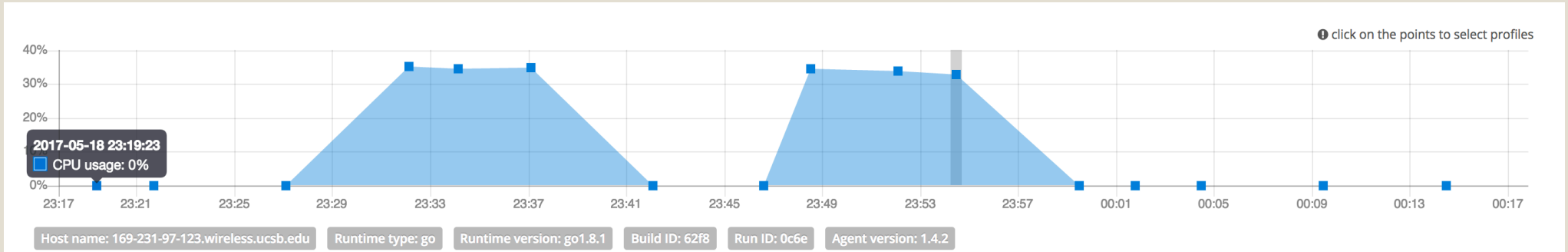
Small steps every day

```
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
```

```
[ShuyanmatoMacBook-Pro:stats shuyanli$ go test -bench . -benchmem -cpuprofile pro]
f.cpu
BenchmarkAddTagsToName-8         3000000               437 ns/op             208
 B/op          10 allocs/op
```

```
BenchmarkAddTagsToName-8         5000000               359 ns/op             160
 B/op           2 allocs/op
PASS
```

stackImpect



net/http.(*conn).serve (/usr/local/go/src/net/http/server.go:1830)	13.625%	545 samples	↓ 1%
net/http.(*conn).serve (/usr/local/go/src/net/http/server.go:1763)	10.4%	416 samples	↑ 7%
runtime.mcall (/usr/local/go/src/runtime/asm_amd64.s:269)	5.25%	210 samples	↓ 22%
net/http.(*conn).serve (/usr/local/go/src/net/http/server.go:1825)	0.975%	39 samples	↓ 17%
net/http.(*conn).serve (/usr/local/go/src/net/http/server.go:1815)	0.875%	35 samples	↑ 9%
runtime.mstart (/usr/local/go/src/runtime/proc.go:1149)	0.5%	20 samples	↓ 17%
net/http.(*connReader).backgroundRead (/usr/local/go/src/net/http/server.go:656)	0.5%	20 samples	↑ 11%
runtime.gcBgMarkWorker (/usr/local/go/src/runtime/mgc.go:1537)	0.2%	8 samples	↓ 38%
net/http.(*conn).serve (/usr/local/go/src/net/http/server.go:1826)	0.075%	3 samples	↓ 25%
runtime.morestack (/usr/local/go/src/runtime/asm_amd64.s:398)	0.075%	3 samples	↓ 50%
runtime.bgsweep (/usr/local/go/src/runtime/mgcsweep.go:59)	0.05%	2 samples	↑ 100%
runtime._System (/usr/local/go/src/runtime/proc.go:3173)	0.05%	2 samples	↑ 100%
net/http.(*conn).serve (/usr/local/go/src/net/http/server.go:1827)	0.05%	2 samples	↑ 100%
net/http.(*conn).serve (/usr/local/go/src/net/http/server.go:1766)	0.025%	1 sample	
net/http.(*conn).serve (/usr/local/go/src/net/http/server.go:1854)	0.025%	1 sample	↑ 0%
runtime.gcBgMarkWorker (/usr/local/go/src/runtime/mgc.go:1581)	0.025%	1 sample	↓ 83%

Tear program apart

- Writing a **benchmark** can help figuring out the details of the program without keep running the program repeatedly. More important, sometime allocation number of one function is high due to the functions it calls. Sometimes these functions hide very deep. Using a stackImpect cannot help to find them out but using a benchmark can.
- What's more, disassemble the function can also help. For example:

Tear program apart

```
         .          .    10f00fb: CMPQ BX, DI
         .          .    10f00fe: JGE 0x10f0122
      10ms       10ms    10f0100: MOVZX 0(SI)(DI*1), R8
         .          .    10f0105: CMPL $0x2f, R8
         .          .    10f0109: JA 0x10f0185
         .          .    10f010b: CMPL $0x9, R8
         .          .    10f010f: JNE 0x10f0163
         .          .    10f0111: CMPQ AX, DI
         .          .    10f0114: JAE 0x10f015c
         .          .    10f0116: MOVB $0x2d, 0(DX)(DI*1)
         .          .    10f011a: INCQ DI
         .          .    10f011d: CMPQ BX, DI
         .          .    10f0120: JL 0x10f0100
         .          .    10f0122: MOVQ $0x0, 0(SP)
         .          .    10f012a: MOVQ DX, 0x8(SP)
         .          .    10f012f: MOVQ AX, 0x10(SP)
         .          .    10f0134: MOVQ CX, 0x18(SP)
         .      230ms    10f0139: CALL runtime.slicebytetostring(SB)
         .          .    10f013e: MOVQ 0x28(SP), AX
         .          .    10f0143: MOVQ 0x20(SP), CX
         .          .    10f0148: MOVQ CX, 0x50(SP)
         .          .    10f014d: MOVQ AX, 0x58(SP)
         .          .    10f0152: MOVQ 0x30(SP), BP
         .          .    10f0157: ADDQ $0x38, SP
```

Thank you

- Q&A