# Profiling Go Programs

November 6, 2013

John Graham-Cumming

# The Golden Rule

- Premature optimization is the root of all evil – Hoare

- My version…
  - Don’t waste programmer cycles saving the wrong CPU cycles
  - (applies to memory as well)

- Measure, measure, measure
  - Then you can optimize

# Some Go Measurement Tools

- Timing
  - Shell `time` command
  - `time.Now(), time.Duration()`
  - `pprof.StartCPUProfile(f), pprof.StopCPUProfile()`
  - `go tool pprof http://localhost:6060/debug/pprof/profile`

- Memory
  - Shell `ps` command
  - `runtime.ReadMemStats(&m)`
  - `runtime.WriteHeapProfile(w)`
  - `go tool pprof http://localhost:6060/debug/pprof/heap`

# CPU PROFILING

# Example: badly implemented LRU Cache

```
package lru1

import “time”

type Item struct {
        key string
        value string
        last time.Time
}

type Cache struct {
        cap int
        data map[string]*Item
}

func NewCache(cap int) (*Cache) {
        return &Cache{cap, make(map[string]*Item)}
}
```

# Example: badly implemented LRU Cache

```go
func (c *Cache) makeSpace() {
    old := &Item{last: time.Now()}
    var key string
    for k, v := range c.data {
        if v.last.Before(old.last) {
            old = v
            key = k
        }
    }

    delete(c.data, key)
}

func (c *Cache) Put(key, value string) {
    if len(c.data) == c.cap {
        c.makeSpace()
    }

    c.data[key] = &Item{value, time.Now()}
}
```

# Example: badly implemented LRU Cache

```
func (c *Cache) Get(key string) (*Item) {
     if c.data[key] != nil {
          c.data[key].last = time.Now()
     }

     return c.data[key]
}
```

# Bad LRU Cache

- Use it to cache relationship between email addresses and their domain’s MX record
- Feed in 10M email addresses in arrival order; cache 10,000 MX records

```
% time ./lrutest1 < top10M
9929964 total 2565368 misses
82.39s user 25.22s system 99% cpu 1:47.61 total
```

- So 1m48s

# Let’s profile it!

- Use the pprof profiler

```
func main() {
    f, _ := os.Create("lrutest1.cpuprofile")
    pprof.StartCPUProfile(f)
    defer pprof.StopCPUProfile()

    ...
}
```

- Creates `lrutest1.cpuprofile`
- `go tool pprof lrutest1 lrutest1.cpuprofile`

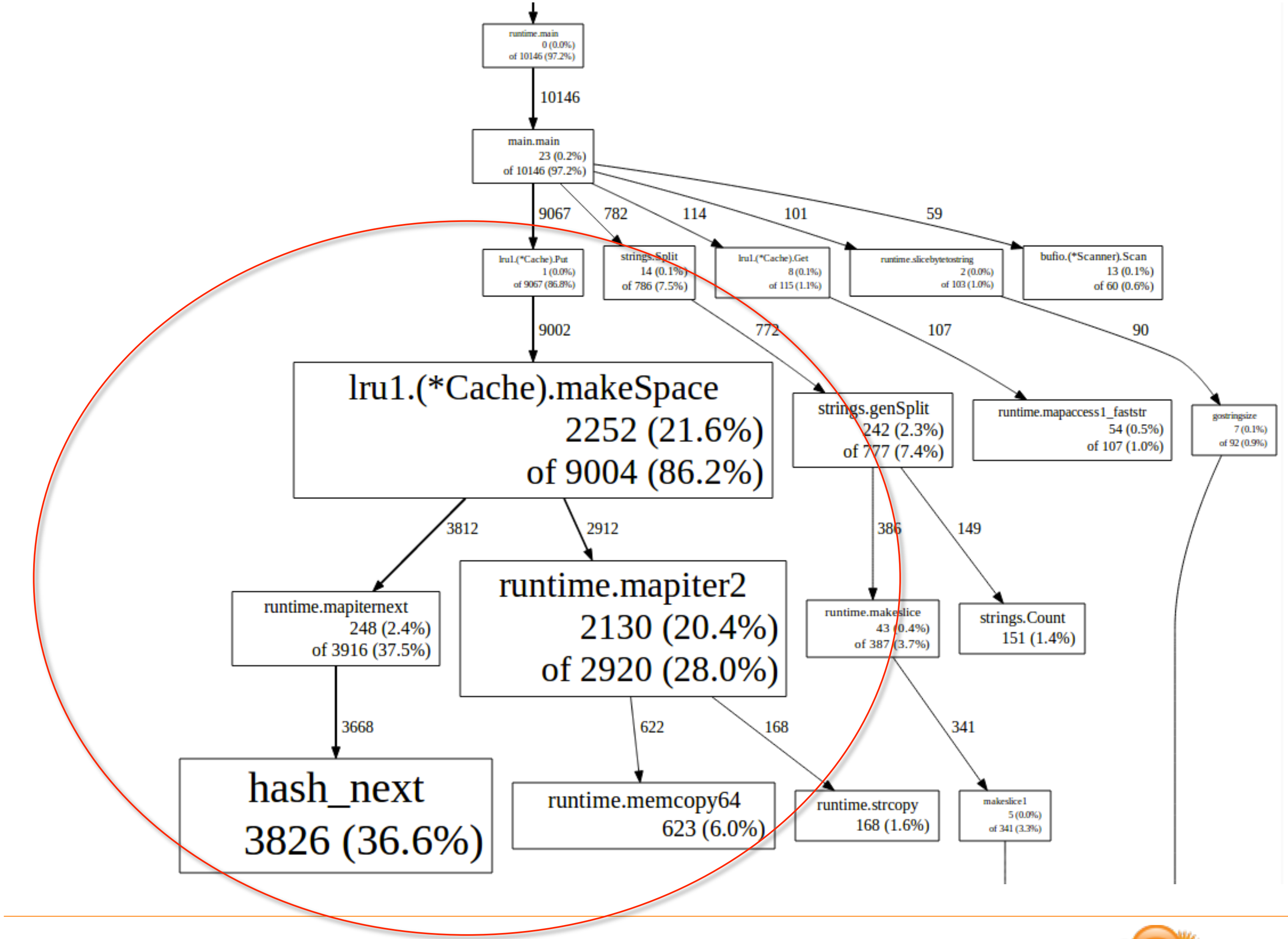
# pprof command-line

```
% go tool pprof lrutest1 lrutest1.cpuprofile
Welcome to pprof!  For help, type 'help'.
(pprof) top
Total: 10440 samples
    3826  36.6%  36.6%     3826  36.6% hash_next
    2252  21.6%  58.2%     9004  86.2% lru1.(*Cache).makeSpace
    2130  20.4%  78.6%     2920  28.0% runtime.mapiter2
     623   6.0%  84.6%      623   6.0% runtime.memcopy64
     248   2.4%  87.0%     3916  37.5% runtime.mapiternext
     242   2.3%  89.3%      777   7.4% strings.genSplit
     168   1.6%  90.9%      168   1.6% runtime.strcopy
     151   1.4%  92.3%      151   1.4% strings.Count
      64   0.6%  93.0%      195   1.9% scanblock
      62   0.6%  93.5%      390   3.7% runtime.mallocgc
```

# pprof command-line

```
(pprof) top -cum
Total: 10440 samples
       0   0.0%   0.0%    10146  97.2% gosched0
      23   0.2%   0.2%    10146  97.2% main.main
       0   0.0%   0.2%    10146  97.2% runtime.main
       1   0.0%   0.2%     9067  86.8% lru1.(*Cache).Put
    2252  21.6%  21.8%     9004  86.2% lru1.(*Cache).makeSpace
     248   2.4%  24.2%     3916  37.5% runtime.mapiternext
    3826  36.6%  60.8%     3826  36.6% hash_next
    2130  20.4%  81.2%     2920  28.0% runtime.mapiter2
      14   0.1%  81.4%      786   7.5% strings.Split
     242   2.3%  83.7%      777   7.4% strings.genSplit
```

# pprof web view



# Live profiling

- net/http/pprof

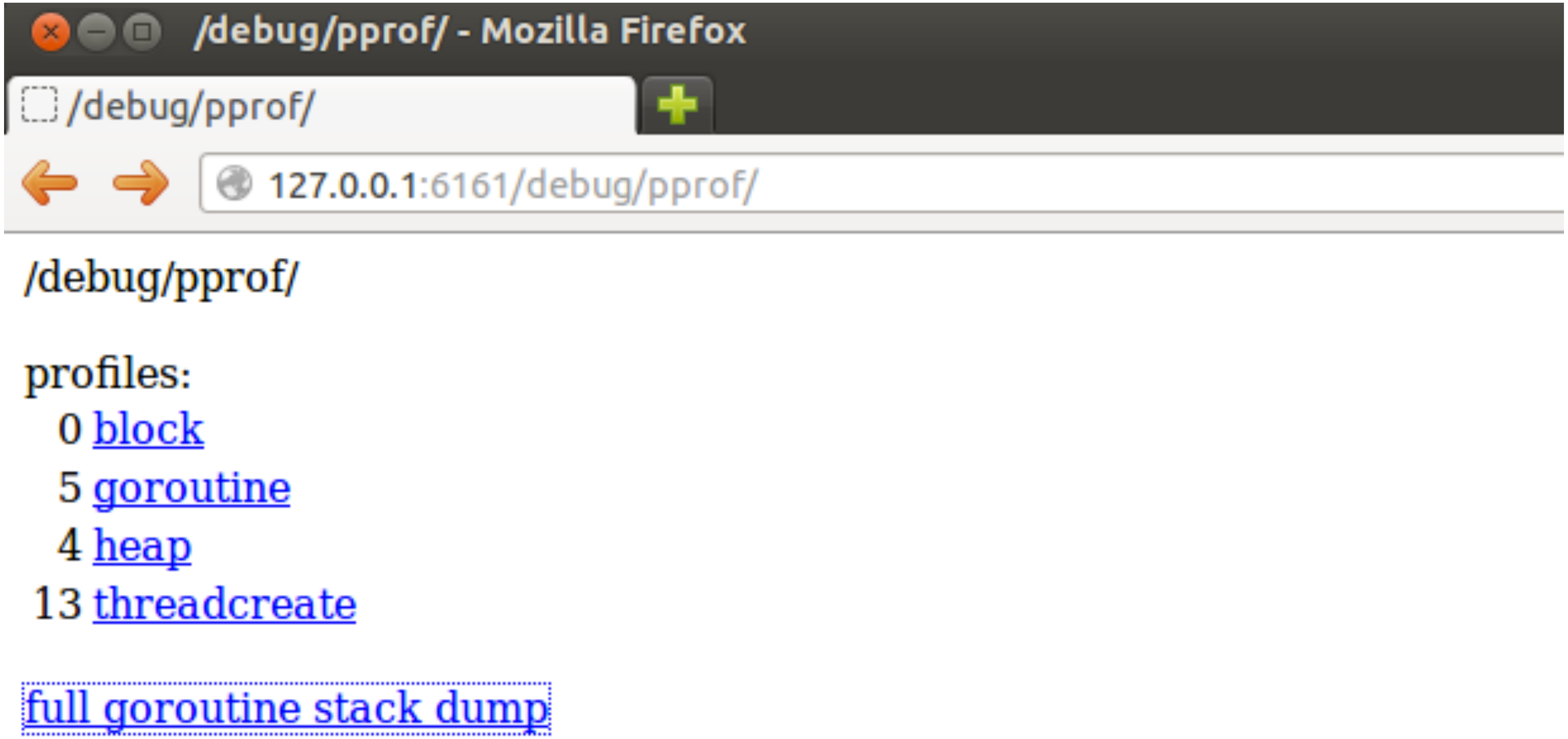
```
import _ “net/http/pprof”

go func() {
        log.Println(http.ListenAndServe("127.0.0.1:6161", nil))
}()
```

- /pprof/debug/heap
- /pprof/debug/profile

# Live command-line profiling

```
% go tool pprof  http://127.0.0.1:6161/debug/pprof/profile
Read http://127.0.0.1:6161/debug/pprof/symbol
Be patient...
Wrote profile to [...]
Welcome to pprof!  For help, type 'help'.
(pprof) top
Total: 2948 samples
    1146  38.9%  38.9%     1146  38.9% hash_next
     642  21.8%  60.7%     2618  88.8% lru1.(*Cache).makeSpace
     582  19.7%  80.4%      806  27.3% runtime.mapiter2
     176   6.0%  86.4%      176   6.0% runtime.memcopy64
      86   2.9%  89.3%     1194  40.5% runtime.mapiternext
      51   1.7%  91.0%      176   6.0% strings.genSplit
      48   1.6%  92.6%       48   1.6% runtime.strcopy
      43   1.5%  94.1%       43   1.5% runtime.futex
      43   1.5%  95.6%       43   1.5% strings.Count
      14   0.5%  96.0%       85   2.9% runtime.makeslice
(pprof)
```

# Live browser profiling



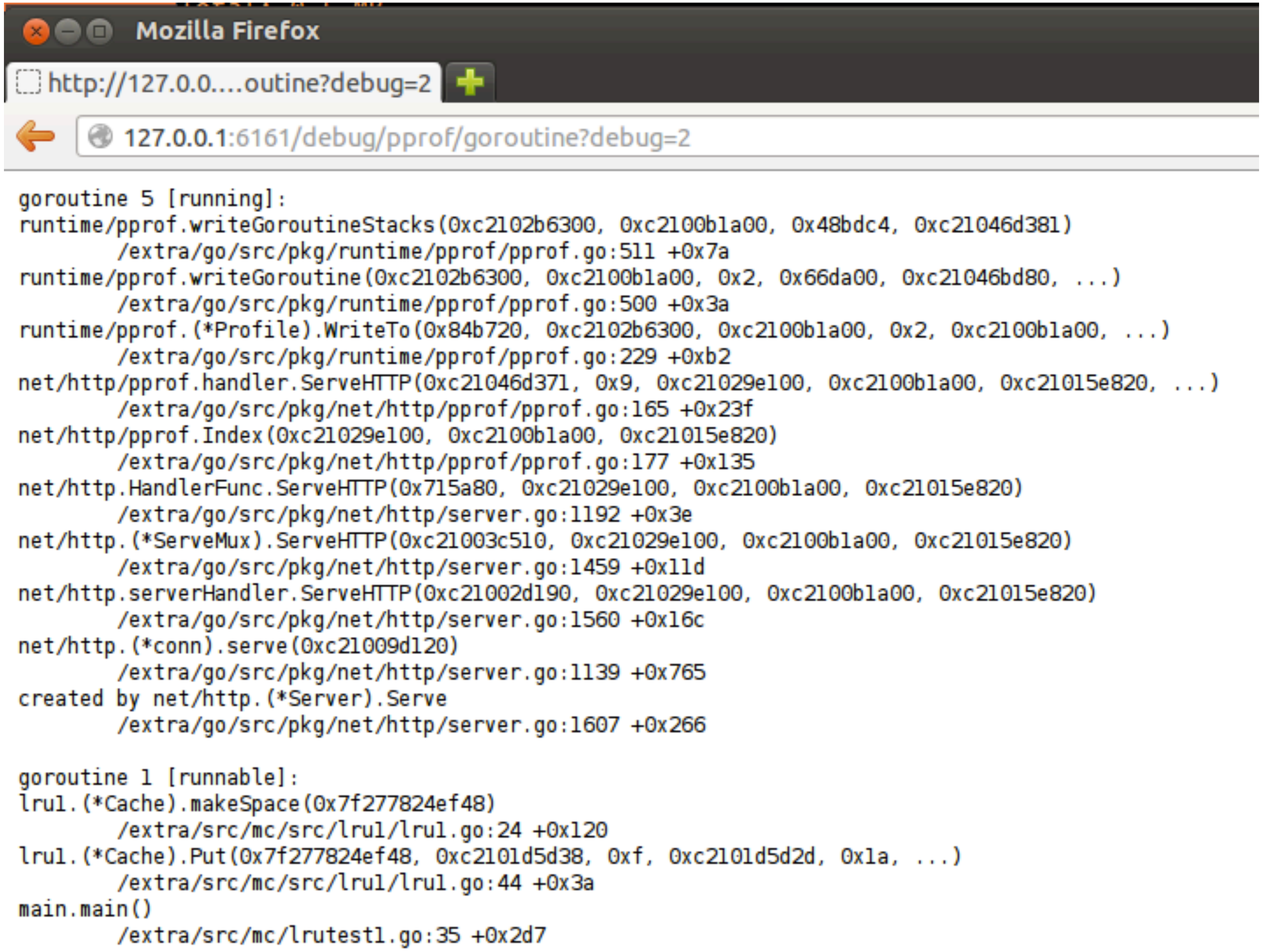
/debug/pprof/

profiles:

```
 0 block
 5 goroutine
 4 heap
13 threadcreate
```

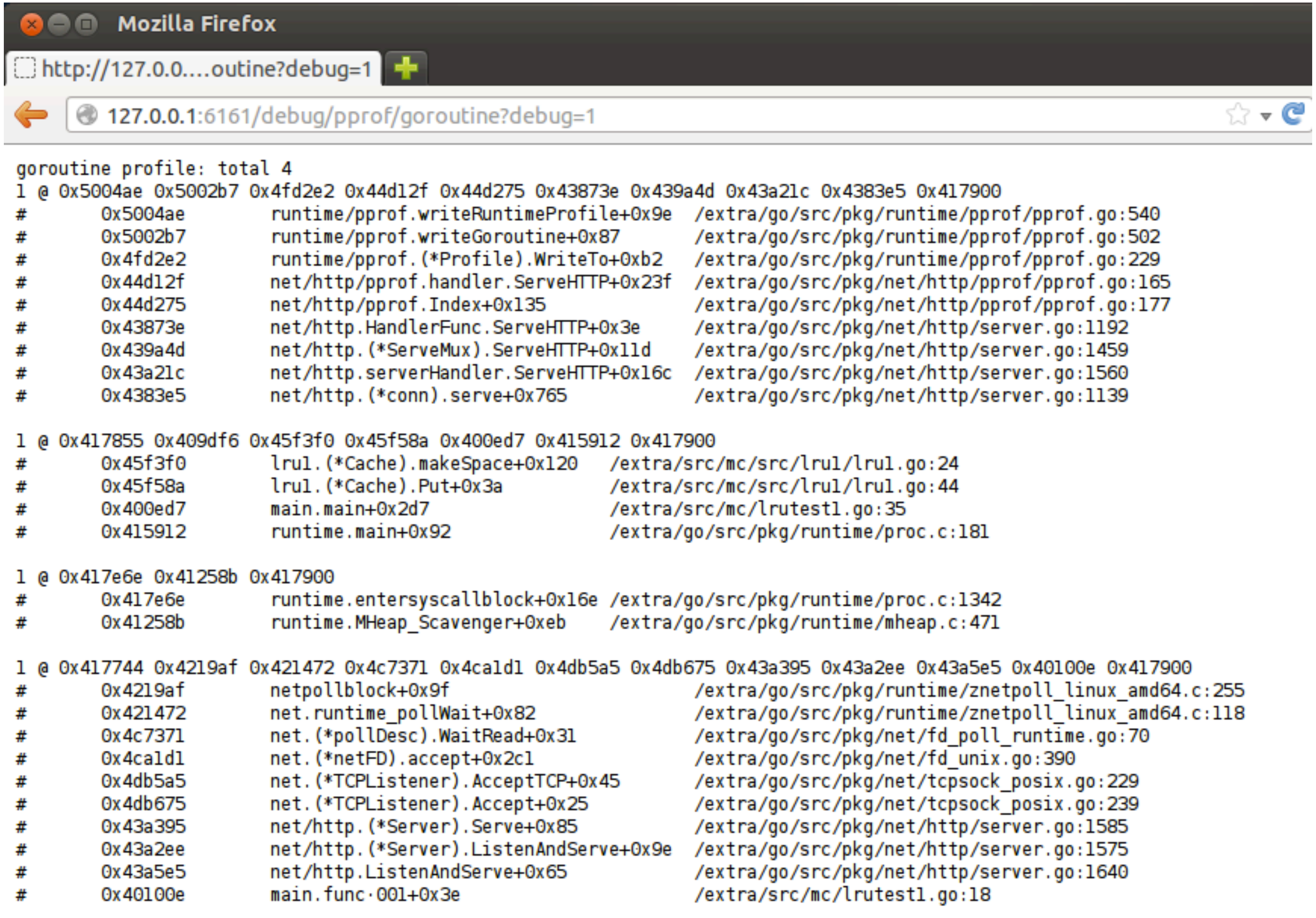
full goroutine stack dump

# Full goroutine stack dump


Mozilla Firefox
http://127.0.0....outine?debug=2
127.0.0.1:6161/debug/pprof/goroutine?debug=2

```
goroutine 5 [running]:
runtime/pprof.writeGoroutineStacks(0xc2102b6300, 0xc2100b1a00, 0x48bdc4, 0xc21046d381)
        /extra/go/src/pkg/runtime/pprof/pprof.go:511 +0x7a
runtime/pprof.writeGoroutine(0xc2102b6300, 0xc2100b1a00, 0x2, 0x66da00, 0xc21046bd80, ...)
        /extra/go/src/pkg/runtime/pprof/pprof.go:500 +0x3a
runtime/pprof.(*Profile).WriteTo(0x84b720, 0xc2102b6300, 0xc2100b1a00, 0x2, 0xc2100b1a00, ...)
        /extra/go/src/pkg/runtime/pprof/pprof.go:229 +0xb2
net/http/pprof.handler.ServeHTTP(0xc21046d371, 0x9, 0xc21029e100, 0xc2100b1a00, 0xc21015e820, ...)
        /extra/go/src/pkg/net/http/pprof/pprof.go:165 +0x23f
net/http/pprof.Index(0xc21029e100, 0xc2100b1a00, 0xc21015e820)
        /extra/go/src/pkg/net/http/pprof/pprof.go:177 +0x135
net/http.HandlerFunc.ServeHTTP(0x715a80, 0xc21029e100, 0xc2100b1a00, 0xc21015e820)
        /extra/go/src/pkg/net/http/server.go:1192 +0x3e
net/http.(*ServeMux).ServeHTTP(0xc21003c510, 0xc21029e100, 0xc2100b1a00, 0xc21015e820)
        /extra/go/src/pkg/net/http/server.go:1459 +0x11d
net/http.serverHandler.ServeHTTP(0xc21002d190, 0xc21029e100, 0xc2100b1a00, 0xc21015e820)
        /extra/go/src/pkg/net/http/server.go:1560 +0x16c
net/http.(*conn).serve(0xc21009d120)
        /extra/go/src/pkg/net/http/server.go:1139 +0x765
created by net/http.(*Server).Serve
        /extra/go/src/pkg/net/http/server.go:1607 +0x266

goroutine 1 [runnable]:
lru1.(*Cache).makeSpace(0x7f277824ef48)
        /extra/src/mc/src/lru1/lru1.go:24 +0x120
lru1.(*Cache).Put(0x7f277824ef48, 0xc2101d5d38, 0xf, 0xc2101d5d2d, 0x1a, ...)
        /extra/src/mc/src/lru1/lru1.go:44 +0x3a
main.main()
        /extra/src/mc/lrutest1.go:35 +0x2d7
```

# Running goroutines


Mozilla Firefox
http://127.0.0....outine?debug=1
127.0.0.1:6161/debug/pprof/goroutine?debug=1

```
goroutine profile: total 4
1 @ 0x5004ae 0x5002b7 0x4fd2e2 0x44d12f 0x44d275 0x43873e 0x439a4d 0x43a21c 0x4383e5 0x417900
#       0x5004ae        runtime/pprof.writeRuntimeProfile+0x9e  /extra/go/src/pkg/runtime/pprof/pprof.go:540
#       0x5002b7        runtime/pprof.writeGoroutine+0x87       /extra/go/src/pkg/runtime/pprof/pprof.go:502
#       0x4fd2e2        runtime/pprof.(*Profile).WriteTo+0xb2   /extra/go/src/pkg/runtime/pprof/pprof.go:229
#       0x44d12f        net/http/pprof.handler.ServeHTTP+0x23f  /extra/go/src/pkg/net/http/pprof/pprof.go:165
#       0x44d275        net/http/pprof.Index+0x135              /extra/go/src/pkg/net/http/pprof/pprof.go:177
#       0x43873e        net/http.HandlerFunc.ServeHTTP+0x3e     /extra/go/src/pkg/net/http/server.go:1192
#       0x439a4d        net/http.(*ServeMux).ServeHTTP+0x11d    /extra/go/src/pkg/net/http/server.go:1459
#       0x43a21c        net/http.serverHandler.ServeHTTP+0x16c  /extra/go/src/pkg/net/http/server.go:1560
#       0x4383e5        net/http.(*conn).serve+0x765            /extra/go/src/pkg/net/http/server.go:1139

1 @ 0x417855 0x409df6 0x45f3f0 0x45f58a 0x400ed7 0x415912 0x417900
#       0x45f3f0        lru1.(*Cache).makeSpace+0x120   /extra/src/mc/src/lru1/lru1.go:24
#       0x45f58a        lru1.(*Cache).Put+0x3a          /extra/src/mc/src/lru1/lru1.go:44
#       0x400ed7        main.main+0x2d7                 /extra/src/mc/lrutest1.go:35
#       0x415912        runtime.main+0x92               /extra/go/src/pkg/runtime/proc.c:181

1 @ 0x417e6e 0x41258b 0x417900
#       0x417e6e        runtime.entersyscallblock+0x16e /extra/go/src/pkg/runtime/proc.c:1342
#       0x41258b        runtime.MHeap_Scavenger+0xeb    /extra/go/src/pkg/runtime/mheap.c:471

1 @ 0x417744 0x4219af 0x421472 0x4c7371 0x4ca1d1 0x4db5a5 0x4db675 0x43a395 0x43a2ee 0x43a5e5 0x40100e 0x417900
#       0x4219af        netpollblock+0x9f                       /extra/go/src/pkg/runtime/znetpoll_linux_amd64.c:255
#       0x421472        net.runtime_pollWait+0x82               /extra/go/src/pkg/runtime/znetpoll_linux_amd64.c:118
#       0x4c7371        net.(*pollDesc).WaitRead+0x31           /extra/go/src/pkg/net/fd_poll_runtime.go:70
#       0x4ca1d1        net.(*netFD).accept+0x2c1               /extra/go/src/pkg/net/fd_unix.go:390
#       0x4db5a5        net.(*TCPListener).AcceptTCP+0x45       /extra/go/src/pkg/net/tcpsock_posix.go:229
#       0x4db675        net.(*TCPListener).Accept+0x25          /extra/go/src/pkg/net/tcpsock_posix.go:239
#       0x43a395        net/http.(*Server).Serve+0x85           /extra/go/src/pkg/net/http/server.go:1585
#       0x43a2ee        net/http.(*Server).ListenAndServe+0x9e  /extra/go/src/pkg/net/http/server.go:1575
#       0x43a5e5        net/http.ListenAndServe+0x65            /extra/go/src/pkg/net/http/server.go:1640
#       0x40100e        main.func·001+0x3e                      /extra/src/mc/lrutest1.go:18
```

# Better LRU implementation

```
package lru2

import "container/list”

type Item struct {
        key string
        value string
}

type Cache struct {
        cap int
        data map[string]*list.Element
        l *list.List
}

func NewCache(cap int) (*Cache) {
        return &Cache{cap, make(map[string]*list.Element),
    list.New()}
}
```

# Better LRU implementation

```
func (c *Cache) Get(key string) (*Item) {
        if c.data[key] != nil {
                c.l.MoveToFront(c.data[key])
                return c.data[key].Value.(*Item)
        }

        return nil
}

func (c *Cache) Put(key, value string) {
        if len(c.data) == c.cap {
                delete(c.data, c.l.Back().Value.(*Item).key)
                c.l.Remove(c.l.Back())
        }

        c.data[key] = c.l.PushFront(&Item{key, value})
}
```

# Better LRU Cache

- Use it to cache relationship between email addresses and their domain's MX record
- Feed in 10M email addresses in arrival order; cache 10,000 MX records

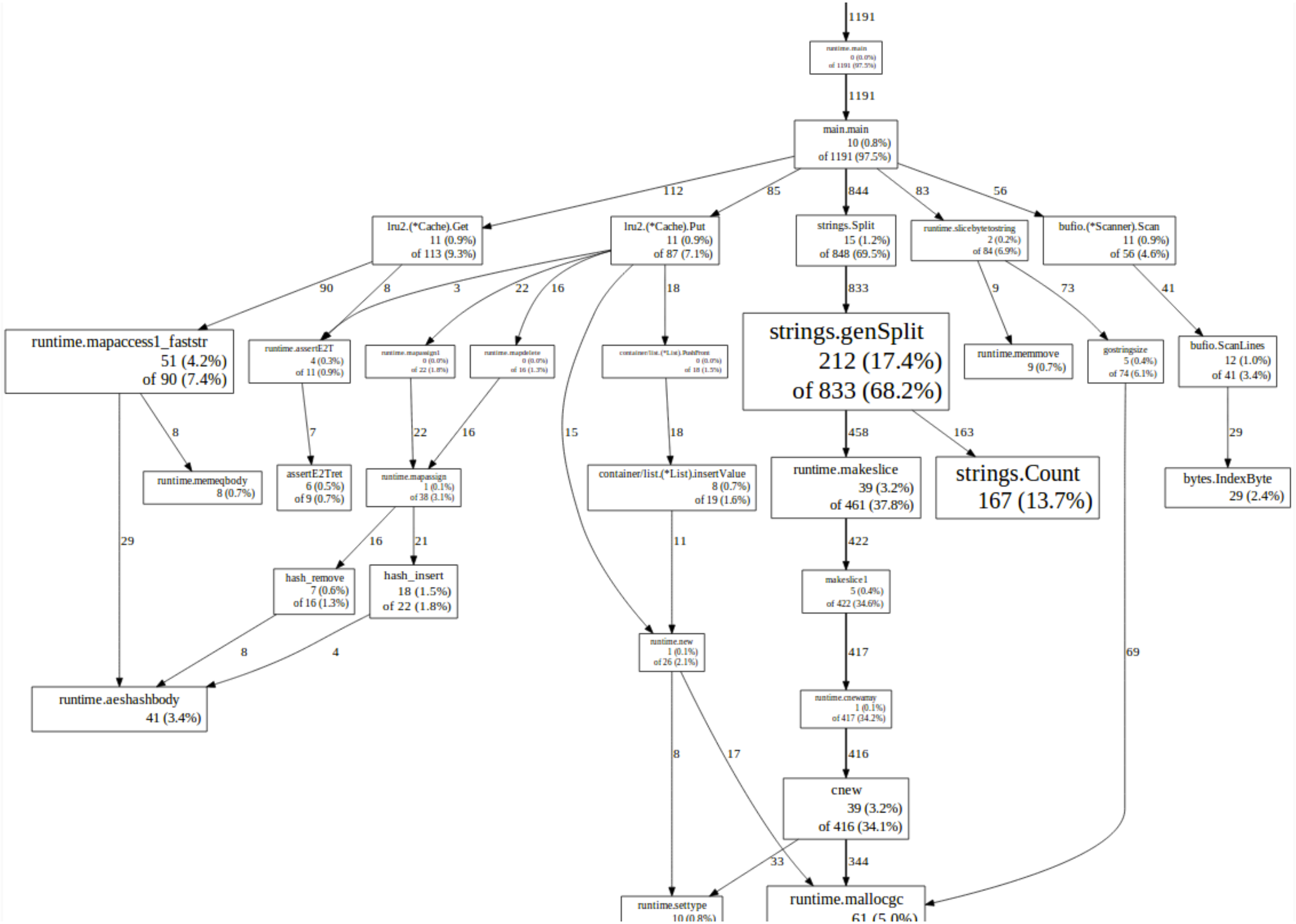
```
% time ./lrutest2 < top10M
9929964 total 2565368 misses
12.19s user 1.14s system 105% cpu 12.652 total
```

- So 12.3s

# pprof command-line

```
% go tool pprof lrutest2 lrutest2.cpuprofile
Welcome to pprof!  For help, type 'help'.
(pprof) top
Total: 1221 samples
     212  17.4%  17.4%      833  68.2% strings.genSplit
     167  13.7%  31.0%      167  13.7% strings.Count
      74   6.1%  37.1%       77   6.3% sweepspan
      71   5.8%  42.9%      206  16.9% scanblock
      61   5.0%  47.9%       61   5.0% markonly
      61   5.0%  52.9%      435  35.6% runtime.mallocgc
      51   4.2%  57.1%       51   4.2% flushptrbuf
      51   4.2%  61.3%       90   7.4% runtime.mapaccess1_faststr
      49   4.0%  65.3%       49   4.0% runtime.futex
      41   3.4%  68.6%       41   3.4% runtime.aeshashbody
```

# pprof web view



# Let’s try random eviction

```
package lru3

import "math/rand"
import "fmt"

type Item struct {
        key string
        value string
}

type Cache struct {
        cap int
        data map[string]*Item
        keys []string
}

func NewCache(cap int) (*Cache) {
        return &Cache{cap, make(map[string]*Item),
   make([]string, 0, cap)}
}
```

# Let’s try random eviction

```
func (c *Cache) Get(key string) (*Item) {
        return c.data[key]
}

func (c *Cache) Put(key, value string) {
        if len(c.keys) == c.cap {
                evict := rand.Intn(c.cap)
                delete(c.data, c.keys[evict])
                c.keys = append(c.keys[:evict],
                    c.keys[evict+1:]...)
        }

        c.keys = append(c.keys, key)

        c.data[key] = &Item{key, value}
}
```

# Random Eviction

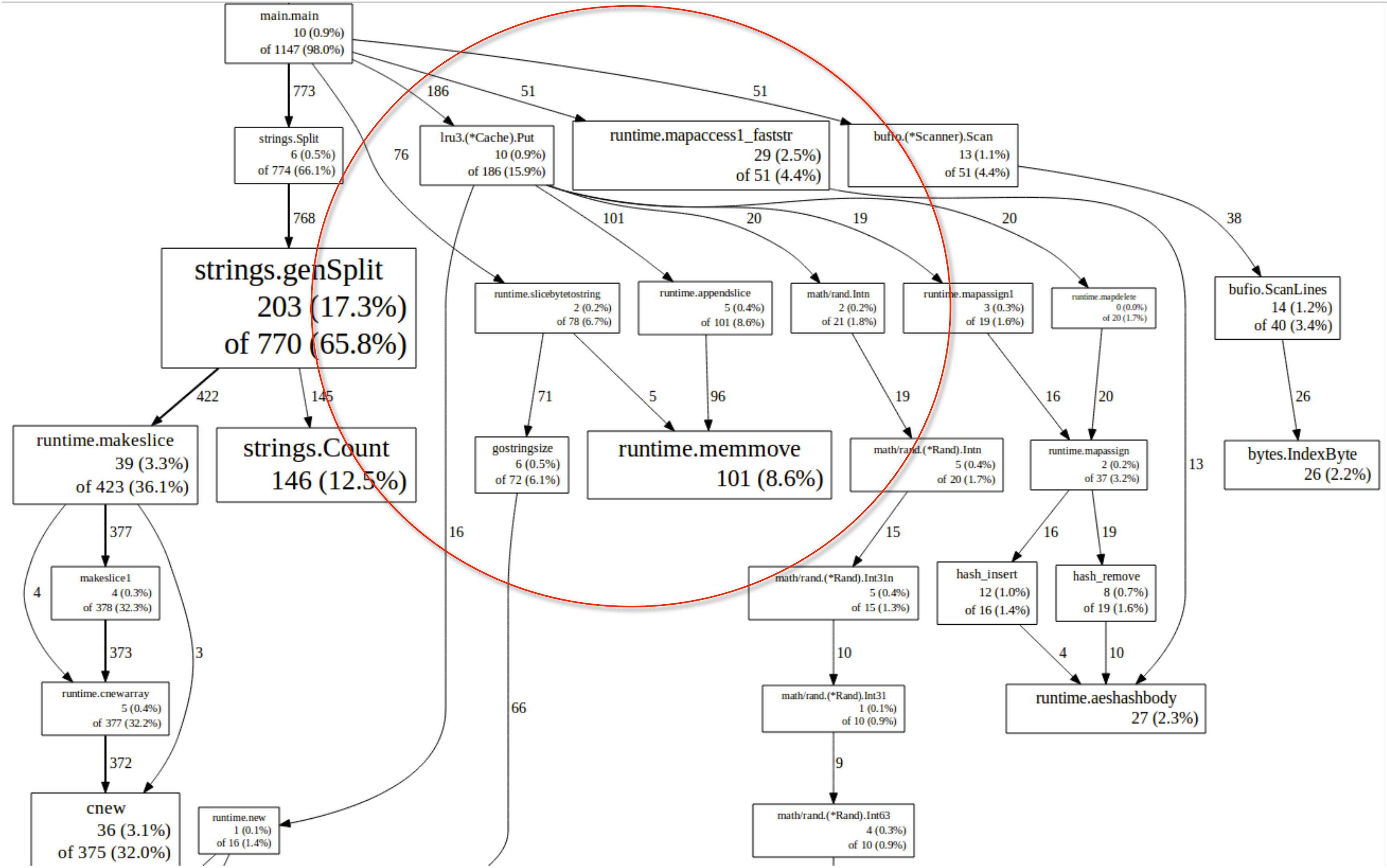
- Same speed for a different algorithm

```
% time ./lrutest3 < top10M
9929964 total 2820060 misses
11.76s user 1.02s system 105% cpu 12.130 total
```

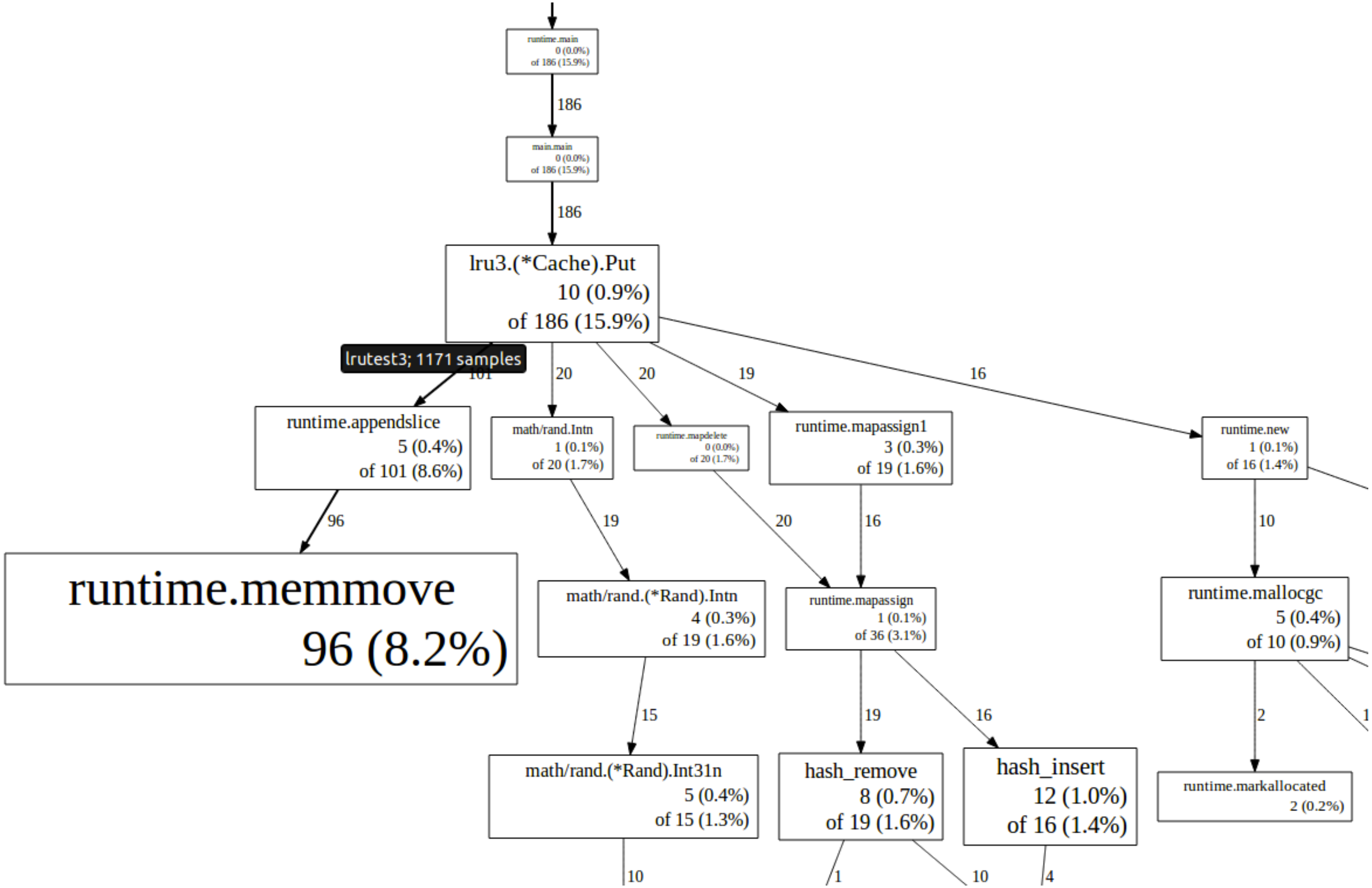
- So 12.1s

```
% go tool pprof lrutest3 lrutest3.cpuprofile
(pprof) top
Total: 1171 samples
     203  17.3%  17.3%      770  65.8% strings.genSplit
     146  12.5%  29.8%      146  12.5% strings.Count
     101   8.6%  38.4%      101   8.6% runtime.memmove
      70   6.0%  44.4%       77   6.6% sweepspan
      61   5.2%  49.6%      380  32.5% runtime.mallocgc
      60   5.1%  54.7%      146  12.5% scanblock
      54   4.6%  59.4%       54   4.6% markonly
```

# pprof web view



# web Put


runtime.main
0 (0.0%)
of 186 (15.9%)
186
main.main
0 (0.0%)
of 186 (15.9%)
186
lru3.(*Cache).Put
10 (0.9%)
of 186 (15.9%)
lrutest3; 1171 samples
101
20
20
19
16
runtime.appendslice
5 (0.4%)
of 101 (8.6%)
math/rand.Intn
1 (0.1%)
of 20 (1.7%)
runtime.mapdelete
0 (0.0%)
of 20 (1.7%)
runtime.mapassign1
3 (0.3%)
of 19 (1.6%)
runtime.new
1 (0.1%)
of 16 (1.4%)
96
19
20
16
10
runtime.memmove
96 (8.2%)
math/rand.(*Rand).Intn
4 (0.3%)
of 19 (1.6%)
runtime.mapassign
1 (0.1%)
of 36 (3.1%)
runtime.mallocgc
5 (0.4%)
of 10 (0.9%)
15
19
16
2
1
math/rand.(*Rand).Int31n
5 (0.4%)
of 15 (1.3%)
hash_remove
8 (0.7%)
of 19 (1.6%)
hash_insert
12 (1.0%)
of 16 (1.4%)
runtime.markallocated
2 (0.2%)
10
1
10
4

# Eliminate slice operation

```
package lru4

import "math/rand"

type Item struct {
        key string
        value string
}

type Cache struct {
        cap int
        data map[string]*Item
        keys []string
}

func NewCache(cap int) (*Cache) {
        return &Cache{cap, make(map[string]*Item),
    make([]string, cap)}
}
```

# Eliminate slice operation

```
func (c *Cache) Get(key string) (*Item) {
        return c.data[key]
}

func (c *Cache) Put(key, value string) {
        slot := len(c.data)
        if len(c.data) == c.cap {
                slot = rand.Intn(c.cap)
                delete(c.data, c.keys[slot])
        }

        c.keys[slot] = key
        c.data[key] = &Item{key, value}
}
```

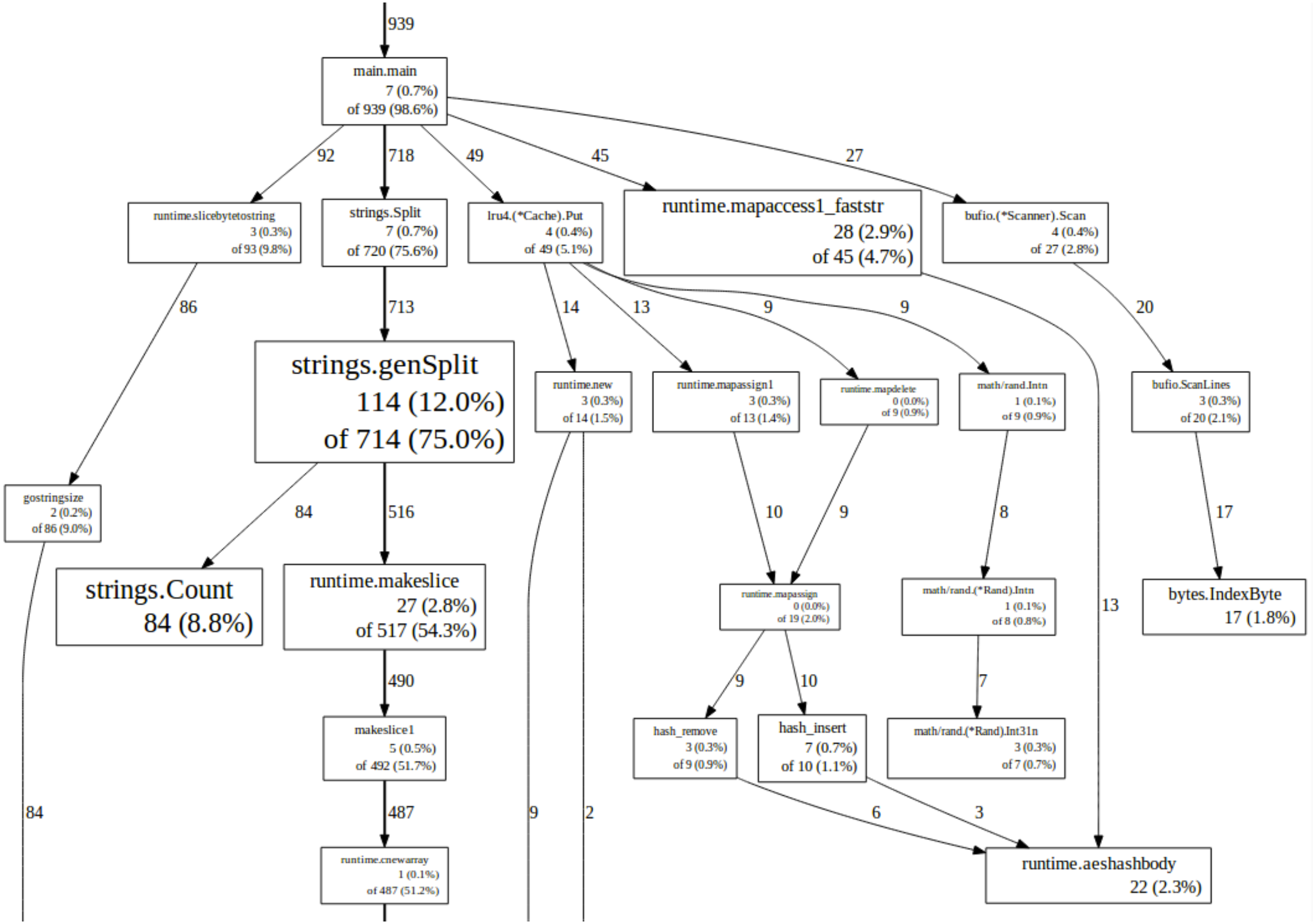
# Eliminate slice operation

- Slightly faster

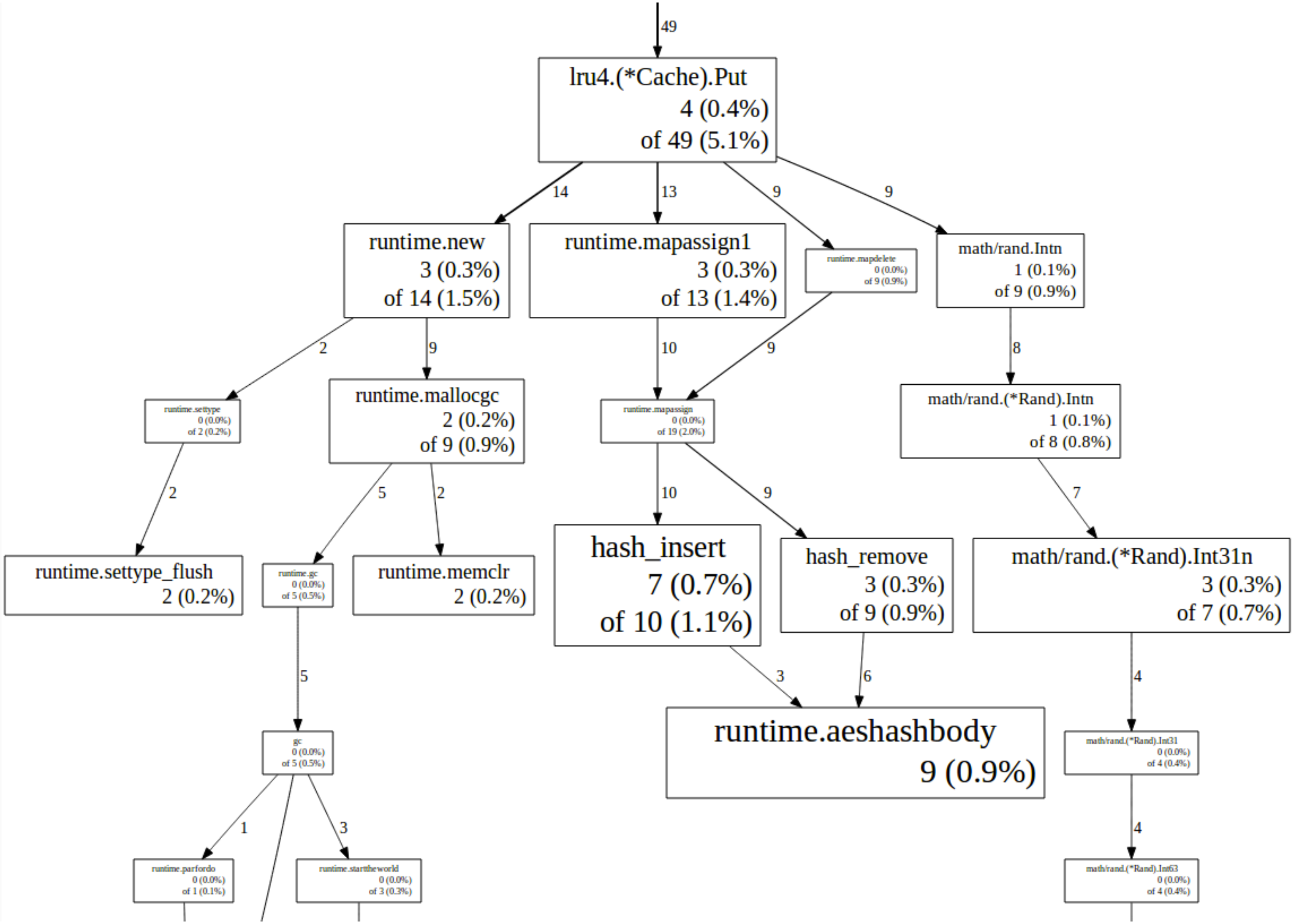
```
% time ./lrutest4 < top10M
9929964 total 2819425 misses
6.51s user 4.83s system 105% cpu 10.787 total
```

- So 10.8s

# Slice operations now gone



# web Put


49
lru4.(*Cache).Put
4 (0.4%)
of 49 (5.1%)
14
13
9
9
runtime.new
3 (0.3%)
of 14 (1.5%)
runtime.mapassign1
3 (0.3%)
of 13 (1.4%)
runtime.mapdelete
0 (0.0%)
of 9 (0.9%)
math/rand.Intn
1 (0.1%)
of 9 (0.9%)
2
9
10
9
8
runtime.settype
0 (0.0%)
of 2 (0.2%)
runtime.mallocgc
2 (0.2%)
of 9 (0.9%)
runtime.mapassign
0 (0.0%)
of 19 (2.0%)
math/rand.(*Rand).Intn
1 (0.1%)
of 8 (0.8%)
2
5
2
10
9
7
runtime.settype_flush
2 (0.2%)
runtime.gc
0 (0.0%)
of 5 (0.5%)
runtime.memclr
2 (0.2%)
hash_insert
7 (0.7%)
of 10 (1.1%)
hash_remove
3 (0.3%)
of 9 (0.9%)
math/rand.(*Rand).Int31n
3 (0.3%)
of 7 (0.7%)
5
3
6
4
gc
0 (0.0%)
of 5 (0.5%)
runtime.aeshashbody
9 (0.9%)
math/rand.(*Rand).Int31
0 (0.0%)
of 4 (0.4%)
1
3
4
runtime.parfordo
0 (0.0%)
of 1 (0.1%)
runtime.starttheworld
0 (0.0%)
of 3 (0.3%)
math/rand.(*Rand).Int63
0 (0.0%)
of 4 (0.4%)

# Also...

- What’s blocking on synchronization primitives?
- What’s causing the creation of OS threads?

# MEMORY RECYCLING

# Memory Statistics

- Read with `runtime.ReadMemStats(&m)`

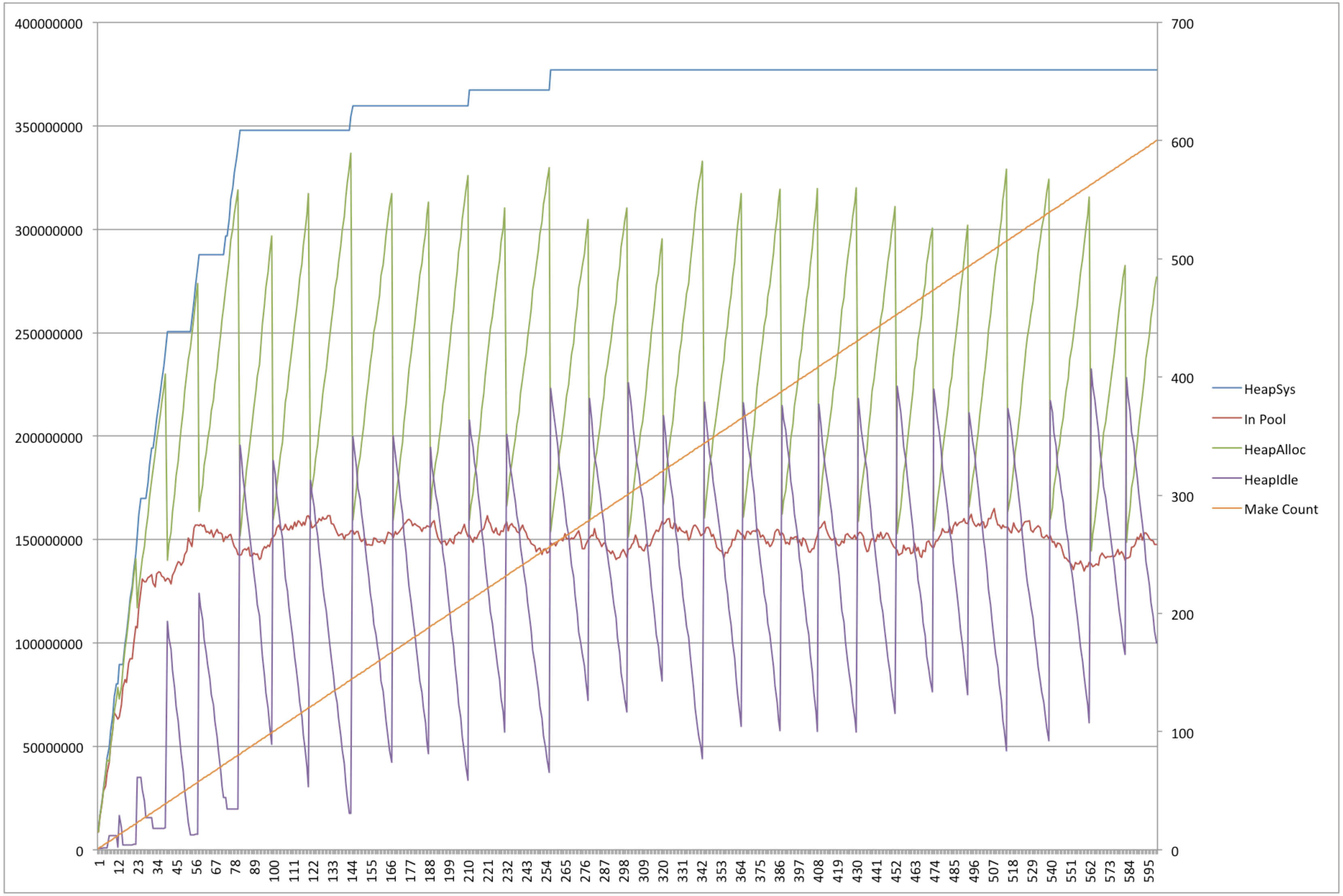
- The `MemStats` struct has tons of members

- Useful ones for looking at heap
  - `HeapInuse` - # bytes in the heap allocated to things
  - `HeapIdle` - # bytes in heap waiting to be used
  - `HeapSys` - # bytes obtained from OS
  - `HeapReleased` - # bytes released to OS

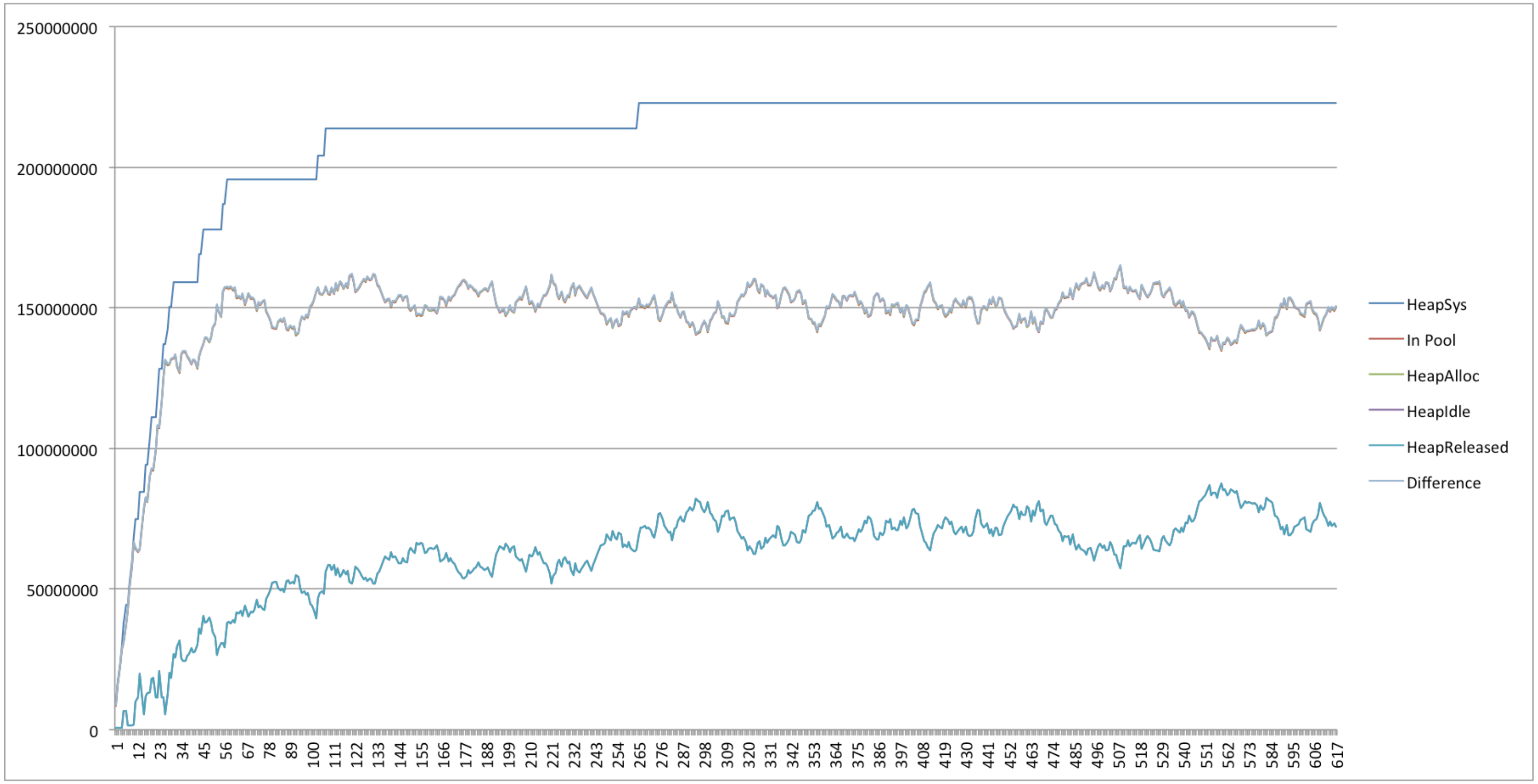
# Test garbage making program

```
func makeBuffer() []byte {
    return make([]byte, rand.Intn(5000000)+5000000)
}

func main() {
    pool := make([][]byte, 20)

    makes := 0
    for {
        b := makeBuffer()
        makes += 1

        i := rand.Intn(len(pool))
        pool[i] = b

        time.Sleep(time.Second)
    }
}
```

# What happens



# debug.FreeOSMemory()



# Use a buffered channel

```
func main() {
    pool := make([][]byte, 20)
    idle:= make(chan []byte, 5)

    makes := 0
    for {
        var b []byte
        select {
        case b = <-idle:
        default:
            makes += 1
            b = makeBuffer()
        }
        i := rand.Intn(len(pool))
        if pool[i] != nil {
            select {
            case idle<- pool[i]:
                pool[i] = nil
            default:
            }
        }

        pool[i] = b

        time.Sleep(time.Second)
    }
}
```

# `select` for non-blocking receive

A buffered channel makes a simple queue

```
idle:= make(chan []byte, 5)

select {
case b = <-idle:

default:
    makes += 1
    b = makeBuffer()
}
```

Try to get from the idle queue

Idle queue empty? Make a new buffer

# `select` for non-blocking send

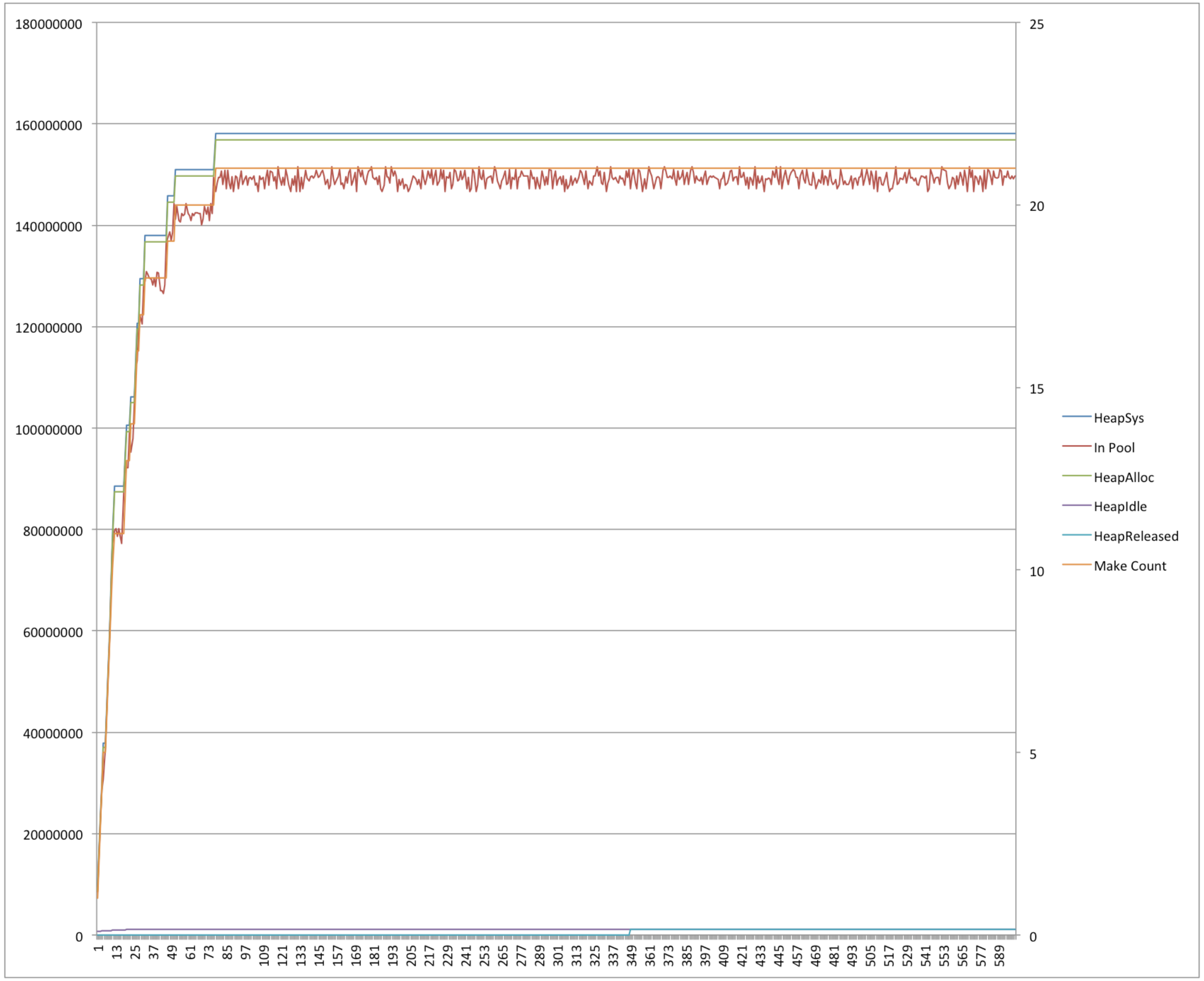
A buffered channel makes a simple queue

```
idle:= make(chan []byte, 5)

select {
case buffer <- pool[i]:
        pool[i] = nil

default:
}
```

Try to return buffer to the idle queue

Idle queue full? GC will have to deal with the buffer

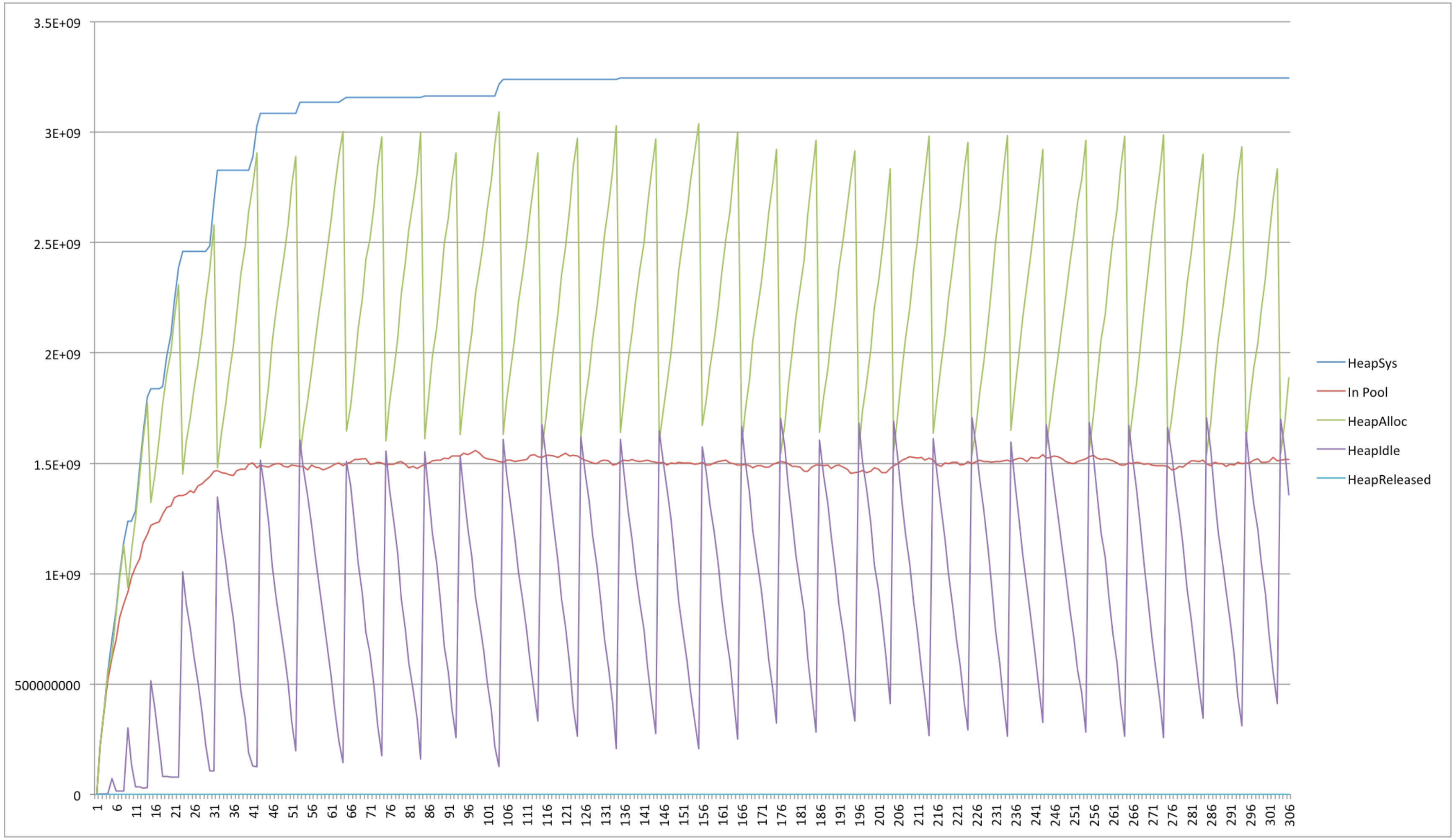
# What happens



# More realistic: 20 goroutines

```
func main() {
    pool := make([][]byte, 200)

    for i := 0; i < 10; i++ {
        go func(offset int) {
            for {
                b := makeBuffer()
                j := offset+rand.Intn(20)
                pool[j] = b

                time.Sleep(time.Millisecond * time.Duration(rand.Intn(1000))
            }
         }(i*20)
    }
}
```

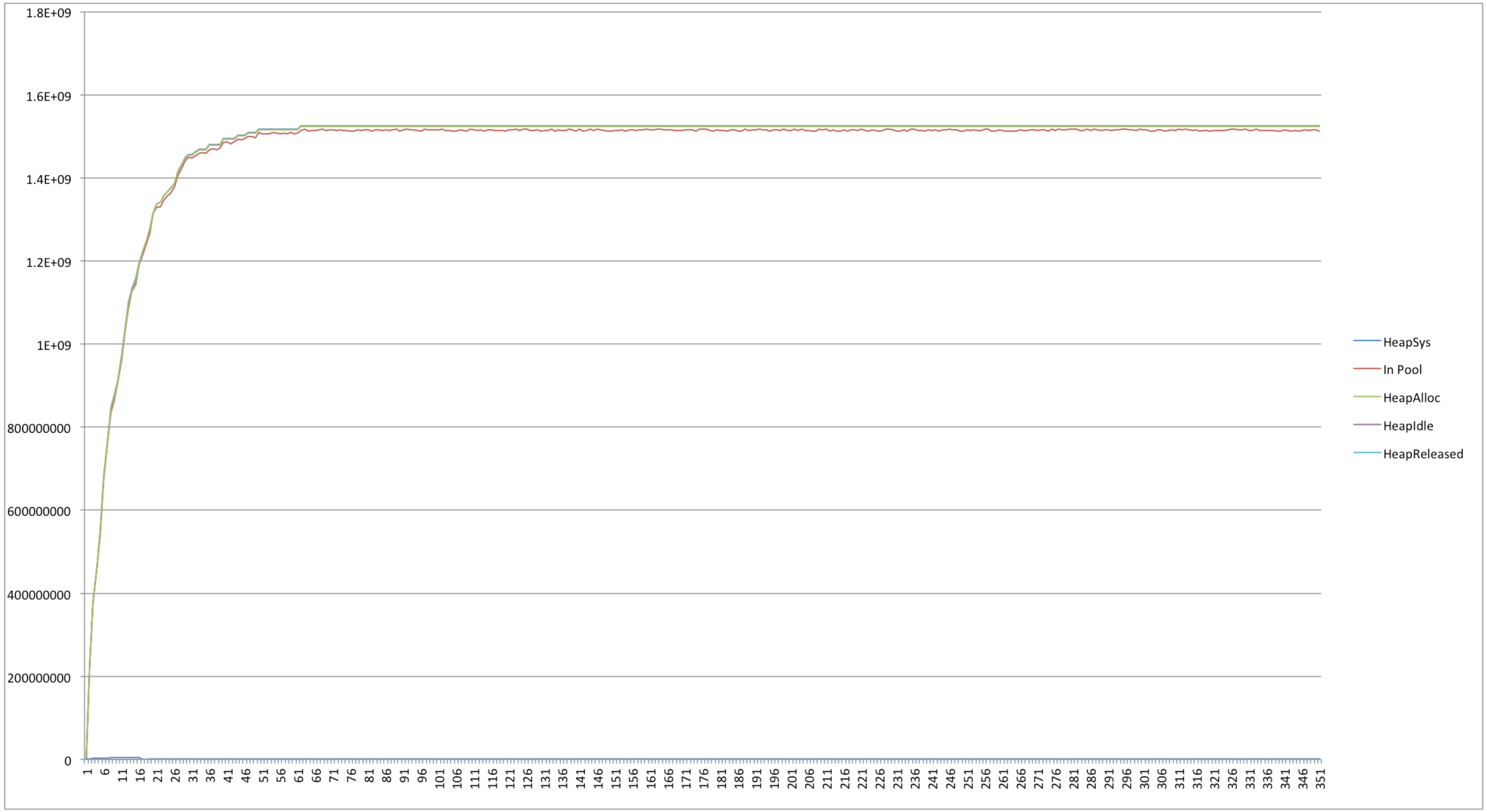
# What happens



# Shared across goroutines

```
func main() {
    buffer := make(chan []byte, 5)

    pool := make([][]byte, 200)
    for i := 0; i < 10; i++ {
        go func(offset int) {
            for {
                var b []byte
                select {
                    case b = <-buffer:
                    default: b = makeBuffer()
                }
                j := offset+rand.Intn(20)
                if pool[j] != nil {
                    select {
                        case buffer <- pool[j]: pool[j] = nil
                        default:
                    }
                }
                pool[j] = b
                time.Sleep(time.Millisecond * time.Duration(rand.Intn(1000))
            }
         }(i*20)
    }
```

# What Happens

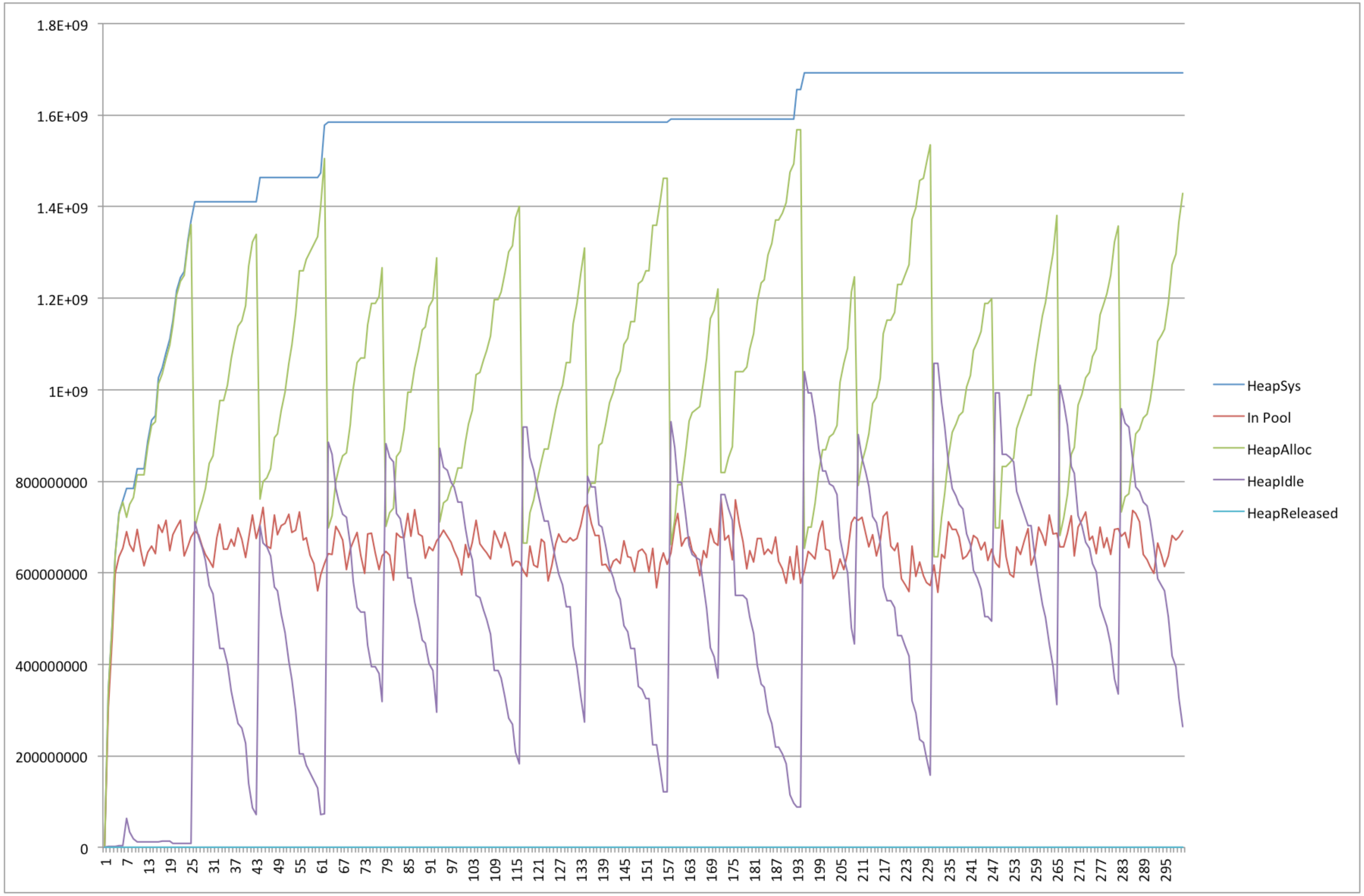


# More realistic example

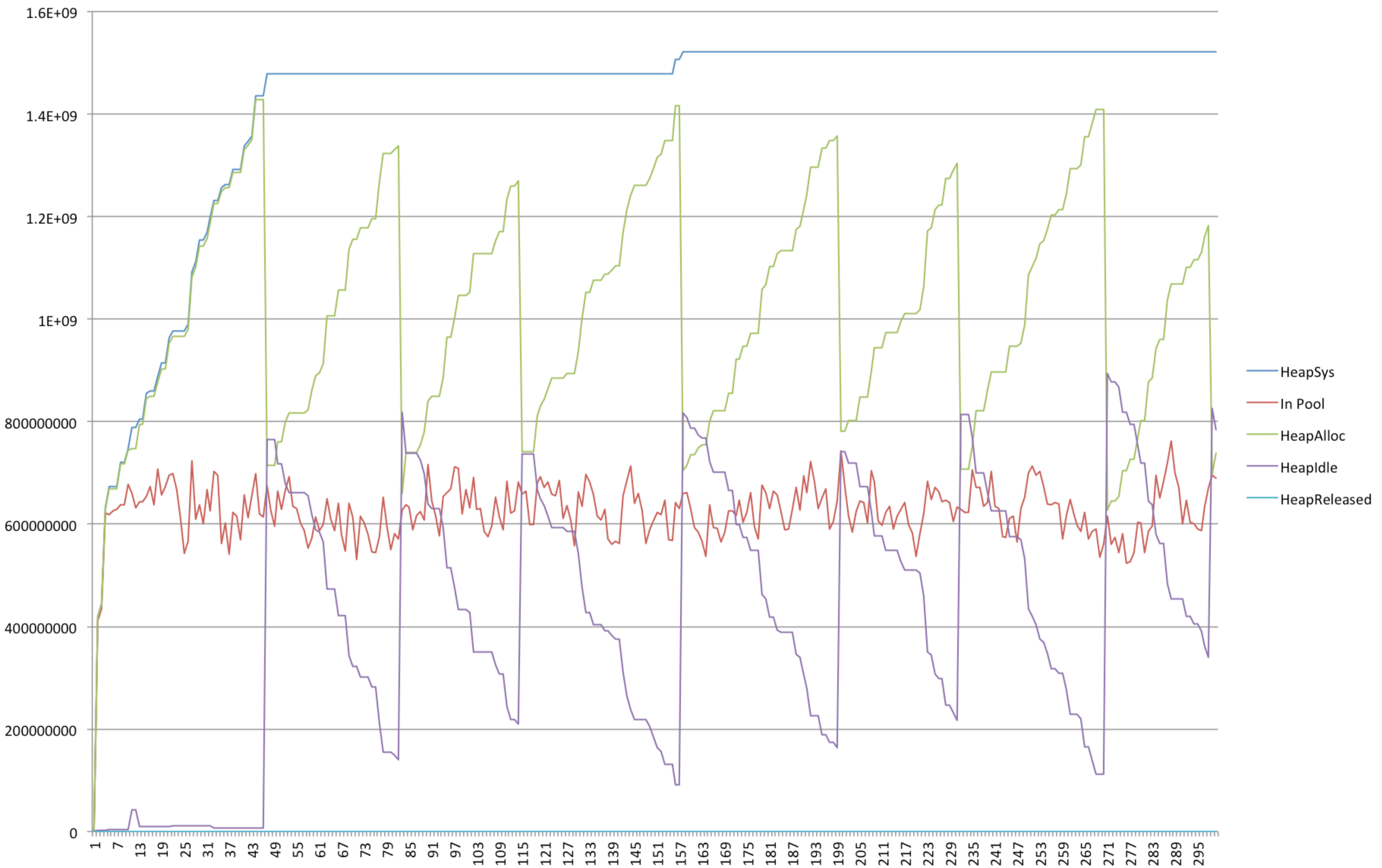
- Alter code to
  - Always try to give back a random buffer from the pool
  - 50% of the time get a new one

- Should create more garbage

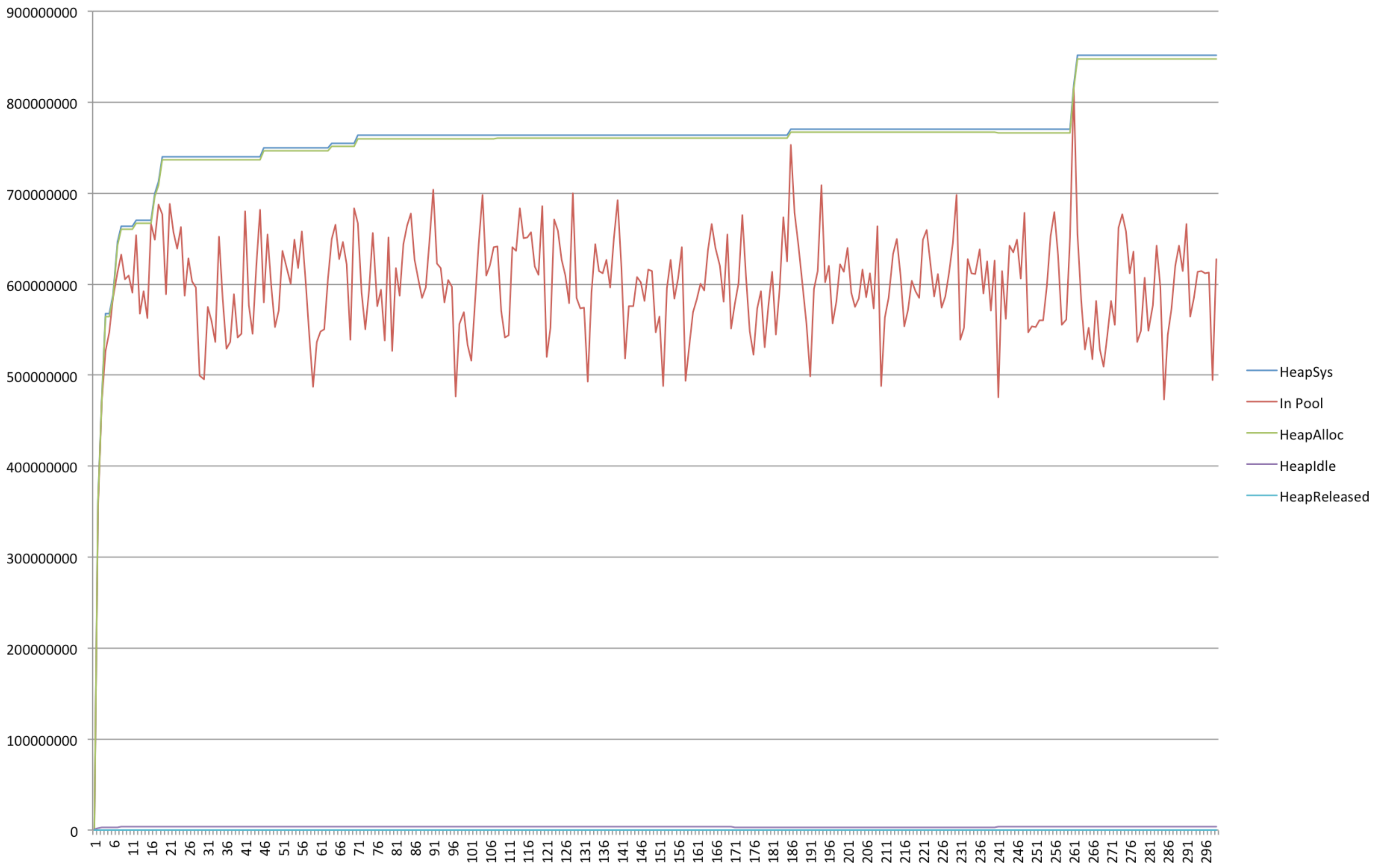
# Idle length 5



# Idle length 20



# Idle length 50



# Also

- This works for things other than []byte
    - Can be done with arbitrary types
    - Just need some way to reset

- There's a proposal to add something like this to the Go package library
    - sync.Cache/sync.Pool
    - Follow https://code.google.com/p/go/issues/detail?id=4720